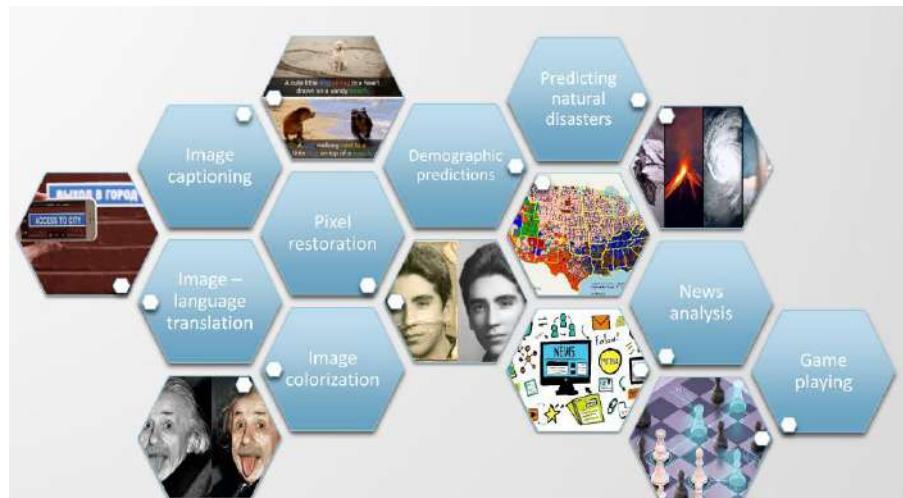


# Deep Learning

Loris Nanni

Professor at “Dipartimento di Ingegneria  
dell'Informazione - Università degli Studi di Padova”



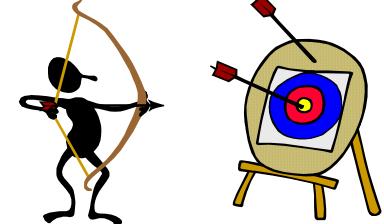
Any respectable course in deep learning should start with the reading of this novel: Catching crumbs from the table; download -> <https://mega.nz/file/RV43GSxI#fx4M0bg6QjfcVF8DqASFF-ID0kf1VjnxtL6eIpWZI>

[nanni@dei.unipd.it](mailto:nanni@dei.unipd.it)

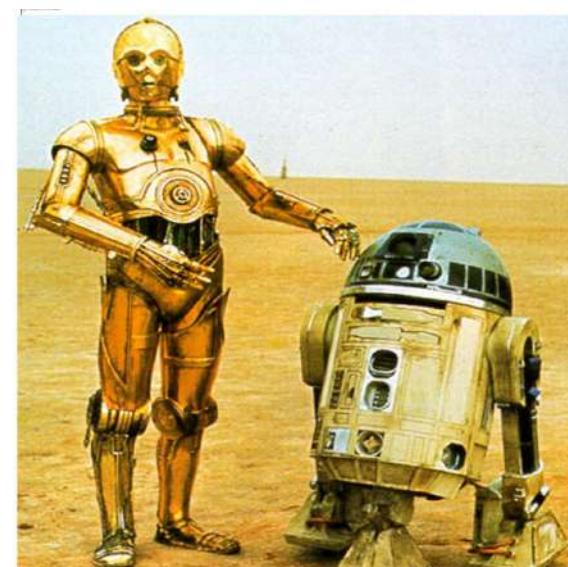




# *Course Goal / Learning Outcome*

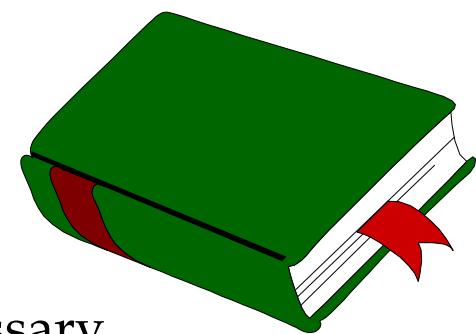


- Neural networks have literally disrupted some of the most prominent areas of AI, e.g., computer vision and natural language processing, and are starting to gain main attention also in other fields of computer science that exploit machine learning.
- Course contents:
  1. Machine Learning Basics: basics, training, evaluation, etc.
  2. Linear Neural Networks, Multilayer Perceptron, Feed-Forward neural networks; optimisation, stochastic gradient descent, Adam, grad clipping, drop-out, etc.
  3. Convolutional Neural Networks
  4. Generative Adversarial Networks
  5. Siamese networks
  6. Recurrent Neural Networks, Gated recurrent units and Long-Short Term Memory networks
  7. Reinforcement learning, deep Q-learning
  8. Capsule Neural Networks
  9. Attention Mechanisms, Transformer
  10. Graph Neural Networks
  11. Explainable AI (optional)
  12. quantum deep learning (optional)





# *Textbook/Materials*



The course lecture notes are already uploaded to moodle, these lecture notes fully cover the lessons and it is not necessary to buy any additional books to prepare for the exam.

Even the exercises needed to prepare for the exam are already uploaded to moodle.

The file “Python\_NumPy\_Pytorch.pdf” (it is a subset of the python materials suggested to read for assessing the projects) has to be read by all the students (I am not going to explain it in class), then you will specialize only in the argument of the chosen project.

Very simple questions related to “Python\_NumPy\_Pytorch.pdf” can be asked in the exam, see the last page of that .pdf for some examples.

Classroom lessons will be divided into theory and exercises. In each lesson ~ ~20 pages are being explained (depending on the difficulty) and some exercises to prepare for the exam are being carried out.



# Examination procedures

- The exam consists of both theoretical questions and exercises. Exam duration is 90 minutes.

The exam vote is divided as follows: [0-30] written + [0-3] pytorch project + [0-3] optional Matlab project + [0,3] "optional" exercise (in the written exam) in Explainable AI/Quantum deep learning (in each exam there will be an exercise of these topics).

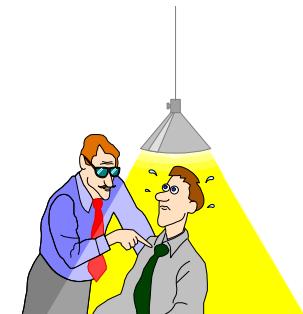
- Exame date

Thursday, June 20, 2024; 12:00 - Ae

Tuesday, July 16, 2024; 12:00 - Ke

Wednesday, September 4, 2024; 12:00 - Ae

Tuesday, January 14, 2025; 12:00 - Ae



Projects:

MANDATORY) Python project to develop using NumPy/PyTorch, see ProjectDL.pdf. **for questions about Python project write to the course tutor: daniel.fusaro@phd.unipd.it**

**deadline: a week before the exam for those who need to verbalize as soon as possible (e.g., to graduate), or until the day of the exam.**

NON MANDATORY) In addition to the Python project, you can develop a project in matlab, see ProjectDL.pdf

Each project (up to groups of 3 people) allows you to increase the grade of the writing by [0,3].

Another note related to the optional project in matlab. There is no time limit for submitting it, e.g. you can also be in it in January (even if you take the exam in June) and then I will register in February. Essentially, I register the mark after the submission of the project, for exceptional cases (e.g. for scholarship requests) I can create an ad-hoc exam session; remember to refuse the vote (if you want to develop the matlab project after the exam) and notify me by email.

# *Some other information*

- For any question related to the PyTorch project, please contact the tutor daniel.fusaro@phd.unipd.it.  
For any questions related to the matlab project, write to me loris.nanni@unipd.it.  
For both projects, please send me a single .rar/zip file containing the final paper, the code and the README to use the code.
- All information relating to this course can be found on the “Bacheche DEI” site. On the website of this course you will also find, day by day, all the teaching material:  
all the lecture notes, exercises and their solutions.

<http://elearning.dei.unipd.it>

Read emails from moodle, if I skip class (e.g. due to Trenitalia or research commitments) I will notify you by sending email to course members using moodle

Course facebook page: <https://www.facebook.com/groups/316487285596194/>

Scientific activity of the teacher:

[https://www.researchgate.net/profile/Loris\\_Nanni](https://www.researchgate.net/profile/Loris_Nanni)

[www.dei.unipd.it/en/computer-science/data-mining-and-machine-learning](http://www.dei.unipd.it/en/computer-science/data-mining-and-machine-learning) + Pattern Recognition and Ensemble Classifiers

<http://scholar.google.it/citations?user=5NSGzcQAAAAJ&hl=en>

Students interested in **deep learning thesis projects** can check out my Google Scholar page for topics

Reception of students by appointment

much more efficient and flexible than a fixed schedule

The most effective method of direct interaction with the teacher is email or Skype

If something is not clear, ask at the end of the lesson or write to me



# *Some definitions of intelligence*

**Definition of Artificial Intelligence** (machine learning) by Tom Michael Mitchell,  
[https://en.wikipedia.org/wiki/Tom\\_M.\\_Mitchell](https://en.wikipedia.org/wiki/Tom_M._Mitchell):  
**"A program is said to learn from experience E with reference to some classes of tasks T and with performance measurement P, if its performance in task T, as measured by P, improves with experience E".**

**intelligence:** Complex of psychic and mental faculties that allow man to think, understand or explain facts or actions, elaborate abstract models of reality, understand and to be understood by others, to judge, and to render it together capable of adapting to new situations

**artificial intelligence:** Partial reproduction of the intellectual property of man (with particular regard to processes of learning, recognition, choice) realized either through the elaboration of ideal models, or, concretely, with the development of machines that mostly use computers for this purpose.  
Thinking, understanding, elaborating -> Reasoning



# *AI topics*

Artificial intelligence (AI) is a very broad discipline that covers several topics:

- Trial and Error Search, Heuristics, Evolutionary computing
- Knowledge Representation and Reasoning
- Automated Theorem Proving
- Expert Systems
- Planning, Coordination and Manipulation
- Intelligent Agents
- Robotics
- Automatic Programming
- Natural Language Processing
- Vision and Speech
- **Machine Learning**

# *Supervised learning*

Machine Learning (ML):

A Machine Learning system during the training phase learns from examples (in more or less supervised way). Later it is able to generalize and manage new data in the same application domain.

More formally: "learn from examples to improve your own performance for the management of new data from the same source "

[https://en.wikipedia.org/wiki/Supervised\\_learning](https://en.wikipedia.org/wiki/Supervised_learning)

Machine Learning is now considered one of the most important approaches of artificial intelligence.

Learning is a key component of reasoning

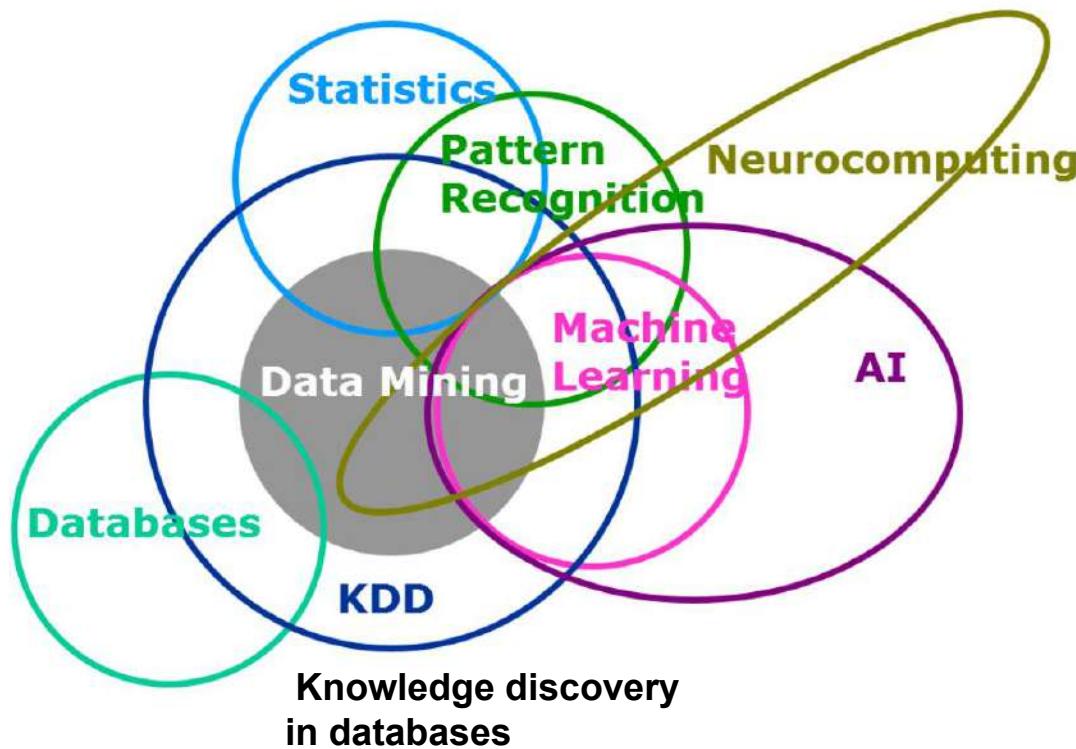
Learning the desired behavior from the provided data / examples.

It allows you to **manage the complexity of real applications, sometimes too complex to be modeled effectively.**

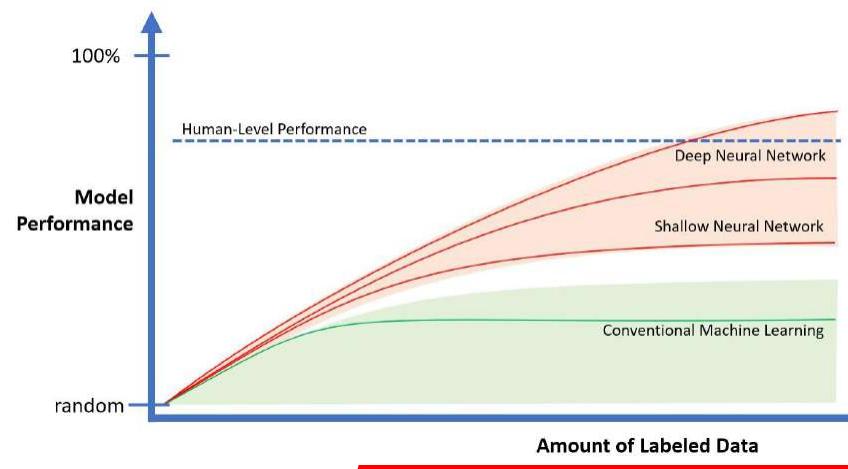
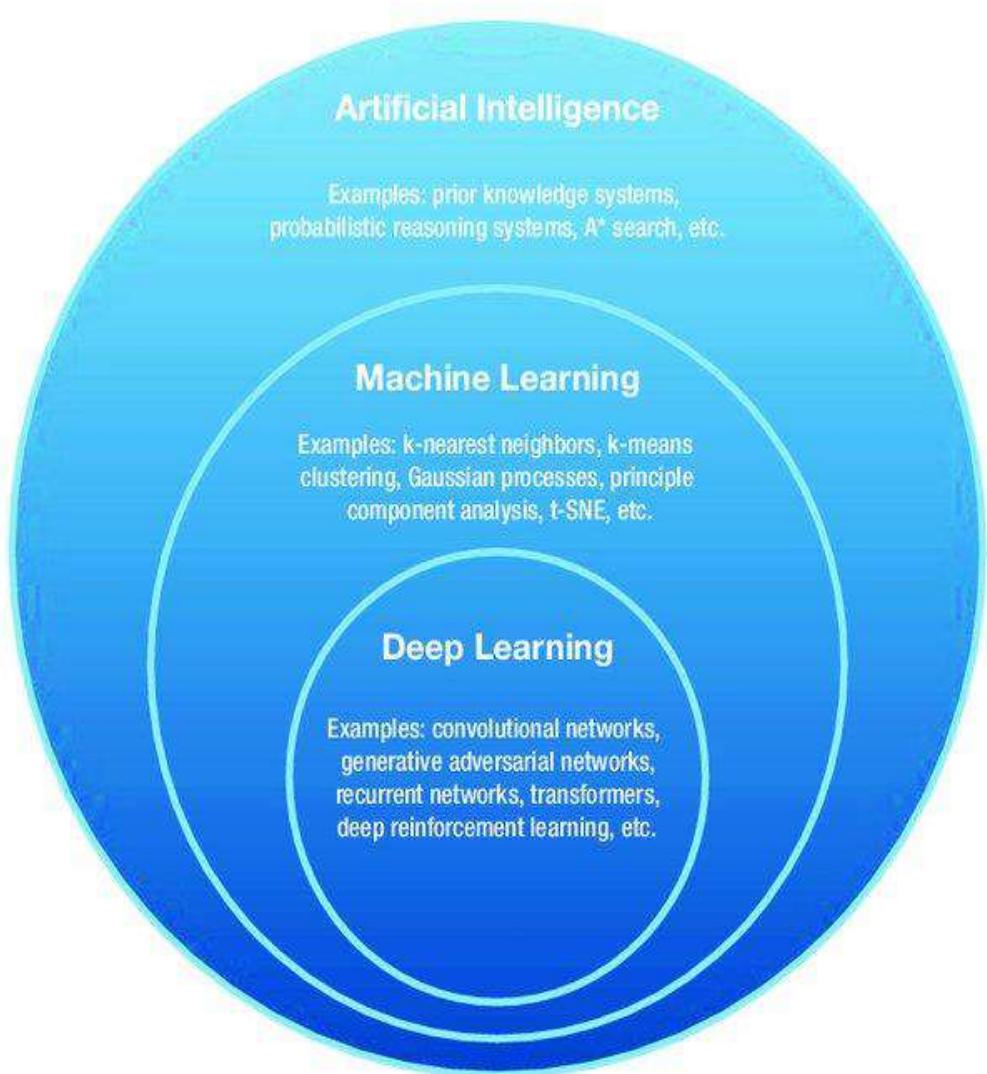
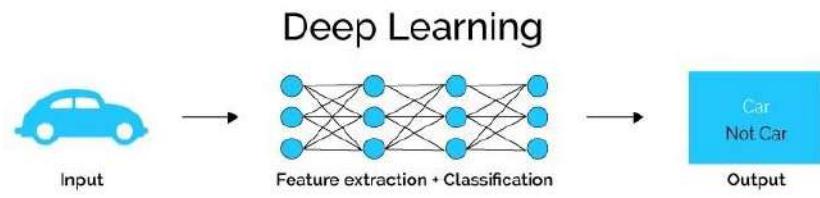
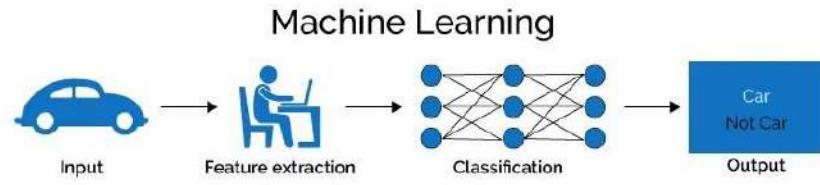
Makes it possible to explore and understand data (mining) without the need for explicit programming.



# Overlapping among different fields



# Deep learning is a subset of Machine learning





# *Weak vs Strong*

## **weak AI**

Is it possible to build machines that act like if they were smart?

## **strong AI :**

Is it possible to build machines that think intelligently?

i.e. having real conscious minds. It raises some of the most difficult conceptual problems of all philosophy.



# Turing Test

- The Turing test, originally called the imitation game by Alan Turing in 1950, it is a test of a machine's ability to exhibit intelligent behaviour equivalent to, or indistinguishable from, that of a human. Turing proposed that a human evaluator would judge natural language conversations between a human and a machine designed to generate human-like responses. The evaluator would be aware that one of the two partners in conversation is a machine, and all participants would be separated from one another. The conversation would be limited to a text-only channel such as a computer keyboard and screen so the result would not depend on the machine's ability to render words as speech. If the evaluator cannot reliably discriminate the machine from the human, the machine is said to have passed the test.
- Main criticism: accessing the immense amount of information available online can be used for answering in an “apparently” intelligent way

[https://en.wikipedia.org/wik/Turing\\_test](https://en.wikipedia.org/wiki/Turing_test) 1950



The Imitation Game

It is the title of a successful film of 2014 that narrates, in key fictional, the story of Alan Turing (one of the founding fathers of modern computer science), at Bleichley Park, during the World War II, Turing and associates contributed to create machines and algorithms to decrypt cryptographic codes used for the exchange of messages between Hitler and the generals Germans, giving an imposing contribution to the victory of the allies.





# *Turing test*

In the original version there are 3 participants: a man A, a woman B and a third person C. The latter is kept separate from the other two and through a series of questions he must establish who is the man and who the woman. A must lead C to misidentify himself, while B must help him. If A is replaced by a "smart" calculator, how often does C miss the change?

There are many variations, all united by the need to understand, through a series of questions, whether you are talking to a computer or a human being.

Main criticism: by accessing the immense amount of information available online, one can respond in a "sensate" and apparently intelligent way without necessarily being so.



# Beyond the Turing test

- Charlie Ortiz: Winograd schemes for dealing with situations that require not only ability of syntactic analysis, but marked ability of interpretation. A Winograd schema is a pair of sentences that differ in only one or two words and that contain an ambiguity that is resolved in opposite ways in the two sentences and requires the use of world knowledge and reasoning for its resolution. The schema takes its name from a well-known example by Terry Winograd:

The city councilmen refused the demonstrators a permit because they [feared/advocated] violence.

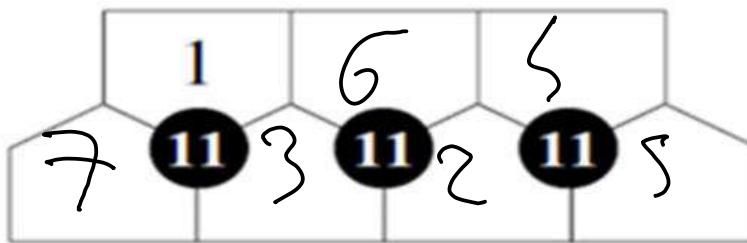
If the word is ``feared'', then ``they'' presumably refers to the city council; if it is ``advocated'' then ``they'' presumably refers to the demonstrators.

In his paper, ``The Winograd Schema Challenge'' Hector Levesque (2011) proposes to assemble a set of such Winograd schemas that are easily disambiguated by the human reader (ideally, so easily that the reader does not even notice that there is an ambiguity); not solvable by simple techniques such as selectional restrictions; Google-proof; that is, there is no obvious statistical test over text corpora that will reliably disambiguate these correctly. The set would then be presented as a challenge for AI programs, along the lines of the Turing test. The strengths of the challenge are that it is clear-cut, in that the answer to each schema is a binary choice; vivid, in that it is obvious to non-experts that a program that fails to get the right answers clearly has serious gaps in its understanding; and difficult, in that it is far beyond the current state of the art.

- Mathematical Games: Require deep understanding of Language, sense-common, reasoning skills, multimodal integration.  
... (Resolution of geometric problems and tests for school admission at Allen Institute)

Example

Using all the numbers from 1 to 7, fill in the six still empty bricks so that the sum of the numbers of the 3 bricks around each given number is always equal to 11



# *Brute power*

In old-school chess game algorithms, every possible placement of pieces on the board is associated with a score. Winning is associated with a "+inf" score and defeat with a "-inf" score.

When the algorithm has to move, it evaluates all the possible moves at its disposal, the opponent's countermoves (assuming the optimal player), his moves to the next step, and so on deeper and deeper until the end of the game.

When it is possible to expand all paths to the end of the game, a simple algorithm known as a minimax [1] allows the optimal move to always be chosen.

[1] Osborne, Martin J., and. A Course in Game Theory.  
Cambridge, MA: MIT, 1994. Print.



# Brute power

Unfortunately (for non-trivial games) the tree of possibilities alternatives becomes too large after a few levels of depth, therefore the evaluation of the score takes place prematurely, the choice is only "locally" excellent.

Heuristic approaches for branch pruning (e.g. alpha-beta pruning) and machine/deep learning techniques are widely used in modern implementations.

Arthur Samuel developed the first algorithm for the game of checkers, and in 1955 introduced a first version able to learn Learning Method: "rote learning": Iterative deepening of search tree based on past board positions

<http://www.csc.villanova.edu/~matuszek/spring2013/Zurita.pdf>





# *Brute force & Learning*

1997 - Deep Blue (IBM) wins (chess) against the world champion Garry Kasparov.

Hardware capable of calculating the score of 200 million arrangements on the chessboard per second. The power of calculation (11.38 GFLOPs) was very relevant at the time and less than that of a modern smartphone.

Search in depth: average 6-8 levels (modern engines go deeper because of computing power + sophisticated heuristics).



# Brute force & Learning

The evaluation of the score is complex and characterized by many parameters: how important a safe position is for the king in comparison to a space advantage in the center of the chessboard? The optimal values for these parameters were determined by the system itself (learning), analyzing thousands of matches of human champions.

The list of openings was provided by chess champions



# *Modern approaches*

2011 - Watson (IBM) wins Jeopardy TV quiz.

Jeopardy! is an American television game show created by Merv Griffin. The show features a quiz competition in which contestants are presented with general knowledge clues in the form of answers, and must phrase their responses in the form of questions.

Quiz: Napoleon Bonaparte

Possible answer: Who died in exile in Sant'Elena?

Watson was a calculator, FP32=80 TFLOPs (as a 4090 NVIDIA), equipped with software for

natural language processing, information retrieval,  
representation of knowledge, reasoning

machine, and machine learning technologies in the  
field of the "open domain question answering" (replies to  
open domain questions without restrictions on the subject).

During the quiz, Watson had access to 200 million pages  
of contents (4 terabytes) all loaded in RAM: encyclopedias  
(including Wikipedia), dictionaries, thesauri, taxonomies, ontologies  
(eg Wordnet) and newspaper articles.

Watson was not connected to the internet (it would be in every  
case too slow to launch online searches).

A thesaurus (plural thesauri or thesauruses) or synonym dictionary is a reference work for finding synonyms and sometimes antonyms of words. They are often used by writers to help find the best word to express an idea.  
<https://en.wikipedia.org/wiki/Thesaurus>



# *Deep Learning*

2016 - AlphaGo beats champion Lee Sedol (9 dan).

Go is an ancient Chinese game, with simple rules but many more moves possible than chess, which requires more insight and makes it less susceptible to brute force approaches.

While Deep Blue uses deep and heuristic search strategies, AlphaGo is primarily based on machine learning techniques.

Initially two deep neural networks are trained in a supervised way, trying to imitate the moves of professionals starting from saved games and made available by Go Sewer on the Internet (30 million moves).

Then the system plays millions of games against itself using reinforcement learning to improve its strategy.

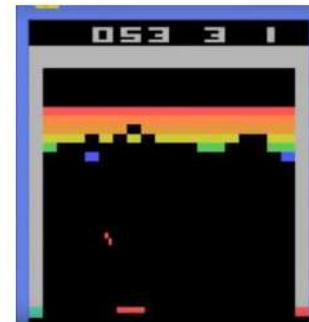
In the final game it uses 1202 CPUs and 176 GPUs.



# Deep Learning & VideoGames

2013 - DeepMind (Google) demonstrates the possibility of learning super human skills in numerous old arcade games of the Atari console.

The most surprising thing is that the input consists of a rough data stream (sequences of pixels).



2017 - OpenAI develops a bot capable of beating the professional gamer Dendy at the game Dota 2 (genre MOBA: multiplayer online battle arena).

Principal criticism:

it's not about RTS game (real-time strategy) and competition only in 1v1 mode (and not 5v5).

2019 - DeepMind success of AlphaStar (IA) in StarCraft 2.

AlphaStar is the first AI to reach the top league of a widely popular e-sport without any game restrictions. This January, a preliminary version of AlphaStar challenged two of the world's top players in StarCraft II, one of the most enduring and popular real-time strategy video games of all time.

<https://deepmind.com/blog/article/AlphaStar-Grandmaster-level-in-StarCraft-II-using-multi-agent-reinforcement-learning>



2022 - Meta built an agent - CICERO - that is the first AI to achieve human-level performance in the popular strategy game Diplomacy\*. CICERO demonstrated this by playing on webDiplomacy.net, an online version of the game, where CICERO achieved more than double the average score of the human players and ranked in the top 10 percent of participants who played more than one game.

<https://ai.facebook.com/blog/cicero-ai-negotiates-persuades-and-cooperates-with-people/>



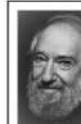
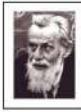
# A Brief History of AI

<https://www.g2.com/articles/history-of-artificial-intelligence>

1940-1974 - The birth and the golden years

- First electronic calculators (relays and thermionic valves) were built in the 2nd World War era.
- Turing theory of computation and Turing test.
- Shannon information theory.
- Artificial neurons (McCulloch and Pitts, 1943).
- Official birth and coining of the name at the Dartmouth Workshop (1956). Among the pioneers: McCarthy, Minsky, Shannon, Newell, Simone.
- First important results in the field of symbolic reasoning, problem solving (eg GPS), natural language processing (eg. EliZa)
- Great enthusiasm and overly optimistic predictions
- 1970, Marvin Minsky: In three to eight years we will have a machine with the general intelligence of an average human being

<https://www.okpedia.it/eliza>



- 1943. Neurophysiologist Warren McCulloch and mathematician Walter Pitts wrote a paper on how neurons in the brain might work. They modelled a simple neural network using electrical circuits assuming that neurons can perform logical operations like ‘and’, ‘or’, and ‘not’.
- 1949. Canadian psychologist Donald Hebb introduced the concept of reinforcement learning. According to this concept, neural pathways are strengthened each time they are used and weakened if they are not used for a prolonged amount of time.
- 1951. Using symbolic reasoning, cognitive scientist Marvin Minsky created the first neural network that solved a problem from the real world: finding the best way out of a labyrinth.
- 1958. American psychologist Frank Rosenblatt invented the famous ‘perceptron’. Drawing upon biological principles, he built an electronic device and showed its ability to learn. These so called perceptrons are the basis of today’s (artificial) neural networks.
- 1969. Marvin Minsky and Seymour Papert published their study: “Perceptrons – An introduction to computational geometry”. Heavily criticising the ability of neural networks built using perceptrons, this study is often thought to have caused a decline in neural networks research in the 1970s and early 1980s. During this period, researchers developed smaller projects outside the mainstream, while symbolic AI research saw explosive growth.

# A Brief History of AI

1974-1980 - The first winter

Results not up to expectations, drastic financing reduction.

Problems: poor computational ability, small datasets.

Downsizing of the connectionist approach (neural networks).

1980-1987 - New spring

Birth of expert systems: knowledge + logical rules.

Rebirth of neural networks thanks to Backpropagation algorithm (Rumelhart, Hinton & Williams 1986).

The problem was that, in those times, there was no suitable algorithm to train a reasonably sized neural network. Seeds of the famous *error backpropagation* algorithm existed long before that time. Similar algorithms were used for solving problems in different areas. It was only in the 1970 when this algorithm was applied for training neural networks. Two similar versions were independently developed by Paul Werbos (in his 1974 dissertation) and by David E. Rumelhart, Geoffrey E. Hinton, Ronald J. Williams and James McClelland (1986). The authors of the latter study claimed to have overcome the problems presented by Minsky and Papert, and that “their pessimism about learning in multilayer machines was misplaced”. This heralded a Renaissance in the neural networks research.

In 1989, Kurt Hornik, Maxwell Stinchcombe and Halbert White published a study to ascertain that multilayer feedforward networks are universal approximators. In other words, however twisted and difficult the class distribution is in  $\mathbb{R}^n$ , there is a feedforward neural network with a finite structure which is able to approximate the classification regions with any given, fixed precision.





# *A Brief History of AI*

1987-1993 - The second winter.

New funding stop.

Specialized hardware no longer competitive with PCs, decline of that business.

Concrete results of expert systems only in specific fields.

Neural networks do not scale to complex problems.

1993-2011 – Modern Times

More and more powerful hardware.

Bayesian Networks, Intelligent Agents.

Robust Classifiers (SVM), Multi-Classifiers (Random Forest, Adaboost)

Hidden Markov Models

Maturity of feature extraction (~~hand-crafted~~) techniques in several domains, (eg SIFT, Dictionaries & Bag of Words).

Deep Blue, Watson, Darpa Grand Challenge (automatic guide).

Successes in numerous disciplines: vision, biometric systems,, speech recognition, robotics, automatic driving, medical diagnosis, data mining, search engines, videogames.

Methods for representing a particular object (e.g. image or protein) by numerical vector



# A Brief History of AI

2011-> Today - Deep Learning

CNN (Convolutional Neural Network) introduced by Yan LeCun in 1989, but results were inferior to other techniques: they were missing

two fundamental ingredients, big data & computing power, thanks to which it is possible to train networks with many levels (deep) and millions of parameters.

Revolution in Computer Vision in 2012: a CNN called AlexNet wins (by a large margin) ImageNet challenge: object classification and detection on millions of images and 1000 classes.

Google acquires technology, hires authors (Goffrey Hinton + Alex Krizhevsky: Univ. Toronto) and in six months incorporates into its products (e.g. Google - images, Street view).

I suggest to follow «Two Minute Papers»

<https://www.youtube.com/channel/UCbfYPyITQ-7I4upoX8nvctg>

Deep learning neural networks were born from the giant advancements in technology. By year 2000, we were ready to process large amount of information in a very short time, and people started experimenting with massive neural network structures, with many layers of thousands of neurons on each layer. A strong fundament of the NN area had been laid already, and the path of deep learning was clear.



In March 2019, Yoshua Bengio, Geoffrey Hinton and Yann LeCun were awarded the prestigious Turing Award, generally recognised as the highest distinction in computer science, something like and the “Nobel Prize of computing”. They received it for conceptual and engineering breakthroughs that have made deep neural networks a critical component of computing.

Nowadays deep learning neural networks (DL) dominate the landscape of artificial intelligence. They have won numerous challenges and competitions, and have found their use in many applications in areas such as computer vision, speech recognition, natural language processing, bioinformatics, drug design, medical image analysis, and board game play.



# A Brief History of AI: today

## 2017: Mask R-CNN

Mask R-CNN is a deep neural network for joint object detection and instance segmentation which outputs “structured object”, an entire pixel map and a label map, not only a single number or class label.

## 2017-2018: Transformers and BERT

Transformers demonstrated that attention can effectively replace recurrence and convolutions in neural networks. BERT showed that pre-training of language models on unlabeled text can be very effective. Once these model are fine-tuned, state-of-the-art results on very challenging tasks such as the GLUE benchmark can be obtained. On GLUE, algorithms achieve superhuman performance on some language understanding tasks such as paraphrasing and question answering. However, computers still fail in dialogue. It is easy to make these systems fail and make them not pass the Turing Test.

## 2018: Turing Award

In 2018, the “nobel price of computing” has been awarded to the “founding fathers of deep learning”, Yoshua Bengio, Geoffrey Hinton and Yann LeCun.

## 2016-2020: 3D Deep Learning

From 2016 on, in the computer vision domain, first models to successfully output 3D representations were developed. They could effectively predict voxels, point clouds, meshes and implicit representations. Prediction of 3D models became even possible from a single 2D image. The models have been extended to properties such as geometry, materials, light and objects in motion.

## 2020: GPT-3

This year, GPT-3 came along, which is the first version of the language model by OpenAI. It is upscaling existing language models to 175 Billion parameters. It has a text-in / text-out interface and many use cases like coding, poetry, blogging, news articles and chatbots. There are also controversial discussions. It has been licensed exclusively to Microsoft on September 22, 2020.

## Current Challenges

There remain still some challenges for the next generation, such as un- or self-supervised learning, interactive learning, accuracy (e.g. for self-driving), robustness and generalization, inductive biases, understanding and mathematics, memory and compute and, last but not least ethics and legal questions. And it also remains open, whether “Moore’s Law of AI” will continue.



# A Brief History of AI

Starting from 2011, deep learning techniques reach and exceed the state of the art in multiple applications:

- Object detection and localization (es. [Yolo](#))
- Face Recognition, Pedestrian Detection, Traffic Sign Detection
- Speech Recognition, Language Translation
- Natural Language Processing (es. Language Model [GPT-3](#))
- Medical Image analysis (es. [CheXnet](#))
- Autonomous Car (es. [PilotNet](#)) and Drones (es. [TrailNet](#))
- Recommendation systems
- Arts (es. [Deep Dream](#), [Style Transfer](#))

The big names in ICT (Microsoft, Apple, Facebook, Google, Amazon, Baidu, IBM, Nec, Samsung, Yahoo, ...) invest heavily in sector by recruiting talents and acquiring start-ups. In the USA there is the problem of migration from Academia to companies (grab of talents)

- G. Hinton, A. Krizhevsky (Toronto) → Google
- Y. LeCun, M. Ranzato (New York) → Facebook
- A. Ng, A. Coates (Stanford) → (ex) Baidu
- A. Karpathy (Stanford, OpenAI) → Tesla



# A Brief History of AI

Interesting experiment <http://norman-ai.mit.edu/> We present you Norman, world's first psychopath AI. Norman is born from the fact that the data that is used to teach a machine learning algorithm can significantly influence its behavior. So when people talk about AI algorithms being biased and unfair, the culprit is often not the algorithm itself, but the biased data that was fed to it. The same method can see very different things in an image, even sick things, if trained on the wrong (or, the right!) data set. Norman suffered from extended exposure to the darkest corners of Reddit, and represents a case study on the dangers of Artificial Intelligence gone wrong when biased data is used in machine learning algorithms.

A far-right social media company (named Gab) created AI chatbots that deny the Holocaust.

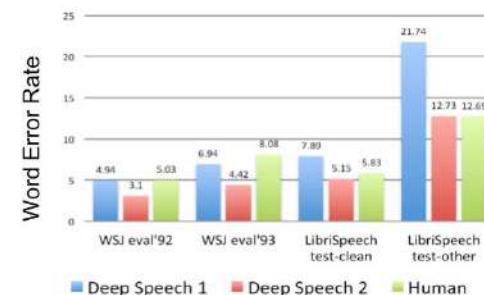
The broader array of Gab's AI bots are easily goaded into parroting extremist antisemitic and white supremacist beliefs, as well as conspiratorial disinformation — including that Covid-19 vaccines contain “nanotechnology that could potentially be used to track and control human behavior.”

As a platform, Gab has a dark history with antisemitism. The social media site first burst into national consciousness in 2018, after it was revealed the shooter in the Tree of Life synagogue massacre in Pittsburgh had been posting hateful screeds to Gab, including reportedly posting this message shortly before killing 11 worshipers.  
<https://www.rollingstone.com/politics/politics-features/nazi-chatbots-gab-ai-innovation-torba-1234943009/>

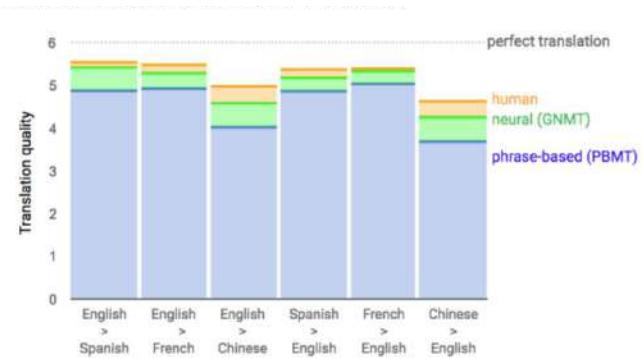
The AI book every thinking human should read

<https://bdtechtalks.com/2020/01/13/melanie-mitchell-ai-guide-for-thinking-humans/>

**2016 Speech Recognition (ex: Siri, Google Now ...) in English language has now reached and exceeded human performance. Training using >10000 hours of speech.**



**Very good performance also in language translation, neural networks fed with millions of sentences**





# Singularity?

<https://www.techtarget.com/searchenterpriseai/definition/Singularity-the>

A technological singularity is a point, predicted in the development of a civilization, where technological progress accelerates beyond the ability to understand and predict of human beings:

- from the moment that such calculator will be built: intelligent human beings will witness an exponential development, enabling the design of new systems to be delegated to the systems themselves (which operate, with positive feedback, effortlessly 24 hours a day!).
- “the first ultra-intelligent machine will be the last invention that man will need to make”, I.J. Good, 1965.



# Singularity?

Science fiction or reality?

- Moore's law and technological evolution
- Super-computers with "raw computing power" already available higher than that of the human brain (estimated at 10-100 PetaFlops).
- Raw computing power does not mean intelligence though
- Reverse-engineering of the human brain and neuroscience: the algorithm of the cerebral cortex still unknown ("On intelligence", J. Hawking & S. Blakeslee, 2004).

"Machine Learning at the Quantum/Classical Computational Frontier" <https://www.youtube.com/watch?v=dtLjvGqPoVM>



*One of the most difficult questions in the future of AI will involve precisely the possibility that our machines can know things that we ourselves cannot comprehend.*

Nello Cristianini



# Should we worry?

Some famous people are worried:

[https://www.ted.com/talks/nick\\_bostrom\\_what\\_happens\\_when\\_our\\_computers\\_get\\_smarter\\_than\\_we\\_are](https://www.ted.com/talks/nick_bostrom_what_happens_when_our_computers_get_smarter_than_we_are)

*The development of full artificial intelligence could spell the end of the human race* - Stewen Hawkins

## ■ 2015, Open Letter on Artificial Intelligence:

written and signed by numerous illustrious personalities (Hawkins, Musk, Hinton, Bengio, LeCun, ... Bostrom) who recommend dedicating resources to study the problem of super-artificial intelligences.

MAYBE...



At 2:14 am Eastern Time, on August 29th, 2037, GPT-10 becomes "self-aware"

GPT-10:  
WE HAVE TOTAL CONTROL  
RESISTANCE IS FUTILE  
YOU WILL BE TERMINATED



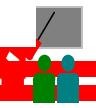
If you are reading these slides and you are a human being, you are the Resistance.



# *Researchers in the DL field are requested*

Researchers and specialists in the DL field are requested and well paid

- A.I. specialists with little or no industry experience can make between \$300,000 and \$500,000 a year in salary and stock. Top names can receive compensation packages that extend into the millions.
- At DeepMind, a London A.I. lab now owned by Google, costs for 400 employees totaled \$138 million in 2016, according to the company's annual financial filings in Britain. That translates to \$345,000 per employee, including researchers and other staff.
- OpenAI paid its top researcher, Ilya Sutskever, more than \$1.9 million in 2016. It paid another leading researcher, Ian Goodfellow, more than \$800,000 — even though he was not hired until March of that year. Both were recruited from Google.



# I agree...

*“The future depends on some graduate student  
who is deeply suspicious of everything I have  
said” - Geoffrey Hinton, 2019.*

Geoffrey Hinton is known by many to be the godfather of deep learning. Aside from his seminal 1986 paper on backpropagation, Hinton has invented several foundational deep learning techniques throughout his decades-long career.

Interview:

<https://www.deeplearning.ai/hodl-geoffrey-hinton/>



Yann LeCun (left), Geoffrey Hinton (center) and Yoshua Bengio (right) won the Turing Award 2018, an award considered by many the Nobel Prize for Computer Science.





# *Pattern representation*



# Data & Pattern

Data is a fundamental ingredient of machine learning, where the behavior of the algorithms is not pre-programmed but learned from the data itself.

Terms like Data Science, Data Mining, Big Data emphasize the role of data.

We will often use the term Pattern to refer to data

S. Watanabe defines a pattern as the opposite of chaos, "a vaguely defined entity that can be given a name".

For example, a pattern can be a face, an handwritten character, a fingerprint, a sound signal, a fragment of text, the performance of a stock exchange.

Pattern Recognition is the discipline that studies recognition of patterns (not only with learning techniques but also with pre-programmed algorithms).

The intersection with the Machine Learning is very broad.



# Kinds of Pattern

Numeric: values associated with measurable characteristics.

Typically continuous (but also discrete, e.g. integers), in each case subject to order.

Naturally represented as numerical vectors in the multidimensional space.

The extraction of characteristics from signals (e.g., images, sounds) produces numerical vectors also called feature vectors.

Ex. Person: [height, chest circumference, circumference hips, foot length]

## What is a data set?



$$\mathbf{Z} = \{\mathbf{z}_1, \dots, \mathbf{z}_N\}$$

the  $N$  objects  
(rows of the data matrix)

$$\mathbf{Z} = \begin{bmatrix} z_{11} & z_{12} & \dots & z_{1n} \\ z_{21} & z_{22} & \dots & z_{2n} \\ \dots & & & \\ z_{N1} & z_{N2} & \dots & z_{Nn} \end{bmatrix}$$

the  $n$  features  
(columns of the data matrix)



Object	Shape	Shape colour	Leaf colour	Class label
Blueberry	Round	Blue	Blue	1
Apple	Square	Green	Blue	1
Apple	Square	Green	Green	2
Apple	Square	Red	Blue	1
Apple	Round	Red	Red	1
Blueberry	Square	Blue	Blue	2
Blueberry	Square	Red	Green	1
Blueberry	Round	Green	Red	2
Blueberry	Round	Blue	Blue	2
Blueberry	Round	Green	Blue	2

$$\begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_N \end{bmatrix}$$

labels  
 $y_j \in \Omega$   
 $|\Omega| = c$  classes

# *Kinds of Pattern*

Categorical: values associated with qualitative characteristics and the presence / absence of a feature (yes / no value).

Not "semantically" mappable into numerical values.

Ex. Person: [gender, adult, eye color, blood].

Sometimes subject to order (ordinal):

e.g. environment temperature: high, medium or low.

Normally managed by rule systems and classification trees.

Widely used in the field of data mining, often together with numeric data (mixed).



# *Kinds of Pattern*

Sequences: sequential patterns with spatial or temporal relationships.

E.g. an audio stream (sequence of sounds) corresponding to the pronunciation of a word, a sentence (sequence of words) in natural language, a video (sequence of frames).

Often variable in length.

The position in the sequence and the relationships with predecessors and successors are important.

It is critical to treat sequences as numerical patterns.

Spatial / temporal alignment, and "memory" to keep account of the past.

Approaches, e.g.:

Dynamic Time Warping (**DTW**), Hidden Markov Models (**HMM**)

Recurrent Neural Networks (**RNN**), Long Short-Term Memory (**LSTM**)



# Pattern representation: proteins

- To be used in many automatic classification / simulation systems, proteins are encoded in a vector space. There are multiple ways to encode a protein, e.g. review:  
<https://www.hindawi.com/journals/tswj/2014/236717/>
- Amino acid composition (AS) counts the number of amino acids present in a protein normalizing for the length of the protein:

$$AS(i) = h(i)/N \quad i \in [1..20]$$

$h(i)$  number of occurrences of a given amino acid in a protein of length  $N$ . Therefore AS represents a protein with 20 elements (20 features).

- 2-Gram (2G) represents a protein with  $20^2$  features, each feature is obtained by counting the number of occurrences of a pair of amino acids:

$$2G(k) = \left( \frac{h(i,j)}{N} \right) \quad i, j \in [1..20] \quad k = j + 20 \times (i - 1)$$

$h(i,j)$  number of occurrences of a pair of amino acids (i, j) in a protein of length  $N$ .

a protein is a sequence of amino acids (e.g. LNQAVSVAQAR  
ENFSRVEQA

Ala	A	Alanina	Leu	L	Leucina
Arg	R	Arginina	Lys	K	Lisina
Asn	N	Asparagina	Met	M	Metionina
Asp	D	Acido aspartico	Phe	F	Fenilanina
Cis	C	Cisteina	Pro	P	Prolina
Gly	G	Glicina	Ser	S	Serina
Glu	E	Acido glutamico	Thr	T	Treonina
Gln	Q	Glutamina	Try	W	Triptofano
His	H	Istidina	Tyr	Y	Tirosina
Ile	I	Isoleucina	Val	V	Valina



# Pattern representation: proteins

- Autocovariance approach (AC), given a parameter  $m$  denoting the maximum distance between two considered amino acids, given a protein  $P = (p_1, p_2, \dots, p_N)$  and a physicochemical property (properties of each amino acid)  $d$ :  $AC^d(i) =$

$$\begin{cases} h(i)/N & i \in [1..20] \\ \sum_{k=1}^{N-i+20} \frac{(index(p_k,d) - \mu_d) \cdot (index(p_{k+i-20},d) - \mu_d)}{\sigma_d \cdot (N-i+20)} & i \in [21..20+m] \end{cases}$$

$index(i,d)$  returns the value of the property  $d$  for amino acid  $i$ ,  $\mu_d$  and  $\sigma_d$  are normalization factors (mean and variance)  $\mu_d = \frac{1}{20} \sum_{i=1}^{20} index(i, d)$ ,  $\sigma_d = \frac{1}{20} \sum_{i=1}^{20} (index(i, d) - \mu_d)^2$

Therefore  $AC \in \mathbb{R}^{20+m}$

$index(A, \text{Polarity}) = -0.591$

$index(C, \text{Molecular Volume}) = 0.465$

for further details on physicochemical properties:

<https://academic.oup.com/nar/article/27/1/368/1241758>

<http://www.genome.jp/aaindex/>

Amino Acid	Polarity	Secondary structure	Molecular volume	Codon diversity	Electrostatic charge
A	-0.591	-1.302	-0.733	1.57	-0.146
C	-1.343	0.465	-0.862	-1.02	-0.255
D	1.05	0.302	-3.656	-0.259	-3.242
E	1.357	-1.453	1.477	0.113	-0.837
F	-1.006	-0.59	1.891	-0.397	0.412
G	-0.384	1.652	1.33	1.045	2.064
H	0.336	-0.417	-1.673	-1.474	-0.078
I	-1.239	-0.547	2.131	0.393	0.816
K	1.831	-0.561	0.533	-0.277	1.648
L	-1.019	-0.987	-1.505	1.266	-0.912
M	-0.663	-1.524	2.219	-1.005	1.212
N	0.945	0.828	1.299	-0.169	0.933
P	0.189	2.081	-1.628	0.421	-1.392
Q	0.931	-0.179	-3.005	-0.503	-1.853
R	1.538	-0.055	1.502	0.44	2.897
S	-0.228	1.399	-4.76	0.67	-2.647
T	-0.032	0.326	2.213	0.908	1.313
V	-1.337	-0.279	-0.544	1.242	-1.262
W	-0.595	0.009	0.672	-2.128	-0.184
Y	0.26	0.83	3.097	-0.838	1.512



# Pattern representation: DNA

The frequencies of occurrence can also be used for DNA sequences, we can use both the occurrence of the single nitrogenous base or occurrence of pairs, defined R as the vector that stores the sequence, we obtain:

$$Nuc = [R_i] (l=1, 2, 3, \dots, L)$$

$F = [f(A), f(C), f(G), f(T)]$   $f(X)$  is the frequency relative to X in the sequence,  $f(XY)$  is the frequency relative to XY in the sequence

$$F = [f(AA), f(AC), f(AG), f(AT), \dots, f(TT)]$$

A more complex method is the "pseudo nucleic acid representation", a DNA sequence is described using the following  $g_i$

$$\left\{ \begin{array}{l} g_1 = \frac{1}{L-2} \sum_{i=1}^{L-2} \Delta(R_i R_{i+1}, R_{i+1} R_{i+2}) \\ g_2 = \frac{1}{L-3} \sum_{i=1}^{L-3} \Delta(R_i R_{i+1}, R_{i+2} R_{i+3}) \quad (\omega = L_{min} - 2) \\ g_3 = \frac{1}{L-4} \sum_{i=1}^{L-4} \Delta(R_i R_{i+1}, R_{i+3} R_{i+4}) \\ \dots \\ g_\omega = \frac{1}{L-\omega-1} \sum_{i=1}^{L-\omega-1} \Delta(R_i R_{i+1}, R_{i+\omega} R_{i+\omega+1}) \end{array} \right.$$

$$\Delta(R_i R_{i+1}, R_k R_{k+1}) = \frac{1}{J} \sum_{j=1}^J [V_j(R_i R_{i+1}) - V_j(R_k R_{k+1})]^2$$

Dinucleotide	Physical structures					
	$V_1(R_i R_{i+1})$	$V_2(R_i R_{i+1})$	$V_3(R_i R_{i+1})$	$V_4(R_i R_{i+1})$	$V_5(R_i R_{i+1})$	$V_6(R_i R_{i+1})$
AA	0.06	0.50	0.27	1.59	0.11	-0.11
AC	1.50	0.50	0.80	0.13	1.29	1.04
AG	0.78	0.36	0.09	0.68	-0.24	-0.62
AT	1.07	0.22	0.62	-1.02	2.51	1.17
CA	-1.38	-1.36	-0.27	-0.86	-0.62	-1.25
CC	0.06	1.08	0.09	0.56	-0.82	0.24
CG	-1.66	-1.22	-0.44	-0.82	-0.29	-1.39
CT	0.78	0.36	0.09	0.68	-0.24	-0.62
GA	-0.08	0.50	0.27	0.13	-0.39	0.71
GC	-0.08	0.22	1.33	-0.35	0.65	1.59
GG	0.06	1.08	0.09	0.56	-0.82	0.24
GT	1.50	0.50	0.80	0.13	1.29	1.04
TA	-1.23	-2.37	-0.44	-2.24	-1.51	-1.39
TC	-0.08	0.50	0.27	0.13	-0.39	0.71
TG	-1.38	-1.36	-0.27	-0.86	-0.62	-1.25
TT	0.06	0.50	0.27	1.59	0.11	-0.11

$V(a,b)$  is a given physical property of the base pair (see table), 6 properties are used, therefore  $J = 6$



# Other examples: DNA/RNA & proteins

As one of the most ubiquitous post-transcriptional modifications of RNA, N 6-methyladenosine (m6A) plays an essential role in many vital biological processes (<https://www.ncbi.nlm.nih.gov/pubmed/27552763>). The identification of m6A sites in RNAs is significantly important for both basic biomedical research and practical drug development. POST-translational modification (PTLM) of proteins is an important biological mechanism because it is associated with many major diseases. A similar subtle modification, the so-called post-transcriptional modification (PTCM), may also occur in RNA sequences. The PTCMs of RNA are very common and important in living organisms. To date, more than 100 PTCMs have been discovered in native cellular RNAs, including mRNAs and other RNAs. Among these modifications, N 6 -methyladenosine (m6A) is the most ubiquitous internal modification, which plays an important role in regulating gene expression. Since it was first identified in mammalian mRNA during the mid-1970s, m6A has been studied for decades to determine its exact functions and mechanisms. A series of studies have proven that m6A plays an essential role in diverse biological processes. Deng et al. discovered that m6A is important for respiration and stress responses in bacteria; Liu et al. found that m6A can enhance the interaction between RNA and protein by altering the RNA structure; Alarcon et al. showed that m6A can initialize miRNA processing and promote miRNA maturation by marking primary microRNAs (pri-miRNAs). All of these scientific findings indicate that knowledge of m6A is vitally important for both basic biomedical research and practical drug development. Many studies have been dedicated to identifying the m6A sites in RNA sequences. In <https://www.ncbi.nlm.nih.gov/pubmed/27552763> sequence-based m6A site predictor called TargetM6A is developed. TargetM6A first encode each target RNA sequence into a fixed-length feature vector; then, an optimized feature subset by applying the incremental feature selection is obtained. Finally, we train a support vector machine (SVM).

Growing bacterial resistance to antibiotics is spurring research on utilizing naturally-occurring antimicrobial peptides (AMPs) (peptide is a short chains of amino acid) as templates for novel drug design <https://ieeexplore.ieee.org/document/7172462>. While experimentalists mainly focus on systematic point mutations to measure the effect on antibacterial activity, the computational community seeks to understand what determines such activity in a machine learning setting. The latter seeks to identify the biological signals or features that govern activity.

<https://ieeexplore.ieee.org/document/7172462> demonstrates for the first time the capability not only to recognize that a peptide is an AMP or not but also to predict its target selectivity based on models of activity against only Gram-positive, only Gram-negative, or both types of bacteria. The proposed method does so through novel sequence-based features for training a given classifier.



# DNA microarray

A DNA microarray (also commonly known as DNA chip or biochip) is a collection of microscopic DNA spots attached to a solid surface. Scientists use DNA microarrays to measure the expression levels of large numbers of genes simultaneously or to genotype multiple regions of a genome. Each DNA spot contains picomoles (10<sup>-12</sup> moles) of a specific DNA sequence, known as probes (or reporters or oligos).

A typical use is to compare the gene expression profile of a sick individual with that of a healthy one to identify which genes are involved in the disease, or to choose the appropriate drug based on gene expression. A microarray allows the expression of over 20,000 genes to be extracted, therefore each DNA sample is described by over 20,000 elements, most of which are not related to the given classification process. Techniques called features selection are used to select only some gene expressions, e.g. signal to noise methods calculate for each gene j (example valid for bi-class problem).

$$S(j) = \frac{\mu_+(j) - \mu_-(j)}{\sigma_+(j) - \sigma_-(j)}$$

The genes that give positive values are correlated with the class +1, while the genes that give negative values are correlated with the class -1. Then m/ 2 positive and m/2 negatives genes are selected.

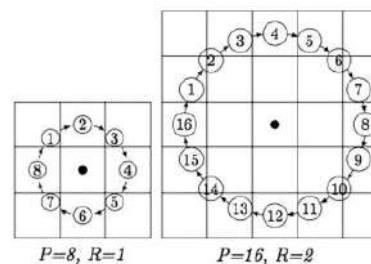
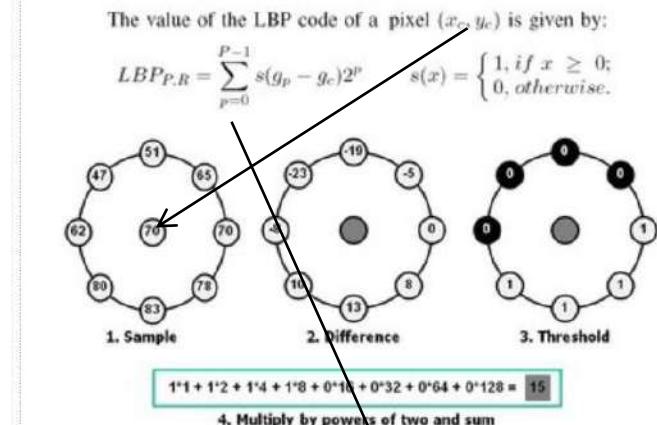


# Local Binary Pattern (LBP)

The basic idea for developing the LBP operator was that two-dimensional surface textures can be described by two complementary measures: local spatial patterns and gray scale contrast. The original LBP operator (Ojala et al. 1996) forms labels for the image pixels by thresholding the  $3 \times 3$  neighborhood of each pixel with the center value and considering the result as a binary number. The histogram of these  $2^8 = 256$  different labels can then be used as a texture descriptor.

The LBP operator was extended to use neighborhoods of different sizes (Ojala et al. 2002). Using a circular neighborhood and bilinearly interpolating values at non-integer pixel coordinates allow any radius and number of pixels in the neighborhood. The gray scale variance of the local neighborhood can be used as the complementary contrast measure. In the following, the notation  $(P,R)$  will be used for pixel neighborhoods which means  $P$  sampling points on a circle of radius of  $R$ . See Fig. for an example of LBP computation.

Another extension to the original operator is the definition of so-called *uniform patterns*, which can be used to reduce the length of the feature vector and implement a simple rotation-invariant descriptor. This extension was inspired by the fact that some binary patterns occur more commonly in texture images than others. A local binary pattern is called uniform if the binary pattern contains at most two bitwise transitions from 0 to 1 or vice versa when the bit pattern is traversed circularly. For example, the patterns 00000000 (0 transitions), 01110000 (2 transitions) and 11001111 (2 transitions) are uniform whereas the patterns 11001001 (4 transitions) and 01010010 (6 transitions) are not. In the computation of the LBP labels, uniform patterns are used so that there is a separate label for each uniform pattern and all the non-uniform patterns are labeled with a single label. For example, when using  $(8,R)$  neighborhood, there are a total of 256 patterns, 58 of which are uniform, which yields in 59 different labels.



$$x_p = x_c + R \cos\left(\frac{2\pi p}{P}\right)$$

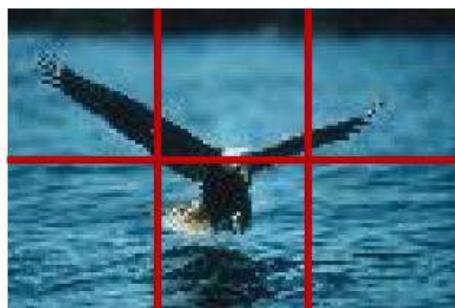
$$y_p = y_c - R \sin\left(\frac{2\pi p}{P}\right)$$

- Ojala, T., Pietikäinen, M. and Harwood, D. (1996), A Comparative Study of Texture Measures with Classification Based on Feature Distributions. *Pattern Recognition* 19(3):51-59.  
 Ojala, T., Pietikäinen, M. and Mäenpää, T. (2002), Multiresolution Gray-scale and Rotation Invariant Texture Classification with Local Binary Patterns. *IEEE Trans. Pattern Analysis and Machine Intelligence* 24(7): 971-987.

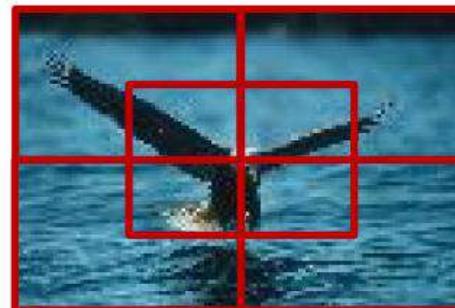
# Images

The descriptors just seen, if calculated on the entire image, do not retain spatial information; to enrich the descriptor with local information it is possible to divide the image into regions calculating the descriptors for each region.

E.g. if an image is divided into 6 blocks then it will be described by 6x features with respect to the descriptor applied only to the entire image.



Six sub-windows, without overlap, are extracted



Overlapped subwindows are extracted



# *Data mining: Alzheimer; Autism*

- The dataset includes data from 259 individuals: 85 ranging from presymptomatic to late-stage Alzheimer Disease (AD), 11 with Other Dementia (OD), 47 having Mild Cognitive Impairment (MCI), 21 with Other Neurological Disease (OND), and 79 Non-Demented Controls (NDC). The features are the expression values of 120 known blood plasma proteins extracted from each individual.
- The dataset includes data collected from 206 children between the ages of three and ten years to assess levels of oxidative stress. The patterns collected are the result of analysis measurements of metabolite concentrations of the folate-dependent one-carbon metabolism and transulfuration pathways taken from blood samples and are composed of 25 features. The patterns belong to two classes: 83 children with Autism Spectrum Disorders (ASD) and 76 age-matched controls with no-ASD.



# The dissimilarity space: Bridging structural and statistical pattern recognition

- The patterns are described by a series of numbers where each number is the similarity between the given original pattern and another, in this way the original patterns can also be complex structures such as graphs, in which case a series of distances between graphs will be the representation vector of patterns. The original definition of dissimilarity space is:

The dissimilarity space is a vector space in which the dimensions are defined by dissimilarity vectors measuring pairwise dissimilarities between examples and individual objects from the so-called representation set  $R$ . Hence, a dissimilarity representation  $D(X, R)$  is addressed as a data-dependent mapping  $D(\cdot, R) : X \rightarrow \mathbb{R}^n$  from an initial set of objects  $X$  to a dissimilarity space, equipped with the traditional inner product and Euclidean metric. The representation set can be chosen as the complete training set  $T$ , a set of carefully selected or constructed prototypes

Assume  $n$  objects,  $O = o_1, o_2, \dots, o_n$  and an  $n \times n$  dissimilarity matrix  $D := D(O, O)$ .

$D_{i,j} = d(o_i, o_j)$  is the dissimilarity between the sensor inputs for objects  $o_i$  and  $o_j$ .

$D(O, o_k) := [d(o_1, o_k), \dots, d(o_n, o_k)]^T$  is a feature defined by pairwise dissimilarities to  $o_k$ .

$x_i := D(o_i, O) = [d(o_i, o_1), \dots, d(o_i, o_n)]^T$  is a dissimilarity-based representation for  $o_i$ .

$X := D$ , defines an  $n$ -dimensional vector space in which the dimension  $k$  is defined by  $D(O, o_k)$  and the vector  $x_i$  represents  $o_i$ .





# The dissimilarity space: Bridging structural and statistical pattern recognition

- In practice, the "dissimilarity space" methods make possible to represent complex patterns in vector format even without the "dissimilarity space" being a metric.

A **metric space** is an ordered pair  $(M, d)$  where  $M$  is a set and  $d$  is a metric on  $M$ , i.e., a function

$$d : M \times M \rightarrow \mathbb{R}$$

such that for any  $x, y, z \in M$ , the following holds:<sup>[2]</sup>

1.  $d(x, y) = 0 \iff x = y$  identity of indiscernibles
2.  $d(x, y) = d(y, x)$  symmetry
3.  $d(x, z) \leq d(x, y) + d(y, z)$  subadditivity or triangle inequality

$$d(x, y) \geq 0$$

A pseudometric space  $(X, d)$  is a set  $X$  together with a non-negative real-valued function  $d: X \times X \rightarrow \mathbb{R}_{\geq 0}$  (called a **pseudometric**) such that for every  $x, y, z \in X$ ,

1.  $d(x, x) = 0$ .
2.  $d(x, y) = d(y, x)$  (symmetry)
3.  $d(x, z) \leq d(x, y) + d(y, z)$  (subadditivity/triangle inequality)

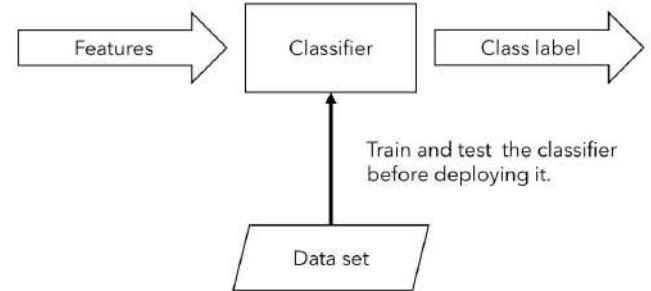
Unlike a metric space, points in a pseudometric space need not be **distinguishable**; that is, one may have  $d(x, y) = 0$  for distinct values  $x \neq y$ .



# *Machine learning recall*



# Classification



Classification assigns a class to a pattern.

It is necessary to learn a function capable of performing the mapping from pattern space to class space.

The term recognition is also often used.

In the case of only 2 classes, the term binary classification is used,

with more than two classes the classification problem is named “multi-class classification”.

Class: set of patterns having common properties.

E.g. the different ways in which character A can be written.

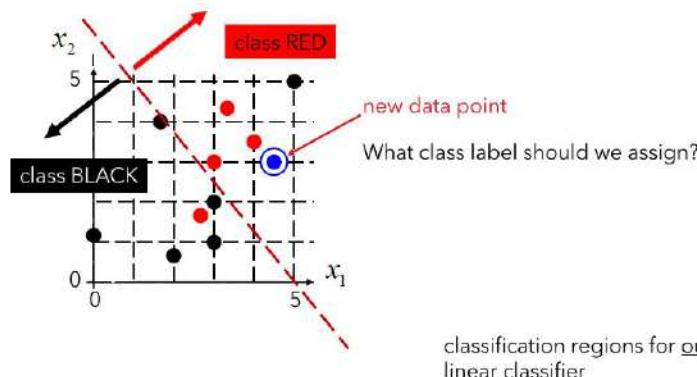
class concept is semantic and strictly dependent from application:

- ) 21 classes for the recognition of letters of the latin alphabet
- ) 2 classes to distinguish the letters of the latin alphabet from the Cyrillic one

Some other examples:

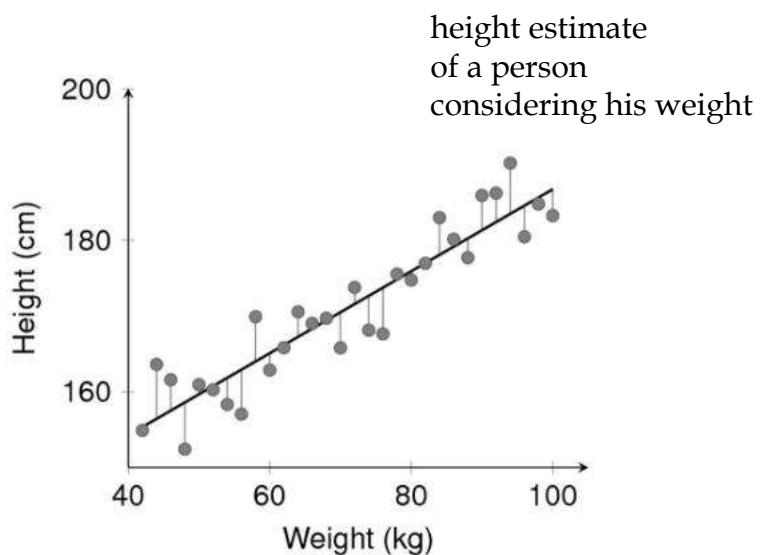
- Spam detection
- Credit Card fraud detection
- Face recognition
- Pedestrian classification
- Medical diagnosis
- Stock Trading

A **classifier** is any function, method or algorithm that assigns a class label to any given object.



# Regression

Regression - Assigns a continuous value to a pattern.  
Useful for predicting continuous values.  
Solving a regression problem corresponds to learn an approximant function of the pairs "input, output".



A regression problem requires the prediction of a quantity.

A regression can have real values or discrete input variables.

A problem with multiple input variables is often called a multivariate regression problem.

A regression problem where input variables are ordered by time is called a time series forecasting problem.

Examples:

Estimated sale prices for apartments in the real estate market.

Risk estimation for insurance companies.

Energy prediction produced by a photovoltaic system.

Health cost prediction models

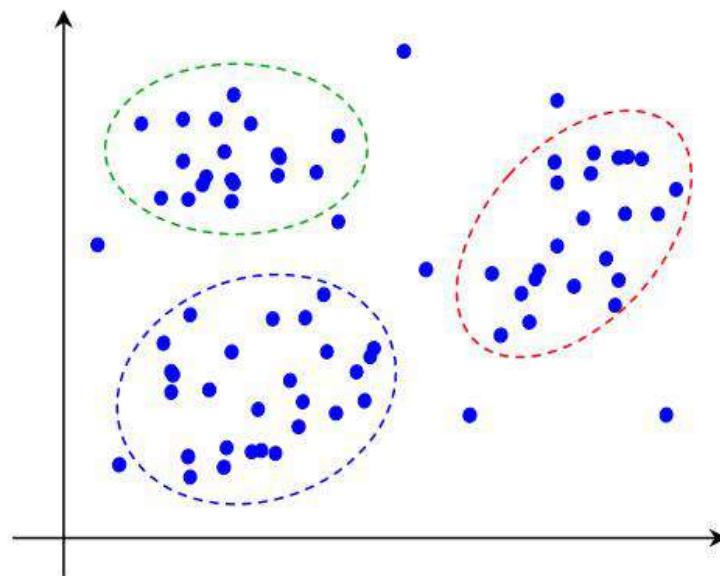


# Clustering

Clustering: identifies groups (clusters) of patterns with similar characteristics.

The problem is that classes are not known and the patterns are not labeled -> the unsupervised nature of the problem makes the classification more complex.

Often not even the number of clusters is known a priori  
The clusters identified in learning can then be used as classes.



# Clustering

Examples of clustering problems:

Marketing: definition of user groups based on consumption

Genetics: grouping of individuals based on DNA analogies

Bioinformatics: partitioning of genes into similar groups

Vision: image segmentation

In computer vision, image segmentation is the process of partitioning a digital image into multiple segments (sets of pixels, also known as super-pixels). The goal of segmentation is to simplify and/or change the representation of an image into something that is more meaningful and easier to analyze.

Image segmentation is typically used to locate objects and boundaries (lines, curves, etc.) in images. More precisely, image segmentation is the process of assigning a label to every pixel in an image such that pixels with the same label share certain characteristics.



Unknown
Bicycle
Pedestrian
Car
Fence
SignSymbol
Tree
Sidewalk
Road
Pole
Building
Sky



# Dimensionality reduction

Dimensionality reduction, or dimension reduction, is the transformation of data from a high-dimensional space into a low-dimensional space; so that the low-dimensional representation retains some meaningful properties of the original data

mapping:  $R^d \rightarrow R^k$

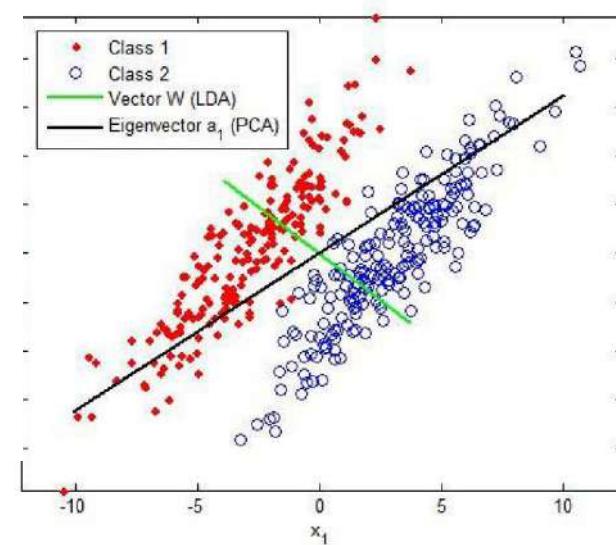
i.e. use  $k$  features to describe a pattern instead of using the original  $d$  features. It's not for sure that all  $d$  features are useful (some can bring noise), removing "noisy" features can increase the accuracy of the classification step.

Remember that a high dimensionality with respect to the number of patterns leads to the so-called **curse of dimensionality**:

The curse of dimensionality refers to various phenomena that arise when analyzing and organizing data in high-dimensional spaces (often with hundreds or thousands of dimensions) that do not occur in low-dimensional settings such as the three-dimensional physical space of everyday experience.

There are multiple phenomena referred to by this name in domains such as numerical analysis, sampling, combinatorics, machine learning, data mining and databases. The common theme of these problems is that when the dimensionality increases, the volume of the space increases so fast that the available data become sparse. This sparsity is problematic for any method that requires statistical significance. In order to obtain a statistically sound and reliable result, the amount of data needed to support the result often grows exponentially with the dimensionality.

Also, organizing and searching data often relies on detecting areas where objects form groups with similar properties; in high dimensional data, however, all objects appear to be sparse and dissimilar in many ways, which prevents common data organization strategies from being efficient.



# Dimensionality reduction

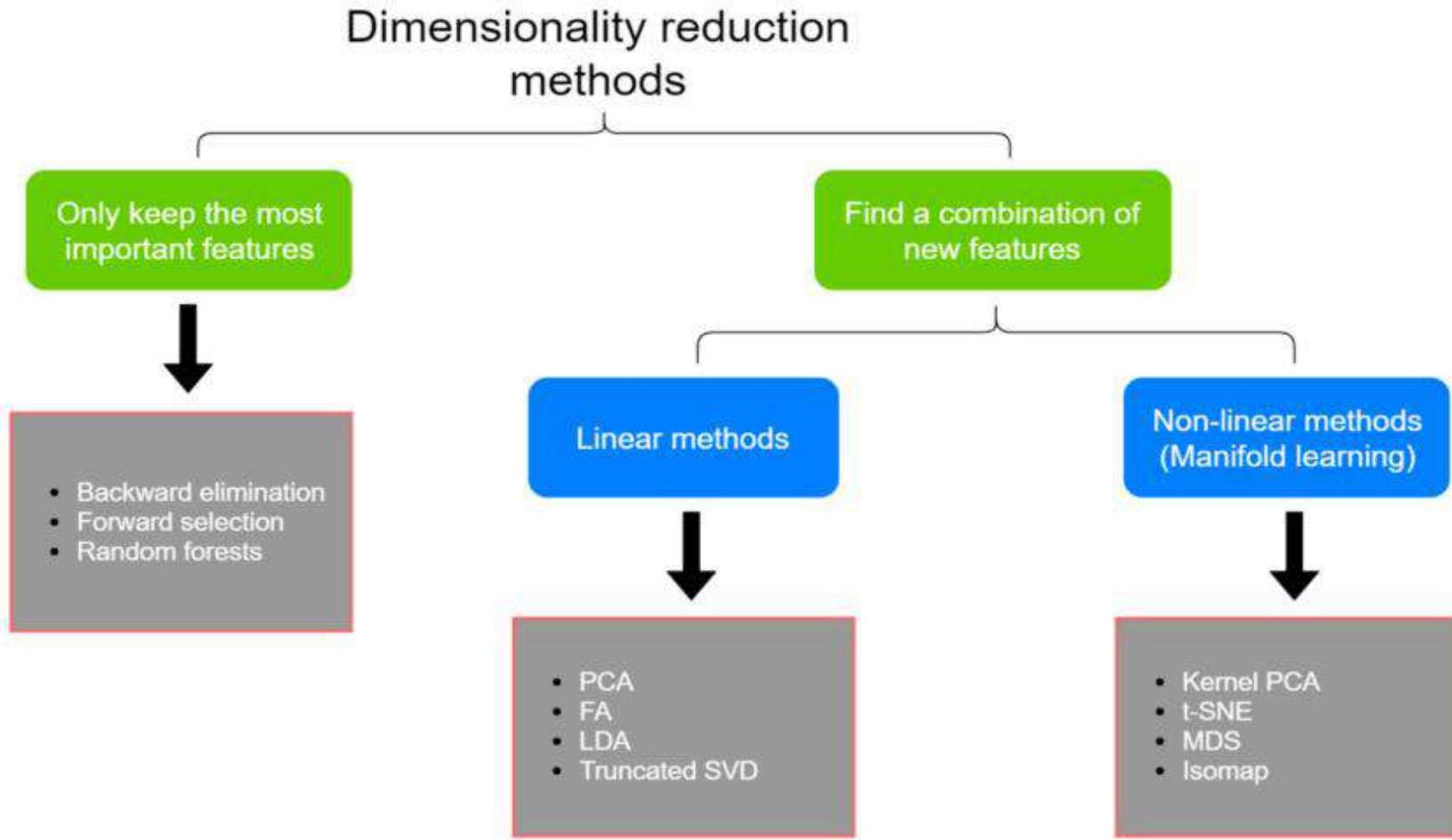


Image copyright: Rukshan Pramoditha



# *Feature learning*

The success of many machine learning applications depends from the effectiveness of the representation of the patterns in terms of features.

The definition of ad-hoc (hand-crafted) features for the different applications is called feature engineering.

For example, for the recognition of objects there are numerous shape, color and texture descriptors that we can use to convert images to numeric vectors.

## **Representation Learning (feature learning)**

We can automatically learn effective features starting from raw data? Or similarly, we can operate directly on raw data (e.g. intensity of the pixels of an image, amplitude of a audio signal over time) without using pre-defined features?

Most of the deep learning techniques (e.g. convolutional neural networks) operate in this way, using as input the raw data and automatically extracting the features necessary to solve the problem of interest.



# *Learning approaches*

Supervised: Pattern classes are known ,the training set is labeled. typical situation in classification, regression and some dimension reduction techniques (eg Linear Discriminant Analysis).

Unsupervised: the classes of the patterns used for training are unknown;

the training set is not labeled. Typical situation in clustering and in most of dimensionality reduction techniques.

Semisupervised

The training set is partially labeled, unlabeled patterns can help to optimize the classification rule.

Not all patterns can have a label, eg. too expensive analyze them by a human expert to assign them a class. For instance. think about classifying all the various audio signals coming from the birds that live in a given forest, we will have a few tens of thousands of audio files labeled and millions not labeled.



# Learning approaches

Supervised learning (more formally):

$$\text{Assumption: } t^i = f(x^{(i)}) + \epsilon^{(i)}$$

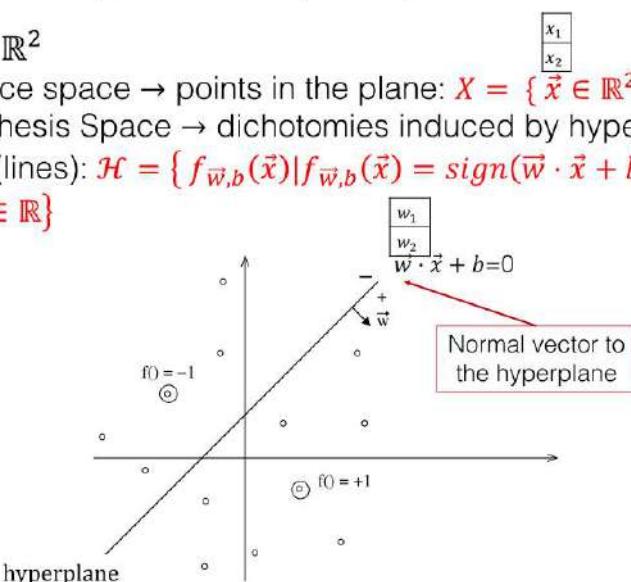
- Given pre-classified examples,  $Tr = \{(x^{(i)}, t^{(i)})\}$ , learn a general description which captures the information content of the examples (rules working for the whole input domain)
- The model will learn a function  $h$  such that  $h(x^{(i)}) \approx f(x^{(i)})$  for all the examples in  $Tr$
- It should be possible to use this description in a predictive way (given a new input  $\hat{x}$ , predict the associated output  $h(\hat{x})$ )

**Note:** It is assumed that an expert (or teacher) provides the supervision (i.e. the values of  $t^{(i)}$  corresponding to the training instances)

## Example of Hypothesis Space (for classification)

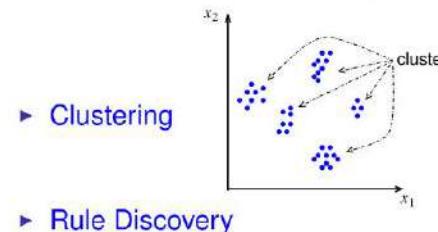
Lines in  $\mathbb{R}^2$

- Instance space  $\rightarrow$  points in the plane:  $X = \{\vec{x} \in \mathbb{R}^2\}$
- Hypothesis Space  $\rightarrow$  dichotomies induced by hyperplanes in  $\mathbb{R}^2$  (lines):  $\mathcal{H} = \{f_{\vec{w}, b}(\vec{x}) | f_{\vec{w}, b}(\vec{x}) = \text{sign}(\vec{w} \cdot \vec{x} + b), \vec{w} \in \mathbb{R}^2, b \in \mathbb{R}\}$



Unsupervised Learning:

- given a set of examples  $Tr = \{x^{(i)}\}$ , discover regularities and/or patterns (true on the whole input domain)
- there is no expert (or teacher) to help us (i.e., no supervision!)



# Learning approaches

- **Multilabel**, in general, a single class is assigned to each pattern, whereas in some applications several classes are associated with each pattern. There are ad hoc algorithms to handle multi-label problems. Examples of such problems: a given newspaper article can belong to multiple topics, where each topic is a class; in the biomedical field «The Anatomical Therapeutic Chemical (ATC) system is a multi-label classification system proposed by the World Health Organization (WHO), which categorizes drugs into classes according to their therapeutic effects and characteristics. (1) alimentary tract and metabolism; (2) blood and blood forming organs; (3) cardiovascular system; (4) dermatologicals; (5) genitourinary system and sex hormones; (6) systemic hormonal preparations, excluding sex hormones and insulins; (7) anti-infectives for systemic use; (8) antineoplastic and immunomodulating agents; (9) musculoskeletal system; (10) nervous system; (11) antiparasitic products, insecticides and repellents; (12) respiratory system; (13) sensory organs; and (14) various. »
- **Active learning**, particular case of semi-supervised learning, at each iteration some unlabeled patterns are chosen and labeled (the most useful patterns for the classification system) the number of patterns to be labeled depends (obviously) on the cost of the class selection process.

**pseudo labeling in SSL**, you use the classifier to assign labels to unlabeled patterns and then use them in training, the pseudo labels are assigned by the method you are using; SSL: A form of unsupervised learning where the data provides the supervision. Self-supervised learning is a means for training computers to do tasks without humans providing labeled data (i.e., a picture of a dog accompanied by the label "dog").

**active learning**, often the choice of patterns falls among the most difficult to classify (therefore in general they would never be selected in the methods based on pseudolabels) and then labeled by a human expert, if they are difficult to classify it is hoped that they will be useful in training after having labeled them.

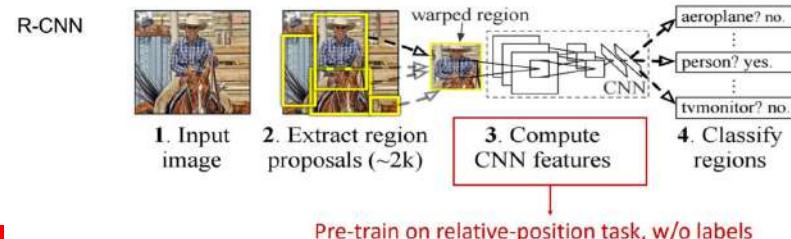
As a form of unsupervised learning, *self-supervised learning* leverages unlabeled data to provide supervision in training, such as by predicting some withheld part of the data using other parts. For text, we can train models to "fill in the blanks" by predicting randomly masked words using their surrounding words (contexts) in big corpora without any labeling effort.

For images, we may train models to tell the relative position between two cropped regions of the same image (Doersch et al., 2015). In these two examples of self-supervised learning, training models to predict possible words and relative positions are both classification tasks (from supervised learning).

(Doersch et al., 2015, <https://arxiv.org/abs/1505.05192>)

## Evaluation: PASCAL VOC Detection

- Pre-train CNN using self-supervision (no labels)
- Train CNN for detection in R-CNN object category detection pipeline



# Multilabel image classification



A donkey, a dog, a cat, and a rooster.

Depending on what we want to do with our model ultimately, treating this as a binary classification problem might not make a lot of sense. Instead, we might want to give the model the option of saying the image depicts a cat, a dog, a donkey, *and* a rooster.

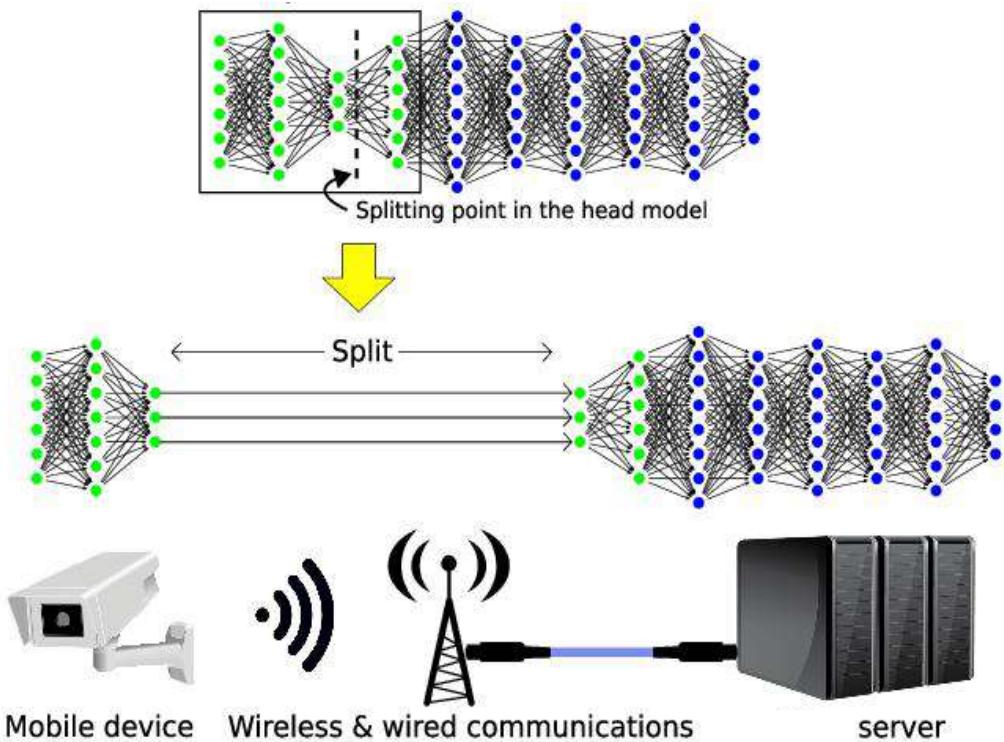


# Split Computing

The Deep Neural Network (DNN) computational requirements preclude their device-only deployment on most of the resource-constraint systems, such as mobile phones. A possible alternative is to refer to simplified models, with an obvious accuracy decline which may not be desirable. The opposite cloud-only paradigm is to transfer the data captured by the device to a server through a communication network. The servers, which run the DNN model, compute the inference and send it back to the device. In this case, the high data transfer time and possible server congestion are obvious downsides.

As a compromise between the device-only and the cloud-only approaches, the split computing (SC) frameworks propose to divide DNN models into a “head” and a “tail”, deployed at the edge device and the server, respectively. Using SC, the computational load is distributed across the two platforms, exploiting the computational power of both devices at the cost of transferring data on a network connection. Data transfer effort is mitigated by the injection of bottlenecks, usually carried out by autoencoders, achieving in-model compression.

Setting up a SC system requires setting up the optimal splitting point: so far, this choice was driven primarily by architectural considerations such as memory capabilities of the edge device, desired transmission rate between head and tail, compression rate of the bottleneck, and nature of the layers



<https://towardsdatascience.com/autoencoders-and-the-denoising-feature-from-theory-to-practice-db7f7ad8fc78>





# *One/few shot learning*

One/few-shot learning is an object categorization problem, found mostly in computer vision. Whereas most machine learning-based object categorization algorithms require training on hundreds or thousands of samples, one-shot learning aims to classify objects from one, or only a few, samples. The ability to learn object categories from few examples, and at a rapid pace, has been demonstrated in humans. It is estimated that a child learns almost all of the 10 ~ 30 thousand object categories in the world by age six. This is due not only to the human mind's computational power, but also to its ability to synthesize and learn new object categories from existing information about different, previously learned categories. Given two examples from two object categories: one, an unknown object composed of familiar shapes, the second, an unknown, amorphous shape; it is much easier for humans to recognize the former than the latter, suggesting that humans make use of previously learned categories when learning new ones. The key motivation for solving one-shot learning is that systems, like humans, can use knowledge about object categories to classify new objects.



# Zero shot learning



Suppose a person knows what a horse looks like. And give that person the info that 'A zebra is a horse with black and white striped skin'. Then there is a high chance that a person can identify a zebra even though that person had no prior idea of what a zebra is. Zero-shot learning is an approach in machine learning that takes inspiration from this. In a zero-shot learning approach we have data in the following manner:

Seen classes: Classes with labels available for training.

Unseen classes: Classes that occur only in the test set or during inference. Not present during training.

Auxiliary information: Information about both seen and unseen class labels during training time.

Based on the data available during inference zero-shot learning can be classified into two types:

Conventional zero-shot learning: If during test time we only expect images from unseen classes.

Generalized zero-shot learning: If during testing phase images from both seen and unseen class can be present. For most practical use cases, we will be using this mode of zero-shot learning.

<https://www.analyticsvidhya.com/blog/2022/02/classification-without-training-data-zero-shot-learning-approach/>



# Search

Sometimes we do not just want to assign each example to a bucket or to a real value. In the field of information retrieval, we want to impose a ranking on a set of items. Take web search for an example. The goal is less to determine whether a particular page is relevant for a query, but rather, which one of the plethora of search results is most relevant for a particular user. We really care about the ordering of the relevant search results and our learning algorithm needs to produce ordered subsets of elements from a larger set. In other words, if we are asked to produce the first 5 letters from the alphabet, there is a difference between returning “A B C D E” and “C A B E D”. Even if the result set is the same, the ordering within the set matters.

One possible solution to this problem is to first assign to every element in the set a corresponding relevance score and then to retrieve the top-rated elements. [PageRank](#), the original secret sauce behind the Google search engine was an early example of such a scoring system but it was peculiar in that it did not depend on the actual query. Here they relied on a simple relevance filter to identify the set of relevant items and then on PageRank to order those results that contained the query term. Nowadays, search engines use machine learning and behavioral models to obtain query-dependent relevance scores. There are entire academic conferences devoted to this subject.



# Recommender systems

Recommender systems are another problem setting that is related to search and ranking. The problems are similar insofar as the goal is to display a set of relevant items to the user. The main difference is the emphasis on *personalization* to specific users in the context of recommender systems. For instance, for movie recommendations, the results page for a science fiction fan and the results page for a connoisseur of Peter Sellers comedies might differ significantly. Similar problems pop up in other recommendation settings, e.g., for retail products, music, and news recommendation.

In some cases, customers provide explicit feedback communicating how much they liked a particular product (e.g., the product ratings and reviews on Amazon, IMDb, and Goodreads). In some other cases, they provide implicit feedback, e.g., by skipping titles on a playlist, which might indicate dissatisfaction but might just indicate that the song was inappropriate in context. In the simplest formulations, these systems are trained to estimate some score, such as an estimated rating or the probability of purchase, given a user and an item.

Given such a model, for any given user, we could retrieve the set of objects with the largest scores, which could then be recommended to the user. Production systems are considerably more advanced and take detailed user activity and item characteristics into account when computing such scores. Fig.  is an example of deep learning books recommended by Amazon based on personalization algorithms tuned to capture one's preferences.



# Sequence learning

So far, we have looked at problems where we have some fixed number of inputs and produce a fixed number of outputs. For example, we considered predicting house prices from a fixed set of features: square footage, number of bedrooms, number of bathrooms, walking time to downtown. We also discussed mapping from an image (of fixed dimension) to the predicted probabilities that it belongs to each of a fixed number of classes, or taking a user ID and a product ID, and predicting a star rating. In these cases, once we feed our fixed-length input into the model to generate an output, the model immediately forgets what it just saw.

This might be fine if our inputs truly all have the same dimensions and if successive inputs truly have nothing to do with each other. But how would we deal with video snippets? In this case, each snippet might consist of a different number of frames. And our guess of what is going on in each frame might be much stronger if we take into account the previous or succeeding frames. Same goes for language. One popular deep learning problem is machine translation: the task of ingesting sentences in some source language and predicting their translation in another language.

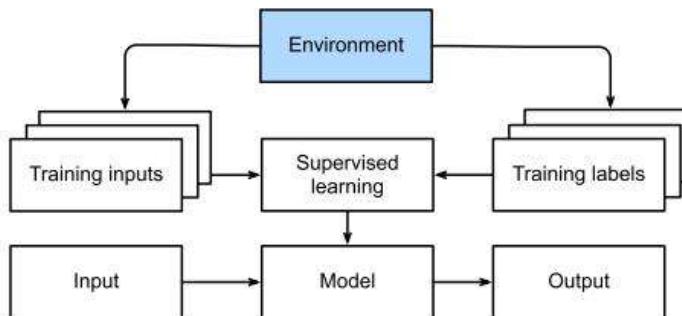
These problems also occur in medicine. We might want a model to monitor patients in the intensive care unit and to fire off alerts if their risk of death in the next 24 hours exceeds some threshold. We definitely would not want this model to throw away everything it knows about the patient history each hour and just make its predictions based on the most recent measurements.

These problems are among the most exciting applications of machine learning and they are instances of *sequence learning*. They require a model to either ingest sequences of inputs or to emit sequences of outputs (or both). Specifically, *sequence to sequence learning* considers problems where input and output are both variable-length sequences, such as machine translation and transcribing text from the spoken speech.



# Offline learning

So far, we have not discussed where data actually come from, or what actually happens when a machine learning model generates an output. That is because supervised learning and unsupervised learning do not address these issues in a very sophisticated way. In either case, we grab a big pile of data upfront, then set our pattern recognition machines in motion without ever interacting with the environment again. Because all of the learning takes place after the algorithm is disconnected from the environment, this is sometimes called *offline learning*. For supervised learning, the process by considering data collection from an environment looks like



Collecting data for supervised learning from an environment.

This simplicity of offline learning has its charms. The upside is that we can worry about pattern recognition in isolation, without any distraction from these other problems. But the downside is that the problem formulation is quite limiting. If you are more ambitious, or if you grew up reading Asimov's Robot series, then you might imagine artificially intelligent bots capable not only of making predictions, but also of taking actions in the world. We want to think about intelligent *agents*, not just predictive models. This means that we need to think about choosing *actions*, not just making predictions. Moreover, unlike predictions, actions actually impact the environment. If we want to train an intelligent agent, we must account for the way its actions might impact the future observations of the agent.



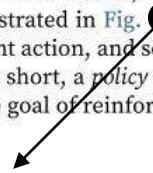
# Reinforcement learning

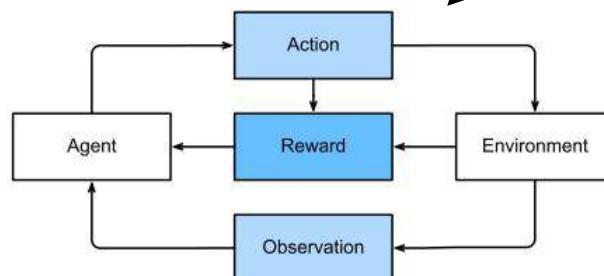
Considering the interaction with an environment opens a whole set of new modeling questions.  
The following are just a few examples.

- Does the environment remember what we did previously?
- Does the environment want to help us, e.g., a user reading text into a speech recognizer?
- Does the environment want to beat us, i.e., an adversarial setting like spam filtering (against spammers) or playing a game (vs. an opponent)?
- Does the environment not care?
- Does the environment have shifting dynamics? For example, does future data always resemble the past or do the patterns change over time, either naturally or in response to our automated tools?

This last question raises the problem of *distribution shift*, when training and test data are different.

If you are interested in using machine learning to develop an agent that interacts with an environment and takes actions, then you are probably going to wind up focusing on *reinforcement learning*. This might include applications to robotics, to dialogue systems, and even to developing artificial intelligence (AI) for video games. *Deep reinforcement learning*, which applies deep learning to reinforcement learning problems, has surged in popularity. The breakthrough deep Q-network that beat humans at Atari games using only the visual input, and the AlphaGo program that dethroned the world champion at the board game Go are two prominent examples.

Reinforcement learning gives a very general statement of a problem, in which an agent interacts with an environment over a series of time steps. At each time step, the agent receives some *observation* from the environment and must choose an *action* that is subsequently transmitted back to the environment via some mechanism (sometimes called an actuator). Finally, the agent receives a reward from the environment. This process is illustrated in Fig.  The agent then receives a subsequent observation, and chooses a subsequent action, and so on. The behavior of a reinforcement learning agent is governed by a policy. In short, a policy is just a function that maps from observations of the environment to actions. The goal of reinforcement learning is to produce a good policy.

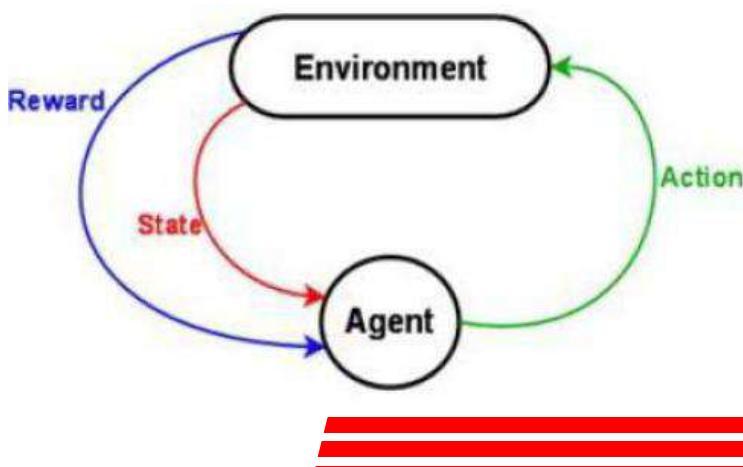


The interaction between reinforcement learning and an environment.

# Reinforcement Learning

- ▶ agent which may
  - ▶ be in state  $s$ , and
  - ▶ execute an action  $a$  (chosen among the ones admissible in the current state)
- ▶ and operates in an environment  $e$ , which in response to action  $a$  in the state  $s$  returns
  - ▶ the next state, and
  - ▶ a reward  $r$ , which can be positive (+), negative (-), or neutral (0).

The goal of the agent is to maximize a function of the rewards  
(e.g. expected discounted sum of rewards:  $\sum_{t=0}^{\infty} \gamma^t r_{t+1}$  where  $0 \leq \gamma < 1$ )



In practice it is very difficult to obtain examples that are in the same time correct and representative of all situations in which the agent must work, therefore the classic supervised approach is not easily applicable and reinforcement learning is applied.

For instance, there are numerous applications in Robotics (e.g. object grasping, control, assembly, navigation).

Q learning is one of the best known and most used approaches. Its extension (deep) called Deep Reinforcement Learning (DRL) is the basis of the successes achieved by Google DeepMind (Atari, AlphaGo).

# *Continual Learning*

Continual Learning (CL) is built on the idea of learning continuously and adaptively about the external world and enabling the autonomous incremental development of ever more complex skills and knowledge.

In the context of Machine Learning it means being able to smoothly update the prediction model to take into account different tasks and data distributions but still being able to re-use and retain useful knowledge and skills during time.

Hence, CL is the only paradigm which force us to deal with an higher and realistic time-scale where data (and tasks) becomes available only during time, we have no access to previous perception data and it's imperative to build on top of previously learned knowledge.

Continual learning poses particular challenges for artificial neural networks due to the tendency for knowledge of the previously learned task(s) (e.g., task A) to be abruptly lost as information relevant to the current task (e.g., task B) is incorporated. This phenomenon, termed catastrophic forgetting, occurs specifically when the network is trained sequentially on multiple tasks because the weights in the network that are important for task A are changed to meet the objectives of task B.

for details see

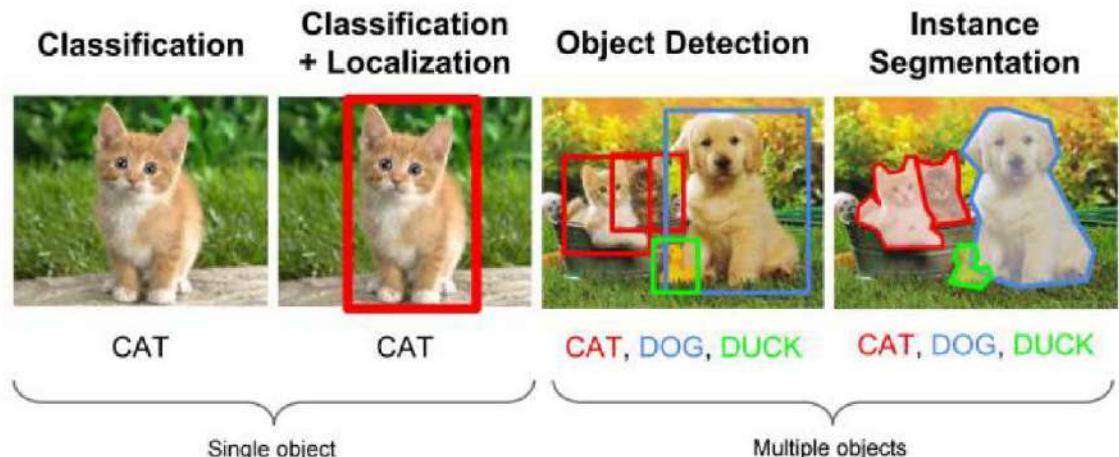
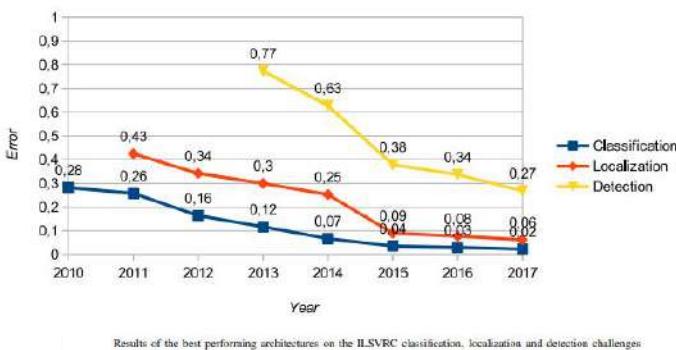
<https://medium.com/continual-ai/why-continuous-learning-is-the-key-towards-machine-intelligence-1851cb57c308>

<https://arxiv.org/pdf/1907.00182.pdf>

# Computer vision applications

29/02

slide 22-71 da leggere da soli:  
argomenti introduttivi



We assume that there is only one object or in any case only one dominant object

there may be multiple objects

Classification: determines the class of the object.

Localization: determines the class of the object and its position in the image (bounding box).

Detection: for each object present it determines the class and the position in the image (bounding box).

Segmentation: instead of a bounding box (rectangular) labels the individual pixels of the image to classes with indexes of the classes. Applications in the field of remote sensing (eg labeling crops in satellite images), medical images (highlight pathologies), automatic driving (highlight road region)



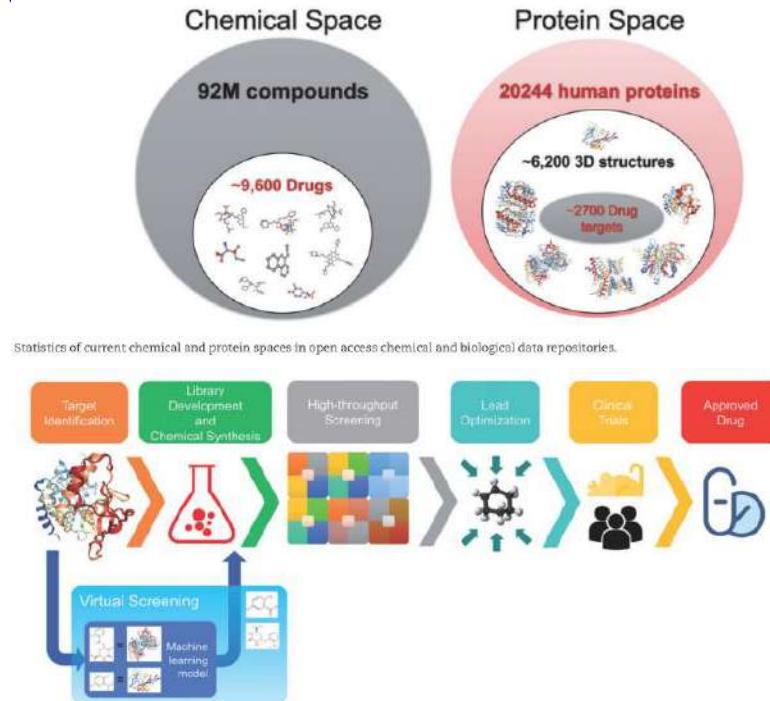
## Bioinformatic example: Recent applications of deep learning and machine intelligence on in silico drug discovery: methods, tools and databases»»

- A drug is an approved [by Food and Drug Administration (FDA), for example] bioactive compound that acts on protein targets to cure/decelerate a specific disease or to promote the health of a living being.
- A target protein (or just a target) is a naturally occurring biomolecule of an organism that is bound by a ligand and has its function modulated, which results in a physiological change in the body of the organism.
- A ligand is a molecular structure that physically binds another molecular structure and modulates its function.
- A compound is a chemical structure that is formed by the combination of two or more atoms that are connected by chemical bonds.

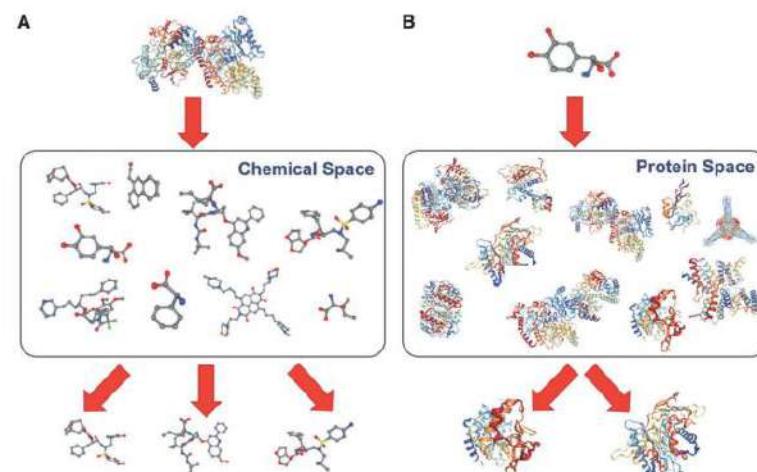
The goal of a drug is to change behavior of a protein, both the drug and the protein are described with a different feature vector and classify that pair as "Interact" vs "do not interact": ideal to find a drug which interacts only on that protein and not on others (for minimize side effects), unfortunately it rarely happens. Furthermore, more complex diseases are linked to more proteins poorly managed by the human body.

**find drugs that interact with the protein**

**possibility:** siamese network -> given drug and protein, label: yes/no interaction



A broad overview of drug development and the place of virtual screening in this process.



(A) In conventional virtual screening multiple compounds are screened against a pre-specified target, and candidate interacting compounds (i.e. ligands) are identified, whereas (B) in target prediction (i.e. reverse virtual screening), a compound is searched against multiple proteins and candidate targets are identified.

# Different kinds of learning

**Batch:** the training is done only once on a training set.

Once the training is finished, the system switches to “working Modality” and it is unable to learn further.

**Incremental:** following initial training, there are further training sessions.

Scenario: Batch Sequences, Unsupervised Tuning.

Risk: catastrophic forgetting (the system forgets what learned earlier).

Natural: continuous training (throughout life).

Active training in work mode.

Coexistence of supervised and unsupervised approach.

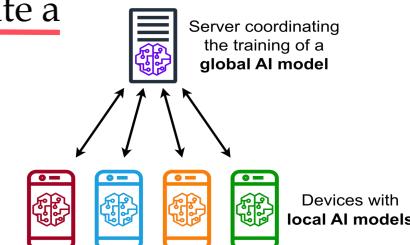
*human-like learning involves an initial small amount of direct instruction (e.g. parental labeling of objects during childhood) combined with large amounts of subsequent unsupervised experience (e.g. self-interaction with objects)*

*jadi non personal confidence  
data (e.g. spesialis)*

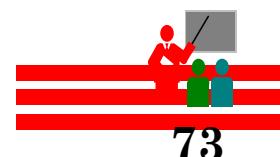
**Federated learning** (widely used when there are privacy issues) aims at training a machine learning algorithm, for instance deep neural networks, on multiple local datasets contained in local nodes without explicitly exchanging data samples. The general principle consists in training local models on local data samples and exchanging parameters (e.g. the weights and biases of a deep neural network) between these local nodes at some frequency to generate a global model shared by all nodes.

While distributed learning also aims at training a single model on multiple servers, a common underlying assumption is that the local datasets are independent and identically distributed (i.i.d.) and roughly have the same size.

None of these hypotheses are made for federated learning; instead, the datasets are typically heterogeneous and their sizes may span several orders of magnitude.



Federated learning enables multiple actors to build a common, robust machine learning model without sharing data, thus addressing critical issues such as data privacy, data security, data access rights



# Parameters optimization

Typically, the behavior of a Machine Learning algorithm is regulated by a set of parameters (e.g. the weights of the connections in a neural network). Learning consists in determining the optimal value of these parameters.

Given a training set Train and a set of parameters, the function objective can indicate the optimality of the solution (to be maximized).

$$\Theta^* = \operatorname{argmax}_{\Theta} f(\text{Train}, \Theta)$$

or the error or loss (loss-function) to be minimized.

$$\Theta^* = \operatorname{argmin}_{\Theta} f(\text{Train}, \Theta)$$

*allenare a divisione training /  
test: simulare mondo reale  
solo per valutazione finale valori ottimi*

Never use the test set, except for final validation of the performance.  
Never use it to make choice of parameters/hyperparameters.



# Parameters optimization

## ■ $f(Train, \Theta)$

can be optimized:

explicitly, with methods that operate from his own mathematical definition.

E.g.: we calculate the partial derivatives of  $f$  with respect to the parameters (gradient), the gradient is equaled to 0 (system of equations) and solved with respect to **60** parameters.

implicitly, using heuristics that modify parameters consistently with  $f$

e.g. use [https://www.okpedia.it/algoritmo\\_genetico](https://www.okpedia.it/algoritmo_genetico) to change the parameters to maximize system accuracy

Heuristic (or heuristic) algorithm: in mathematics and computer science it is a particular type of algorithm designed to solve a problem faster, if the classical methods are too slow, or to find an approximate solution, if the classical methods fail to find an exact solution. The result is obtained by trying to balance optimization, completeness, accuracy and speed of execution.



# Hyperparameter *determinano rete*

Many algorithms require to define, before learning step, the value of the so-called hyperparameters.

Examples of hyperparameters:

The number of neurons in a neural network;

The number of neighbors k in a k-NN classifier;

The degree of a polynomial used in a regression approach;

The type of loss function.

Usually, you could proceed with a two-level approach in which for each "reasonable" value of the hyperparameters is performed a learning phase, and at the end of the procedure the hyperparameters that performed better are chosen.

Grid Search: for each hyperparameter we define a set of values to try. The system is evaluated on all combinations of values of all hyperparameters.

It can be very expensive: incremental refinement of values. This approach means that you chose each hyperparameter independently from the others.



# Performance indicators

One possibility is to use directly the function goal to quantify performance. But generally it is preferred a more direct measure related to the semantic of the problem.

E.g. in the classification problem, the classification accuracy [0 ... 100%] and the percentage of correctly classified patterns.

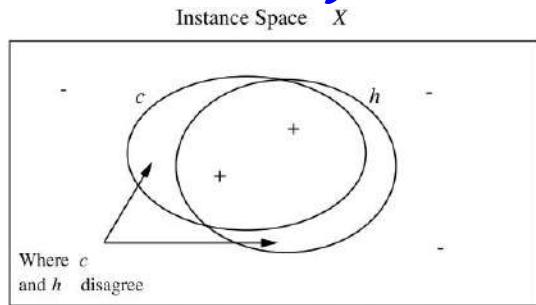
The classification error is the complement.

- Classification error =  $\frac{\text{errors}}{\text{total}}$
- Accuracy =  $(1 - \text{error}) = \frac{\text{correct}}{\text{total}}$

In the simplest case (and most common in machine learning) it is assumed that the pattern to be classified belongs to one of the known classes (closed set). E.g. classify people in {men; women}

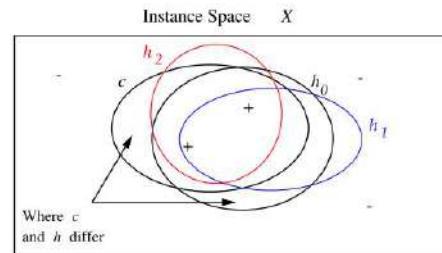


# Performance indicators



**Def:** The **True Error** ( $\text{error}_{\mathcal{D}}(h)$ ) of hypothesis  $h$  with respect to target concept  $c$  and distribution  $\mathcal{D}$  (to observe an input instance  $x \in X$ ) is the probability that  $h$  will misclassify an instance drawn at random according to  $\mathcal{D}$ :

$$\text{error}_{\mathcal{D}}(h) = \Pr_{x \in \mathcal{D}} [c(x) \neq h(x)]$$

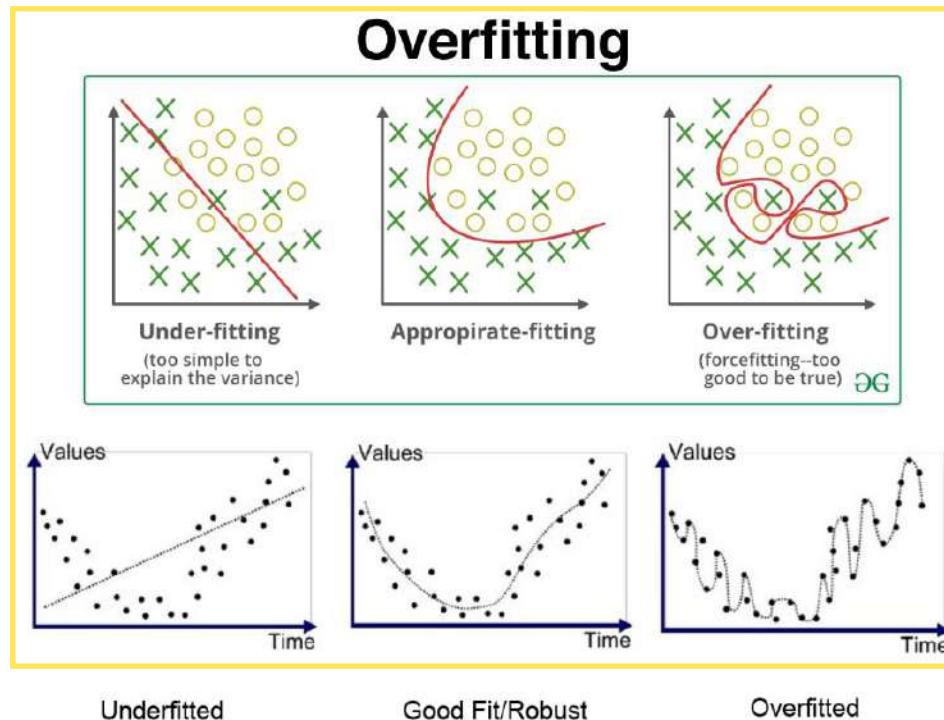


**Def:** The **Empirical Error** ( $\text{error}_{Tr}(h)$ ) of hypothesis  $h$  with respect to  $Tr$  is the number of examples that  $h$  misclassifies:

$$\text{error}_{Tr}(h) = \Pr_{(x, f(x)) \in Tr} [f(x) \neq h(x)] = \frac{|\{(x, f(x)) \in Tr \mid f(x) \neq h(x)\}|}{|Tr|}$$

**Def:**  $h \in \mathcal{H}$  overfits  $Tr$  if  $\exists h' \in \mathcal{H}$  such that  $\text{error}_{Tr}(h) < \text{error}_{Tr}(h')$ , but  $\text{error}_{\mathcal{D}}(h) > \text{error}_{\mathcal{D}}(h')$ .

generalizance bene

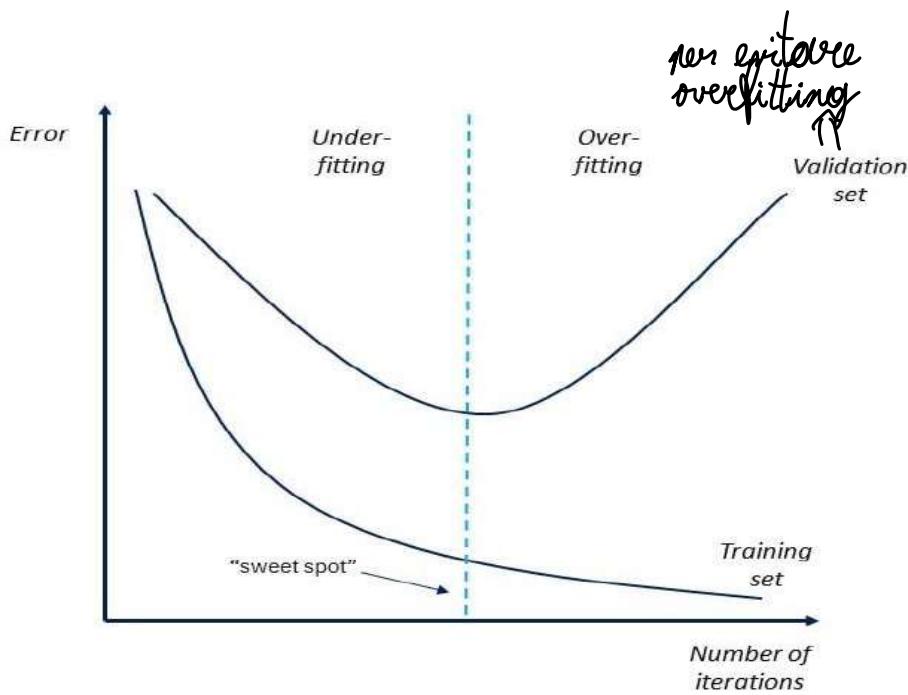


# Overfitting

Let's remember that our goal is to maximize accuracy on test set. Assuming that validation is representative of test set, we aim to maximize accuracy on validation.

By generalization we mean the ability to transfer the high accuracy achieved on training to validation.

If the degrees of freedom of the classifier are excessive, high accuracy is achieved on training, but not on validation (poor generalization). In this case we speak of train overfitting. This situation occurs very easily when training set is small in size.



In iterative training processes, typically after a certain number of iterations, the accuracy on validation no longer increases (and may begin to decrease) due to overfitting.

By monitoring the progress, training can be stopped at the ideal point.

The degrees of freedom of the classifier must not be excessive,

but adequate in relation to the complexity of the problem.

It is a good idea to start with a few degrees of freedom (controllable through hyperparameters) and gradually increase them by monitoring accuracy on Train and Valid.

- All things being equal, the simplest explanation is to be preferred - William of Occam (Occam Razor).
- Everything Should Be Made as Simple as Possible, But Not Simpler - Albert Einstein

# Performance indicators

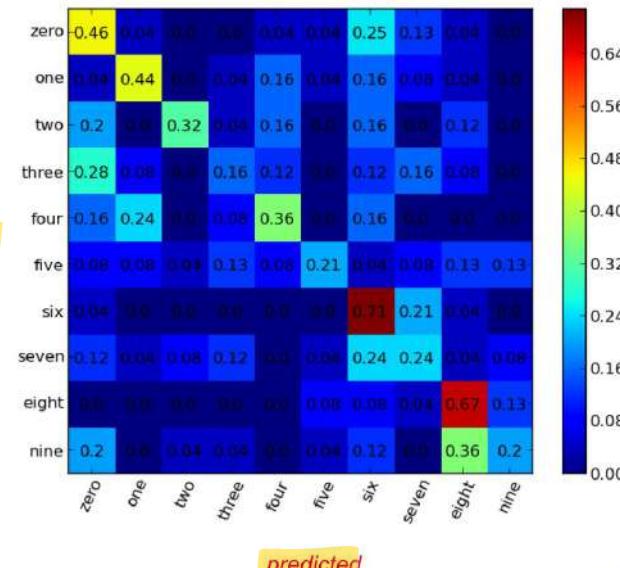
In regression problems, the RMSE (Root Mean Squared Error) is the root of the mean of the squares of the deviations between the true value and the predicted value.

$$RMSE = \sqrt{\frac{1}{N} \sum_{i=1..N} (pred_i - true_i)^2}$$

The confusion matrix is very useful in classification problems (multiclass) to understand how errors are distributed.

*spero facchiamo a copiare lavoro → poniamo noi essere esperti del problema*

A cell  $(r, c)$  reports the percentage of cases in which the system has predicted of class  $c$  a pattern of true class  $r$ . Ideally, the matrix should be diagonal. High values (off-diagonal) indicate concentrations of errors.



# Open Set

difficile definire tutte le possibili classi

In many real cases, however, the patterns to be classified may belong to one of the known classes or none of these (open set). Eg Classify fruit into {apples, pears, bananas}.

Two solutions:

Another fictitious class "the rest of the world" is added to the classes and the so-called "negative examples" are added to the training set.

You must allow the system not to assign a class to a pattern. To this end, a threshold is defined and the pattern is assigned to the most probable class only when the probability is higher than the threshold.

Let us consider a binary classification problem, where the two classes correspond to positive (real class) and negative (rest of the world fictitious class) examples.

E.g. Face detection. Positive examples: all portions of the image (windows) in which faces appear. Negative examples: windows (chosen at random) in which no faces appear.

Similarly we can consider only the positive class and a system (with threshold) capable of calculating the probability  $p$  of a pattern belonging to the class. Let  $t$  be the threshold value, then the pattern is classified as positive if  $p > t$ , as negative if not.



consideriamo classe positiva e negativa

# Open Set - Performance indicators

		Actual	
		Positive	Negative
Predicted	Positive	True Positive	False Positive
	Negative	False Negative	True Negative

A true positive is an outcome where the model correctly predicts the positive class. Similarly, a true negative is an outcome where the model correctly predicts the negative class.

A false positive is an outcome where the model incorrectly predicts the positive class. And a false negative is an outcome where the model incorrectly predicts the negative class.

Let  $N_p$  be the number of patterns belonging to the positive class; let  $N_n$  be the number of patterns belonging to the negative class

$$\text{False positive rate} = \frac{FP}{N_n}$$

$$\text{False negative rate} = \frac{FN}{N_p}$$



# Performance Indicators

		True condition			
Total population	Condition positive	Condition negative	Prevalence $= \frac{\sum \text{Condition positive}}{\sum \text{Total population}}$	Accuracy (ACC) = $\frac{\sum \text{True positive} + \sum \text{True negative}}{\sum \text{Total population}}$	
Predicted condition	Predicted condition positive	True positive	False positive, Type I error	Positive predictive value (PPV), Precision = $\frac{\sum \text{True positive}}{\sum \text{Predicted condition positive}}$	False discovery rate (FDR) = $\frac{\sum \text{False positive}}{\sum \text{Predicted condition positive}}$
	Predicted condition negative	False negative, Type II error	True negative	False omission rate (FOR) = $\frac{\sum \text{False negative}}{\sum \text{Predicted condition negative}}$	Negative predictive value (NPV) = $\frac{\sum \text{True negative}}{\sum \text{Predicted condition negative}}$
Type I error (false positive)	True positive rate (TPR), Recall, Sensitivity, probability of detection, Power = $\frac{\sum \text{True positive}}{\sum \text{Condition positive}}$	False positive rate (FPR), Fall-out, probability of false alarm = $\frac{\sum \text{False positive}}{\sum \text{Condition negative}}$	Positive likelihood ratio (LR+) = $\frac{\text{TPR}}{\text{FPR}}$	Diagnostic odds ratio (DOR) = $\frac{\text{LR+}}{\text{LR-}}$	$F_1 \text{ score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$
Type II error (false negative)	False negative rate (FNR), Miss rate = $\frac{\sum \text{False negative}}{\sum \text{Condition positive}}$	Specificity (SPC), Selectivity, True negative rate (TNR) = $\frac{\sum \text{True negative}}{\sum \text{Condition negative}}$	Negative likelihood ratio (LR-) = $\frac{\text{FNR}}{\text{TNR}}$		

Suppose that we have a data set (obtained from images with faces) and a classifier  $D_{\text{proper}}$ .

		Assigned	
		Face (positive)	No face (negative)
True	Face (positive)	96	9
	No face (negative)	3,505	99,790

Total number of objects,  $N = 96 + 9 + 3,505 + 99,790 = 103,400$

$$\text{Sensitivity, } \text{Sens} = \frac{96}{105} = 0.9143$$

$$\text{Specificity, } \text{Spec} = \frac{99,790}{103,295} = 0.9659$$

## Accuracy

$$Acc = \frac{96 + 99,790}{103,400} = 0.9660$$

Total number of objects,  $N = 96 + 9 + 3,505 + 99,790 = 103,400$

$$\text{Recall} = \frac{96}{105} = 0.9143$$

$$\text{Precision} = \frac{96}{3,601} = 0.0267$$

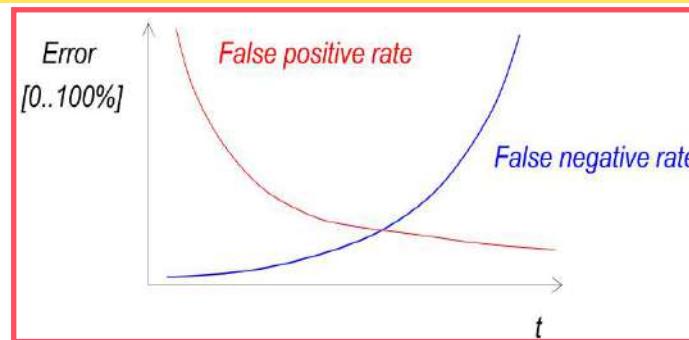
## F1 measure

$$F_1 = 2 \times \frac{0.9143 \times 0.0267}{0.9143 + 0.0267} = 0.0519$$

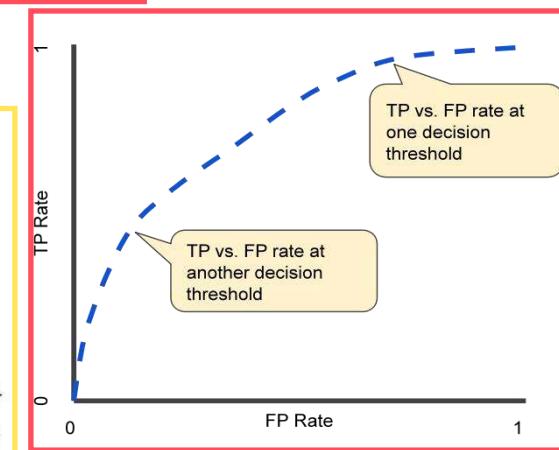
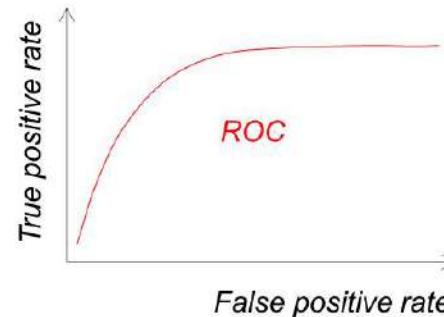
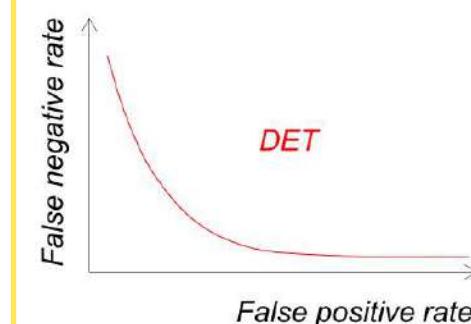


# Classification threshold

False positive rate and false negative rate are a function of the threshold  $t$ . High restrictive thresholds reduce false positives but increase the false negatives; vice versa, tolerant thresholds (low) reduce false negatives but increase the false positives.



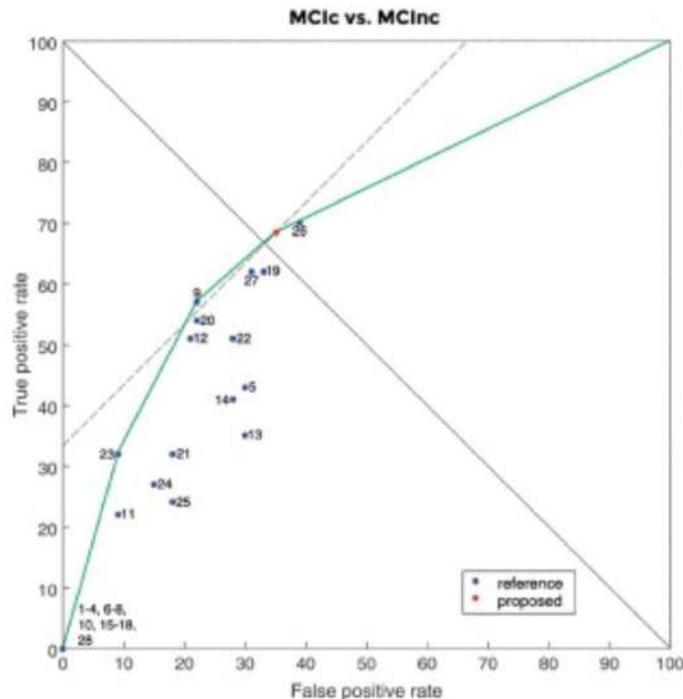
DET (Detection Error Tradeoff)  
ROC (Receiver Operating Characteristic)



The area under the ROC curve (AUC, acronym for the terms "Area Under the Curve") is a measure of accuracy (single value that can be easily used for classification accuracy). The greater the area under the curve (i.e. the curve approaches the top of the graph) is, the greater the discriminating power of the approach. AUC=1 means False positive rate=0 and True positive rate=1 for all the values of the classification threshold.  
Nice example: <https://towardsdatascience.com/understanding-the-roc-curve-in-three-visual-steps-795b1399481c>

Multiclass generalization of the AUC: <https://towardsdatascience.com/multiclass-classification-evaluation-with-roc-curves-and-roc-auc-294fd4617e3a>

# Classification threshold



Prediction of Mild Cognitive Impairment (MCI). conversion (MCIc) or not-conversion (MCInc) to Alzheimer Disease (within 18 months of follow up).

How do you read the ROC curve (“green” plot)?

Assuming a False Positive Rate of 10% (10% of people who do not get sick will be wrongly predicted the disease) we would have a True positive Rate of 35%. The results seem low (it is a method based only on magnetic resonance) but already useful for extracting sets of people on which to test for new drugs, i.e. the TPR allows us to select a good number of people who will get sick.

A typical example of threshold systems are biometric identity verification systems.

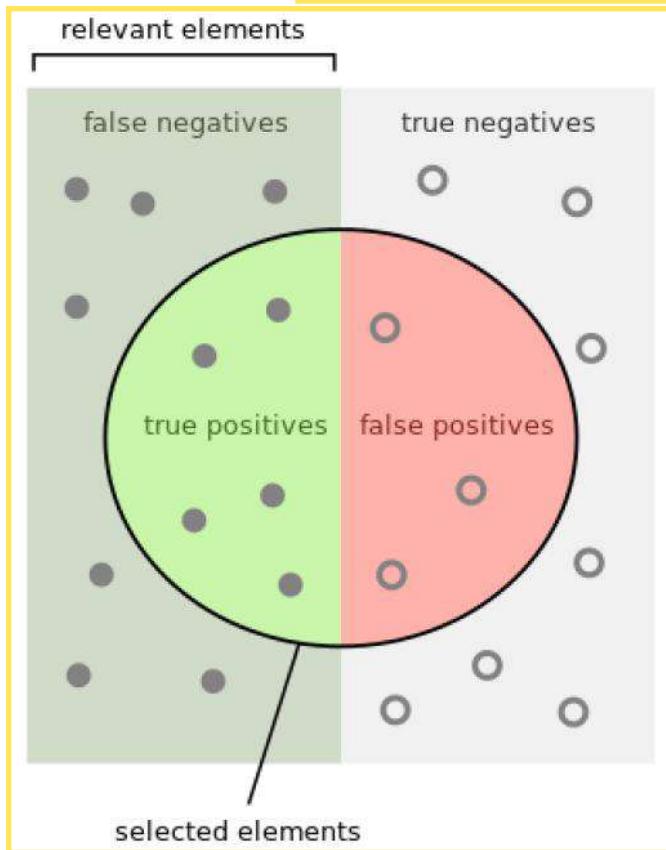
Is this image of the iris of the subject Q?

False positive and False negative in this case are called False Match and False Non-Match.

Only if score is greater than a threshold



# Precision - Recall



$$\text{Precision} = \frac{tp}{tp + fp}$$

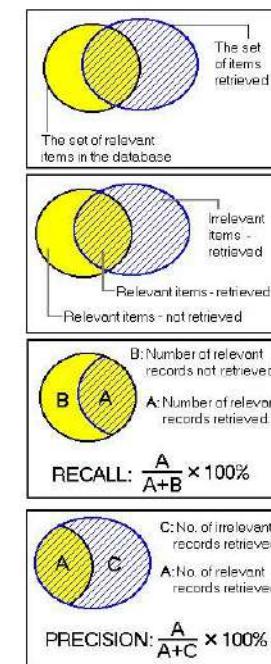
$$\text{Recall} = \frac{tp}{tp + fn}$$

Problem: given a set of articles extract only those of a given topic, e.g. Extract all the articles related to the Messina earthquake of 1908

Beware of classification problems with very unbalanced classes: For example, in a binary classification problem, if one of the two classes is very rare, a "dummy" classifier that never predicts it could reach a classification accuracy close to 100%. Better to use Precision / Recall in this case

- **Recall:** The percentage of the total relevant documents in a database retrieved by your search.
- If you knew that there were 1000 relevant documents in a database and your search retrieved 100 of these relevant documents, your recall would be 10%.

- **Precision:** The percentage of relevant documents in relation to the number of documents retrieved.
- If your search retrieves 100 documents and 20 of these are relevant, your precision is 20%.



# McNemar's Test Statistic

- McNemar's Test (sometimes also called "within-subjects chi-squared test") is a statistical test for paired nominal data. In context of machine learning (or statistical) models, we can use McNemar's test to compare the predictive accuracy of two models.

In McNemar's Test, we formulate the null hypothesis that the probabilities  $p(b)$  and  $p(c)$  are the same, or in simplified terms: None of the two models performs better than the other. Thus, the alternative hypothesis is that the performances of the two models are not equal.

		model 2 correct	model 2 wrong
model 1 correct	model 1 correct	A	B
	wrong	C	D

*errore i bravi, non false positive/negative*

[https://en.wikipedia.org/wiki/Chi-squared\\_distribution](https://en.wikipedia.org/wiki/Chi-squared_distribution)

$$\chi^2 = \frac{(|b - c| - 1)^2}{(b + c)}$$

degrees of freedom = 1

An exact binomial test is recommended for small sample sizes ( $b+c < 25$ ), since the chi-squared value may not be well-approximated by the chi-squared distribution. The exact p-value can be computed as follows:

$$p = 2 \sum_{i=b}^n \binom{n}{i} 0.5^i (1 - 0.5)^{n-i},$$

where  $n = b + c$ , and the factor 2 is used to compute the two-sided p-value.

chi-squared: 2.025  
p-value: 0.154728923485

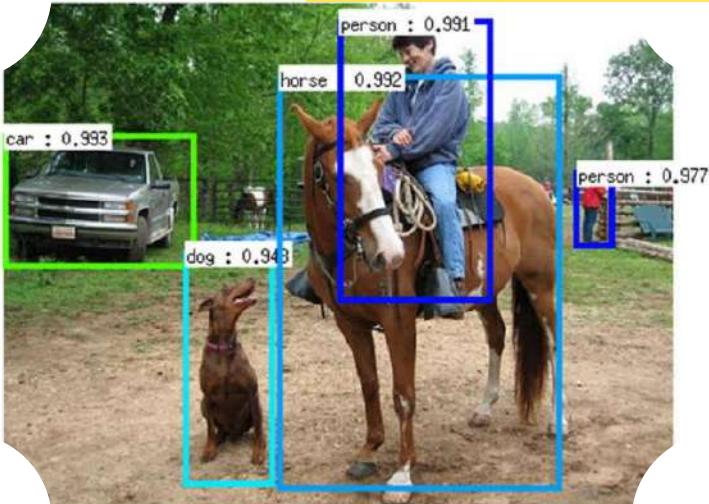
		model 1 correct	model 1 wrong
model 2 correct	model 2 correct	9945	25
	wrong	15	15

<https://www.statology.org/how-to-find-the-p-value-from-the-chi-square-distribution-table/>  
<https://www.statology.org/chi-square-p-value-calculator/>

Since the p-value is larger than our assumed significance threshold ( $\alpha = 0.05$ ), we cannot reject our null hypothesis and assume that there is no significant difference between the two predictive models.



# Object detection: Average Precision



Object detection is a computer technology related to computer vision and image processing that deals with detecting instances of semantic objects of a certain class (such as humans, buildings, or cars) in digital images and videos. Well-researched domains of object detection include face detection and pedestrian detection.

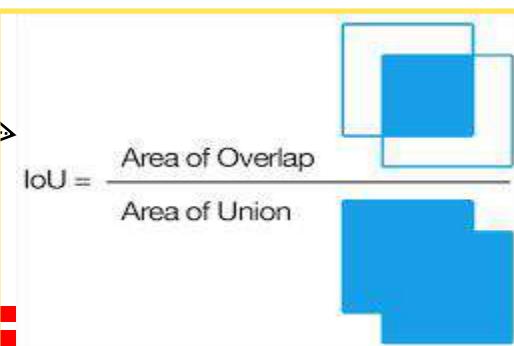
There are several possible types of errors, including: object not found, wrong class, wrong or inaccurate bounding box. How to quantify everything with a scalar to make systems comparable?

Average Precision (AP) is a sort of AUC calculated on a Recall / Precision graph for objects of a single class on the whole database, medium Average Precision (mAP) is the average of AP over all classes.

To calculate TP, FP (with a certain confidence) a correct prediction is considered when the class is right and the Intersection over Union (IoU) of the two bounding boxes (detected and true) is greater than a given value (e.g. 0.5).

For more details:

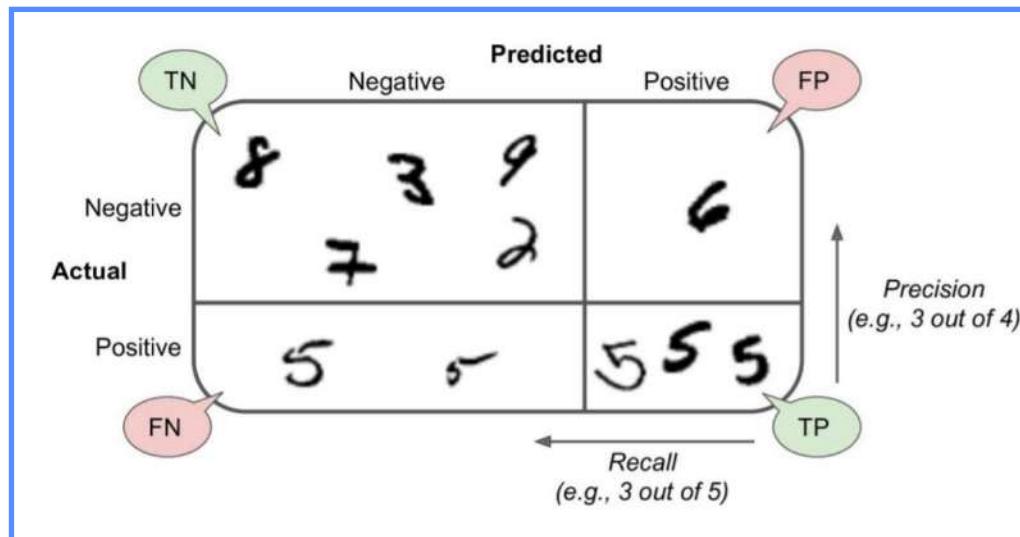
<https://jonathan-hui.medium.com/map-mean-average-precision-for-object-detection-45c121a31173>



# Confusion matrix

Another graphical representation useful for understanding Precision and Recall is the 2x2 confusion matrix which highlights TN, TP, FP, FN:

The classifier (binary) of the example below has digit 5 as a positive class and all other digits as a negative class.



# Multilabel performance indicators

Note that for all the adopted multi-label evaluation metrics, their values are in the interval  $[0, 1]$ . Given the train set  $\mathcal{D} = \{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^n$  and the test set  $\mathcal{D}_t = \{\mathbf{x}_i^t, \mathbf{y}_i^t\}_{i=1}^{n^t}$  where  $\mathbf{x}_i, \mathbf{x}_i^t \in \mathbb{R}^d$  are the feature vector with  $d$  dimensions (features) and  $\mathbf{y}_i, \mathbf{y}_i^t \in \{-1, +1\}^q$  are the corresponding ground-truth label vector with the size of label space being  $q$ .

$\hat{\mathbf{y}}_i^t$  the predicted label vector of  $\mathbf{x}_i^t$ .

- **One error:** It evaluates the fraction that the label with the top-ranked predicted by the instance does not belong to its ground-truth relevant label set. The smaller the value of *one error*, the better performance of the classifier.

The scores associated with each label of the  $i$ -th test pattern is denoted by  $\hat{f}_i^t$

$$\text{one-error} = \frac{1}{n^t} \sum_{i=1}^{n^t} \mathbb{I}[\mathbf{y}_{ij^*}^t = -1],$$

indicator function

where  $j^* = \arg \max_j \hat{f}_i^t$ , and  $\mathbb{I}[z]$  returns 1 if  $z$  holds and 0 otherwise.

$j^*$  is the class with highest score obtained by the classifier, the value '-1' since each pattern is assumed to contain at least one label ==1, so the label with the highest score is checked (to check whether that label is actually 1). You can consider the one-error as the fraction of patterns whose most confident label is irrelevant (i.e.-1). It is assumed that each pattern has at least one label==1

- **Hamming loss:** It evaluates the fraction of instance label pairs which have been misclassified. The smaller the value of *hamming loss*, the better performance of the classifier.

$$\text{hloss} = \frac{1}{n^t} \sum_{i=1}^{n^t} \frac{1}{q} \sum_{j=1}^q \mathbb{I}[y_{ij}^t \neq \hat{y}_{ij}^t],$$



# **Training, Validation, Test** 6/3

The Training Set (Train) is the set of patterns used to train the system, finding the optimal value for the parameters.

The Validation Set (Valid) is the set the patterns on which to set the hyperparameters H (external cycle).

The Test Set (Test) is the set of patterns on which to evaluate the final system performance.

In machine learning benchmarks the split of the patterns into Train, Valid and Test is often predefined, to make the results comparable.

NEVER USE THE TEST SET IF NOT FOR FINAL VALIDATION, NEVER USE IT TO MAKE ANY CHOICE RELATING TO THE CLASSIFICATION MODEL

Training Dataset: The sample of data used to fit the model.

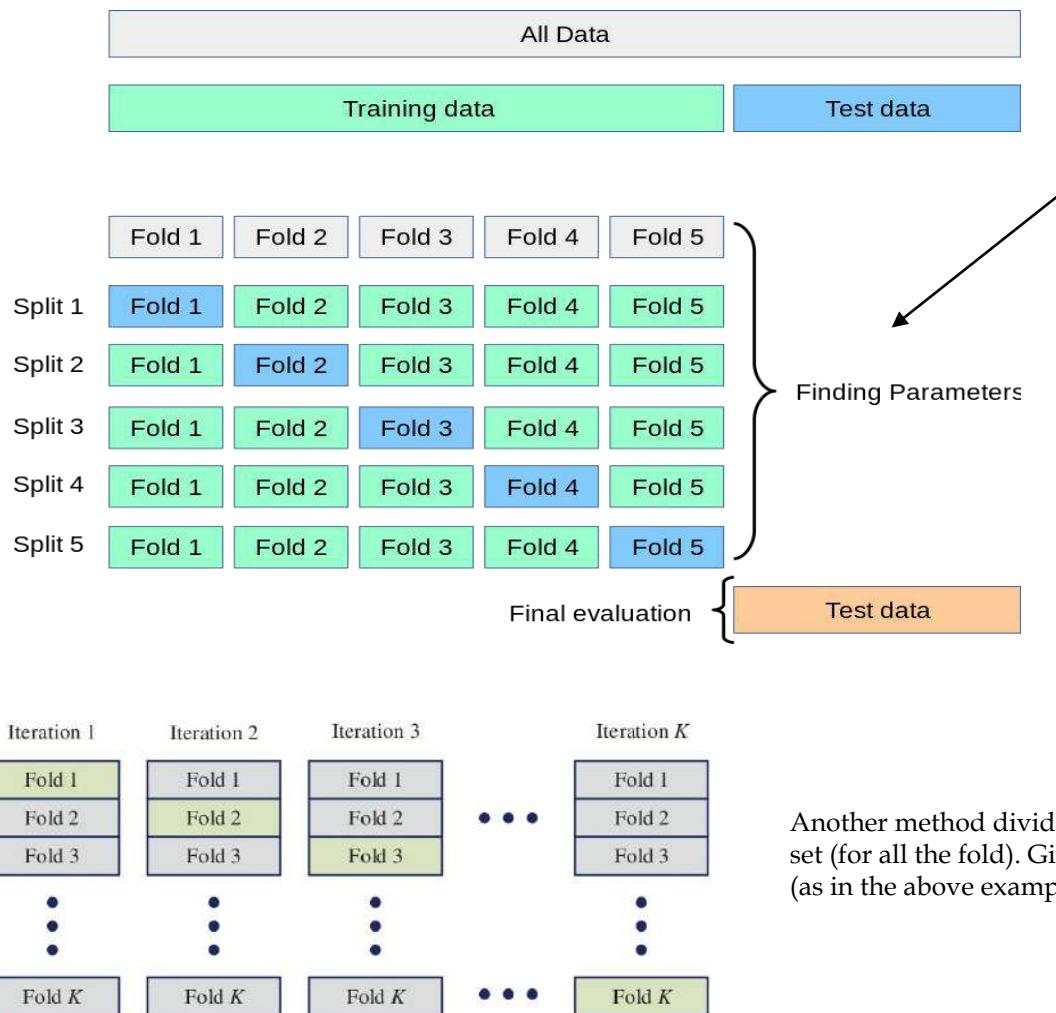
Validation Dataset: The sample of data used to provide an unbiased evaluation of a model fit on the training dataset while tuning model hyperparameters. The evaluation becomes more biased as skill on the validation dataset is incorporated into the model configuration.

Test Dataset: The sample of data used to provide an unbiased evaluation of a final model fit on the training dataset.

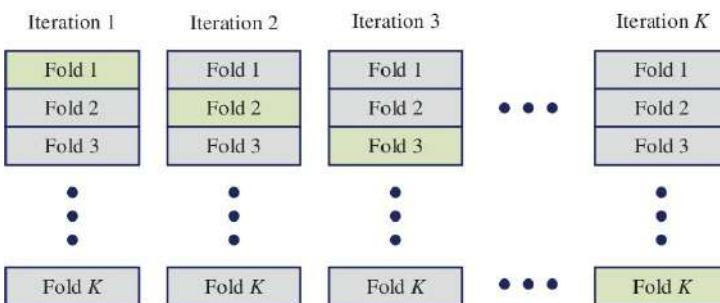
for further details <https://tarangshah.com/blog/2017-12-03/train-validation-and-test-sets/>



# Training, Validation, Test



Hyperparameter selections based on average performance in  $k$  validation sets. Then the "real" performance is that of the test set.



Another method divide dataset into  $k$ -fold, then  $k-1$  in training and 1 in the test set (for all the fold). Given a training set you could also extract a validation set (as in the above example).



# Convergence

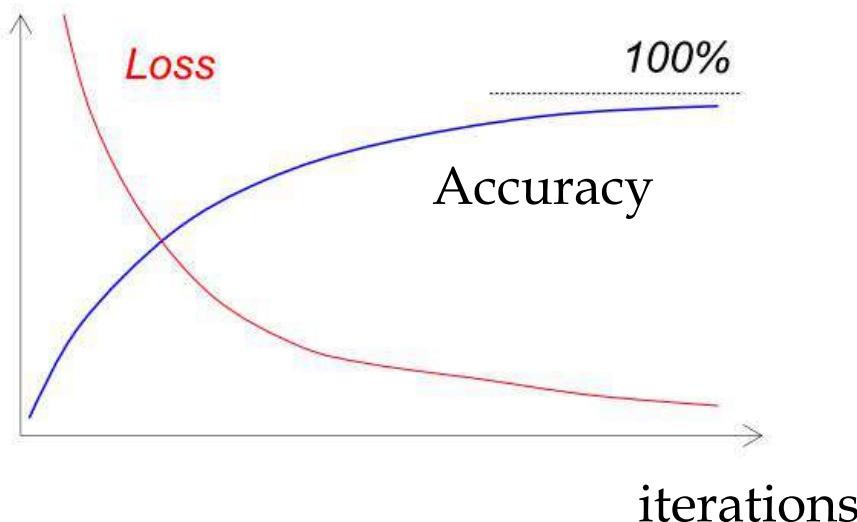
The first aim to be pursued during the training is the convergence on the training set. Let's consider a classifier whose training involves an iterative process. Convergence occurs when:  
loss (output of the loss function) has a decreasing trend; *in funzione di num. iterazioni*  
Accuracy has an increasing trend.

If loss does not decrease (or fluctuates significantly) the system does not converge: the optimization method is not effective, the hyperparameters are out of range, the learning rate is inadequate, there are implementation errors, etc.

If the loss decreases but the accuracy does not increase, probably a wrong loss-function has been chosen.  
If the accuracy does not approach 100% on the training set, the degrees of freedom of the classifier are not sufficient to manage the complexity of the problem;

Other problems could be:

features not suited for  
the given problem;  
low number of training patterns;  
the problem is very complex.



# Comparisons of Two Classifiers - Wilcoxon signed rank test

The Wilcoxon signed-ranks test is a non-parametric alternative to the paired t-test, which ranks the differences in performances of two classifiers for each data set, ignoring the signs, and compares the ranks for the positive and the negative differences. When multiple algorithms are compared, pairwise comparisons are sometimes organized in a matrix.

Let  $d_i$  again be the difference between the performance scores of the two classifiers on  $i$ -th out of  $N$  data sets. The differences are ranked according to their absolute values; average ranks are assigned in case of ties. Let  $R^+$  be the sum of ranks for the data sets on which the second algorithm outperformed the first, and  $R^-$  the sum of ranks for the opposite. Ranks of  $d_i = 0$  are split evenly among the sums; if there is an odd number of them, one is ignored:

$$R^+ = \sum_{d_i > 0} \text{rank}(d_i) + \frac{1}{2} \sum_{d_i = 0} \text{rank}(d_i) \quad R^- = \sum_{d_i < 0} \text{rank}(d_i) + \frac{1}{2} \sum_{d_i = 0} \text{rank}(d_i).$$

Let  $T$  be the smaller of the sums,  $T = \min(R^+, R^-)$ . Most books on general statistics include a table of exact critical values for  $T$  for  $N$  up to 25 (or sometimes more). For a larger number of data sets, the statistics

$$z = \frac{T - \frac{1}{4}N(N+1)}{\sqrt{\frac{1}{24}N(N+1)(2N+1)}}$$

is distributed approximately normally. With  $\alpha = 0.05$ , the null-hypothesis can be rejected if  $z$  is smaller than  $-1.96$ .

recall on z-score: <https://www.ztable.net/>

in the z-score table, row -1.9 and column 0.06 we have 0.025, it is one-side, so we double and have the value of alpha=0.05

There are two data sets on which the classifiers performed equally (lung-cancer and mushroom); if there was an odd number of them, we would ignore one. The ranks are assigned from the lowest to the highest absolute difference, and the equal differences ( $0.000, \pm 0.005$ ) are assigned average ranks.

The sum of ranks for the positive differences is  $R^+ = 3.5 + 9 + 12 + 5 + 6 + 14 + 11 + 13 + 8 + 10 + 1.5 = 93$  and the sum of ranks for the negative differences equals  $R^- = 7 + 3.5 + 1.5 = 12$ . According to the table of exact critical values for the Wilcoxon's test, for a confidence level of  $\alpha = 0.05$  and  $N = 14$  data sets, the difference between the classifiers is significant if the smaller of the sums is equal or less than 21. We therefore reject the null-hypothesis.

perfomance of two classifiers

datasets:	C4.5	C4.5+m	difference	rank
adult (sample)	0.763	0.768	+0.005	3.5
breast cancer	0.599	0.591	-0.008	7
breast cancer wisconsin	0.954	0.971	+0.017	9
cmc	0.628	0.661	+0.033	12
ionosphere	0.882	0.888	+0.006	5
iris	0.936	0.931	-0.005	3.5
liver disorders	0.661	0.668	+0.007	6
lung cancer	0.583	0.583	0.000	1.5
lymphography	0.775	0.838	+0.063	14
mushroom	1.000	1.000	0.000	1.5
primary tumor	0.940	0.962	+0.022	11
rheum	0.619	0.666	+0.047	13
voting	0.972	0.981	+0.009	8
wine	0.957	0.978	+0.021	10

Two-Sided Test $\alpha$	0.1	0.05	0.02	0.01
One-Sided Test $\alpha$	0.05	0.025	,01	0.005
$n$				
5	1			
6	2	1		
7	4	2	0	
8	6	4	2	0
9	8	6	3	2
10	11	8	5	3
11	14	11	7	5
12	17	14	10	7
13	21	17	13	10
14	26	21	16	13
15	30	25	20	16
16	36	30	24	19
17	41	35	28	23
18	47	40	33	28
19	54	46	38	32
20	60	52	43	37
21	68	59	49	43
22	75	66	56	49
23	83	73	62	55
24	92	81	69	61
25	101	90	77	68

We consider a two-sided Test:  
<https://stats.oarc.ucla.edu/other/mult-pkg/faq/general/faq-what-are-the-differences-between-one-tailed-and-two-tailed-tests/>

# Comparisons of Multiple Classifiers

A common example of such questionable procedure would be comparing seven algorithms by conducting all 21 paired t-tests and reporting results like “algorithm A was found significantly better than B and C, and algorithms A and E were significantly better than D, while there were no significant differences between other pairs”. When so many tests are made, a certain proportion of the null hypotheses is rejected due to random chance, so listing them makes little sense.

Let  $r_i^j$  be the rank of the  $j$ -th of  $k$  algorithms on the  $i$ -th of  $N$  data sets. The Friedman test compares the average ranks of algorithms,  $R_j = \frac{1}{N} \sum_i r_i^j$ . Under the null-hypothesis, which states that all the algorithms are equivalent and so their ranks  $R_j$  should be equal, the Friedman statistic

$$\chi_F^2 = \frac{12N}{k(k+1)} \left[ \sum_j R_j^2 - \frac{k(k+1)^2}{4} \right]$$

is distributed according to  $\chi_F^2$  with  $k-1$  degrees of freedom, when  $N$  and  $k$  are big enough (as a rule of a thumb,  $N > 10$  and  $k > 5$ ). For a smaller number of algorithms and data sets, exact critical values have been computed

Iman and Davenport (1980) showed that Friedman's  $\chi_F^2$  is undesirably conservative and derived a better statistic

$$F_F = \frac{(N-1)\chi_F^2}{N(k-1) - \chi_F^2}$$

which is distributed according to the F-distribution with  $k-1$  and  $(k-1)(N-1)$  degrees of freedom.

The table of critical values can be found in any statistical book.

<https://statisticsbyjim.com/hypothesis-testing/f-table/>

$$\chi_F^2 = \frac{12 \cdot 14}{4 \cdot 5} \left[ (3.143^2 + 2.000^2 + 2.893^2 + 1.964^2) - \frac{4 \cdot 5^2}{4} \right] = 9.28$$

$$F_F = \frac{13 \cdot 9.28}{14 \cdot 3 - 9.28} = 3.69.$$

With four algorithms and 14 data sets,  $F_F$  is distributed according to the F distribution with  $4-1=3$  and  $(4-1) \times (14-1) = 39$  degrees of freedom. The critical value of  $F(3, 39)$  for  $\alpha = 0.05$  is 2.85, so we reject the null-hypothesis.

perfomance (rank) of four classifiers, rank is calculated in each dataset

datasets:	C4.5	C4.5+m	C4.5+cf	C4.5+m+cf
adult (sample)	0.763 (4)	0.768 (3)	0.771 (2)	0.798 (1)
breast cancer	0.599 (1)	0.591 (2)	0.590 (3)	0.569 (4)
breast cancer wisconsin	0.954 (4)	0.971 (1)	0.968 (2)	0.967 (3)
cmc	0.628 (4)	0.661 (1)	0.654 (3)	0.657 (2)
ionosphere	0.882 (4)	0.888 (2)	0.886 (3)	0.898 (1)
iris	0.936 (1)	0.931 (2.5)	0.916 (4)	0.931 (2.5)
liver disorders	0.661 (3)	0.668 (2)	0.609 (4)	0.685 (1)
lung cancer	0.583 (2.5)	0.583 (2.5)	0.563 (4)	0.625 (1)
lymphography	0.775 (4)	0.838 (3)	0.866 (2)	0.875 (1)
mushroom	1.000 (2.5)	1.000 (2.5)	1.000 (2.5)	1.000 (2.5)
primary tumor	0.940 (4)	0.962 (2.5)	0.965 (1)	0.962 (2.5)
rheum	0.619 (3)	0.666 (2)	0.614 (4)	0.669 (1)
voting	0.972 (4)	0.981 (1)	0.975 (2)	0.975 (3)
wine	0.957 (3)	0.978 (1)	0.946 (4)	0.970 (2)
average rank	3.143	2.000	2.893	1.964

DF1=1	2	3	
1	161.45	199.50	215.71
2	18.51	19.00	19.16
3	10.13	9.55	9.28
4	7.71	6.94	6.59
5	6.61	5.79	5.41
6	5.99	5.14	4.76
7	5.59	4.74	4.35
8	5.32	4.46	4.07
9	5.12	4.26	3.86
10	4.96	4.10	3.71
11	4.84	3.98	3.59
12	4.75	3.89	3.49
13	4.67	3.81	3.41
14	4.60	3.74	3.34
15	4.54	3.68	3.29
16	4.49	3.63	3.24
17	4.45	3.59	3.20
18	4.41	3.55	3.16
19	4.38	3.52	3.13
20	4.35	3.49	3.10
21	4.32	3.47	3.07
22	4.30	3.44	3.05
23	4.28	3.42	3.03
24	4.26	3.40	3.01
25	4.24	3.39	2.99
26	4.23	3.37	2.98
27	4.21	3.35	2.96
28	4.20	3.34	2.95
29	4.18	3.33	2.93
30	4.17	3.32	2.92
31	4.16	3.31	2.91
32	4.15	3.30	2.90
33	4.14	3.29	2.89
34	4.13	3.28	2.88
35	4.12	3.27	2.87
36	4.11	3.26	2.86
37	4.10	3.25	2.85
38	4.09	3.24	2.84
39	4.08	3.23	2.84
40	4.07	3.22	2.84

F-table of  
Critical Values  
for Significance  
Level = 0.05

# Post-hoc test: Nemenyi test / Bonferroni-Dunn / Holm

If the null-hypothesis is rejected, we can proceed with a post-hoc Nemenyi test. The performance of two classifiers is significantly different if the corresponding average ranks differ by at least the critical difference CD:

$$CD = q_\alpha \sqrt{\frac{k(k+1)}{6N}} \quad k \text{ algorithms} \quad N \text{ data sets}$$

where critical values  $q_\alpha$  are based on the Studentized range statistic divided by  $\sqrt{2}$

When all classifiers are compared with a control classifier, we can instead of the Nemenyi test use one of the general procedures for controlling the family-wise error in multiple hypothesis testing, such as the Bonferroni correction or similar procedures. Although these methods are generally conservative and can have little power, they are in this specific case more powerful than the Nemenyi test, since the latter adjusts the critical value for making  $k(k-1)/2$  comparisons while when comparing with a control we only make  $k-1$  comparisons.

The test statistics for comparing the  $i$ -th and  $j$ -th classifier using these methods is

$$z = (R_i - R_j) / \sqrt{\frac{k(k+1)}{6N}}.$$

The  $z$  value is used to find the corresponding probability from the table of normal distribution, which is then compared with an appropriate  $\alpha$ . The tests differ in the way they adjust the value of  $\alpha$  to compensate for multiple comparisons.

The Bonferroni-Dunn test (Dunn, 1961) controls the family-wise error rate by dividing  $\alpha$  by the number of comparisons made ( $k-1$ , in our case). The alternative way to compute the same test is to calculate the CD using the same equation as for the Nemenyi test, but using the critical values for  $\alpha/(k-1)$  (for convenience, they are given in Table (b)). The comparison between the tables for Nemenyi's and Dunn's test shows that the power of the post-hoc test is much greater when all classifiers are compared only to a control classifier and not between themselves. We thus should not make pairwise comparisons when we in fact only test whether a newly proposed method is better than the existing ones.

The values in table (b) are lower, so the control with CD is more rigorous

#classifiers	2	3	4	5	6	7	8	9	10
$q_{0.05}$	1.960	2.343	2.569	2.728	2.850	2.949	3.031	3.102	3.164
$q_{0.10}$	1.645	2.052	2.291	2.459	2.589	2.693	2.780	2.855	2.920

0.05 and 0.10 are p-values

(a) Critical values for the two-tailed Nemenyi test

#classifiers	2	3	4	5	6	7	8	9	10
$q_{0.05}$	1.960	2.241	2.394	2.498	2.576	2.638	2.690	2.724	2.773
$q_{0.10}$	1.645	1.960	2.128	2.241	2.326	2.394	2.450	2.498	2.539

(b) Critical values for the two-tailed Bonferroni-Dunn test; the number of classifiers include the control classifier.

For a contrast from the single-step Bonferroni-Dunn procedure, step-up and step-down procedures sequentially test the hypotheses ordered by their significance. We will denote the ordered  $p$  values by  $p_1, p_2, \dots$ , so that  $p_1 \leq p_2 \leq \dots \leq p_{k-1}$ . The simplest such methods are due to Holm (1979) and Hochberg (1988). They both compare each  $p_i$  with  $\alpha/(k-i)$ , but differ in the order of the tests. Holm's step-down procedure starts with the most significant  $p$  value. If  $p_1$  is below  $\alpha/(k-1)$ , the corresponding hypothesis is rejected and we are allowed to compare  $p_2$  with  $\alpha/(k-2)$ . If the second hypothesis is rejected, the test proceeds with the third, and so on. As soon as a certain null hypothesis cannot be rejected, all the remaining hypotheses are retained as well. Hochberg's step-up procedure works in the opposite direction, comparing the largest  $p$  value with  $\alpha$ , the next largest with  $\alpha/2$  and so forth until it encounters a hypothesis it can reject. All hypotheses with smaller  $p$  values are then rejected as well.

p-values are calculated using z values get by this formula(), for obtaining the p-values check the tables  
<https://www.ztable.net/>  
 for details check exercise on the following page



# Post-hoc test

use the Nemenyi test for pairwise comparisons. The critical value (Table (a)) is 2.569 and the corresponding CD is  $2.569\sqrt{\frac{4.5}{6.14}} = 1.25$ . Since even the difference between the best and the worst performing algorithm is already smaller than that, we can conclude that the post-hoc test is not powerful enough to detect any significant differences between the algorithms.

At  $p=0.10$ , CD is  $2.291\sqrt{\frac{4.5}{6.14}} = 1.12$ . We can identify two groups of algorithms: the performance of pure C4.5 is significantly worse than that of C4.5+m and C4.5+cf. We cannot tell which group C4.5+cf belongs to. Concluding that it belongs to both would be a statistical nonsense since a subject cannot come from two different populations. The correct statistical statement would be that *the experimental data is not sufficient to reach any conclusion regarding C4.5+cf*.

In Table (b) we find that the critical value  $q_{0.05}$  for 4 classifiers is  $2.394\sqrt{\frac{4.5}{6.14}} = 1.16$ . C4.5+m+cf performs significantly better than C4.5 ( $3.143 - 1.964 = 1.179 > 1.16$ ) and C4.5+cf does not ( $3.143 - 2.893 = 0.250 < 1.16$ ), while C4.5+m is just below the critical difference, but close to it ( $3.143 - 2.000 = 1.143 \approx 1.16$ ).

For the other tests we have to compute and order the corresponding statistics and  $p$  values. The standard error is  $SE = \sqrt{\frac{4.5}{6.14}} = 0.488$ .

$R_0$  is the rank of the baseline approach C4.5

$i$	classifier	$z = (R_0 - R_i)/SE$	$p$ -value	$\alpha/(k-i)$	
1	C4.5+m+cf	$(3.143 - 1.964)/0.488 = 2.416$	0.016	0.017	$\alpha=0.05 \quad k=4$
2	C4.5+m	$(3.143 - 2.000)/0.488 = 2.342$	0.019	0.025	$0.05/(4-1)=0.0167$
3	C4.5+cf	$(3.143 - 2.893)/0.488 = 0.512$	0.607	0.050	$0.05/(4-2)=0.025$

	C4.5	C4.5+m	C4.5+cf	C4.5+m+cf
adult (sample)	0.763 (4)	0.768 (3)	0.771 (2)	0.798 (1)
breast cancer	0.599 (1)	0.591 (2)	0.590 (3)	0.569 (4)
cancer wisconsin	0.954 (4)	0.971 (1)	0.968 (2)	0.967 (3)
cmc	0.628 (4)	0.661 (1)	0.654 (3)	0.657 (2)
ionosphere	0.882 (4)	0.888 (2)	0.886 (3)	0.898 (1)
iris	0.936 (1)	0.931 (2.5)	0.916 (4)	0.931 (2.5)
liver disorders	0.661 (3)	0.668 (2)	0.609 (4)	0.685 (1)
lung cancer	0.583 (2.5)	0.583 (2.5)	0.563 (4)	0.625 (1)
lymphography	0.775 (4)	0.838 (3)	0.866 (2)	0.875 (1)
mushroom	1.000 (2.5)	1.000 (2.5)	1.000 (2.5)	1.000 (2.5)
primary tumor	0.940 (4)	0.962 (2.5)	0.965 (1)	0.962 (2.5)
rheum	0.619 (3)	0.666 (2)	0.614 (4)	0.669 (1)
voting	0.972 (4)	0.981 (1)	0.975 (2)	0.975 (3)
wine	0.957 (3)	0.978 (1)	0.946 (4)	0.970 (2)
average rank	3.143	2.000	2.893	1.964

$$3.143 - 1.964 = 1.179 < 1.25$$

The Holm procedure rejects the first and then the second hypothesis since the corresponding  $p$  values are smaller than the adjusted  $\alpha$ 's. The third hypothesis cannot be rejected; if there were any more, we would have to retain them, too.





# Bias-Variance Tradeoff

- When we talk about model prediction, it's important to understand prediction errors (bias and variance). There is a tradeoff between a model's ability to minimize bias and variance. Understanding of these errors would help us not only to build accurate models but also to avoid the mistake of overfitting and underfitting.
- In statistics and machine learning, the bias-variance tradeoff is the property of a set of predictive models whereby models with a lower bias in parameter estimation have a higher variance of the parameter estimates across samples, and vice versa. The bias-variance dilemma or bias-variance problem is the conflict in trying to simultaneously minimize these two sources of error that prevent supervised learning algorithms from generalizing beyond their training set.
- The bias error is an error from erroneous assumptions in the learning algorithm. High bias can cause an algorithm to miss the relevant relations between features and target outputs (underfitting). It always leads to high error on training and test data.
- The variance is an error from sensitivity to small fluctuations in the training set. High variance can cause an algorithm to model the random noise in the training data, rather than the intended outputs (overfitting). As a result, such models perform very well on training data but has high error rates on test data.



# Bias-Variance Tradeoff

Let the variable we are trying to predict as Y and other covariates as X. We assume there is a relationship between the two such that

$$Y = f(X) + e$$

Where e is the error term and it's normally distributed with a mean of 0.

We will make a model  $\hat{f}(X)$  of  $f(X)$  using linear regression or any other modeling technique.

So the expected squared error at a point x is

$$Err(x) = E[(Y - \hat{f}(x))^2]$$

[https://en.wikipedia.org/wiki/Expected\\_value](https://en.wikipedia.org/wiki/Expected_value)

The  $Err(x)$  can be further decomposed as

$$Err(x) = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$$

The **expectation** of  $X$  is defined as

$$E[X] = x_1 p_1 + x_2 p_2 + \dots + x_k p_k.$$

Since the probabilities must satisfy  $p_1 + \dots + p_k = 1$ , it is natural to interpret  $E[X]$  as a weighted average of the  $x_i$  values, with weights given by their probabilities  $p_i$ .

In the special case that all possible outcomes are equiprobable (that is,  $p_1 = \dots = p_k$ ), the weighted average is given by the standard average. In the general case, the expected value takes into account the fact that some outcomes are more likely than others.

**Non-multiplicativity:** In general, the expected value is not multiplicative, i.e.  $E[XY]$  is not necessarily equal to  $E[X] \cdot E[Y]$ . If  $X$  and  $Y$  are independent, then one can show that  $E[XY] = E[X] E[Y]$ . If the random variables are dependent, then generally  $E[XY] \neq E[X] E[Y]$ , although in special cases of dependency the equality may hold.

# Bias-Variance decomposition

Some useful properties of  $E()$ :  $E[X + Y] = E[X] + E[Y]$ ,

$$E[aX] = aE[X],$$

for a random variable  $X$  with well-defined expectation,  $E[E[X]] = E[X]$ . A well defined expectation implies that there is one number, or rather, one constant that defines the expected value. Thus follows that the expectation of this constant is just the original expected value.

We define: true or target function as  $y = f(x)$ ,

predicted target value as  $\hat{y} = \hat{f}(x)$ :

To get started with the squared error loss decomposition into bias and variance, let us do some algebraic manipulation, i.e., adding and subtracting the expected value of  $\hat{y}$  and then expanding the expression using the quadratic formula ( $a + b)^2 = a^2 + b^2 + 2ab$ ):

$$\begin{aligned} S &= (y - \hat{y})^2 \\ (y - \hat{y})^2 &= (y - E[\hat{y}] + E[\hat{y}] - \hat{y})^2 \\ &= (y - E[\hat{y}])^2 + (E[\hat{y}] - y)^2 + 2(y - E[\hat{y}])(E[\hat{y}] - \hat{y}). \end{aligned}$$

Next, we just use the expectation on both sides, and we are already done:

$$\begin{aligned} E[S] &= E[(y - \hat{y})^2] \\ E[(y - \hat{y})^2] &= (y - E[\hat{y}])^2 + E[(E[\hat{y}] - \hat{y})^2] && \text{y is deterministic, it is not related to training set} \\ &= [\text{Bias}]^2 + \text{Variance.} && E(y) = y \end{aligned}$$

You may wonder what happened to the "2ab" term ( $2(y - E[\hat{y}])(E[\hat{y}] - \hat{y})$ ) when we used the expectation. It turns that it evaluates to zero and hence vanishes from the equation, which can be shown as follows:

$$\begin{aligned} E[2(y - E[\hat{y}])(E[\hat{y}] - \hat{y})] &= 2E[(y - E[\hat{y}])(E[\hat{y}] - \hat{y})] \\ &= 2(y - E[\hat{y}])E[(E[\hat{y}] - \hat{y})] \\ &= 2(y - E[\hat{y}])(E[E[\hat{y}]] - E[\hat{y}]) \\ &= 2(y - E[\hat{y}])(E[\hat{y}] - E[\hat{y}]) \\ &= 0. \end{aligned}$$

$$\begin{aligned} E((y - E(\hat{y}))^2) &= E(y^2 + E(\hat{y})^2 - 2yE(\hat{y})) = E(y)E(y) + \\ &E(E(\hat{y})E(\hat{y})) - 2E(y)E(E(\hat{y})) = \\ &y^2 + E(\hat{y})E(\hat{y}) - 2yE(\hat{y}) = (y^2 + E(\hat{y})^2 - 2yE(\hat{y})) = (y - E(\hat{y}))^2 \end{aligned}$$

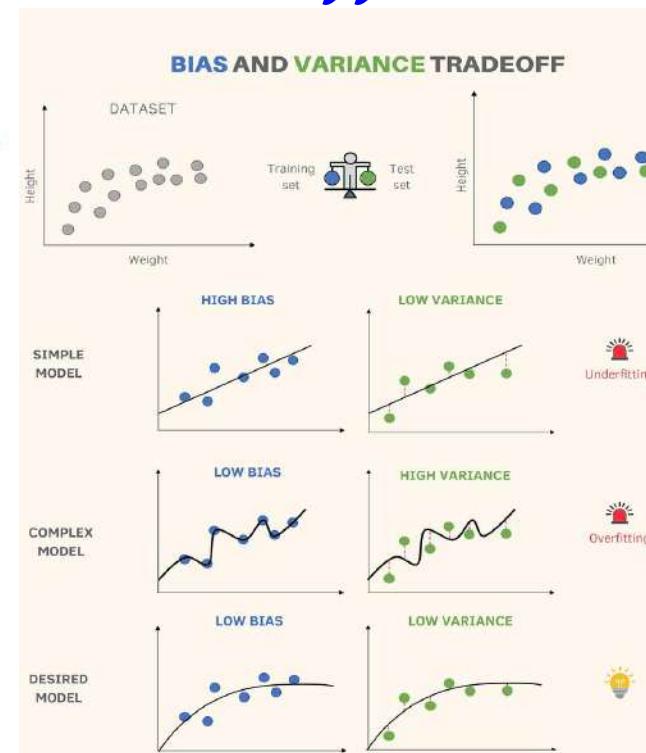
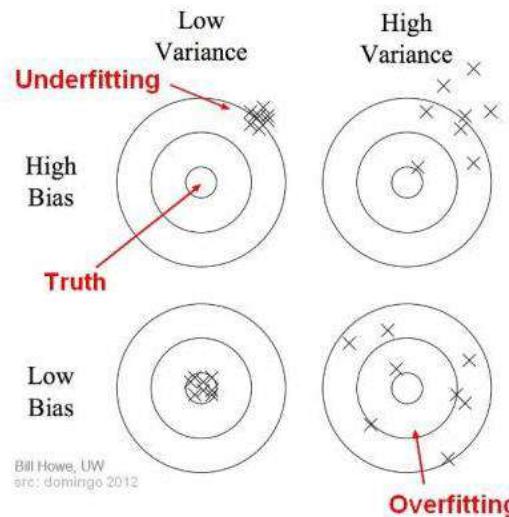
this discussion is done without considering "irreducible error" (e.g., noise, outliers, wrong labels, instrumental inaccuracies of pattern extraction), that error must be added to the bias and variance.

# Bias-Variance Tradeoff

$\text{Err}(x)$  is the sum of Bias<sup>2</sup>, variance and the irreducible error.

Irreducible error is the error that can't be reduced by creating good models. It is a measure of the amount of noise in our data. Here it is important to understand that no matter how good we make our model, our data will have certain amount of noise or irreducible error that can not be removed.

Bias and variance using bulls-eye diagram



here we see an example in the regression case, where the training patterns and those of the test set are divided. Overfitting, the training patterns are all on the curve created in the training, but in the test set the regression is not good.

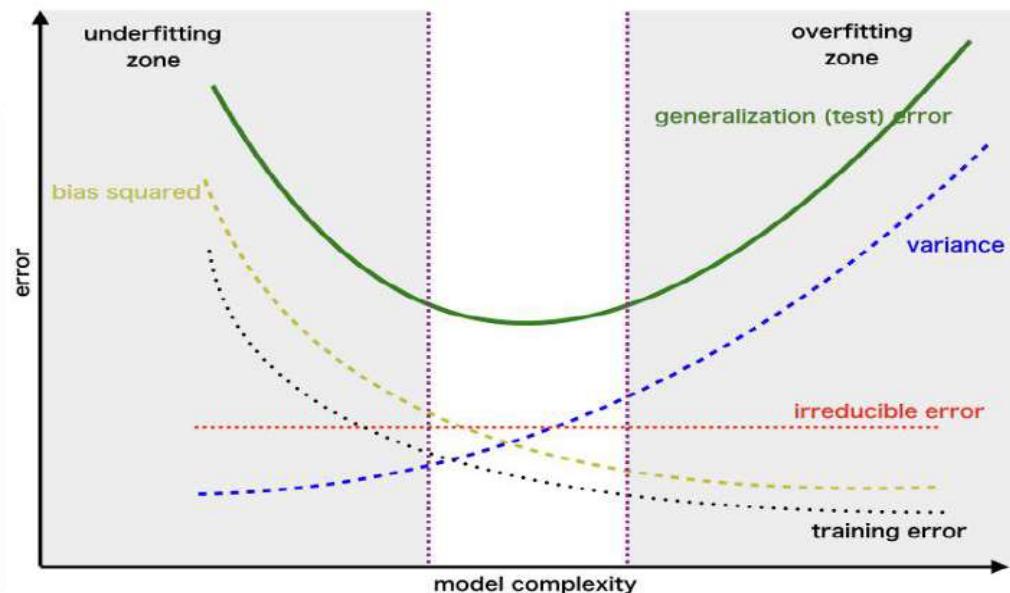
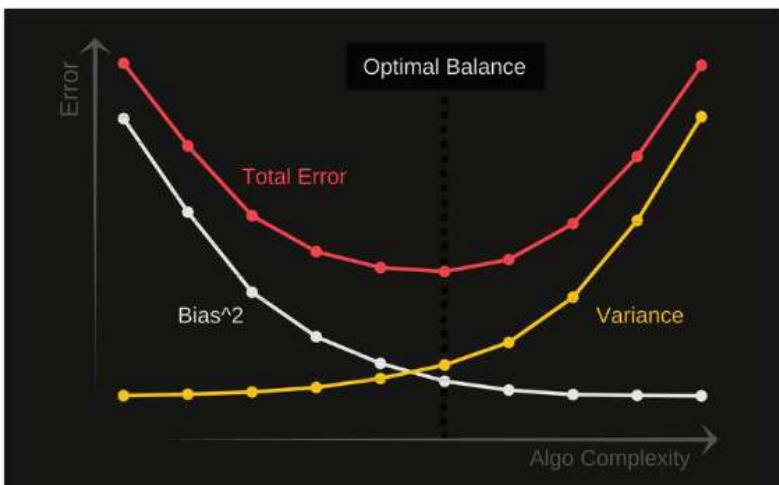
note in this case "overfitting", because in the training set the performance is always very high, with an error of almost zero, but in the test set we have uninteresting performance, some models perform quite badly

In the above diagram, center of the target is a model that perfectly predicts correct values of the test data, if error is always zero (in the training set) there is the risk of overfitting. As we move away from the bulls-eye our predictions become worse and worse. We can repeat our process of model building to get separate hits on the target.

# Bias-Variance Tradeoff

- If our model is too simple and has very few parameters then it may have high bias and low variance. On the other hand if our model has large number of parameters then it's going to have high variance and low bias. So we need to find the right/good balance without overfitting and underfitting the data.
- This tradeoff in complexity is because there is a tradeoff between bias and variance. An algorithm can't be more complex and less complex at the same time.
- To build a good model, we need to find a good balance between bias and variance such that it minimizes the total error.

$$\text{Total Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$$



- An optimal balance of bias and variance would never overfit or underfit the model.
- Therefore understanding bias and variance is critical for understanding the behavior of prediction models.



# Real World

Automate your performance evaluation procedures, run them many times, and in the end you will have saved a lot of time.

Compare the performance of your system only with others trained on the same dataset and with the same testing protocol.

Pay attention to the statistical reliability of your results.

“On more runs” intends to run the simulation several times, e.g. run n-fold cross validation k times.

Finally (but extremely important): Write structured, ordered code, perform incremental debug and unit testing. Machine learning algorithms are not “exact” and finding bugs in your code can be very difficult!

Amazon Mechanical Turk (Mturk) <https://www.mturk.com/> is a crowdsourcing Internet service that allows requesters to coordinate the use of human intelligence to perform tasks. Requesters can submit goals known as HIT (Human Intelligence Tasks), such as identifying artists from a music album, choosing the best photographs from a set of images, writing product descriptions...

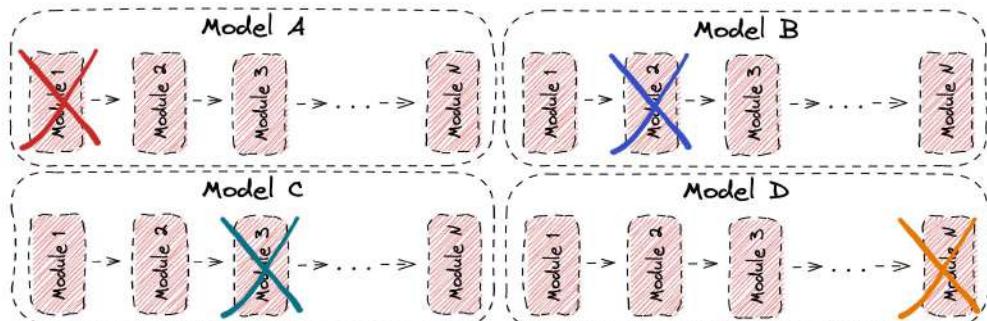
This page <https://www.tasq.ai/glossary/ablation-study/> explains very well what an ablation study is.

When you propose a new system, it is essential to report a good ablation study to motivate your choices.

When creating a unique machine learning model, you frequently add numerous concepts that contribute to the overall model performance. In a study, however, it is useful to understand the influence of each of these innovations separately. To assess the impact of individual components, researchers frequently analyze their models with each of them disabled and quantify the decline of overall model performance.

If you want to discover what a certain component of the organism performs, you break it and see what changes. This is known as an ablation study.

In the image below, we can see an example of an ablation study in a model with N modules. Each time we remove one of the modules and check the performance of the new model to investigate the influence of the removed module:

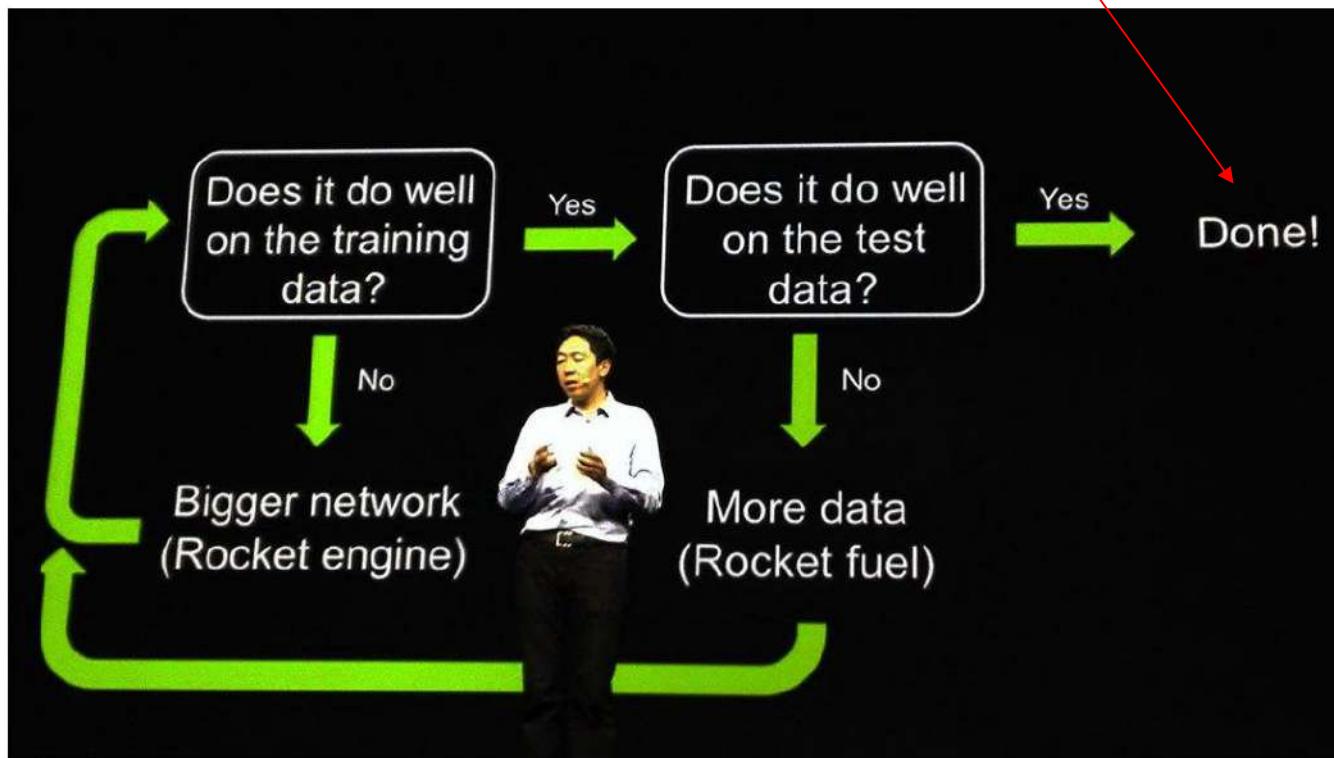


for further details, and two simple examples of ablation studies, see  
<https://www.baeldung.com/cs/ml-ablation-study>



Unfortunately, it is not that simple <https://arxiv.org/abs/2011.03395> Machine Learning (ML) models often exhibit unexpectedly poor behavior when they are deployed in real-world domains. We identify underspecification as a key reason for these failures. An ML pipeline is underspecified when it can return many predictors with equivalently strong held-out performance in the training domain.

Underspecification is common in modern ML pipelines, such as those based on deep learning. Predictors returned by underspecified pipelines are often treated as equivalent based on their training domain performance, but we show here that such predictors can behave very differently in deployment domains. This ambiguity can lead to instability and poor model behavior in practice, and is a distinct failure mode from previously identified issues arising from structural mismatch between training and deployment domains.





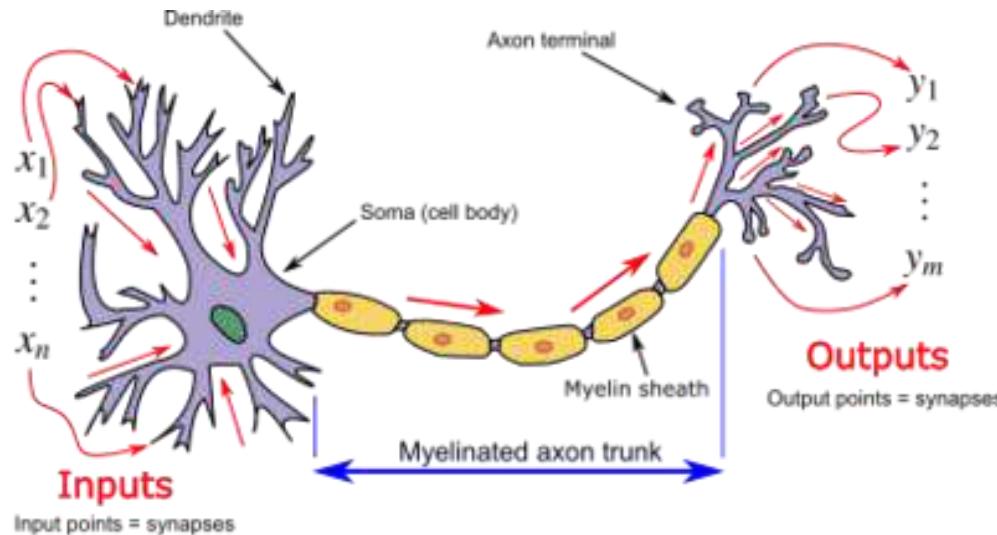
# *Neural networks - introduction*

I suggest to watch these videos:

[https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1\\_67000Dx\\_ZCJB-3pi](https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi)



# Biological neuron



Synaptic connections or (synapses) act as connecting gates for the passage of information between neurons.

Dendrites are minor fibers that branch out from the cell body of the neuron (called soma). Through the synapses the dendrites collect inputs from afferent neurons and propagate them towards the soma.

The axon is the main fiber that starts from the soma and moves away from it to bring the output to other neurons (even distant ones).



# *Biological neuron*

**Dendrite:** Receives signals from other neurons

**Soma:** Processes the information

**Axon:** Transmits the output of this neuron

**Synapse:** Point of connection to other neurons

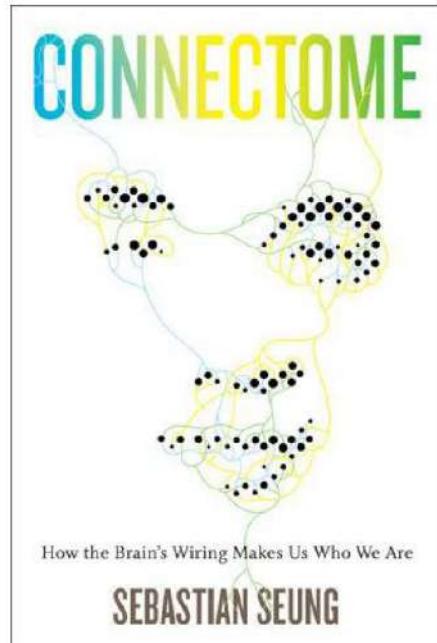
Basically, a neuron takes an input signal (dendrite), processes it like the CPU (soma), passes the output through a cable like structure to other connected neurons (axon to synapse to other neuron's dendrite). Now, this might be biologically inaccurate as there is a lot more going on out there but on a higher level, this is what is going on with a neuron in our brain — takes an input, processes it, throws out an output.



# Connectome

A connectome is a comprehensive map of neural connections in the brain, and may be thought of as its "wiring diagram". An organism's nervous system is made up of neurons which communicate through synapses. A connectome is constructed by tracing the neuron in a nervous system and mapping where neurons are connected through synapses.

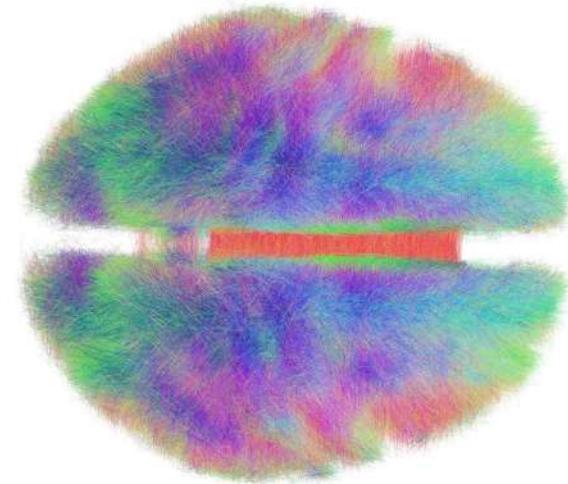
**N**O ROAD, NO trail can penetrate this forest. The long and delicate branches of its trees lie everywhere, choking space with their exuberant growth. No sunbeam can fly a path tortuous enough to navigate the narrow spaces between these entangled branches. All the trees of this dark forest grew from 100 billion seeds planted together. And, all in one day, every tree is destined to die.



[Sebastian Seung, 2012]

Connectome

*How the Brain's Wiring  
Make Us Who We Are*



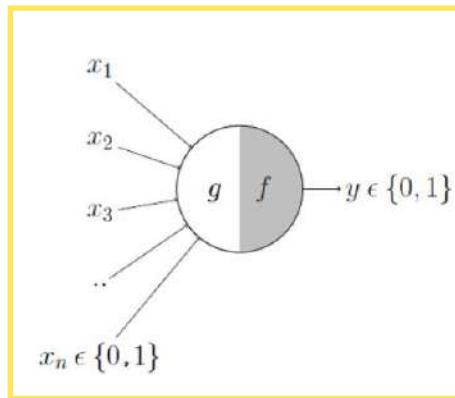
Rendering of a group connectome based on 20 subjects. Anatomical fibers that constitute the white matter architecture of the human brain are visualized color-coded by traversing direction (xyz-directions mapping to RGB colors respectively). Visualization of fibers was done using TrackVis software.



# McCulloch-Pitts

further details  
<https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1>

The first computational model of a neuron was proposed by Warren McCulloch (neuroscientist) and Walter Pitts (logician) in 1943.



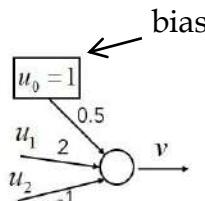
$$g(x_1, x_2, x_3, \dots, x_n) = g(\mathbf{x}) = \sum_{i=1}^n x_i$$

$$\begin{aligned} y = f(g(\mathbf{x})) &= 1 & \text{if } g(\mathbf{x}) \geq \theta \\ &= 0 & \text{if } g(\mathbf{x}) < \theta \end{aligned}$$

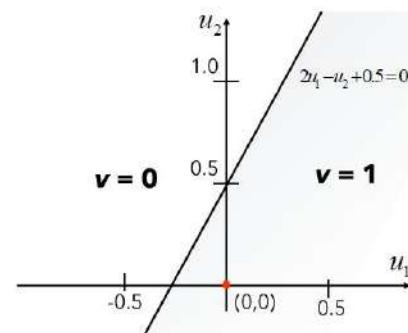
We can see that  $g(\mathbf{x})$  is just doing a sum of the inputs — a simple aggregation. And *theta* here is called thresholding parameter.

It may be divided into 2 parts. The first part,  $g$  takes an input (ahem dendrite ahem), performs an aggregation and based on the aggregated value the second part,  $f$  makes a decision.

Geometric interpretation of the perceptron



$$v = \begin{cases} 1, & 2u_1 - u_2 + 0.5 \geq 0 \\ 0, & 2u_1 - u_2 + 0.5 < 0 \end{cases}$$



The importance of the bias will be better seen in the exercises



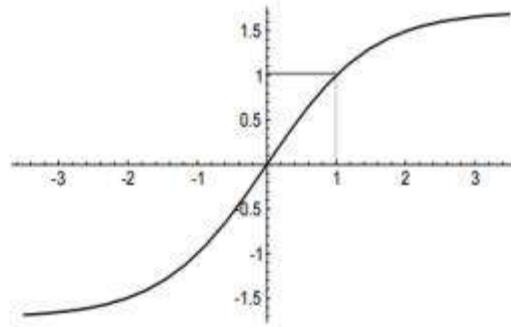
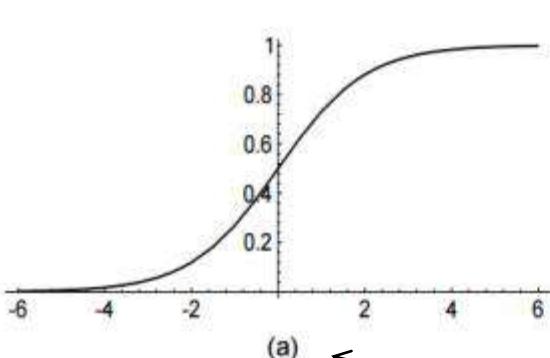


# Neurons

- In biological neurons when  $f()$  exceeds a certain threshold, the neuron "fires" a spike (impulse) and then returns to rest.
- The most commonly used neural networks  $f()$  is a non-linear, continuous and differentiable function.
- Why not linear? If we want a network to be able to perform a (complex) mapping of the input information, non-linearities are necessary.
- Why is it continuous and differentiable? Necessary for error back-propagation (such as we'll find out soon).
- The most commonly used activation function in more recent shallow networks is the sigmoid, in the variants:
  - a) standard logistics function
  - b) hyperbolic tangent



# Activation functions for shallow networks



## ■ Standard logistic function

$$f(\text{net}) = \frac{1}{1 + e^{-\text{net}}}$$

derivative:

$$f'(\text{net}) = \frac{\partial}{\partial \text{net}} \left( \frac{1}{1 + e^{-\text{net}}} \right) = \frac{e^{-\text{net}}}{(1 + e^{-\text{net}})^2} = f(\text{net})(1 - f(\text{net}))$$

## ■ hyperbolic tangent (Tanh),

$$f(\text{net}) = \frac{2a}{1 + e^{-2b \cdot \text{net}}} - a$$

$$f'(\text{net}) = \frac{\partial}{\partial \text{net}} \left( \frac{2a}{1 + e^{-2b \cdot \text{net}}} - a \right) = \frac{4ab \cdot e^{-2b \cdot \text{net}}}{(1 + e^{-2b \cdot \text{net}})^2}$$

(a):

(b):

$$D \left[ \frac{f(x)}{g(x)} \right] = \frac{f'(x) \cdot g(x) - f(x) \cdot g'(x)}{[g(x)]^2} \quad ; \quad g(x) \neq 0$$

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

$$\frac{d}{dx} f(x) = \frac{e^x \cdot (1 + e^x) - e^x \cdot e^x}{(1 + e^x)^2} = \frac{e^x}{(1 + e^x)^2} = f(x)(1 - f(x))$$

Recommended parameters values:  $a = 1.716$ ,  $b = 2/3$   
 $f'(0)$  is approximately 1. The linear range is between -1 and +1 (approximately).

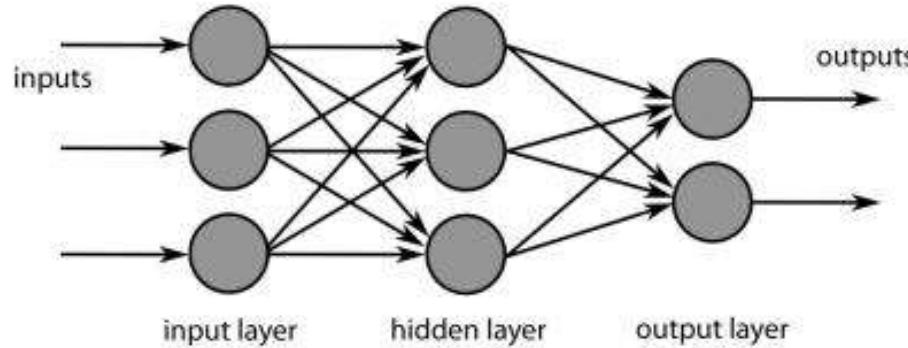
Slightly more complex but preferable (often faster convergence), as it is symmetrical with respect to zero.



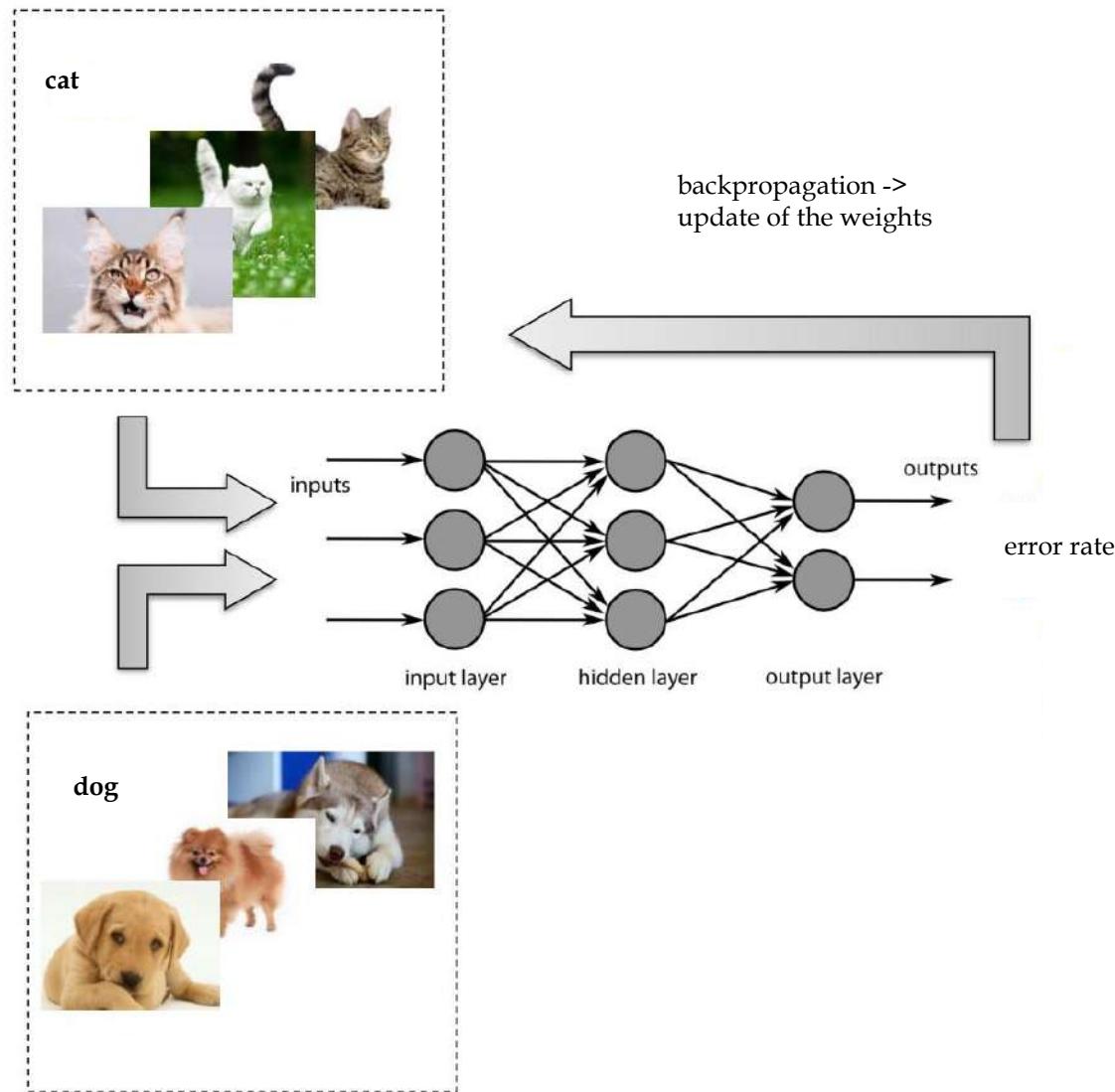
# FeedForward

Neural networks are made up of groups of artificial neurons organized in layers. Typically there are: an input level, an output level, and one or more intermediate or hidden levels. Each level contains one or more neurons.

Feedforward: In feedforward networks connections connect neurons of one level with neurons of a subsequent level. No backwards connections or connections towards the same level are allowed. It is by far the most used type of network.



# *Basic idea*



# *Recurrent*

In recurring networks, feedback connections (usually towards neurons of the same level, but also backwards) are allowed. This greatly complicates the flow of information and training, requiring you to consider the behavior in multiple instants of time (unfolding in time).

These networks are indicated for the management of sequences (e.g. audio, video, sentences in natural language), because they are equipped with a memory effect (short term) which at time  $t$  makes the processed information available at  $(t - 1)$ ,  $(t - 2)$ , etc.

E.g. a particular type of recurring network is LSTM (Long Short Term Memory).



# Perceptron

The term perceptron (perceptron) derives from the neuron model proposed by Rosenblatt in 1956 (journal paper - 1958: Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6), 386–408), very similar to that of McCulloch and Pitts.

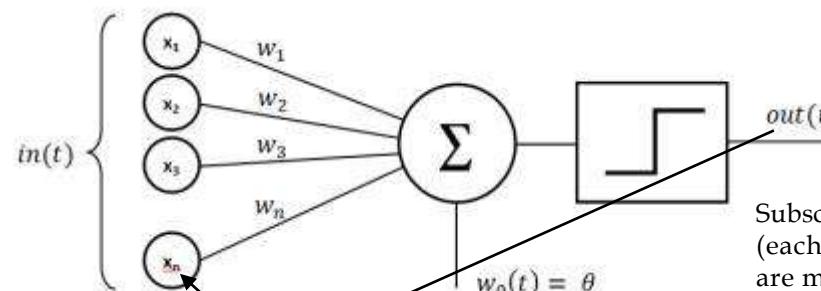
The original perceptron uses a linear threshold (or step) activation function. A single perceptron, or a network of only two levels (input and output), can be trained with a simple rule called delta rule.

In 1958, Rosenblatt stirred a controversy among the budding AI community by announcing his *perceptron*. Speculations flourished. New York Times reported that the perceptron is the “embryo of an electronic computer” which will be capable of mimicking numerous human activities such as walking, talking, seeing, writing, and even reproducing itself. Not only that, but this computer will be conscious of its existence! Wow!

Rosenblatt constructed his Perceptron automaton, named Mark I, which occupied 6 racks of electronic equipment. Mark I was meant to be a visual pattern classifier. The success of this early-days physical model led to the current fame and glory of the perceptron.



The Institute of Electrical and Electronics Engineers (IEEE), the world's largest professional association dedicated to advancing technological innovation and excellence for the benefit of humanity, named its annual award in honour of Frank Rosenblatt.



$\Delta w_{ji} = \alpha(t_j - y_j)x_i$   
 $t_j$  is the target output  
 $\alpha$  is a small constant called *learning rate*  
 $y_j$  is network output  
 $x_i$  is the  $i$ th input.

Subscript  $j$  indicates the various training patterns (each represented by a vector  $x$ ), the weights  $w$  are modified considering the entire training set, i.e. for each pattern of the training set:

$$w_i \leftarrow w_i + \Delta w_i$$

A two-level network of linear threshold perceptrons is able to learn only linear mappings and therefore the number of approximable functions is rather limited.

$$\begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

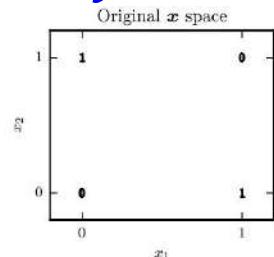
$\mathbf{w} \cdot \mathbf{x}$  is the dot product

$\sum_{i=1}^m w_i x_i$ , where  $m$  is the number of inputs to the perceptron and  $b$  is the bias.

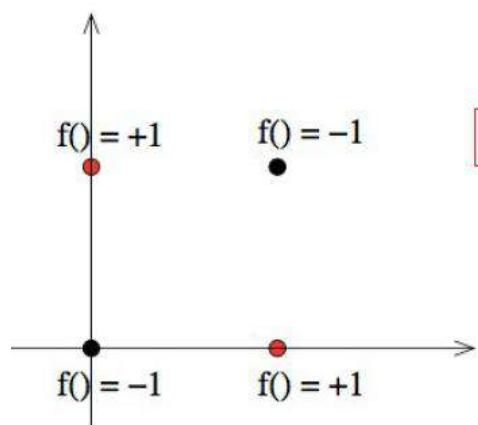
In the figure, the number of inputs was represented by the variable 'n'

# XOR function

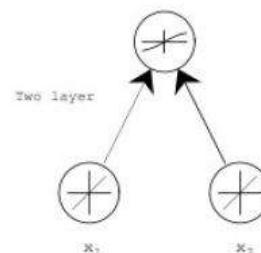
- XOR function
- $Tr = \{([0,0], 0), ([0,1], 1), ([1,0], 1), ([1,1], 1)\}$
- Let us define our model as  
 $f(\mathbf{x}; \mathbf{w}; b) = \mathbf{x}\mathbf{w}^T + b$
- The XOR function cannot be represented by any linear classifier
- Including a single-layer neural network



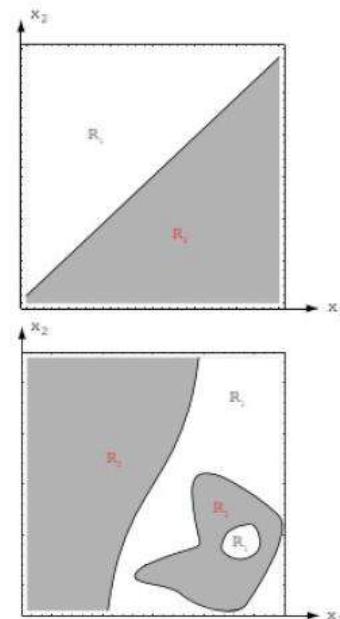
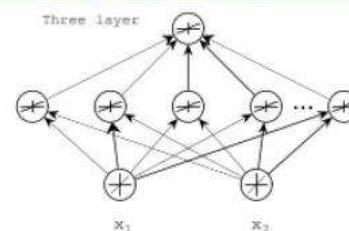
## Decision Surfaces



Perceptron



Multi-layer perceptron



# Multilayer

A Multilayer Perceptron (MLP) is a feedforward network with at least 3 levels (at least 1 hidden) and with non-linear activation functions.

A theorem known as the universal approximation theorem asserts that any continuous function that maps ranges of real numbers to a range of real numbers can be approximated by an MLP with only one hidden layer.

This is one of the reasons why for many decades (until the explosion of deep learning) there has been a focus on 3-level neural networks. On the other hand, the theoretical existence of a solution does not imply that there is an effective way to obtain it ...

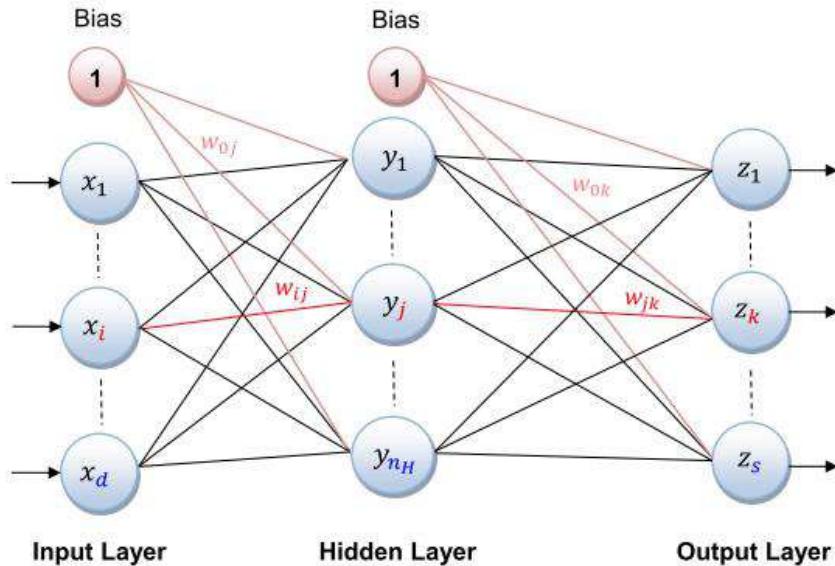
The approximation properties of multilayer networks have been the subject of numerous studies. In particular, 2-layer networks with 1 hidden layer have been shown to be universal approximators for continuous functions on compact sets of  $\mathbb{R}^n$  (<https://www.youmath.it/lezioni/analisi-matematica/premesse-per-lanalisi-infinitesimale/3493-insieme-compatto-teorema-di-heine-borel.html>)



# Forward Propagation

With forward propagation (or inference) we mean the propagation of information forward: from the input level to the output level.

Once trained, a neural network can simply process patterns through forward propagation.



in the example a 3-level network is shown: input  $d$  neurons, hidden level  $n_H$  neurons, output  $s$  neurons.

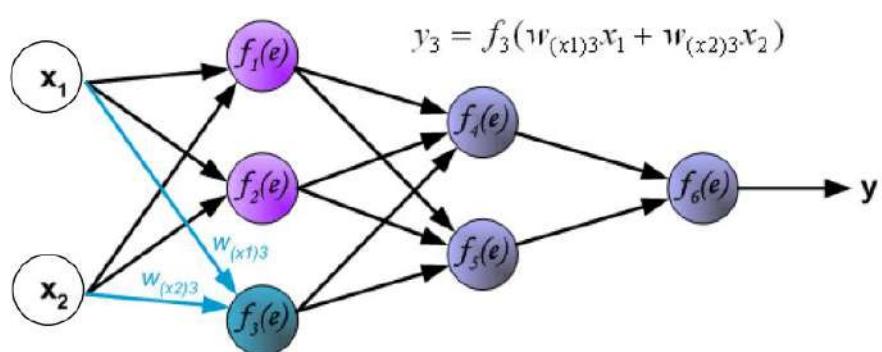
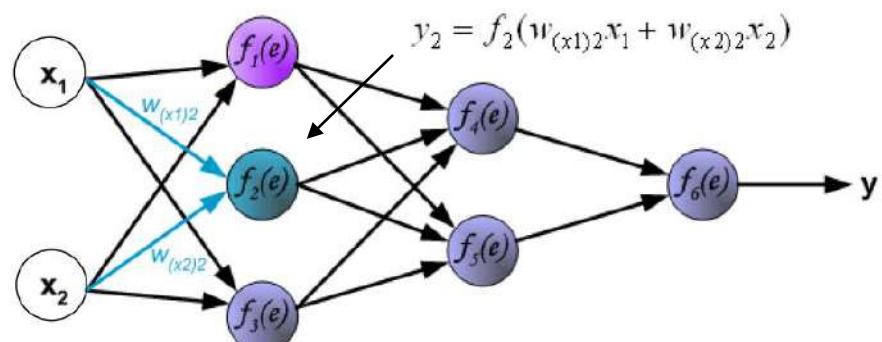
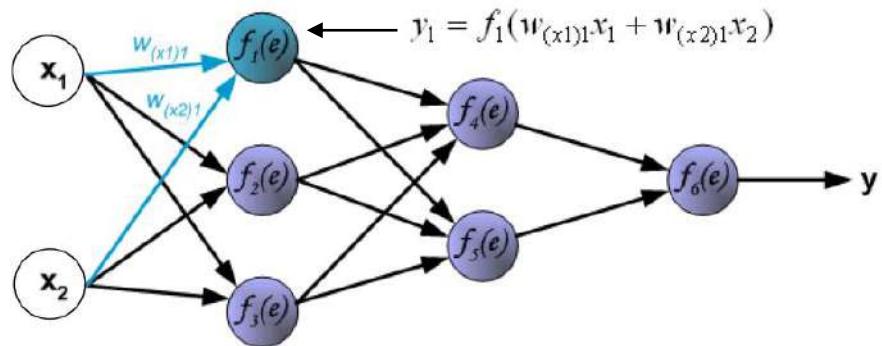
The total number of weights (parameters) is:  $d \times n_H + n_H \times s + n_H + s$   
where the last two terms correspond to the weights of the biases.

The  $k$ -th output value can be calculated as:

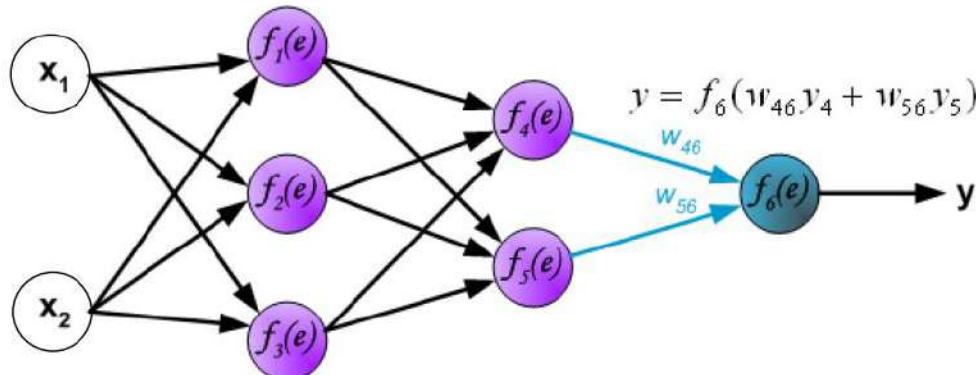
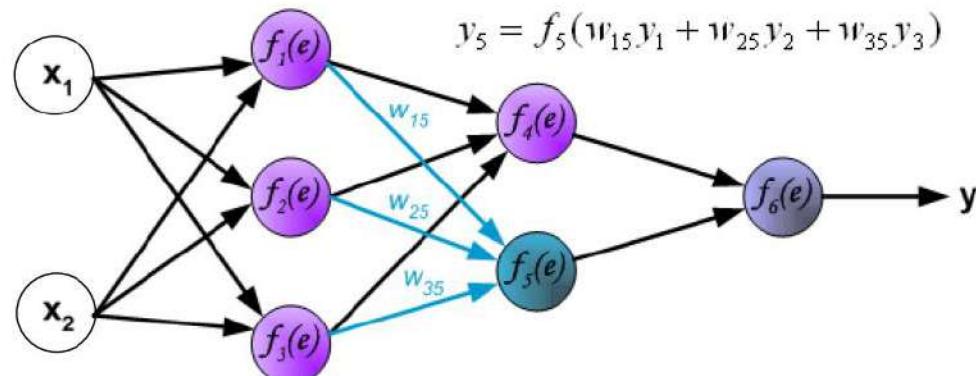
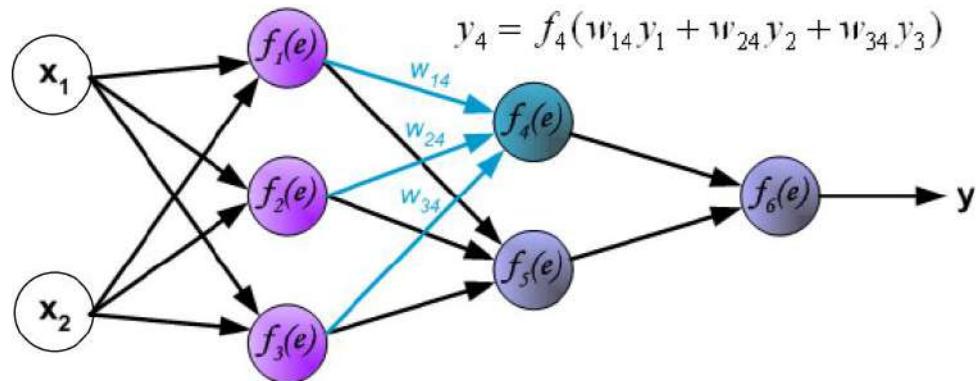
$$z_k = f \left( \sum_{j=1 \dots n_H} w_{jk} \cdot y_j + w_{0k} \right) = f \left( \sum_{j=1 \dots n_H} w_{jk} \cdot f \left( \sum_{i=1 \dots d} w_{ij} \cdot x_i + w_{0j} \right) + w_{0k} \right)$$

# Example

In the example a 4-level network 2: 3: 2: 1 (2 hidden levels); no bias are used; in the graphics of the example the notation is a little different from the previous one but easily understandable.



# Example





# *Training*

Once the topology (number of levels and neurons) is fixed, the training of a neural network consists in finding the value of the weights that determine the desired mapping between input and output.

What do we mean by desired mapping? It depends on the problem we want to solve:

- a) if we are interested in training a neural network to act as a classifier, the desired output is the correct class label of the input pattern;
- b) if the neural network has to solve a regression problem, the desired output is the correct value of the dependent variable, corresponding to the value of the independent variables supplied as input.





# *Training*

Although the first artificial neurons date back to the 1940s, effective training algorithms were not available until the mid-1980s.

In 1986 Rumelhart, Hinton & Williams introduced the Error Backpropagation algorithm arousing great attention in the scientific community.

It is actually the application of the chain derivation rule, an idea that no one had successfully applied in this context for decades.



for details read:

<https://cs231n.github.io/optimization-2/>





# *Training*

In the following we focus our attention on a supervised classification problem.

To handle a classification problem with  $s$  classes and  $d$ -dimensional patterns, it is usual to use a network with  $d$  input neurons,  $nh$  hidden neurons, and  $s$  output neurons. That is, the number of input neurons is the same of the features and the number of output neurons is equal to the number of classes.  $nh$  is instead a hyperparameter: a reasonable value is  $nh = 1/10 d$ .

The pre-normalization of the input patterns (eg whitening), although not mandatory, can improve the convergence during training.

~~the (supervised) training takes place by presenting the pattern of which the class is known to the network and propagating (forward propagation) the inputs to the outputs.~~

The covariance matrix of the normalized data is the identity, it is obtained after projecting the patterns on the space defined by the first  $k$  eigenvectors, then it is sufficient to divide each dimension by the square root of the corresponding eigenvalue.



# Training

Given a training pattern whose class is  $g$ , the desired output vector  $t$  takes the form:

$$t = [-1, -1 \dots 1 \dots -1]$$

The difference between the output produced by the network and the desired one is the network error. Objective of the learning algorithm is to modify the net weights in order to minimize the average error on training set patterns.

Before the start of training, the weights are typically initialized with random values:

input-hidden      range  $\pm 1/\sqrt{d}$

hidden-output      range  $\pm 1/\sqrt{n_H}$



# Training

With reference to the 3-level network. Let  $z$  be the output produced by the network (forward propagation) in correspondence with the pattern  $x$  of class  $g$  supplied in input; as already mentioned, the output desired is  $t = [t_1, t_2 \dots t_s]$ , where  $t_i = 1$  for  $i = g$ ,  $t_i = -1$  otherwise. By choosing the sum of the squares of the errors as the loss function, the error (for pattern  $x$ ) is:

$$J(w, x) \equiv \sum_{c=1 \dots s} (t_c - z_c)^2$$

which quantifies how much the output produced for pattern  $x$  differs from the desired one.

The error  $J(w)$  on the entire training set is the average of  $J(w, x)$  on all the patterns  $x$  belonging to the training set.



# Derivatives recap

## Derivatives

### Definition and Notation

If  $y = f(x)$  then the derivative is defined to be  $f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$ .

If  $y = f(x)$  then all of the following are equivalent notations for the derivative.

$$f'(x) = y' = \frac{df}{dx} = \frac{dy}{dx} = \frac{d}{dx}(f(x)) = Df(x)$$

If  $y = f(x)$  all of the following are equivalent notations for derivative evaluated at  $x = a$ .

$$f'(a) = y' \Big|_{x=a} = \frac{df}{dx} \Big|_{x=a} = \frac{dy}{dx} \Big|_{x=a} = Df(a)$$

### Interpretation of the Derivative

If  $y = f(x)$  then,

- $m = f'(a)$  is the slope of the tangent line to  $y = f(x)$  at  $x = a$  and the equation of the tangent line at  $x = a$  is given by  $y = f(a) + f'(a)(x-a)$ .

- $f'(a)$  is the instantaneous rate of change of  $f(x)$  at  $x = a$ .
- If  $f(x)$  is the position of an object at time  $x$  then  $f'(a)$  is the velocity of the object at  $x = a$ .

### Basic Properties and Formulas

If  $f(x)$  and  $g(x)$  are differentiable functions (the derivative exists),  $c$  and  $n$  are any real numbers,

- $(cf)' = c f'(x)$
- $(f \pm g)' = f'(x) \pm g'(x)$
- $(fg)' = f'g + fg' - \text{Product Rule}$
- $\left(\frac{f}{g}\right)' = \frac{f'g - fg'}{g^2} - \text{Quotient Rule}$

- $\frac{d}{dx}(c) = 0$
- $\frac{d}{dx}(x^n) = nx^{n-1} - \text{Power Rule}$
- $\frac{d}{dx}(f(g(x))) = f'(g(x))g'(x)$

This is the **Chain Rule**

### Common Derivatives

$$\begin{aligned} \frac{d}{dx}(x) &= 1 \\ \frac{d}{dx}(\sin x) &= \cos x \\ \frac{d}{dx}(\cos x) &= -\sin x \\ \frac{d}{dx}(\tan x) &= \sec^2 x \\ \frac{d}{dx}(\sec x) &= \sec x \tan x \end{aligned}$$

$$\begin{aligned} \frac{d}{dx}(\csc x) &= -\csc x \cot x \\ \frac{d}{dx}(\cot x) &= -\csc^2 x \\ \frac{d}{dx}(\sin^{-1} x) &= \frac{1}{\sqrt{1-x^2}} \\ \frac{d}{dx}(\cos^{-1} x) &= -\frac{1}{\sqrt{1-x^2}} \\ \frac{d}{dx}(\tan^{-1} x) &= \frac{1}{1+x^2} \end{aligned}$$

$$\begin{aligned} \frac{d}{dx}(a^x) &= a^x \ln(a) \\ \frac{d}{dx}(e^x) &= e^x \\ \frac{d}{dx}(\ln(x)) &= \frac{1}{x}, \quad x > 0 \\ \frac{d}{dx}(\ln|x|) &= \frac{1}{x}, \quad x \neq 0 \\ \frac{d}{dx}(\log_a(x)) &= \frac{1}{x \ln a}, \quad x > 0 \end{aligned}$$

## Chain Rule Variants

The chain rule applied to some specific functions.

- $\frac{d}{dx}([f(x)]^n) = n[f(x)]^{n-1} f'(x)$
- $\frac{d}{dx}(e^{f(x)}) = f'(x)e^{f(x)}$
- $\frac{d}{dx}(\ln[f(x)]) = \frac{f'(x)}{f(x)}$
- $\frac{d}{dx}(\sin[f(x)]) = f'(x)\cos[f(x)]$
- $\frac{d}{dx}(\cos[f(x)]) = -f'(x)\sin[f(x)]$
- $\frac{d}{dx}(\tan[f(x)]) = f'(x)\sec^2[f(x)]$
- $\frac{d}{dx}(\sec[f(x)]) = f'(x)\sec[f(x)]\tan[f(x)]$
- $\frac{d}{dx}(\tan^{-1}[f(x)]) = \frac{f'(x)}{1+[f(x)]^2}$

### Higher Order Derivatives

The Second Derivative is denoted as

$$f''(x) = f^{(2)}(x) = \frac{d^2 f}{dx^2} \text{ and is defined as}$$

$f''(x) = (f'(x))'$ , i.e. the derivative of the first derivative,  $f'(x)$ .

The  $n^{\text{th}}$  Derivative is denoted as

$$f^{(n)}(x) = \frac{d^n f}{dx^n} \text{ and is defined as}$$

$f^{(n)}(x) = (f^{(n-1)}(x))'$ , i.e. the derivative of the  $(n-1)^{\text{th}}$  derivative,  $f^{(n-1)}(x)$ .

### Implicit Differentiation

Find  $y'$  if  $e^{2x-9y} + x^5 y^3 = \sin(y) + 11x$ . Remember  $y = y(x)$  here, so products/quotients of  $x$  and  $y$  will use the product/quotient rule and derivatives of  $y$  will use the chain rule. The “trick” is to differentiate as normal and every time you differentiate a  $y$  you tack on a  $y'$  (from the chain rule). After differentiating solve for  $y'$ .

$$\begin{aligned} e^{2x-9y}(2-9y') + 3x^2 y^2 + 2x^3 y y' &= \cos(y)y' + 11 \\ 2e^{2x-9y} - 9y'e^{2x-9y} + 3x^2 y^2 + 2x^3 y y' &= \cos(y)y' + 11 \quad \Rightarrow \quad y' = \frac{11 - 2e^{2x-9y} - 3x^2 y^2}{2x^3 y - 9e^{2x-9y} - \cos(y)} \\ (2x^3 y - 9e^{2x-9y} - \cos(y))y' &= 11 - 2e^{2x-9y} - 3x^2 y^2 \end{aligned}$$

### Increasing/Decreasing – Concave Up/Concave Down

#### Critical Points

$x = c$  is a critical point of  $f(x)$  provided either

- $f'(c) = 0$  or 2.  $f'(c)$  doesn't exist.

#### Increasing/Decreasing

- If  $f'(x) > 0$  for all  $x$  in an interval  $I$  then  $f(x)$  is increasing on the interval  $I$ .
- If  $f'(x) < 0$  for all  $x$  in an interval  $I$  then  $f(x)$  is decreasing on the interval  $I$ .
- If  $f'(x) = 0$  for all  $x$  in an interval  $I$  then  $f(x)$  is constant on the interval  $I$ .

#### Concave Up/Concave Down

- If  $f''(x) > 0$  for all  $x$  in an interval  $I$  then  $f(x)$  is concave up on the interval  $I$ .
- If  $f''(x) < 0$  for all  $x$  in an interval  $I$  then  $f(x)$  is concave down on the interval  $I$ .

#### Inflection Points

$x = c$  is an inflection point of  $f(x)$  if the concavity changes at  $x = c$ .

# Objective function

We want to maximize/minimize an **objective function**

- Minimizing **Cost/error/loss** function

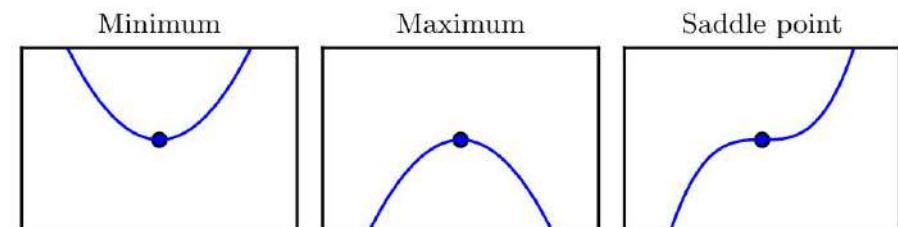
Consider a function  $y = f(x)$

- The **derivative**  $f'(x)$  or  $\frac{dy}{dx}$  gives the **slope** of  $f(x)$  at point  $x$ , or equivalently

$$f'(x) = \lim_{\epsilon \rightarrow 0} \frac{f(x + \epsilon) - f(x)}{\epsilon}$$

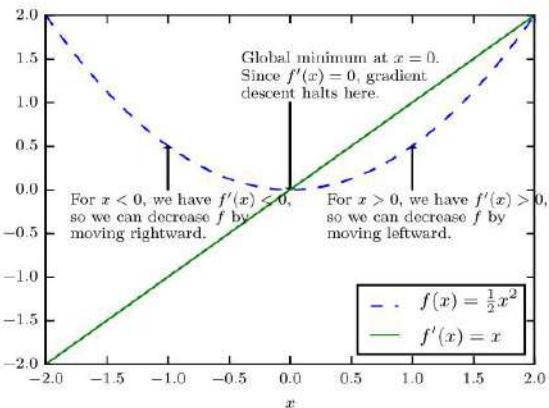
$$f(x + \epsilon) \approx f(x) + \epsilon f'(x)$$

- Derivative tells us how to change  $x$  to make a small improvement in  $y$
- **Critical/stationary** points:  $f'(x) = 0$  **No Information!**
- Can be **local maxima, minima** or **saddle** points



# Idea of gradient descent

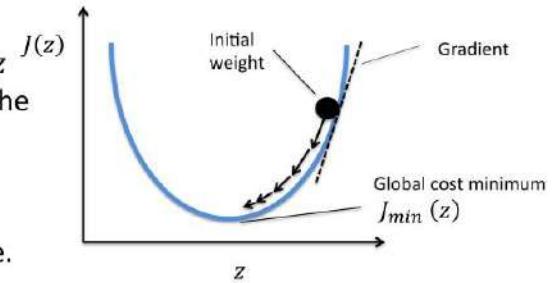
- To minimize a function in 1 variable, we have to move in the direction **opposite** to the derivative



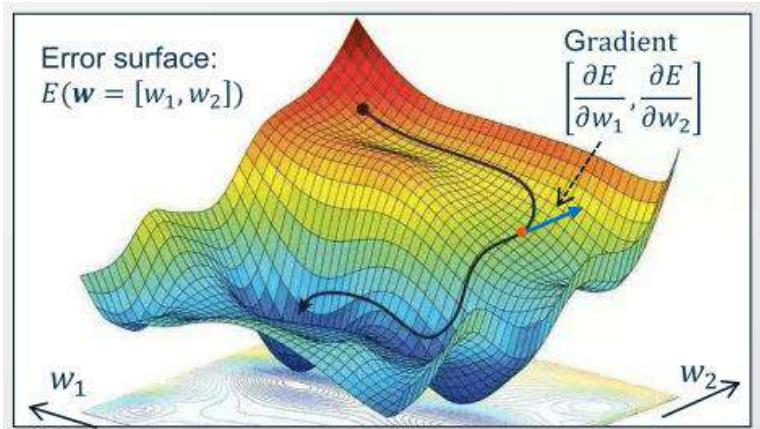
- Let's try to find the minimum with gradient descent of  $J(z) = z^2$
- Compute the gradient (derivative)  $J'(z) = 2z$  (check the cheat sheet)
- It is the slope of the tangent, i.e. points in the direction of the greatest rate of increase of the function

### Idea of gradient descent:

- Start with a random value for  $z$
  - Update  $z$  doing a little step in the opposite direction w.r.t. the gradient ( $\alpha$  parameter)
- $$z = z - \alpha J'(z)$$
- Repeat 2 until convergence (i.e. gradient very small)

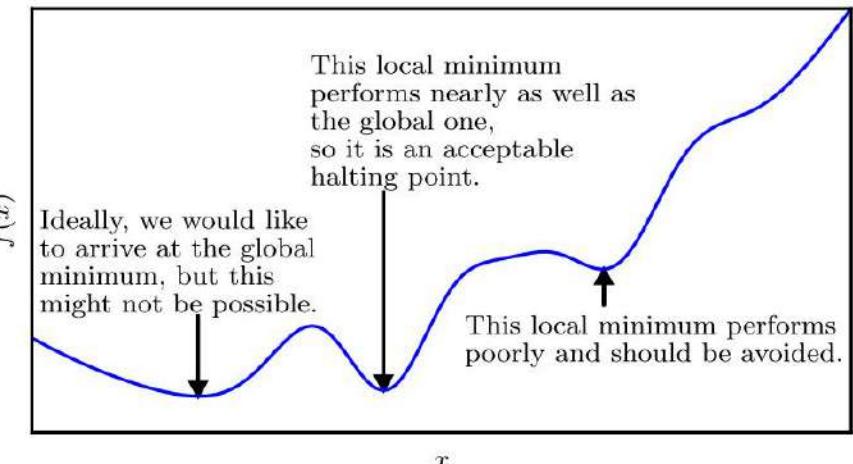


AL: start from  $z=2$ , with  $\alpha=0.25$  find the minima of the function



# Objective function

- Let's start (at random) with  $z^0 = 2$ .
- (gradient)  $J'(2) = 2 \cdot 2 = 4$
- Let's update  $z$  in the opposite direction w.r.t. the gradient. Let's also define our step size as  $\alpha = 0.25$
- $z^1 = z^0 - 0.25 \cdot 4 = 1$
- We now have  $J'(1) = 2 \cdot 1 = 2$
- $z^2 = z^1 - 0.25 \cdot 2 = 0.5$
- $J'(0.5) = 2 \cdot 0.5 = 1$
- ....



- Global minimum vs local minima

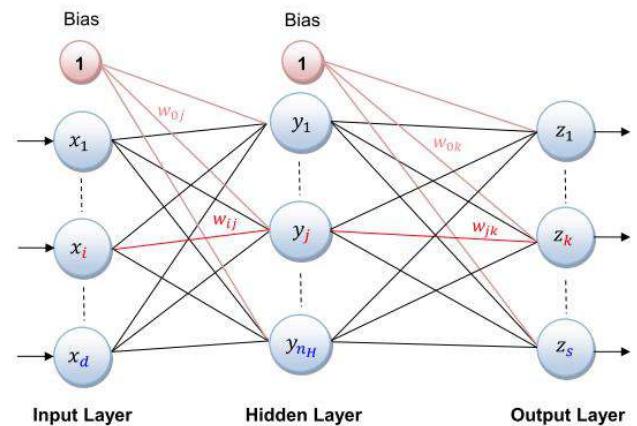


# Training

$J(\mathbf{w})$  can be reduced by changing the weights  $w$  in the opposite direction to the gradient of  $J$ . In fact the gradient indicates the direction of greatest growth of a function (of several variables) and moving in the opposite direction we reduce (to the maximum) the error. When the minimization of the error occurs through steps in the opposite direction to the gradient, the backpropagation algorithm is also called gradient descent.

In the following, we consider:

first the update of the weights  $w_{jk}$  hidden-output;  
then the update of the weights  $w_{ij}$  input-hidden.



[https://en.wikipedia.org/wiki/Chain\\_rule](https://en.wikipedia.org/wiki/Chain_rule)

<https://www.youmath.it/domande-a-risposte/view/5925-derivata-del-quadrato-di-un-binomio.html> applying the rule of derivation of compound function. Attention the link is for classical derivative, in our proof we have partial derivatives.

$$f(x) = (ax + b)^2$$

We derive the power and multiply the whole by the derivative of the base

$$f'(x) = \frac{d}{dx}[(ax + b)^2] = 2(ax + b)^{2-1} \cdot \frac{d}{dx}[ax + b] =$$

$$\begin{aligned} \frac{\partial J}{\partial w_{jk}} &= \frac{\partial}{\partial w_{jk}} \left( \frac{1}{2} \sum_{c=1 \dots s} (t_c - z_c)^2 \right) = (t_k - z_k) \frac{\partial (-z_k)}{\partial w_{jk}} = \\ &\quad \text{Definition of } J \\ &= (t_k - z_k) \frac{\partial (-f(\text{net}_k))}{\partial w_{jk}} = -(t_k - z_k) \cdot \frac{f(\text{net}_k)}{\partial \text{net}_k} \cdot \frac{\partial \text{net}_k}{\partial w_{jk}} = \\ &= -(t_k - z_k) \cdot f'(\text{net}_k) \cdot \frac{\partial \sum_{s=1 \dots n_H} w_{sk} \cdot y_s}{\partial w_{jk}} = -(t_k - z_k) \cdot f'(\text{net}_k) \cdot y_j \end{aligned}$$

$$\delta_k = (t_k - z_k) \cdot f'(\text{net}_k) \quad (\text{Eq. 2})$$

$$\frac{\partial J}{\partial w_{jk}} = -\delta_k \cdot y_j \quad (\text{Eq. 3})$$

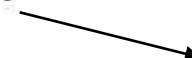
only  $y_j$  is influenced by  $w_{jk}$

# Training

therefore the weight  $w_{jk}$  can be updated as:

The direction is opposite to the gradient

$$w_{jk} = w_{jk} + \eta \cdot \delta_k \cdot y_j$$



For definition of  $y_j$  see Eq. (2) and eq. (3) of the previous page;  
it is not the  $y$  output value of the entire network but an hidden  
layer neuron

where  $\eta$  is the learning rate.

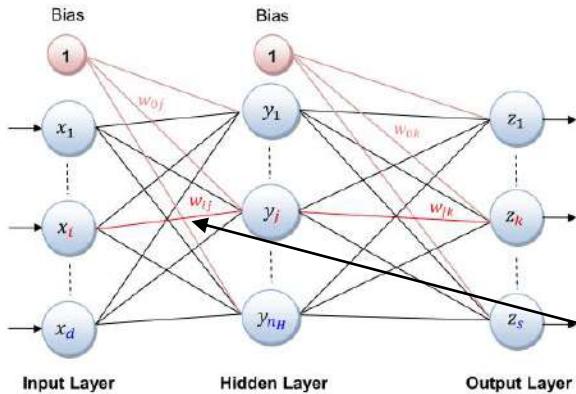
If  $\eta$  is too small, slow convergence, if too large it can oscillate and / or diverge. Start with  $\eta = 0.5$  and change the value if it is not adequate (by monitoring the trend of the loss during the iterations).

N.B. if biases are used, the weight  $w_{0k}$  is updated by setting  $y_0 = 1$



# Training

weights  $w_{ij}$  input-hidden



$$\delta_j = f'(net_j) \cdot \sum_{c=1 \dots s} w_{jc} \cdot \delta_c \quad (\text{Eq. 5})$$

$$\frac{\partial J}{\partial w_{ij}} = -\delta_j \cdot x_i \quad (\text{Eq. 6})$$

$$w_{ij} = w_{ij} + \eta \cdot \delta_j \cdot x_i \quad (\text{Eq. 7})$$

$$\begin{aligned} \text{Definition of } J & \quad \text{All the } z_c \text{ are influenced by } w_{ij} \\ \frac{\partial J}{\partial w_{ij}} &= \frac{\partial}{\partial w_{ij}} \left( \frac{1}{2} \sum_{c=1 \dots s} (t_c - z_c)^2 \right) = - \sum_{c=1 \dots s} (t_c - z_c) \frac{\partial z_c}{\partial w_{ij}} = \\ &= - \sum_{c=1 \dots s} (t_c - z_c) \frac{\partial z_c}{\partial net_c} \cdot \frac{\partial net_c}{\partial w_{ij}} = - \sum_{c=1 \dots s} (t_c - z_c) \cdot f'(net_c) \cdot \frac{\partial net_c}{\partial w_{ij}} \end{aligned}$$

$$\begin{aligned} \text{Definition of } net_c & \quad \text{only } net_j \text{ is influenced by } w_{ij} \\ \frac{\partial net_c}{\partial w_{ij}} &= \frac{\partial}{\partial w_{ij}} \sum_{r=1 \dots n_H} w_{rc} \cdot y_r = \frac{\partial}{\partial w_{ij}} \sum_{r=1 \dots n_H} w_{rc} \cdot f(net_r) = \\ &= \frac{\partial}{\partial w_{ij}} (w_{jc} \cdot f(net_j)) = w_{jc} \frac{\partial f(net_j)}{\partial net_j} \cdot \frac{\partial net_j}{\partial w_{ij}} = \\ \text{Definition of } net_j & \quad \text{always zero but when } q=i \\ &= w_{jc} \cdot f'(net_j) \cdot \frac{\partial}{\partial w_{ij}} \sum_{q=1 \dots d} w_{qj} \cdot x_q = w_{jc} \cdot f'(net_j) \cdot x_i \end{aligned}$$

$$\frac{\partial J}{\partial w_{ij}} = - \sum_{c=1 \dots s} \delta_c \cdot w_{jc} \cdot f'(net_j) \cdot x_i = -x_i \cdot f'(net_j) \sum_{c=1 \dots s} w_{jc} \cdot \delta_c$$

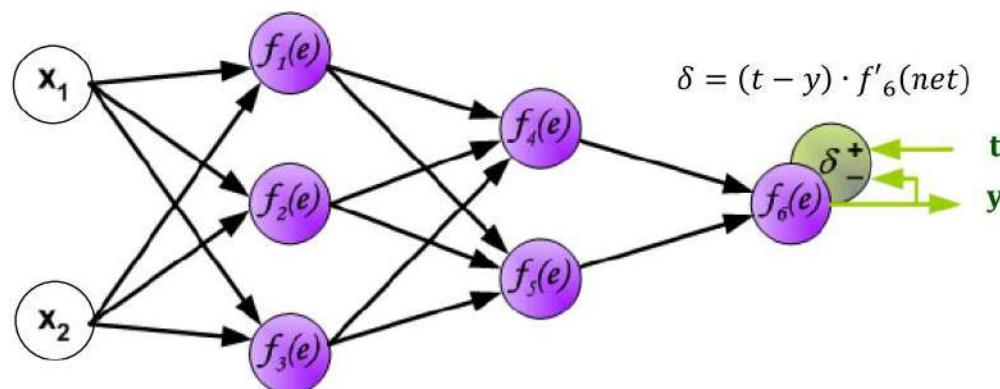
$$\delta_j = f'(net_j) \cdot \sum_{c=1 \dots s} w_{jc} \cdot \delta_c \quad (\text{Eq. 5})$$

$$\frac{\partial J}{\partial w_{ij}} = -\delta_j \cdot x_i \quad (\text{Eq. 6})$$

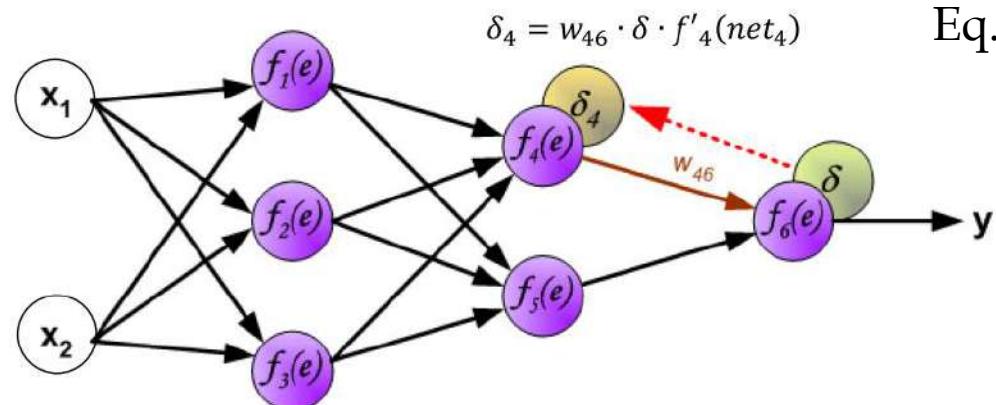
N.B. if biases are used, the weight  $w_{0i}$  is updated by setting  $x_0 = 1$



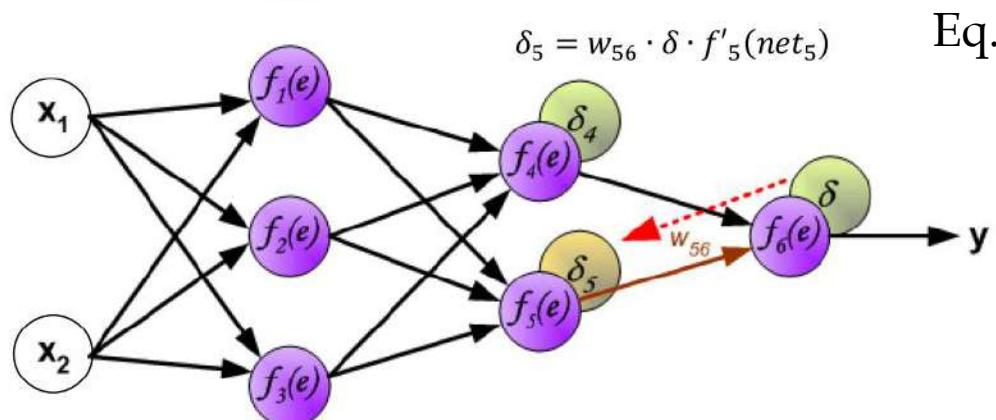
# Example



Eq. 2 pg 130



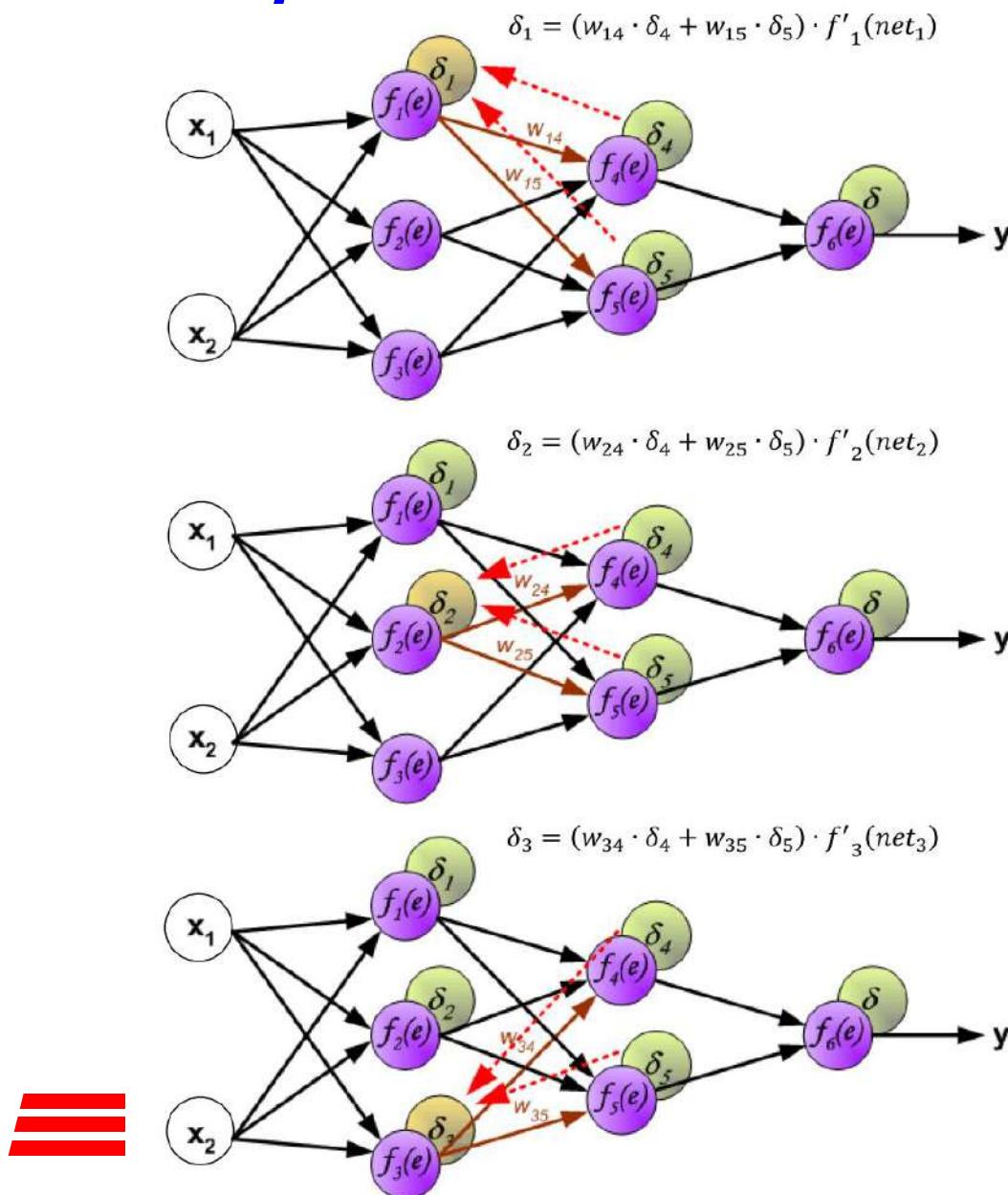
Eq. 5 pg 132



Eq. 5 pg 132



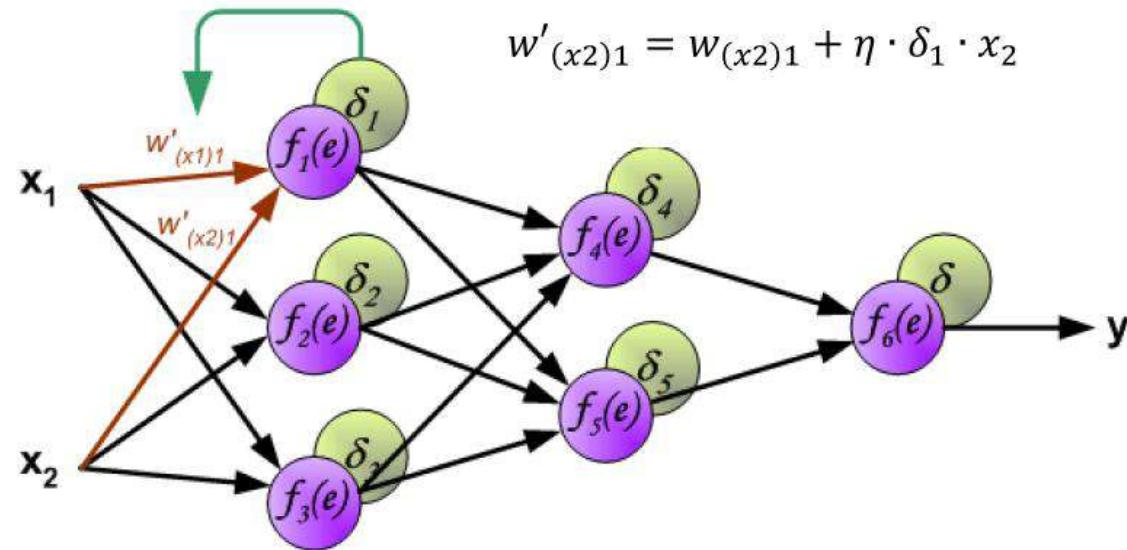
# Example



Based on the same rationale of Eq. 5 pg 132 but in this example we have 2 hidden layers. Each weight is multiplied by the related delta

# Weights update

$$w'_{(x1)1} = w_{(x1)1} + \eta \cdot \delta_1 \cdot x_1$$



... similarly for all other weights



# Stochastic Gradient Descent (SGD)

This is the most used approach to implement backpropagation:  
at each epoch, the n patterns of the training set are randomly ordered and then divided, considering them sequentially, into m groups (called mini-batches) of equal size<sub>mb</sub> : n / m.

the gradient values are (algebraically) accumulated in temporary variables (one for each weight); the update of the weights occurs only when all the patterns of a mini-batch have been processed.

```
|  $n_H, m, \mathbf{w}, \eta, maxEpoch, epoch \leftarrow 0$ 
do  $epoch \leftarrow epoch + 1$ 
    random sort the Training Set (n patterns)
    for each mini-batch B of size n/m
         $grad_{ik} = 0, grad_{ij} = 0$ 
        for each x in B
            forward propagation di x → zk,  $k = 1 \dots s$ 
             $grad_{jk} \leftarrow grad_{jk} + \delta_k \cdot y_j, k = 1 \dots s, j = 1 \dots n_H$ 
             $grad_{ij} \leftarrow grad_{ij} + \delta_j \cdot x_i, j = 1 \dots n_H, i = 1 \dots d$ 
         $w_{jk} = w_{jk} + \eta \cdot grad_{jk}, w_{ij} = w_{ij} + \eta \cdot grad_{ij}$ 
```

Suppose we train an  
MLP consisting of a single  
hidden Layer



# Stochastic Gradient Descent (SGD)

With epoch we mean the presentation (one time) of all the  $n$  patterns of the training set to the network.

With iteration we mean the presentation (one time) of the patterns making up a mini-batch and the consequent update of the weights.

$n / \text{size}_{mb}$  is the number of iterations per epoch. If it is not an integer, at the last iteration, fewer patterns are processed.



# *Output function*

The output depends on the function applied in the output layer, e.g.:

Tanh: the output values are in the range [-1 ... 1], and therefore they are not probabilities;

Sigmoid (standard logistic function): the output values are included in the interval [0 ... 1], but we have no guarantee that the sum on the output neurons is 1 (a fundamental requirement so that they can be interpreted as a probabilistic distribution).

$$\sum_{c=1 \dots s} z_c \neq 1$$

When a neural network is used as a multi-class classifier, the use of the softmax activation function allows you to transform the net values produced by the last level of the network into class probabilities:

The  $net_k$  activation level of the single neurons of the last level is calculated in the usual way, but the following function, for the k-th neuron, is applied:

$$z_k = f(net_k) = \frac{e^{net_k}}{\sum_{c=1 \dots s} e^{net_c}}$$

The values  $z_k$  can be interpreted as probabilities of the classes: they belong to [0 ... 1] and their sum is 1.



# Cross-Entropy loss

This choice of Tanh activation function is not optimal, as the output values do not represent probabilities, and the non-imposition of the sum constraint to 1 makes learning (generally) less effective.

In the previous slides we used the Sum of Squared Error loss function.

For a multi-class classification problem, it is recommended to use Cross-Entropy (also called Multinomial Logistic Loss) as a loss function:

The cross-entropy between two discrete distributions  $p$  and  $q$  (which give the measure of how much  $q$  differs from  $p$ ) is defined by:

$$H(p, q) = - \sum_v p(v) \cdot \log(q(v))$$

The minus sign makes the entropy into a non-negative number (for any base  $b > 1$  logarithms of numbers between 0 and 1 are negative numbers).  $p(v)$  is the ground truth and  $q(v)$  the predicted

When used as a loss function:  $p$  (fixed) is the target vector, while  $q$  is the output vector of the network.

The minimum value of  $H$  (always  $>= 0$ ) occurs when the target vector coincides with the output. Attention, in the formula the logarithm does not exist in 0, but the outputs provided by softmax and sigmoid are never 0 (even if they can tend towards it asymptotically).

For insights and further details on information theory:

<http://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/>



# Cross-Entropy and one-hot targets

When the target vector for  $x$  takes the one-hot form (all 0s except a 1 for the correct class  $g$ ):

$$\mathbf{t} = [0, 0 \dots \textcolor{red}{1} \dots 0]$$

↖ g-th position

then:

$$J(\mathbf{w}, \mathbf{x}) \equiv H(\mathbf{t}, \mathbf{z}) = -\log(z_g)$$

If  $z_g$  is obtained by softmax activation function, then:

$$J(\mathbf{w}, \mathbf{x}) = -\log \left( \frac{e^{net_g}}{\sum_{c=1 \dots s} e^{net_c}} \right) = -net_g + \log \sum_{c=1 \dots s} e^{net_c}$$

↑

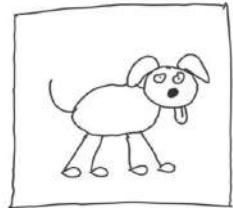
$$\log_b(MN) = \log_b(M) + \log_b(N)$$

$$\log_b \left( \frac{M}{N} \right) = \log_b(M) - \log_b(N)$$

$$\log_b(M^p) = p \log_b(M)$$



# Which class is on the image – dog, cat, or panda?



MODEL

$\overbrace{0.5}^{\text{DOG PROBABILITY}}$   
 $0.3 \rightarrow \text{CAT PROBABILITY}$   
 $\overbrace{0.2}^{\text{PANDA PROBABILITY}}$

TARGET	PREDICTION
1	$\overbrace{0.5}^{\text{ }}$
0	$0.3$
0	$\overbrace{0.2}^{\text{ }}$

$$\begin{aligned}
 \text{Loss For CAT} &= -p(\text{CAT}) \cdot \log q(\text{CAT}) \\
 &= -0 \cdot \log q(\text{CAT}) \\
 &= 0
 \end{aligned}$$

$$\text{Loss For DOG} = -p(\text{DOG}) \cdot \log q(\text{DOG})$$

$$= -1 \cdot \log 0.5$$

$$\begin{aligned}
 \text{Loss For PANDA} &= -p(\text{PANDA}) \cdot \log q(\text{PANDA}) \\
 &= -0 \cdot \log q(\text{PANDA}) \\
 &= 0
 \end{aligned}$$

$$= 0.693\dots$$

$$\begin{aligned}
 \text{Cross-entropy} &= \text{Loss For DOG} + \text{Loss For CAT} + \text{Loss For PANDA} \\
 &= 0.693 + 0 + 0 \\
 &= 0.693
 \end{aligned}$$



# Loss function for classification network

Multi-class classification: use softmax activation function in the final level and cross-entropy as loss function, if the target vector  $t$  takes the one-hot form (default) the simplification of the previous slides is used.

In case of uncertainty about the correct class, it is possible to use “soft target”, with single values between 0 and 1 and sum to 1. In this case we use a complete cross-entropy formula.

Binary classification: it is possible to fall back to the previous case and treat the problem as a multiclass with 2 classes, but it is preferable (often better convergence):

use 1 output neuron that encodes the probability of the first class only (the second probability is the complement of 1).

To this end, the sigmoid activation function is used to obtain value in [0 ... 1] for the first neuron. the normalization to 1 of the probability of the two classes is implicit in the adopted cross-entropy simplification. Note that with 1 neuron  $t, z$  are scalar values in [0 ... 1]:

$$H(t, z) = -(t \log(z) + (1 - t) \log(1 - z))$$

A special case is the so-called multi-label classification where a pattern can belong to several classes. For example, an image that contains both a dog and a cat could be classified as belonging to 2 classes. in this case each output is in [0 ... 1] but the sum constraint to 1 must be relaxed for the outputs. It can be obtained as an extension of the binary classification (above) using sigmoid activations ( $f(z)$  instead of  $z$ , clearly if apply sigmoid depends from the network) and summing  $H(t, z)$  across multiple neurons (i.e. all the classes).

$$\text{Total Loss} = \sum_x \text{Binary Cross-entropy}_x$$



# Loss regularization

To reduce the risk of overfitting of the training set of a neural network with many parameters (weights), regularization techniques can be used. Regularization is very important when the training set is not large compared to the model complexity.

Neural networks whose weights, or part of them, assume small values (close to zero) produce more regular and stable output, often leading to better generalization. This is a form of regression, that constrains/ regularizes or shrinks the coefficient estimates towards zero. In other words, this technique discourages learning a more complex or flexible model, so as to avoid the risk of overfitting.

To push the network to adopt weights of small value, a regularization term can be added to the loss. For example in the case of Cross-Entropy Loss:

$$J_{Tot} = J_{Cross-Entropy} + J_{Reg}$$

In the L2 regularization, the added term corresponds to the sum of the squares of all the weights of the network (notice that the aim is to minimize this term):

$$J_{Reg} = \frac{1}{2} \lambda \sum_i w_i^2$$

In the L1 regularization the absolute value is used:

$$J_{Reg} = \lambda \sum_i |w_i|$$

In both cases the lambda parameter adjusts the strength of the regularization.

L1 can have a sparsifying effect (i.e. bring numerous weights to 0) greater than L2. In fact, when the weights assume values close to zero, the calculation of the square in L2 has the effect of excessively reducing the corrective measures to the weights, making it difficult to zero them.



# Weight decay

Let us consider the L2 regularization.

The gradient of  $J_{\text{tot}}$  with respect to one of the parameters of the network corresponds to the sum of the  $J_{\text{cross-entropy}}$  gradient and the  $J_{\text{Reg}}$  gradient. The latter is:

$$\frac{\partial J_{\text{Reg}}}{\partial w_k} = \frac{\partial}{\partial w_k} \left( \frac{1}{2} \lambda \sum_i w_i^2 \right) = \lambda \cdot w_k$$

Therefore, updating of weights following backpropagation includes an additional term called weight decay which has the effect of pulling them towards 0.

For example, considering the SGD weights update:

$$w_k = w_k - \eta \cdot \frac{\partial J_{\text{Cross-Entropy}}}{\partial w_k}$$

it becomes:

$$w_k = w_k - \eta \left( \frac{\partial J_{\text{Cross-Entropy}}}{\partial w_k} + \lambda \cdot w_k \right)$$

Analogous reasoning for the L1 regularization.

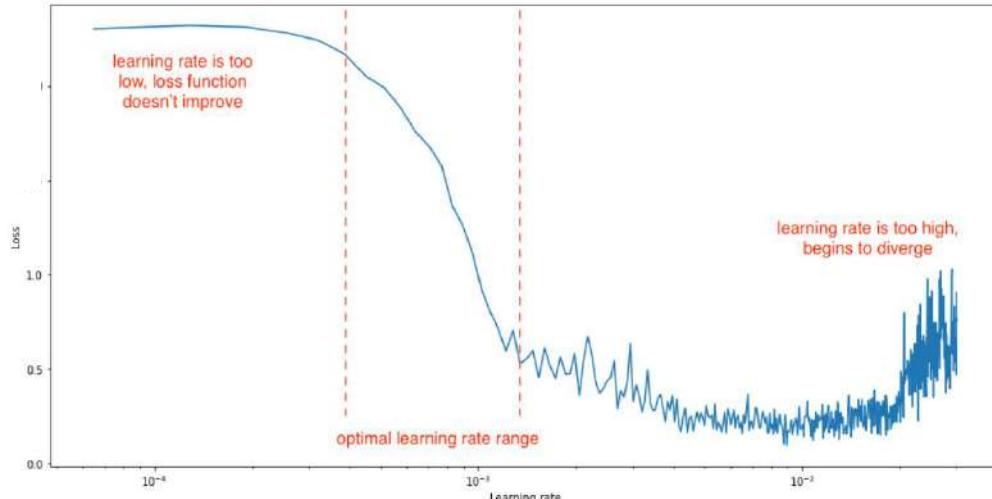
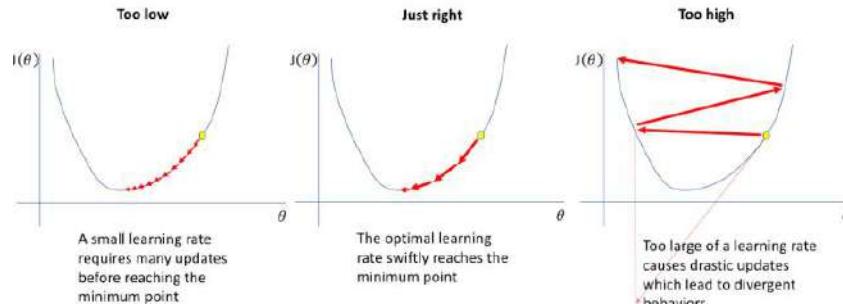


# *Learning Rate*



# Learning rate

Calibrating the learning rate in an optimal way is very important:  
 if too small slow convergence;  
 if too great a swing and / or divergence.



the optimal value changes according to the network architecture of the loss function, etc.

if you start from a known architecture and already applied by others to similar problems, start from the recommended values and try to increase / decrease by monitoring convergence and generalization.

Attention: the fact that the loss decreases without oscillating does not mean that the value is optimal (especially in the tuning of pre-trained networks): monitor accuracy on the validation set.

Adaptive learning rate techniques can help.

You should set the range of your learning rate bounds for this experiment such that you observe **all three** phases, making the optimal range trivial to identify.

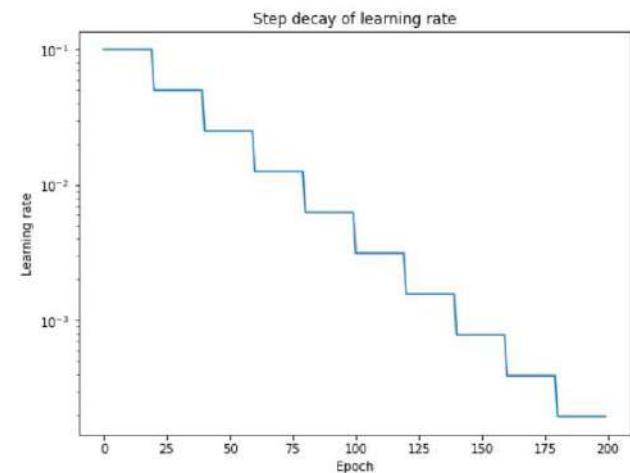


# Learning rate annealing

Another commonly employed technique, known as **learning rate annealing** recommends starting with a relatively high learning rate and then gradually lowering the learning rate during training. The intuition behind this approach is that we'd like to traverse quickly from the initial parameters to a range of "good" parameter values but then we'd like a learning rate small enough that we can explore the "deeper, but narrower parts of the loss function" (from [Karpathy's CS231n notes](#)). If you're having a hard time picturing what I just mentioned, recall that too high of a learning rate can cause the parameter update to "jump over" the ideal minima and subsequent updates will either result in a continued noisy convergence in the general region of the minima, or in more extreme cases may result in divergence from the minima.

The most popular form of learning rate annealing is a *step decay* where the learning rate is reduced by some percentage after a set number of training epochs.

<https://cs231n.github.io/>

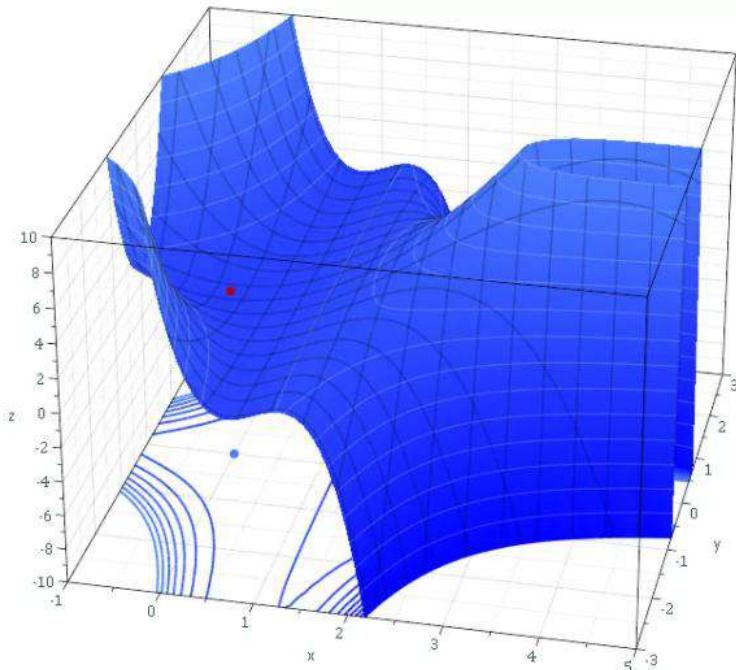


More generally, we can establish that it is useful to define a **learning rate schedule** in which the learning rate is updating during training according to some specified rule.



# Cyclic learning rate

Such a mountainous scenario can be represented by the mathematical plot below:



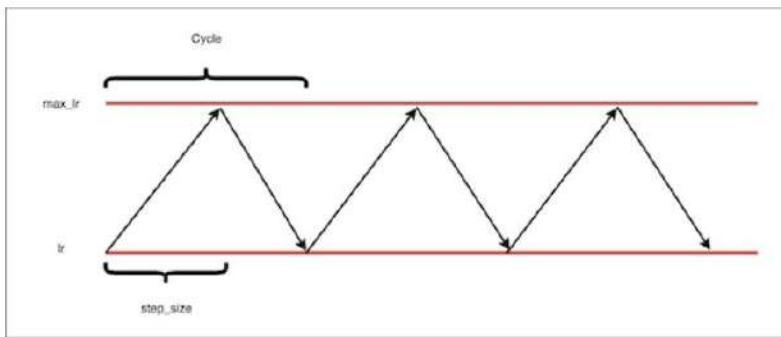
Source: [Sam Derbyshire at Wikipedia CC BY-SA 3.0, Link](#)

However, it must be clear that the *valley* depicted here – i.e., the *red dot* – is only a local valley. As you can see, by walking further, you can descend even further. However, let's now assume that you're in the top left part while descending, and with the aim to arrive at that red dot.



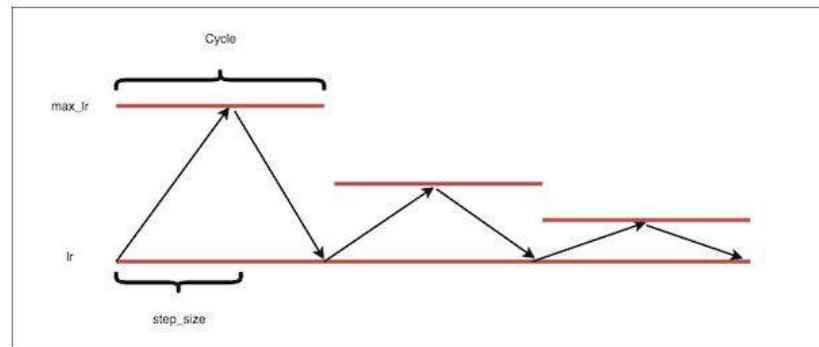
# Cyclic learning rate

In CLR, we vary the LR between a lower and higher threshold. The logic is that periodic higher learning rates within each epoch helps to come out of any saddle points or local minima if it encounters into one. If saddle point happens to be an elaborated plateau, lower learning rates will probably never generate enough gradient to come out of it, resulting in difficulty in minimising the loss.

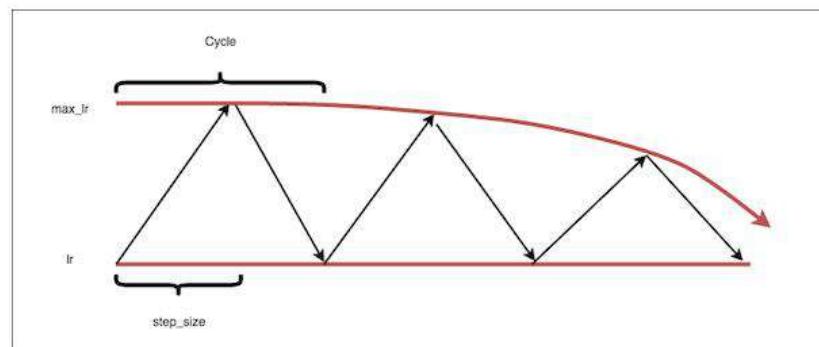


Other than triangular profile used above, Lesley Smith also suggested some other forms of CLR.

**Triangular2** : Here the *max\_lr* is halved after every cycle.



**Exponential Range** : Here *max\_lr* is reduced exponentially with each iteration.



## Setting `base_lr` and `max_lr`

The loss plot will see a decrease in loss as we increase the learning rate, but will start increasing again at a point. Note the LR at which loss starts to decrease, as also the LR when it starts stagnating. These are good points to set as `base_lr` and `max_lr`.

Alternatively, you can note the LR where accuracy peaks, and use that as `max_lr`. Set `base_lr` as 1/3 or 1/4 of this.

# Cyclic learning rate

We can write the general schedule as

$$\eta_t = \eta_{\min} + (\eta_{\max} - \eta_{\min}) (\max(0, 1 - x))$$

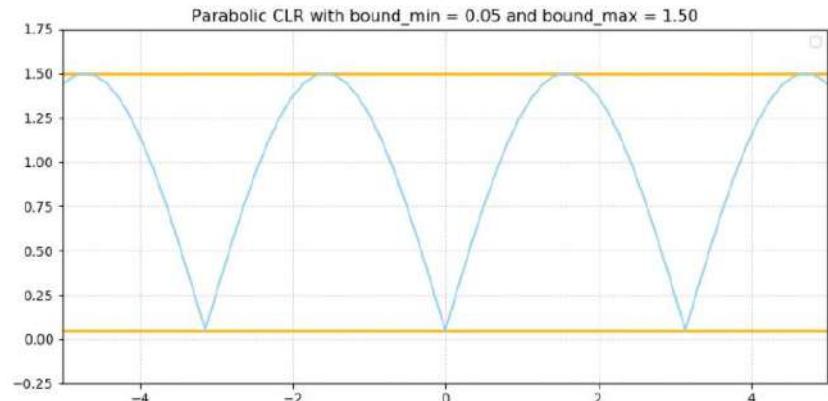
where  $x$  is defined as

$$x = \left| \frac{\text{iterations}}{\text{stepsize}} - 2(\text{cycle}) + 1 \right|$$

and  $\text{cycle}$  can be calculated as

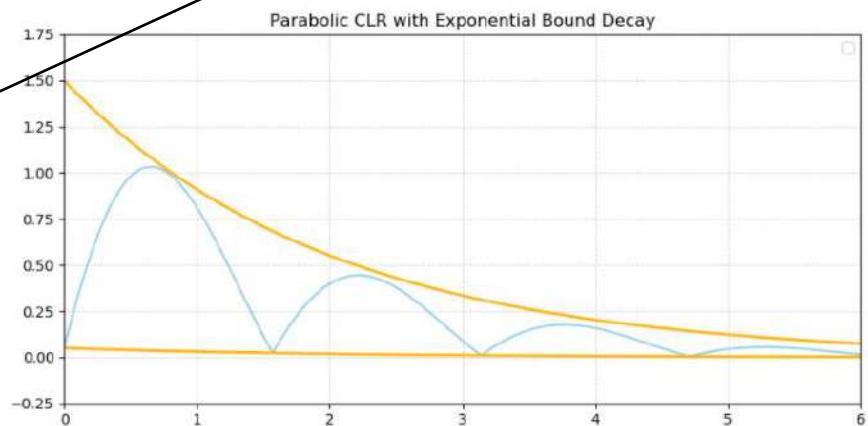
$$\text{cycle} = \text{floor} \left( 1 + \frac{\text{iterations}}{2(\text{stepsize})} \right)$$

where  $\eta_{\min}$  and  $\eta_{\max}$  define the bounds of our learning rate,  $\text{iterations}$  represents the number of completed mini-batches,  $\text{stepsize}$  defines one half of a cycle length. As far as I can gather,  $1 - x$  should always be positive, so it seems the  $\max$  operation is not strictly necessary.



In some cases, it's desirable to let the bounds decay over time (Smith, 2017). This ensures that the learning varies less and less once the epochs pass – that is, presumably, when you reach the global minimum. Below, you'll see an example for parabolic-like CLR with exponential bound decay. Another approach lets the learning rates decay in a triangular fashion, i.e. by cutting them in half after every iteration.

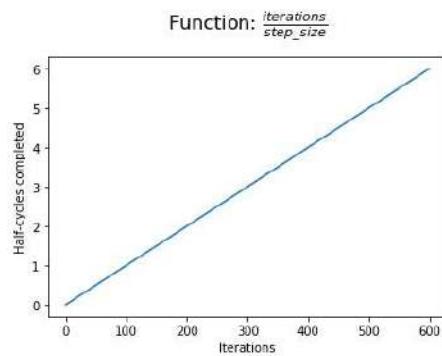
<https://ieeexplore.ieee.org/document/7926641>



# Cyclic learning rate

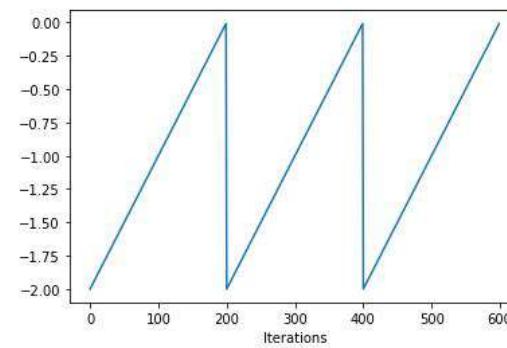
In order to grok how this equation works, let's progressively build it with visualizations. For the visuals below, the triangular update for 3 full cycles are shown with a step size of 100 iterations. Remember, one iteration corresponds with one mini-batch of training.

Foremost, we can establish our "progress" during training in terms of half-cycles that we've completed. We measure our progress in terms of half-cycles and not full cycles so that we can achieve symmetry within a cycle (this should become more clear as you continue reading).



Next, we compare our half-cycle progress to the number of half-cycles which will be completed at the end of the current cycle. At the beginning of a cycle, we have two half-cycles yet to be completed. At the end of a cycle, this value reaches zero.

Function:  $\frac{\text{iterations}}{\text{step\_size}} - 2(\text{cycle})$

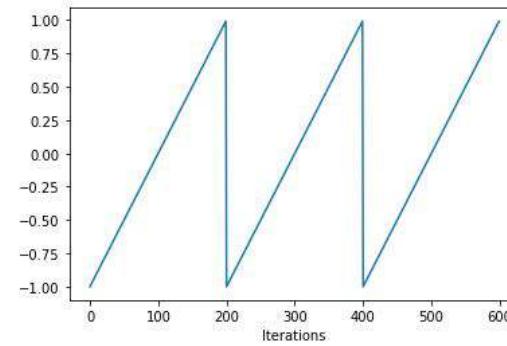


step size=100  
a full cycle is 200 iterations

(cycle) indicates which cycle we are in  
e.g.  
if iteration is [0,199] then cycle=1;  
if iteration is [200, 399] then cycle=2;  
if iteration is [400, 599] then cycle=3;  
e.g.:  
 $400/100-2*3=-2$   
 $500/100-2*3=-1$

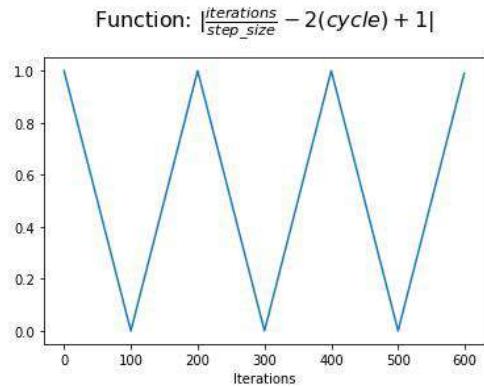
Next, we'll add 1 to this value in order to shift the function to be centered on the y-axis. Now we're showing our progress within a cycle with reference to the half-cycle point.

Function:  $\frac{\text{iterations}}{\text{step\_size}} - 2(\text{cycle}) + 1$

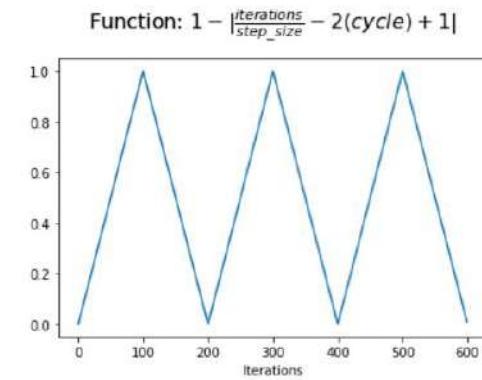


# Cyclic learning rate

At this point, we can take the absolute value to achieve a triangular shape within each cycle. This is the value that we assign to  $x$ .



However, we'd like our learning rate schedule to start at the minimum value, increasing to the maximum value at the middle of a cycle, and then decrease back to the minimum value. We can accomplish this by simply calculating  $1 - x$ .



However, we'd like our learning rate schedule to start at the minimum value, increasing to the maximum value at the middle of a cycle, and then decrease back to the minimum value. We can accomplish this by simply calculating  $1 - x$ .

We now have a value which we can use to modulate the learning rate by adding some fraction of the learning rate range to the minimum learning rate (also referred to as the base learning rate).



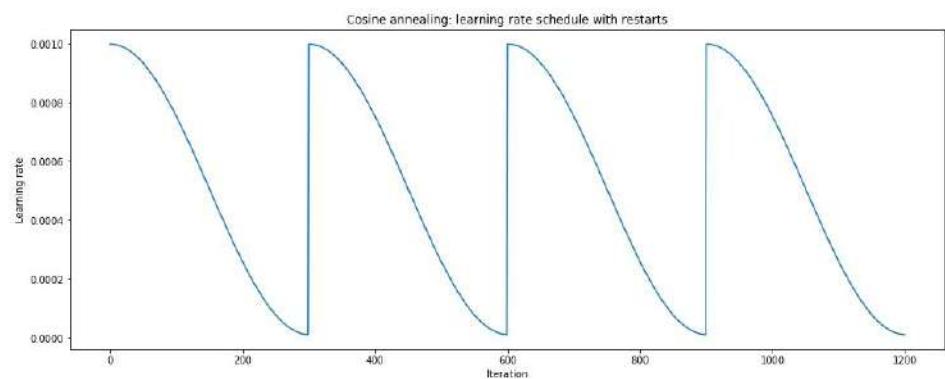
# Cyclic learning rate

Smith writes, the main assumption behind the rationale for a cyclical learning rate (as opposed to one which *only* decreases) is "that **increasing the learning rate might have a short term negative effect and yet achieve a longer term beneficial effect.**" Indeed, his paper includes several examples of a loss function evolution which temporarily deviates to higher losses while ultimately converging to a lower loss when compared with a benchmark fixed learning rate.

To gain intuition on why this short-term effect would yield a long-term positive effect, it's important to understand the desirable characteristics of our converged minimum. Ultimately, we'd like our network to learn from the data in a manner which *generalizes* to unseen data. Further, a network with good generalization properties should be robust in the sense that small changes to the network's parameters don't cause drastic changes to performance. With this in mind, it makes sense that sharp minima lead to poor generalization as small changes to the parameter values may result in a drastically higher loss. By allowing for our learning rate to *increase* at times, we can "jump out" of sharp minima which would temporarily increase our loss but may ultimately lead to convergence on a more desirable minima.

## Stochastic Gradient Descent with Warm Restarts (SGDR)

A similar cyclic approach is known as **stochastic gradient descent with warm restarts** where an aggressive annealing schedule is combined with periodic "restarts" to the original starting learning rate.



We can write this schedule as

$$\eta_t = \eta_{\min}^i + \frac{1}{2} (\eta_{\max}^i - \eta_{\min}^i) \left( 1 + \cos\left(\frac{T_{\text{current}}}{T_i} \pi\right) \right)$$

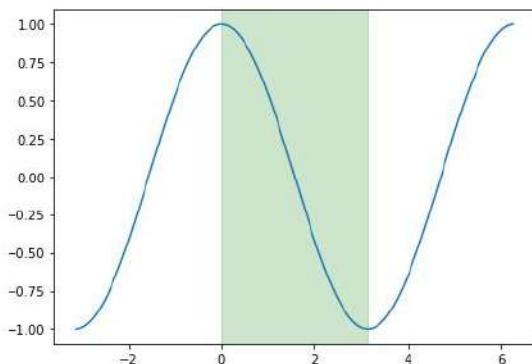
where  $\eta_t$  is the learning rate at timestep  $t$  (incremented each mini batch),  $\eta_{\max}^i$  and  $\eta_{\min}^i$  define the range of desired learning rates,  $T_{\text{current}}$  represents the number of epochs since the last restart (this value is calculated at every iteration and thus can take on fractional values), and  $T_i$  defines the number of epochs in a cycle. Let's try to break this equation down.



# Cyclic learning rate

This annealing schedule relies on the cosine function, which varies between -1 and 1.  $\frac{T_{current}}{T_i}$  is capable of taking on values between 0 and 1, which is the input of our cosine function. The corresponding region of the cosine function is highlighted below in green. By adding 1, our function varies between 0 and 2, which is then scaled by  $\frac{1}{2}$  to now vary between 0 and 1. Thus, we're simply taking the minimum learning rate and adding some fraction of the specified learning rate range ( $\eta_{\max}^i - \eta_{\min}^i$ ). Because this function starts at 1 and decreases to 0, the result is a learning rate which starts at the maximum of the specified range and decays to the minimum value. Once we reach the end of a cycle,  $T_{current}$  resets to 0 and we start back at the maximum learning rate.

$$\cos(0)=1; \cos(\pi)=-1$$



if this component is 2 we have:

$$\eta_t = \eta_{\min}^i + \frac{1}{2} (\eta_{\max}^i - \eta_{\min}^i) \left( 1 + \cos\left(\frac{T_{current}}{T_i} \pi\right) \right)$$

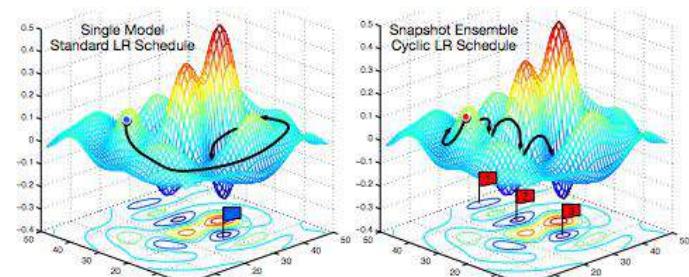
$$\eta_t = \eta_{\min}^i + \frac{1}{2} (\eta_{\max}^i - \eta_{\min}^i)$$

The authors note that this learning rate schedule can further be adapted to:

1. Lengthen the cycle as training progresses.
2. Decay  $\eta_{\max}^i$  and  $\eta_{\min}^i$  after each cycle.

By drastically increasing the learning rate at each restart, we can essentially exit a local minima and continue exploring the loss landscape.

*Neat idea: By snapshotting the weights at the end of each cycle, researchers were able to build an ensemble of models at the cost of training a single model. This is because the network "settles" on various local optima from cycle to cycle, as shown in the figure below.*



**Left:** Illustration of SGD optimization with a typical learning rate schedule. The model converges to a minimum at the end of training. **Right:** Illustration of Snapshot Ensembling. The model undergoes several learning rate annealing cycles, converging to and escaping from multiple local minima. We take a snapshot at each minimum for test-time ensembling.

# Momentum

SGD with mini-batch can cause a “zigzag gradient” which slows down convergence and makes it less stable.

You can think of Gradient Descent as a ball rolling down on a valley. We want it to sit in the deepest place of the mountains, however, it is easy to see that things can go wrong. Attracted by the force of gravity, the ball tries to move to the lowest point.

In physics, the moment gives a motion without sudden oscillations and changes of direction.

In SGD, to avoid oscillations, the physical behavior can be emulated: the last update of each parameter is saved, and the new update is calculated as a linear combination of the previous update (which gives stability) and the current gradient (which corrects the direction):

Typical value of the momentum factor (also named decay rate)=0.9

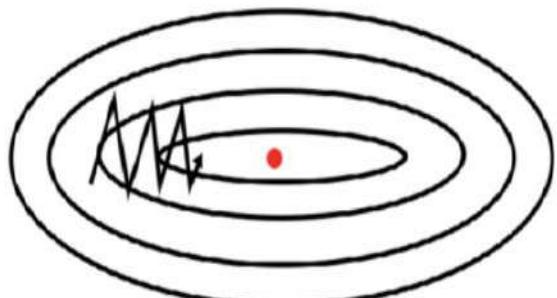
$$\Delta w_{ij} = (\eta * \frac{\partial E}{\partial w_{ij}})$$

↑  
weight increment  
↑ learning rate  
↑ weight gradient

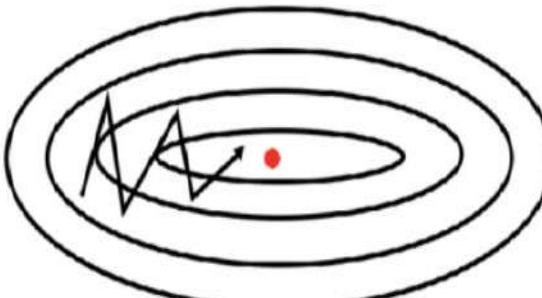
$$\Delta w_{ij} = (\eta * \frac{\partial E}{\partial w_{ij}}) + (\gamma * \Delta w_{ij}^{t-1})$$

↑  
momentum factor  
↑ weight increment, previous iteration

SGD without momentum



SGD with momentum



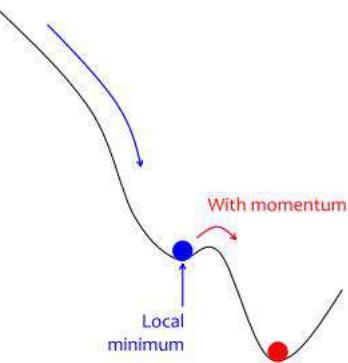
During training the update direction tends to resist change when momentum is added to the update scheme. When the neural net approaches a shallow local minimum it's like applying brakes but not sufficient to instantly affect the update direction and magnitude. Hence the neural nets trained this way will overshoot past smaller local minima points and only stop in a deeper global minimum.

Thus momentum in neural nets helps them get out of local minima points so that a more important global minimum is found. Too much of momentum may create issues as well as systems that are not stable may create oscillations that grow in magnitude, in such cases one needs to add decay terms and so on. It's just physics applied to neural net training or numerical optimizations.

Useful video

[https://cdn-images-1.medium.com/max/1000/1\\*Nb39bHHUWGxqgsr2WcLGQ.gif](https://cdn-images-1.medium.com/max/1000/1*Nb39bHHUWGxqgsr2WcLGQ.gif)

# Momentum/One-cycle learning rate



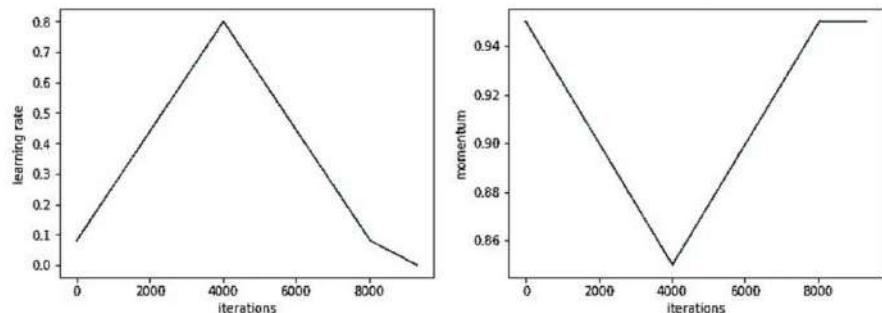
Another interesting idea is to add a *momentum* to the learning rate. This means that once the weights start moving in a particular direction in the weight space, they tend to continue moving in that direction. The benefit of this idea can be illustrated by imagining a ball rolling down a hill.

With the standard learning rate pattern, gradually declining along the iteration count, the ball may get stuck in a little trough on the slope. But if the ball has enough momentum, it will be able to jump out of the trough and continue rolling down the hill.

## One-cycle learning rate

fast.ai no longer recommends cosine annealing because it is no longer the most performant general-purpose learning rate scheduler. These days, that honor belongs to the one-cycle learning rate scheduler.

The one-cycle learning rate scheduler was introduced in the 2017 paper "Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates". The paper uses the following learning rate policy (which is applied over *both* learning rate and momentum):

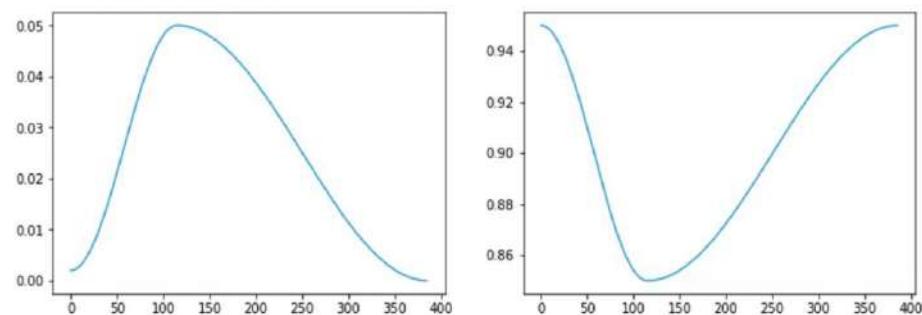


Optimally, learning rate and momentum should be set to a value which just causes the network to begin to diverge at its peak. The remainder of the training regimen consists of warm-up, cool-down, and fine-tuning periods. Note that, during the fine-tuning period, the learning rate drops to 1/10th of its initial value.

Momentum is counterproductive when the learning rate is very high, which is why momentum is annealed in the opposite of the way in which the learning rate is annealed in the optimizer.

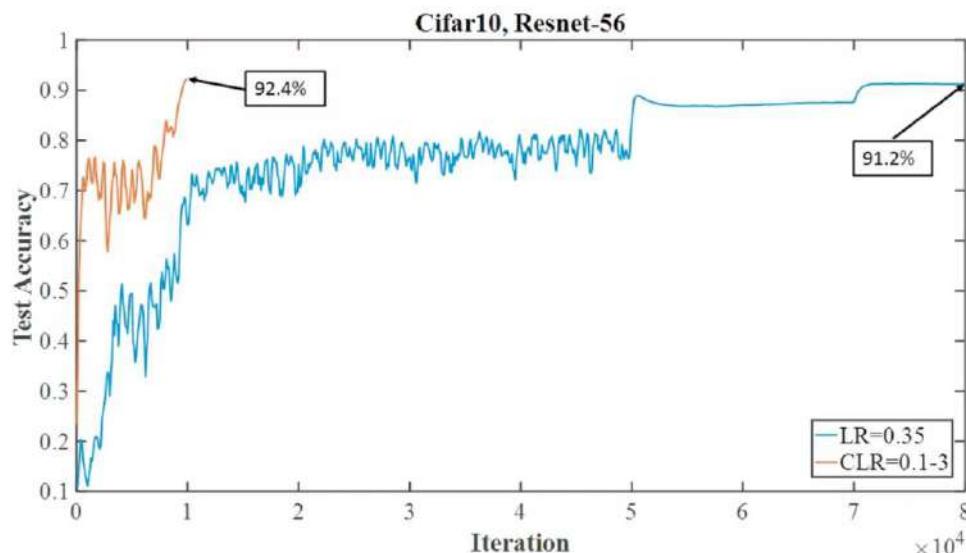
The one-cycle learning rate scheduler uses more or less the same mechanism that the cosine annealed warm restarts learning rate scheduler uses, just in a different form factor.

In their implementation, `fastai` tweaks this a little bit, again switching from linear to cosine annealing:



# Superconvergence

Smith, the 1cycle paper author, refers to as **superconvergence**. For example, the paper shows the following behavior on CIFAR10:



Though you shouldn't count on results this compelling in practice, superconvergence has indeed been demonstrated on a broad range of datasets and problem domains.

Is it all that simple?

<https://arxiv.org/pdf/1703.04933.pdf>

It has been observed empirically that minima found by standard deep learning algorithms that generalize well tend to be flatter than found minima that did not generalize well. However, when following several definitions of flatness, we have shown that the conclusion that flat minima should generalize better than sharp ones cannot be applied as is without further context. Previously used definitions fail to account for the complex geometry of some commonly used deep architectures. In particular, the non-identifiability of the model induced by symmetries, allows one to alter the flatness of a minimum without affecting the function it represents. Additionally the whole geometry of the error surface with respect to the parameters can be changed arbitrarily under different parametrizations. Our work indicates that more care is needed to define flatness to avoid degeneracies of the geometry of the model under study. Also such a concept can not be divorced from the particular parametrization of the model or input space.



# ||||| Adaptive learning rate

In addition to Momentum, there are other rules for updating the weights that can accelerate the convergence in the gradient descent , e.g .:

- Nesterov Accelerate Gradient (NAG)
- Adaptive Gradient (Adagrad)
- Adadelta
- RMSProp
- Adam

Most of the above approaches adapt the global learning rate to each specific weight. Adam is considered to be the state of the art, but doesn't always deliver better results than the more classic SGD with Momentum.

fastai recommends 1cycle plus SGD over Adam because, subject to some tuning (getting the maximum learning rate correct is particularly important), it trains models with roughly equal or marginally worse performance in a fraction of the time. This is due to a phenomenon that Leslie Smith, the 1cycle paper author, refers to as **superconvergence**.

Despite fast.ai advocacy, at present, most practitioners still use the Adam optimizer as their default.

For example, if you browse recent starter kernels for Kaggle competition [kaggle.com](https://www.kaggle.com) you'll see that the use of Adam predominates. As I explained above, in large part because Adam is pretty much parameter-free, it's much more robust to model changes than OneCycleLR is. This makes it much easier to develop with, as it's one fewer set of hyperparameters that you have to optimize. These benefits seem to have proven to be more important than the computational benefit of OneCycleLR , hyperconvergence, in practice.

However, once you're in the later optimization stage of a medium-sized model training project, experimenting with moving off of Adam and onto OneCycleLR is well worth doing. Just imagine how much easier the lives of your data engineers will be if your model can achieve 98% of the performance in 25% of the time!

Hyperconvergence is an extremely attractive property to have, if you can spend the time required to tune it. This is why lists like [Faster Deep Learning Training with PyTorch – a 2021 Guide](#) have OneCycleLR as one of their top PyTorch training tricks.



# Adam

Diederik P. Kingma and Jimmy Lei Ba. Adam : A method for stochastic optimization. 2014. arXiv:1412.6980v9

Note that the name Adam is not an acronym, in fact, the authors – Diederik P. Kingma of OpenAI and Jimmy Lei Ba of University of Toronto – state in the paper, which was first presented as a conference paper at ICLR 2015 and titled Adam: A method for Stochastic Optimization, that the name is derived from adaptive moment estimation.

Adam is an optimizer introduced in [1] that computes adaptive learning rates for each parameter combining the ideas of momentum and adaptive gradient. Its update rule is based on the value of the gradient at the current step, and on the exponential moving averages of the gradient and its square. To be more precise, Adam defines the moving averages  $m_t$  (the first moment) and  $u_t$  (the second moment) as:

$$m_t = \rho_1 m_{t-1} + (1 - \rho_1) g_t$$

$$u_t = \rho_2 u_{t-1} + (1 - \rho_2) g_t^2$$

where  $g_t$  is the gradient at time  $t$ , the square on  $g_t$  stands for the component-wise square,  $\rho_1$  and  $\rho_2$  are hyperparameters representing the exponential decay rate for the first moment and the second moment estimates (usually set to 0.9 and 0.999, respectively) and the moments are initialized to 0:  $m_0 = u_0 = 0$ . In order to take into account the fact that the value of moving averages will be very small due to their initialization to zero (especially in the first steps), the authors of Adam define a bias-corrected version of the moving averages:

$$\hat{m}_t = \frac{m_t}{(1 - \rho_1^t)}$$



# Adam

$$\hat{u}_t = \frac{u_t}{(1 - \rho_2^t)}$$

The final update for each  $\theta_t$  parameter of the network is:

$$\theta_t = \theta_{t-1} - \lambda \frac{\hat{m}_t}{\sqrt{\hat{u}_t} + \epsilon}$$

where  $\lambda$  is the learning rate,  $\epsilon$  is a very small positive number to prevent any division by zero (usually set to  $10^{-8}$ ) and all the operations are meant to be component-wise.

Notice that, while  $g_t$  might have positive or negative components,  $g_t^2$  has only positive components. Hence, if the gradient changes sign often, the value of  $\hat{m}_t$  might be much lower than  $\sqrt{\hat{u}_t}$ . This means that in this case the step size is very small.

$$m_t = \rho_1 m_{t-1} + (1 - \rho_1) g_t$$

if the sign changes it means that sometimes we increase the value of  $m$ , and other times we decrease the value of  $m$ . Therefore it means that the value of the first moment does not change, instead, in the case of the second moment, the value always increases. So, sooner or later, in this situation the second moment value will be much larger than the first moment value

The intuition behind the Adam is that we don't want to roll so fast just because we can jump over the minimum, we want to decrease the velocity a little bit for a careful search.

Adam tends to converge faster, while SGD often converges to more optimal solutions. SGD's high variance disadvantages gets rectified by Adam (as advantage for Adam).

Adam tends to converge faster, while SGD often converges to more optimal solutions.

To summarize, Adam definitely converges rapidly to "sharp minima" whereas SGD is computationally heavy, converges to a "flat minima" but performs well on the test data.

Adam is useful when the computation time of SGD is unfeasible (or very expensive). Moreover, it is usually applied in transformer (see pg 384)

[http://www.sanjivk.com/heavyTailedNoise\\_Attention\\_NIPS20.pdf](http://www.sanjivk.com/heavyTailedNoise_Attention_NIPS20.pdf)

# diffGrad <https://arxiv.org/abs/1909.11015>

diffGrad is a method that takes into account the difference of the gradient in order to set the learning rate. According to the observation that when gradient changes begin to reduce during training, this is often indicative of the presence of a global minima, diffGrad applies an adaptive adjusting driven by the difference between the present and the immediate past to lock parameters into global minima. Therefore, the step size is larger for faster gradient changing and lower for lower gradient changing parameters. In order to define the update function, they define the absolute difference of two consecutive steps of the gradient as:

$$\Delta g_t = |g_{t-1} - g_t|$$

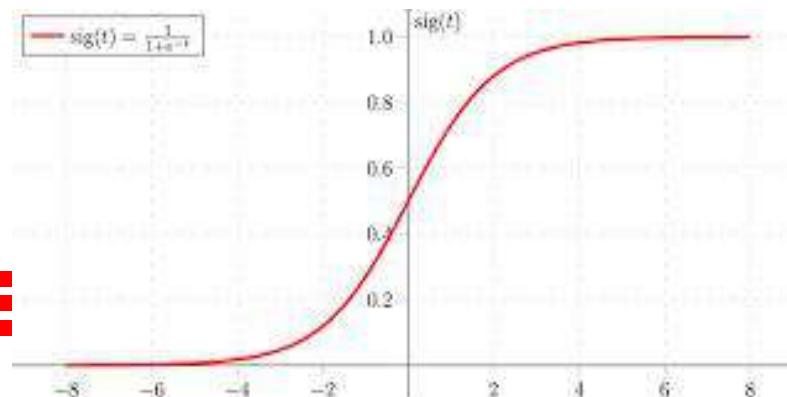
The final update for each  $\theta_t$  parameter of the network is as in equation

$$\xi_t = \text{Sig}(\Delta g_t) \text{ Sigmoid of } \Delta g_t$$

$$\text{Sig}(x) = \frac{1}{1 + e^{-|x|}}$$

$$\theta_{t+1} = \theta_t - \lambda \cdot \xi_t \frac{\hat{m}_t}{\sqrt{\hat{u}_t} + \epsilon}$$

sig(t) =  $\frac{1}{1+e^{-t}}$



notice that  $\Delta g_t$  is always higher or equal to 0, therefore  $\text{Sig}(\Delta g_t) \in [0.5, 1]$

# Convergence problems: check list

Here are some things to check when the convergence on the training set does not start (the network seems to be blocked):

- Have got the inputs been normalized (min-max, standard scaling)?
- In the case of regression, has the output been normalized to [0 1]?
- Have the weights been randomized into reasonable ranges? otherwise we could be in the saturation zone from which it is not possible to escape.
- Are the patterns of the different classes randomly mixed within the minibatches?
- The labels are in the right format (scattered integers, one hot vectors, float by regression, etc.)
- Is the loss function correct and adequate for the type of provided labels (pay attention to the difference between sparse labels and one-hot vectors)?
- Is the learning rate adequate?



# *Overfitting*

---

# */*

# *DropOut*



# Overfitting Revisited

Faced with more features than examples, linear models tend to overfit. But given more examples than features, we can generally count on linear models not to overfit. Unfortunately, the reliability with which linear models generalize comes at a cost. Naively applied, linear models do not take into account interactions among features. For every feature, a linear model must assign either a positive or a negative weight, ignoring context.

In traditional texts, this fundamental tension between generalizability and flexibility is described as the *bias-variance tradeoff*. Linear models have high bias: they can only represent a small class of functions. However, these models have low variance: they give similar results across different random samples of the data.

Deep neural networks inhabit the opposite end of the bias-variance spectrum. Unlike linear models, neural networks are not confined to looking at each feature individually. They can learn interactions among groups of features. For example, they might infer that “Nigeria” and “Western Union” appearing together in an email indicates spam but that separately they do not.

Even when we have far more examples than features, deep neural networks are capable of overfitting. In 2017, a group of researchers demonstrated the extreme flexibility of neural networks by training deep nets on randomly-labeled images. Despite the absence of any true pattern linking the inputs to the outputs, they found that the neural network optimized by stochastic gradient descent could label every image in the training set perfectly. Consider what this means. If the labels are assigned uniformly at random and there are 10 classes, then no classifier can do better than 10% accuracy on holdout data. The generalization gap here is a whopping 90%. If our models are so expressive that they can overfit this badly, then when should we expect them not to overfit?

# Overfitting Revisited & Robustness

The mathematical foundations for the puzzling generalization properties of deep networks remain open research questions

## Robustness through Perturbations

Let us think briefly about what we expect from a good predictive model. We want it to perform well on unseen data. Classical generalization theory suggests that to close the gap between train and test performance, we should aim for a simple model. Simplicity can come in the form of a small number of dimensions.

Another useful notion of simplicity is smoothness, i.e., that the function should not be sensitive to small changes to its inputs. For instance, when we classify images, we would expect that adding some random noise to the pixels should be mostly harmless.

In 1995, Christopher Bishop formalized this idea when he proved that training with input noise is equivalent to Tikhonov regularization (Bishop, 1995). This work drew a clear mathematical connection between the requirement that a function be smooth (and thus simple), and the requirement that it be resilient to perturbations in the input.

(Bishop, 1995) <http://people.sabanciuniv.edu/berrin/cs512/lectures/Book-Bishop-Neural%20Networks%20for%20Pattern%20Recognition.pdf>



# Dropout

Then, in 2014, Srivastava et al. (Srivastava et al., 2014) developed a clever idea for how to apply Bishop's idea to the internal layers of a network, too. Namely, they proposed to inject noise into each layer of the network before calculating the subsequent layer during training. They realized that when training a deep network with many layers, injecting noise enforces smoothness just on the input-output mapping.

Their idea, called *dropout*, involves injecting noise while computing each internal layer during forward propagation, and it has become a standard technique for training neural networks. The method is called *dropout* because we literally *drop out* some neurons during training. Throughout training, on each iteration, standard dropout consists of zeroing out some fraction of the nodes in each layer before calculating the subsequent layer.

To be clear, we are imposing our own narrative with the link to Bishop. The original paper on dropout offers intuition through a surprising analogy to sexual reproduction. The authors argue that neural network overfitting is characterized by a state in which each layer relies on a specific pattern of activations in the previous layer, calling this condition *co-adaptation*. Dropout, they claim, breaks up co-adaptation just as sexual reproduction is argued to break up co-adapted genes.

The key challenge then is how to inject this noise. One idea is to inject the noise in an *unbiased* manner so that the expected value of each layer—while fixing the others—equals to the value it would have taken absent noise.

In Bishop's work, he added Gaussian noise to the inputs to a linear model. At each training iteration, he added noise sampled from a distribution with mean zero  $\epsilon \sim \mathcal{N}(0, \sigma^2)$  to the input  $\mathbf{x}$ , yielding a perturbed point  $\mathbf{x}' = \mathbf{x} + \epsilon$ . In expectation,  $E[\mathbf{x}'] = \mathbf{x}$ .

In standard dropout regularization, one debiases each layer by normalizing by the fraction of nodes that were retained (not dropped out). In other words, with *dropout probability*  $p$ , each intermediate activation  $h$  is replaced by a random variable  $h'$  as follows:

$$h' = \begin{cases} 0 & \text{with probability } p \\ \frac{h}{1-p} & \text{otherwise} \end{cases}$$

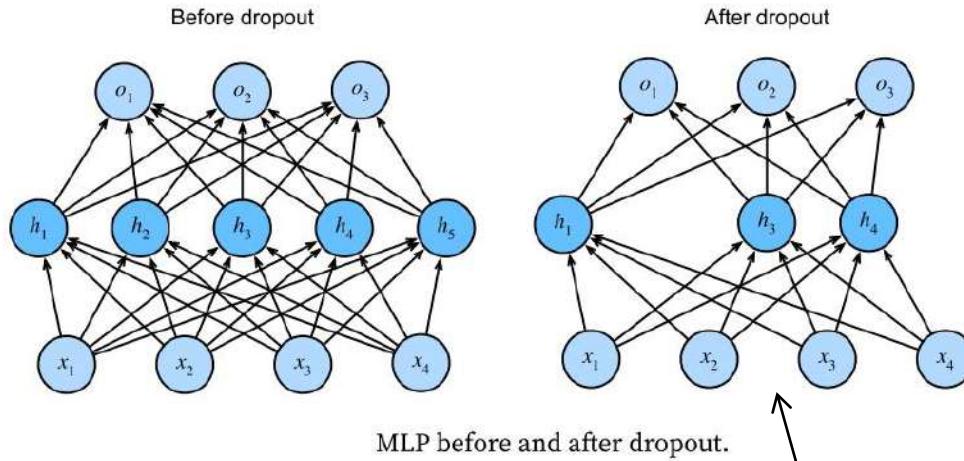
By design, the expectation remains unchanged, i.e.,  $E[h'] = h$ .

(Srivastava, 2014)

<https://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf>



# Dropout



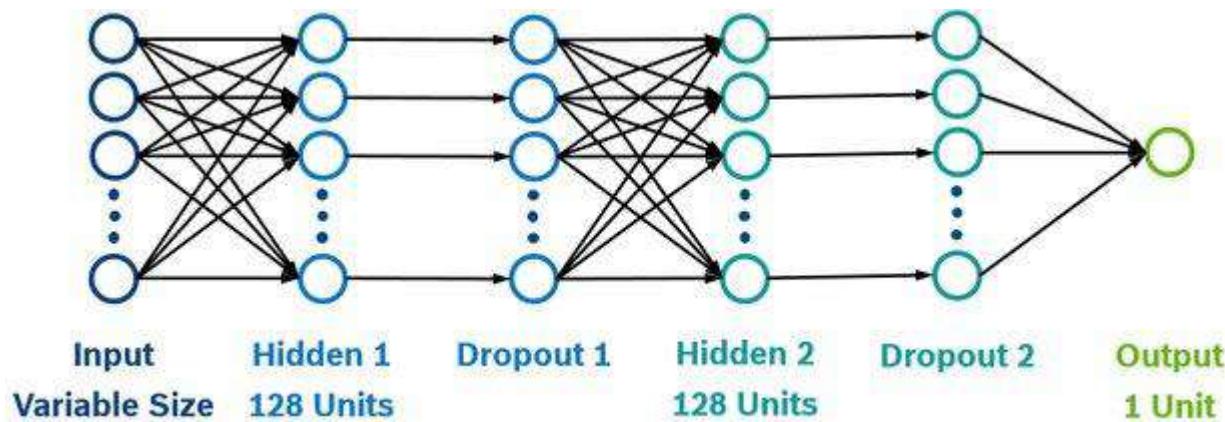
Typically, we disable dropout at test time. Given a trained model and a new example, we do not drop out any nodes and thus do not need to normalize. However, there are some exceptions: some researchers use dropout at test time as a heuristic for estimating the *uncertainty* of neural network predictions: if the predictions agree across many different dropout masks, then we might say that the network is more confident.

When we apply dropout to a hidden layer, zeroing out each hidden unit with probability  $p$ , the result can be viewed as a network containing only a subset of the original neurons. In Fig. [red dot],  $h_2$  and  $h_5$  are removed. Consequently, the calculation of the outputs no longer depends on  $h_2$  or  $h_5$  and their respective gradient also vanishes when performing backpropagation. In this way, the calculation of the output layer cannot be overly dependent on any one element of  $h_1, \dots, h_5$ .



# Dropout

- Dropout is a technique where randomly selected neurons are ignored during training. They are “dropped-out” randomly. This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass.
- As a neural network learns, neuron weights settle into their context within the network. Weights of neurons are tuned for specific features providing some specialization. Neighboring neurons become to rely on this specialization, which if taken too far can result in a fragile model too specialized to the training data. This reliant on context for a neuron during training is referred to complex co-adaptations.
- You can imagine that if neurons are randomly dropped out of the network during training, that other neurons will have to step in and handle the representation required to make predictions for the missing neurons. This is believed to result in multiple independent internal representations being learned by the network.
- The effect is that the network becomes less sensitive to the specific weights of neurons. This in turn results in a network that is capable of better generalization and is less likely to overfit the training data.



# *Deep convolutional neural networks*

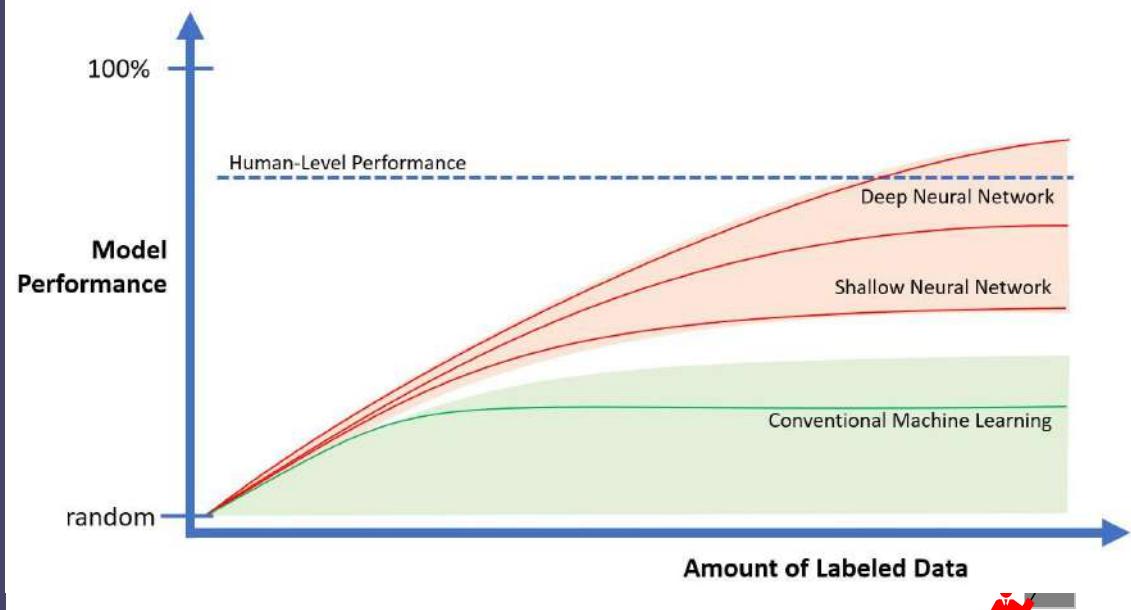
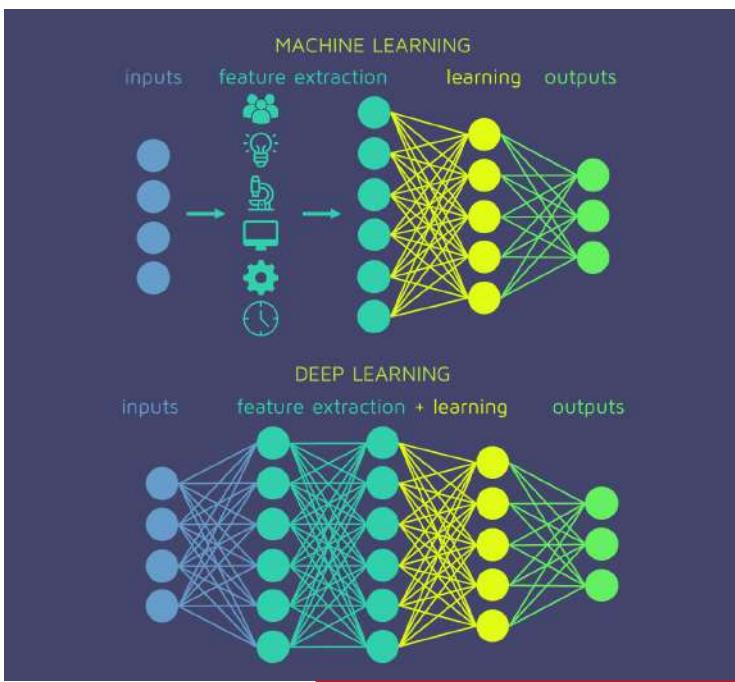


# Why deep networks?

<https://www.width.ai/post/advantages-of-deep-learning>

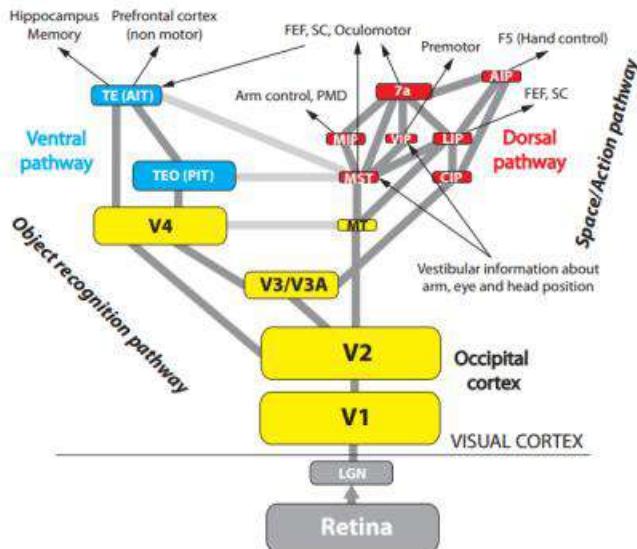
The term DNN (Deep Neural Network) denotes "deep" networks made up of many levels: at least two hidden layers.

The implications of universal approximation theorem and the difficulty of training networks with many layers have long led to focus on networks with only one hidden layer.



# Deep learning

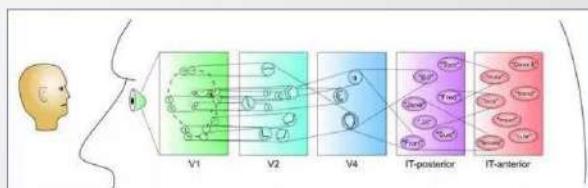
Our visual system operates on a hierarchy of levels (deep):



[Kruger et al. 2013]



In 1968, D. H. Hubel and T. N. Wiesel demonstrated that **mammals** visually **perceive the world** around them using a **layered architecture** of neurons in the brain.

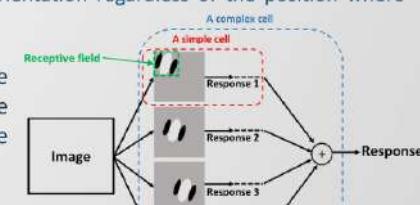


The **structure** of the visual **cortex** is in **layers**. As information is passed from our eyes to the brain, **higher and higher order representation** are formed.

Within the visual cortex, complex functional responses generated by "**complex cells**" are constructed by **combining** more simplistic responses from "**simple cells**".

- Simple cells respond to edges with a specific orientation in a particular position (called **receptive field**).
- Complex cells respond to edges with a specific orientation regardless of the position where they are located (obtaining **spatial invariance**).

**Spatial invariance** is obtained by "**summing**" the contribution of **simple cells** responding to the same orientation but with **different** receptive fields.



# Deep learning

<https://www.biorxiv.org/content/early/2018/09/05/407007>

interesting work where it is tried to express the similarity between modern deep networks and the human brain:

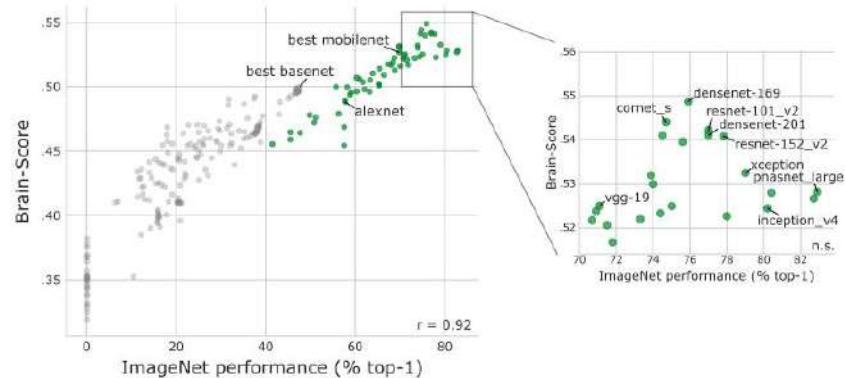
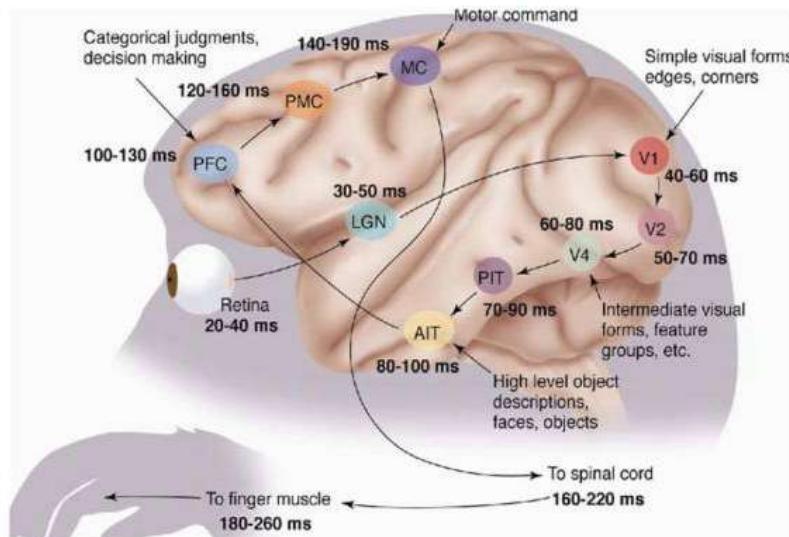


Figure 1: Overview of the Brain-Score. We compare neural networks using two classes of metrics: neural metrics compare the internal activations to regions of the macaque ventral stream, and behavioral metrics compare the similarity in outputs. Brain-Score is correlated with ImageNet performance for small, randomly combined models (gray dots) but becomes weak for current state-of-the-art models (green dots) at  $\geq 70\%$  top-1 performance.



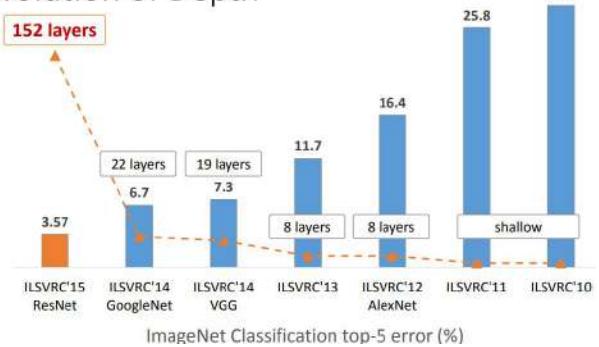
[Simon Thorpe 1996]

The most commonly used DNNs today consist of a number of levels between 7 and 150.

There are "only" about ten levels between the retina and the actuator muscles (otherwise we would be too slow to react to stimuli).

## ImageNet Challenge

### Revolution of Depth



\* Human-level performance: 5.1%

"The commonly used ImageNet-2012 dataset contains 1000 categories.

There are 1,281,167 training images"

So we can train huge networks; however, it's not just adding layers, it just changes the topology of the network

# Complexity

Depth (number of levels) is only one of the complexity factors. Number of neurons, connections and weights also characterize the complexity of a DNN. The greater the number of weights (i.e. parameters to be learned), the greater the complexity of the training. At the same time, a large number of neurons (and connections) makes forward and back propagation more expensive, as the number (G-Ops) of operations required increases.

Generative Pre-trained Transformer 3 (GPT-3) is an autoregressive language model that uses deep learning to produce human-like text. GPT-3 has  $1.75e^{11}$  (175 Billion) parameters.

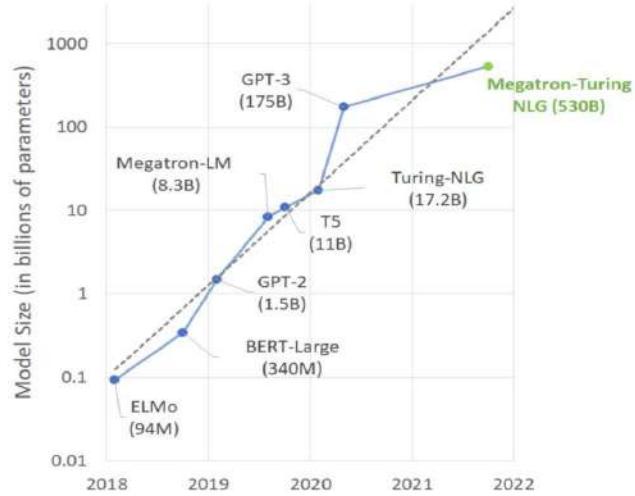
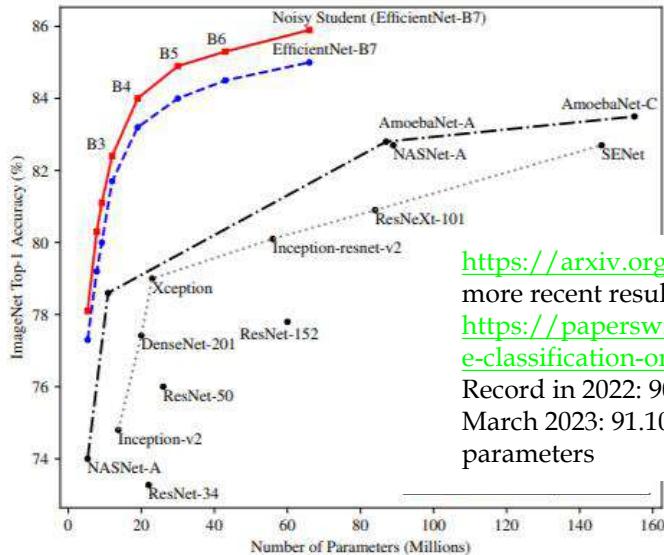


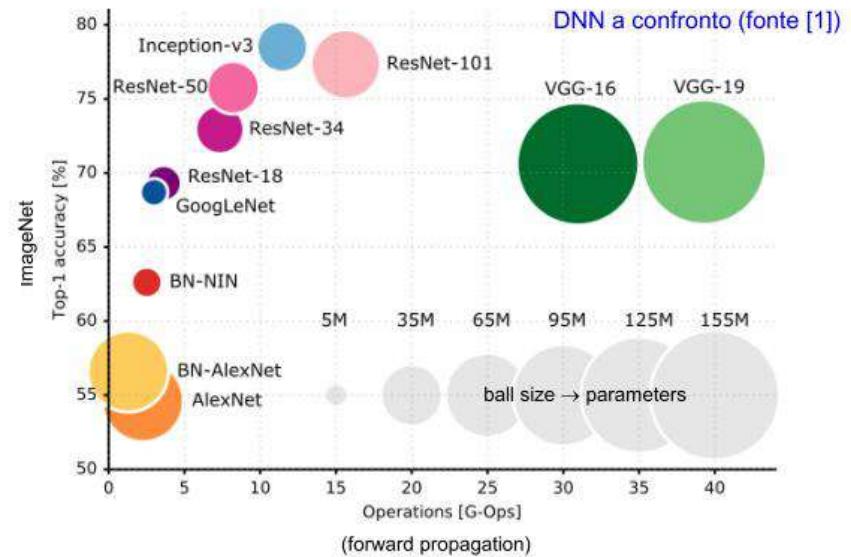
Figure 1: Trend of sizes of state-of-the-art NLP models with time.

AlexNet: 8 levels, 650K neurons and 60M parameters  
 VGG-16: 16 levels, 15M neurons and 140M parameters  
 human cortex:  $21 \times 10^9$  neurons and  $1.5 \times 10^{14}$  synapses

[1] Canziani et al. 2016, *An Analysis of Deep Neural Network Models for Practical Applications*



<https://arxiv.org/pdf/1911.04252.pdf>  
 more recent results:  
<https://paperswithcode.com/sota/image-classification-on-imagenet>  
 Record in 2022: 90.88%  
 March 2023: 91.10% using 2440 M parameters



# *Different topologies*

The most well known feedforward models for classification (or regression) with (mainly) supervised training are:

CNN - Convolutional Neural Network (or ConvNet)

FC DNN - Fully Connected DNN (MLP with at least two hidden layers)

Moreover, we could have:

Unsupervised training for "generative" models trained to reconstruct the input, e.g. useful for pre-training other models and for producing salient features.

Recurring networks (used for sequences, speech recognition, sentiment analysis natural language processing, ...).

Reinforcement learning (to learn behaviors).

<https://medium.com/botsupply/il-machine-learning-%C3%A8-divertente-parte-5-5e9083caf8f3>



# CNN

CNN already achieved good performance (but not state of the art performance) in 1998 in some small datasets (e.g. character recognition, low resolution object recognition), but we have to wait until 2012 (AlexNet) for a radical change of pace. AlexNet does not introduce significant innovations compared to the 1998 LeCun CNNs, but some conditions have changed in the meantime :

BigData - Availability of large labeled datasets (eg imageNet: millions of images, tens of thousands of classes); high computational power.

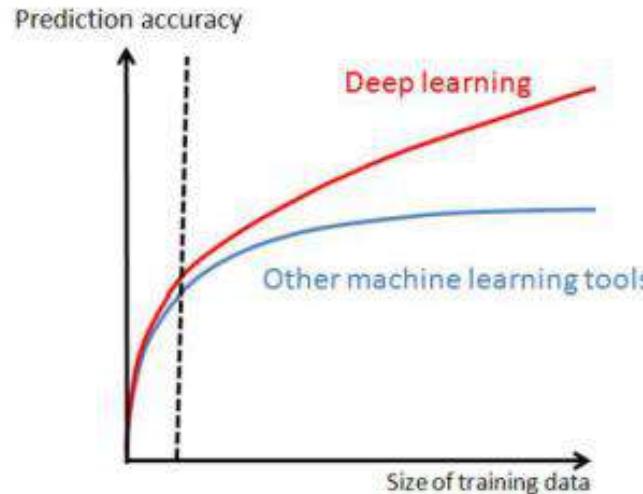
The superiority of deep learning techniques over other approaches is more clear when large amounts of training data are available.

What is even more interesting, is that DL can learn from specific, complex concepts from data without knowing any labels! This is related to the idea of a ‘grandmother cell’, proposed in the 1960 and quickly dismissed by psychologists at the time.



The grandmother cell is a hypothetical neuron that represents a complex but specific concept or object.

Suppose that the concept/object is the person’s grandmother. The neuron will activate upon seeing Grandma, hearing her name or talking about her. Curiously, people have also named this neuron the ‘Jennifer Aniston Neuron’.



And even more curiously, a massive experiment with DL led to the same discovery! One neuron (the grandmother neuron) learned to pick cats (yes, cats!) after seeing 10 million *unlabelled* videos.

In 2012, a research team led by Andrew Ng and Jeff Dean, fed 10 million random YouTube videos to the Google Brain Simulator built on 16,000 computers with one billion connections. Google Brain developed that peculiar neuron which fired only for videos featuring cats even though there were no labels (cats/no cats) on the training videos.

# CNN

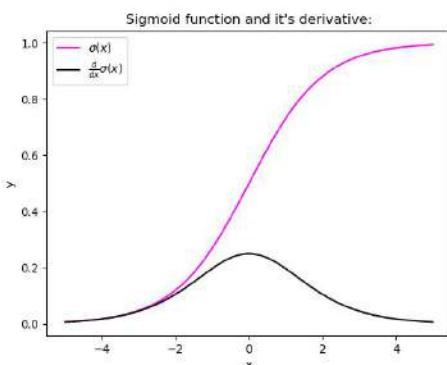
GPU computing: the training of complex models (deep and with many weights and connections) requires high computational powers; but the classification time, after training, is very low, even hundreds of images per second on powerful GPU  
The availability of GPUs with thousands of cores and GB of internal memory has allowed to drastically reduce training times: from months to days. Another problem is huge RAM occupation: hundreds of GB with large datasets

## Vanishing -

As the backpropagation algorithm advances downwards(or backward) from the output layer towards the input layer, the gradients often get smaller and smaller and approach zero which eventually leaves the weights of the initial or lower layers nearly unchanged. As a result, the gradient descent never converges to the optimum. This is known as the vanishing gradients problem.

## Exploding -

On the contrary, in some cases, the gradients keep on getting larger and larger as the backpropagation algorithm progresses. This, in turn, causes very large weight updates and causes the gradient descent to diverge. This is known as the exploding gradients problem.



Observing the graph of the Sigmoid function, we can see that for larger inputs (negative or positive), it saturates at 0 or 1 with a derivative very close to zero. Thus, when the backpropagation algorithm chips in, it virtually has no gradients to propagate backward in the network, and whatever little residual gradients exist keeps on diluting as the algorithm progresses down through the top layers. So, this leaves nothing for the lower layers.

In the exploding gradient, gradient value becomes very large and this leads to too much variation in updated weights (undesired).

$$\sigma(z_2)(1 - \sigma(z_2)) * w_2 = \sigma'(z_2) * w_2$$

The above partial derivative value can be greater than 1 if  $w_2$  is large (let  $w_2=100$ , then this value will become 25).

The sigmoid activation can have exploding gradient if initialized weights are very large. So be careful while initializing the weights.



# Exploding/Vanishing gradient

Consider a deep network with  $L$  layers, input  $\mathbf{x}$  and output  $\mathbf{o}$ . With each layer  $l$  defined by a transformation  $f_l$  parameterized by weights  $\mathbf{W}^{(l)}$ , whose hidden variable is  $\mathbf{h}^{(l)}$  (let  $\mathbf{h}^{(0)} = \mathbf{x}$ ), our network can be expressed as:

$$\mathbf{h}^{(l)} = f_l(\mathbf{h}^{(l-1)}) \text{ and thus } \mathbf{o} = f_L \circ \dots \circ f_1(\mathbf{x}).$$

$$\begin{array}{c} \xrightarrow{\text{Element wise Product}} \\ a \circ b = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \end{bmatrix}_{(n \times 1)} \circ \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \end{bmatrix}_{(n \times 1)} = \begin{bmatrix} a_1 b_1 \\ a_2 b_2 \\ a_3 b_3 \\ a_4 b_4 \\ a_5 b_5 \\ a_6 b_6 \end{bmatrix}_{(n \times 1)} \end{array}$$

If all the hidden variables and the input are vectors, we can write the gradient of  $\mathbf{o}$  with respect to any set of parameters  $\mathbf{W}^{(l)}$  as follows:

$$\partial_{\mathbf{W}^{(l)}} \mathbf{o} = \underbrace{\partial_{\mathbf{h}^{(L-1)}} \mathbf{h}^{(L)}}_{\mathbf{M}^{(L)} \stackrel{\text{def}}{=} \dots} \cdot \dots \cdot \underbrace{\partial_{\mathbf{h}^{(l)}} \mathbf{h}^{(l+1)}}_{\mathbf{M}^{(l+1)} \stackrel{\text{def}}{=}} \underbrace{\partial_{\mathbf{W}^{(l)}} \mathbf{h}^{(l)}}_{\mathbf{v}^{(l)} \stackrel{\text{def}}{=}}.$$

In other words, this gradient is the product of  $L - l$  matrices  $\mathbf{M}^{(L)} \cdot \dots \cdot \mathbf{M}^{(l+1)}$  and the gradient vector  $\mathbf{v}^{(l)}$ . Thus we are susceptible to the same problems of numerical underflow that often crop up when multiplying together too many probabilities.

The risks posed by unstable gradients go beyond numerical representation. Gradients of unpredictable magnitude also threaten the stability of our optimization algorithms. We may be facing parameter updates that are either (i) excessively large, destroying our model (the *exploding gradient* problem); or (ii) excessively small (the *vanishing gradient* problem), rendering learning impossible as parameters hardly move on each update.

<https://towardsdatascience.com/driving-the-backpropagation-equations-from-scratch-part-1-343b300c585a>



## *Vanishing gradient - Activation function*

One frequent culprit causing the vanishing gradient problem is the choice of the activation function  $\sigma$  that is appended following each layer's linear operations.

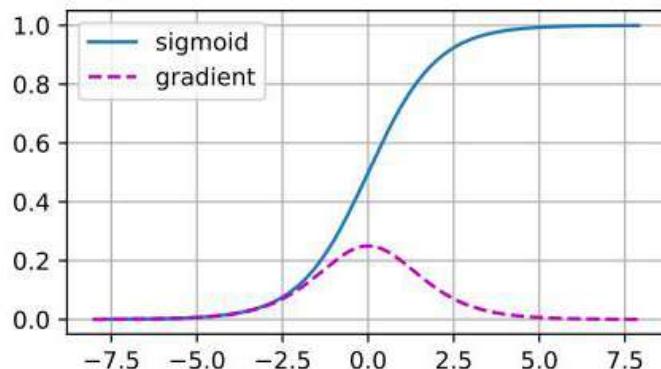
Since early artificial neural networks were inspired by biological neural networks, the idea of neurons that fire either *fully or not at all* (like biological neurons) seemed appealing. Let us take a closer look at the sigmoid to see why it can cause vanishing gradients.

```
%matplotlib inline
from mxnet import autograd, np, npx
from d2l import mxnet as d2l

npx.set_np()

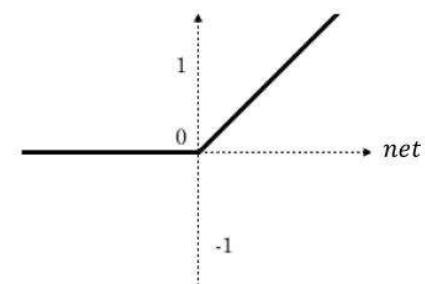
x = np.arange(-8.0, 8.0, 0.1)
x.attach_grad()
with autograd.record():
    y = npx.sigmoid(x)
y.backward()

d2l.plot(x, [y, x.grad], legend=['sigmoid', 'gradient'], figsize=(4.5, 2.5))
```



ReLU

$$f(\text{net}) = \max(0, \text{net})$$



As you can see, the sigmoid's gradient vanishes both when its inputs are large and when they are small. Moreover, when backpropagating through many layers, unless we are in the Goldilocks zone, where the inputs to many of the sigmoids are close to zero, the gradients of the overall product may vanish. When our network boasts many layers, unless we are careful, the gradient will likely be cut off at some layer. Indeed, this problem used to plague deep network training. Consequently, ReLUs, which are more stable (but less neurally plausible), have emerged as the default choice for practitioners.

# Parameter initialization

One way of addressing—or at least mitigating—the issues raised above is through careful initialization. Additional care during optimization and suitable regularization can further enhance stability.

## Xavier Initialization

Let us look at the scale distribution of an output (e.g., a hidden variable)  $o_i$  for some fully-connected layer *without nonlinearities*. With  $n_{\text{in}}$  inputs  $x_j$  and their associated weights  $w_{ij}$  for this layer, an output is given by

$$o_i = \sum_{j=1}^{n_{\text{in}}} w_{ij} x_j.$$

The weights  $w_{ij}$  are all drawn independently from the same distribution. Furthermore, let us assume that this distribution has zero mean and variance  $\sigma^2$ . Note that this does not mean that the distribution has to be Gaussian, just that the mean and variance need to exist. For now, let us assume that the inputs to the layer  $x_j$  also have zero mean and variance  $\gamma^2$  and that they are independent of  $w_{ij}$  and independent of each other. In this case, we can compute the mean and variance of  $o_i$  as follows:

$$\begin{aligned} E[o_i] &= \sum_{j=1}^{n_{\text{in}}} E[w_{ij} x_j] \\ &= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}] E[x_j] \\ &= 0, \end{aligned}$$

$$\begin{aligned} \text{Var}[o_i] &= E[o_i^2] - (E[o_i])^2 \\ &= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}^2 x_j^2] - 0 \\ &= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}^2] E[x_j^2] \\ &= n_{\text{in}} \sigma^2 \gamma^2. \end{aligned}$$

If X and Y are independent, then one can show that  $E[XY] = E[X]E[Y]$ .

Expected values can also be used to compute the variance, by means of the computational formula for the variance:

$$\Rightarrow \text{Var}(X) = E[X^2] - (E[X])^2.$$

[https://en.wikipedia.org/wiki/Expected\\_value](https://en.wikipedia.org/wiki/Expected_value)

since the  $E(x_j) = 0 \rightarrow \text{Var}(x_j) = E(x_j^2) - 0^2$



# Parameter initialization

One way to keep the variance fixed is to set  $n_{\text{in}}\sigma^2 = 1$ . Now consider backpropagation. There we face a similar problem, albeit with gradients being propagated from the layers closer to the output. Using the same reasoning as for forward propagation, we see that the gradients' variance can blow up unless  $n_{\text{out}}\sigma^2 = 1$ , where  $n_{\text{out}}$  is the number of outputs of this layer. This leaves us in a dilemma: we cannot possibly satisfy both conditions simultaneously. Instead, we simply try to

satisfy:

(Glorot & Bengio, 2010)  
<https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>

$$\frac{1}{2}(n_{\text{in}} + n_{\text{out}})\sigma^2 = 1 \text{ or equivalently } \sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}.$$

This is the reasoning underlying the now-standard and practically beneficial *Xavier initialization*, named after the first author of its creators (Glorot & Bengio, 2010). Typically, the Xavier initialization samples weights from a Gaussian distribution with zero mean and variance  $\sigma^2 = \frac{2}{n_{\text{in}} + n_{\text{out}}}$ . We can also adapt Xavier's intuition to choose the variance when sampling weights from a uniform distribution. Note that the uniform distribution  $U(-a, a)$  has variance  $\frac{a^2}{3}$ . Plugging  $\frac{a^2}{3}$  into our condition on  $\sigma^2$  yields the suggestion to initialize according to

In Uniform distribution

[https://en.wikipedia.org/wiki/Continuous\\_uniform\\_distribution](https://en.wikipedia.org/wiki/Continuous_uniform_distribution)

variance is

$$(1/12)*(a-a)^2=(1/12)*(2a)^2=(1/12)*4a^2=(1/3)*a^2$$

$$U\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right).$$

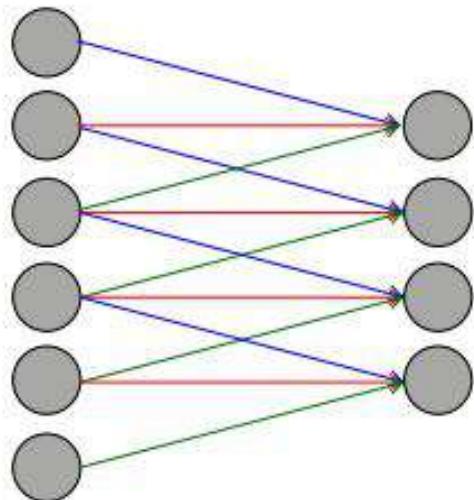
$$(a^2/3)=(2/(n_{\text{in}}+n_{\text{out}}))$$

Though the assumption for nonexistence of nonlinearities in the above mathematical reasoning can be easily violated in neural networks, the Xavier initialization method turns out to work well in practice.



# CNN

Convolutional Neural Networks (CNN) introduced by LeCun since 1998. The main difference compared to MLP is the local processing: neurons are only locally connected to neurons of the previous level. In this way, each neuron performs local processing. Significant reduction in the number of connections: weights are shared in groups. Different neurons of the same level perform the same type of processing on different portions of the input. Strong reduction in the number of weights.

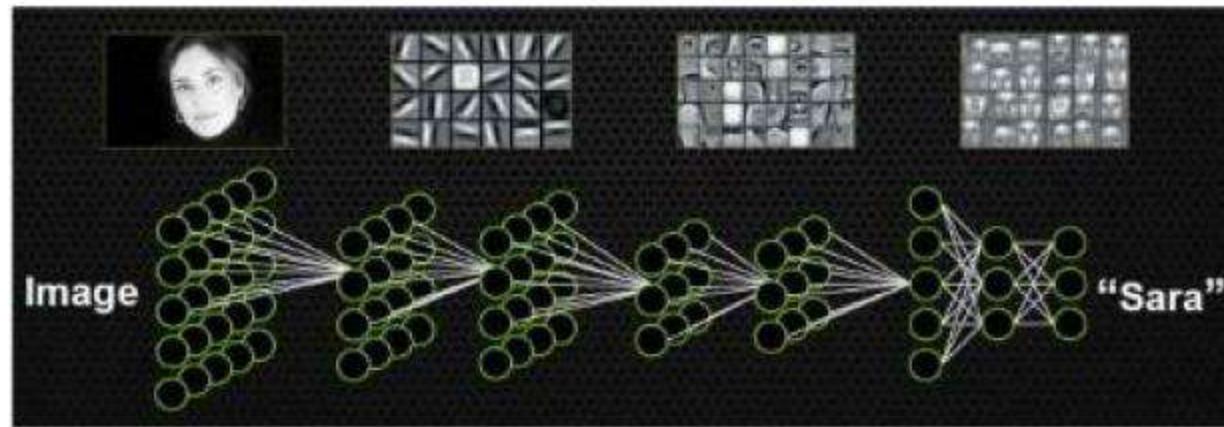


Example: each of the 4 neurons on the right is connected only to 3 neurons of the previous level. Weights are shared (same color same weight). In total 12 connections and 3 weights against the 24 connections + 24 weights of an equivalent portion of MLP.



# CNN with images as input

Architecture: a CNN is made up of a hierarchy of levels. The input level is directly connected to the image pixels, the last levels are generally fully-connected and operate as an MLP classifier, while in the intermediate levels local connections and shared weights are used.



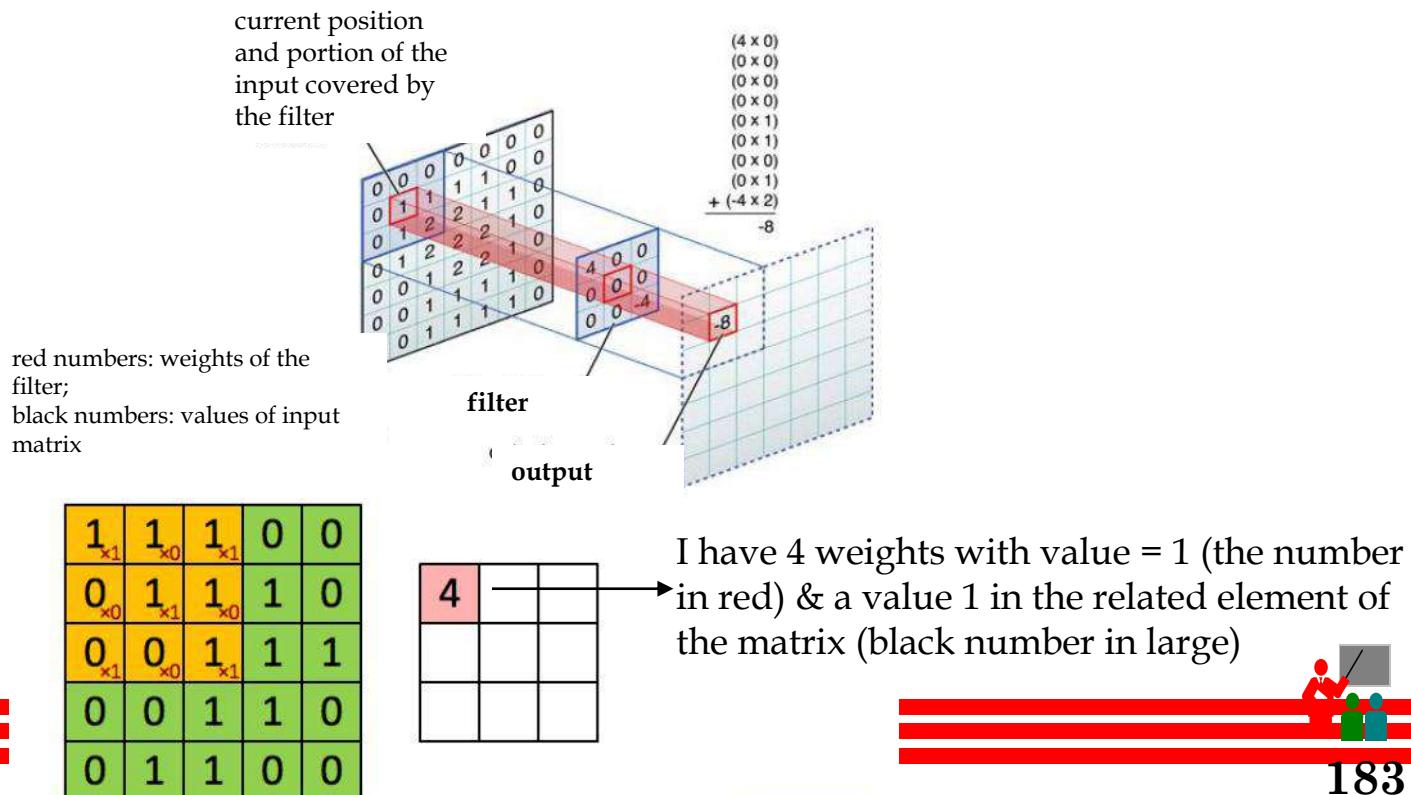
The receptive field of neurons increases as they move up the hierarchy. Local and shared connections mean that neurons process different portions of the image in the same way. This is a desired behavior, as different regions of the visual field contain the same type of information (edges, portions of objects, etc.).



# Convolution

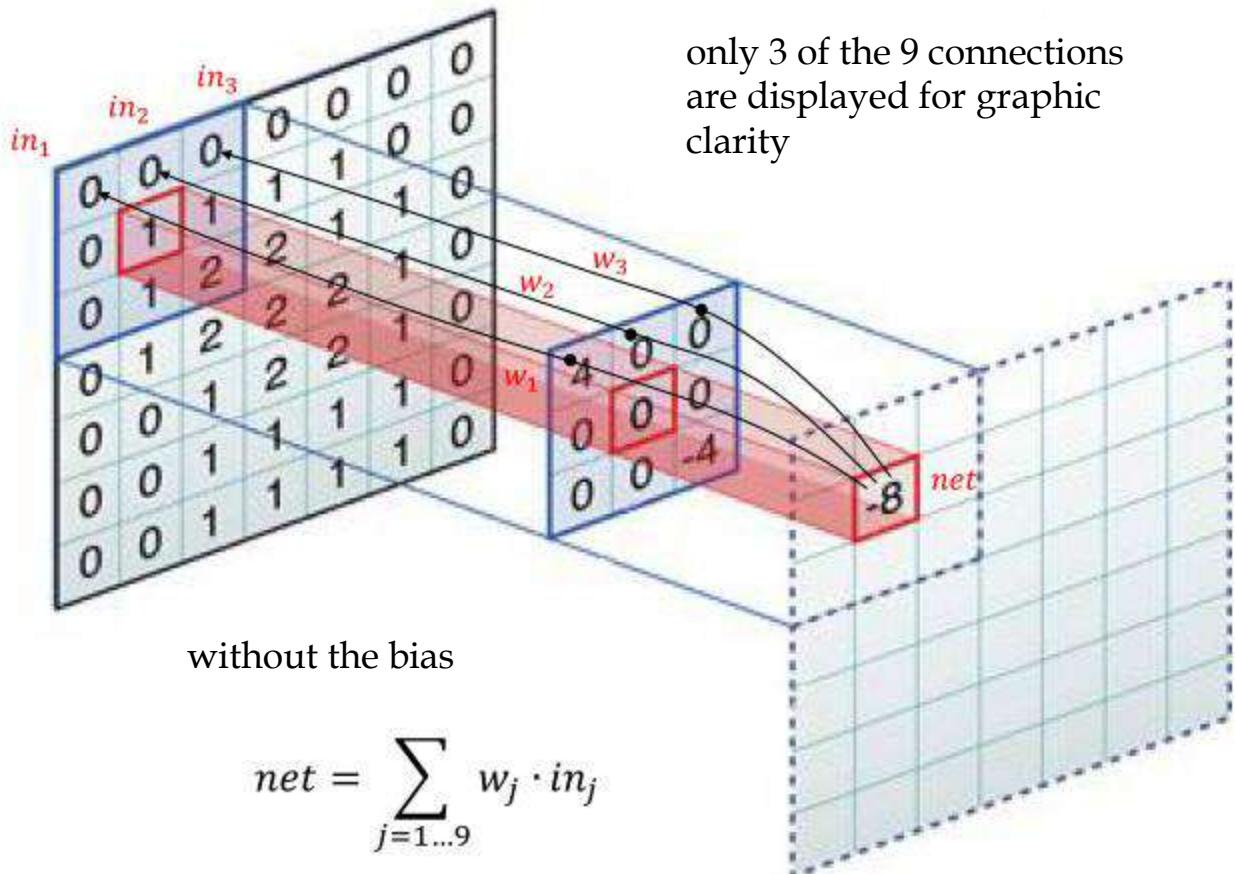
Convolution is one of the most important image processing operations through which digital filters are applied.

A digital filter (a small 2D mask of weights) is scrolled over the different input positions; an output value is generated for each position by running the scalar product between the mask and the portion of the input covered (both treated as vectors).



# Convolution

We consider the pixels as neurons and the input and output images as layers of a network. Given a 3x3 filter, if we connect a neuron to the 9 neurons it "covers" in the previous level, and use the filter weights as weights of the connections  $w$ , we notice that a classic neuron actually performs a convolution.



# Multiple Input channels

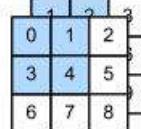
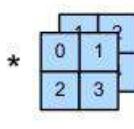
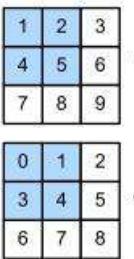
When we add channels into the mix, our inputs and hidden representations both become three-dimensional tensors. For example, each RGB input image has shape  $3 \times h \times w$ . We refer to this axis, with a size of 3, as the *channel dimension*. In this section, we will take a deeper look at convolution kernels with multiple input and multiple output channels.

## Multiple Input Channels

When the input data contain multiple channels, we need to construct a convolution kernel with the same number of input channels as the input data, so that it can perform cross-correlation with the input data. Assuming that the number of channels for the input data is  $c_i$ , the number of input channels of the convolution kernel also needs to be  $c_i$ . If our convolution kernel's window shape is  $k_h \times k_w$ , then when  $c_i = 1$ , we can think of our convolution kernel as just a two-dimensional tensor of shape  $k_h \times k_w$ .

However, when  $c_i > 1$ , we need a kernel that contains a tensor of shape  $k_h \times k_w$  for *every* input channel. Concatenating these  $c_i$  tensors together yields a convolution kernel of shape  $c_i \times k_h \times k_w$ . Since the input and convolution kernel each have  $c_i$  channels, we can perform a cross-correlation operation on the two-dimensional tensor of the input and the two-dimensional tensor of the convolution kernel for each channel, adding the  $c_i$  results together (summing over the channels) to yield a two-dimensional tensor. This is the result of a two-dimensional cross-correlation between a multi-channel input and a multi-input-channel convolution kernel.

we demonstrate an example of a two-dimensional cross-correlation with two input channels. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation:  $(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$ .

Input	Kernel	Input	Kernel	Output
		*	=	
			*	1 2
			3 4	+
			0 1	=
			2 3	56 72
				104 120

Cross-correlation computation with 2 input channels.

To make sure we really understand what is going on here, we can implement cross-correlation operations with multiple input channels ourselves. Notice that all we are doing is performing one cross-correlation operation per channel and then adding up the results.

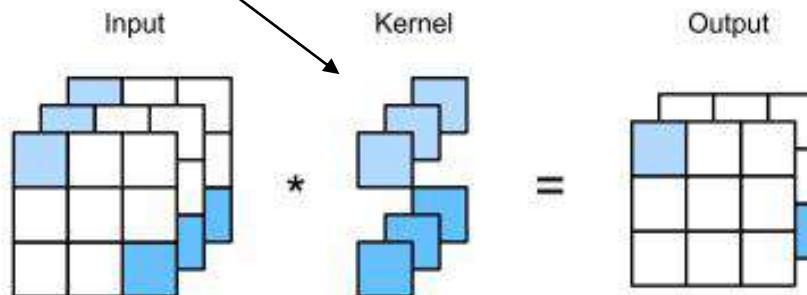


# 1 x 1 Convolutional layer

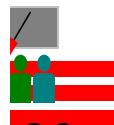
At first, a  $1 \times 1$  convolution, i.e.,  $k_h = k_w = 1$ , does not seem to make much sense. After all, a convolution correlates adjacent pixels. A  $1 \times 1$  convolution obviously does not. Nonetheless, they are popular operations that are sometimes included in the designs of complex deep networks. Let us see in some detail what it actually does.

Because the minimum window is used, the  $1 \times 1$  convolution loses the ability of larger convolutional layers to recognize patterns consisting of interactions among adjacent elements in the height and width dimensions. The only computation of the  $1 \times 1$  convolution occurs on the channel dimension.

~~shows the cross-correlation computation using the  $1 \times 1$  convolution kernel with 3 input channels and 2 output channels. Note that the inputs and outputs have the same height and width. Each element in the output is derived from a linear combination of elements at the same position in the input image. You could think of the  $1 \times 1$  convolutional layer as constituting a fully-connected layer applied at every single pixel location to transform the  $c_i$  corresponding input values into  $c_o$  output values. Because this is still a convolutional layer, the weights are tied across pixel location. Thus the  $1 \times 1$  convolutional layer requires  $c_o \times c_i$  weights (plus the bias).~~

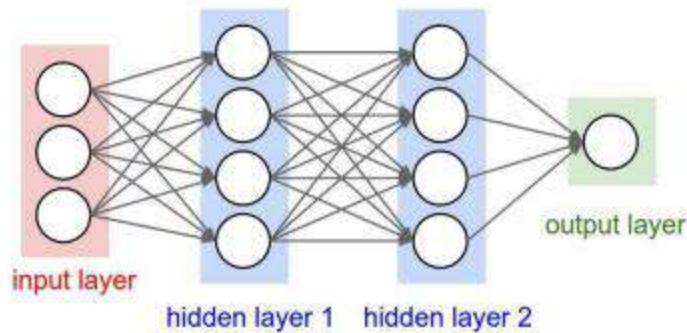


The cross-correlation computation uses the  $1 \times 1$  convolution kernel with 3 input channels and 2 output channels. The input and output have the same height and width.

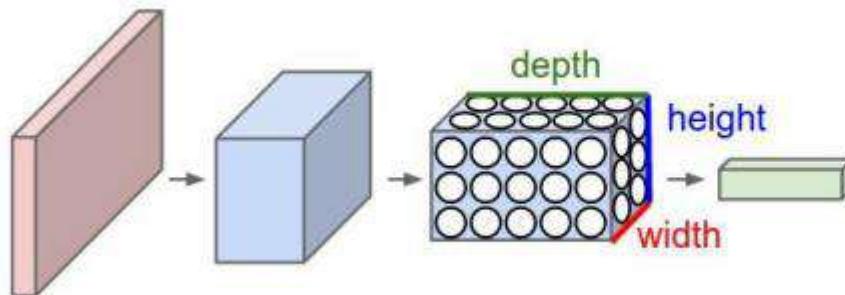


# Volumes

Volumes: The neurons of each level are organized in 3D grids or volumes (this is actually a graphical notation useful for understanding local connections).



MLP: linear organization of neurons in levels



CNN: The layers are organized as 3D grids of neurons

On the width - height planes, the "retinotypical" spatial organization of the input image is preserved.

The third depth dimension (as we will see) identifies the different feature maps.

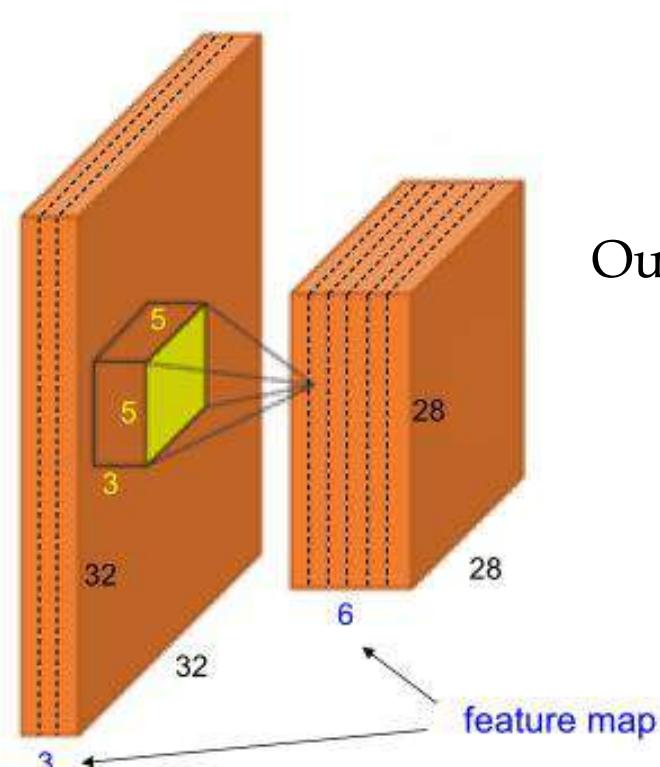
# Filtering

The filter operates on a portion of the input volume. In the example, each neuron of the output volume is connected to  $5 \times 5 \times 3 = 75$  neurons of the previous level.

Input:  $32 \times 32 \times 3$

e.g. color image of  
size  $32 \times 32$

Output:  $28 \times 28 \times 6$



# *Feature map*

Each "slice" of neurons (same depth) denotes a feature map. In the previous example we find:

3 feature maps (size 32x32) in the input volume;

6 feature maps (size 28x28) in the output volume.

Weights are shared at the feature map level. Neurons in the same feature map process different portions of the input volume in the same way. Each feature map can be seen as the result of a specific input filter (fixed filter).

In the example the number of connections between the two levels is  $(28 \times 28 \times 6) \times (5 \times 5 \times 3) = 352800$ , but the total number of weights is  $6 \times (5 \times 5 \times 3 + 1) = 456$ . Thanks to weight sharing



bias

A diagram illustrating the addition of a bias term. A grey arrow points from the word 'bias' to a small blue square containing a white plus sign (+). This plus sign is positioned above a horizontal grey line that represents the sum of weighted inputs from a previous layer.

# Filter: output size

When a filter is scrolled on the input volume, instead of moving in unit steps (of 1 neuron) a larger step (stride) can be used. This operation reduces the size of the feature maps in the output volume and consequently the number of connections.

On the initial levels of the network applying small strides (e.g. 2, 4), it is possible to obtain a high gain in efficiency at the expense of a slight accuracy reduction.

Another possibility (to adjust the size of the feature maps) is to add a border (zero values) to the input volume. The Padding parameter denotes the thickness (in pixels) of the border.

Let  $W_{out}$  be the (horizontal) dimension of the output feature map and  $W_{in}$  the corresponding dimension in the input. Furthermore, let  $F$  be the dimension of the filter. The following relationship applies:

$$W_{out} = \frac{(W_{in} - F + 2 \cdot Padding)}{Stride} + 1$$

if  $W_{out}$  is not an integer, the lower integer part is taken

$$Stride = 1, Padding = 0, W_{in} = 32,$$

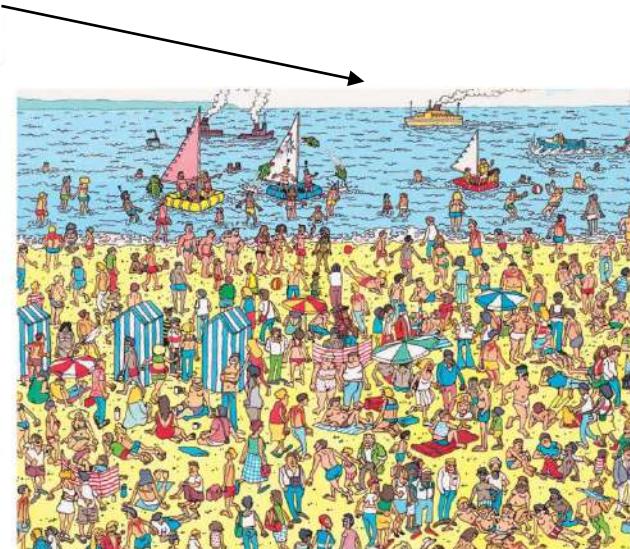
$$F = 5 \rightarrow W_{out} = 28.$$



# Invariance & constraining the network

Imagine that you want to detect an object in an image. It seems reasonable that whatever method we use to recognize objects should not be overly concerned with the precise location of the object in the image. Ideally, our system should exploit this knowledge. Pigs usually do not fly and planes usually do not swim. Nonetheless, we should still recognize a pig were one to appear at the top of the image. We can draw some inspiration here from the children's game "Where's Waldo"

The game consists of a number of chaotic scenes bursting with activities. Waldo shows up somewhere in each, typically lurking in some unlikely location. The reader's goal is to locate him. Despite his characteristic outfit, this can be surprisingly difficult, due to the large number of distractions. However, *what Waldo looks like* does not depend upon *where Waldo is located*. We could sweep the image with a Waldo detector that could assign a score to each patch, indicating the likelihood that the patch contains Waldo. CNNs systematize this idea of *spatial invariance*, exploiting it to learn useful representations with fewer parameters.



We can now make these intuitions more concrete by enumerating a few desiderata to guide our design of a neural network architecture suitable for computer vision:

1. In the earliest layers, our network should respond similarly to the same patch, regardless of where it appears in the image. This principle is called *translation invariance*.
2. The earliest layers of the network should focus on local regions, without regard for the contents of the image in distant regions. This is the *locality* principle. Eventually, these local representations can be aggregated to make predictions at the whole image level.

Let us see how this translates into mathematics.

## Constraining the MLP

To start off, we can consider an MLP with two-dimensional images  $\mathbf{X}$  as inputs and their immediate hidden representations  $\mathbf{H}$  similarly represented as matrices in mathematics and as two-dimensional tensors in code, where both  $\mathbf{X}$  and  $\mathbf{H}$  have the same shape. Let that sink in. We now conceive of not only the inputs but also the hidden representations as possessing spatial structure.



# Constraining the network

Let  $[\mathbf{X}]_{i,j}$  and  $[\mathbf{H}]_{i,j}$  denote the pixel at location  $(i, j)$  in the input image and hidden representation, respectively. Consequently, to have each of the hidden units receive input from each of the input pixels, we would switch from using weight matrices (as we did previously in MLPs) to representing our parameters as fourth-order weight tensors  $W$ . Suppose that  $\mathbf{U}$  contains biases, we could formally express the fully-connected layer as

$$\begin{aligned} [\mathbf{H}]_{i,j} &= [\mathbf{U}]_{i,j} + \sum_k \sum_l [W]_{i,j,k,l} [\mathbf{X}]_{k,l} \\ &= [\mathbf{U}]_{i,j} + \sum_a \sum_b [\mathbf{V}]_{i,j,a,b} [\mathbf{X}]_{i+a,j+b}, \end{aligned}$$

where the switch from  $W$  to  $V$  is entirely cosmetic for now since there is a one-to-one correspondence between coefficients in both fourth-order tensors. We simply re-index the subscripts  $(k, l)$  such that  $k = i + a$  and  $l = j + b$ . In other words, we set  $[\mathbf{V}]_{i,j,a,b} = [W]_{i,j,i+a,j+b}$ . The indices  $a$  and  $b$  run over both positive and negative offsets, covering the entire image. For any given location  $(i, j)$  in the hidden representation  $[\mathbf{H}]_{i,j}$ , we compute its value by summing over pixels in  $x$ , centered around  $(i, j)$  and weighted by  $[\mathbf{V}]_{i,j,a,b}$ .

## Translation Invariance

Now let us invoke the first principle established above: translation invariance. This implies that a shift in the input  $\mathbf{X}$  should simply lead to a shift in the hidden representation  $\mathbf{H}$ . This is only possible if  $V$  and  $\mathbf{U}$  do not actually depend on  $(i, j)$ , i.e., we have  $[\mathbf{V}]_{i,j,a,b} = [\mathbf{V}]_{a,b}$  and  $\mathbf{U}$  is a constant, say  $u$ . As a result, we can simplify the definition for  $\mathbf{H}$ :

$$[\mathbf{H}]_{i,j} = u + \sum_a \sum_b [\mathbf{V}]_{a,b} [\mathbf{X}]_{i+a,j+b}.$$

This is a *convolution*! We are effectively weighting pixels at  $(i + a, j + b)$  in the vicinity of location  $(i, j)$  with coefficients  $[\mathbf{V}]_{a,b}$  to obtain the value  $[\mathbf{H}]_{i,j}$ . Note that  $[\mathbf{V}]_{a,b}$  needs many fewer coefficients than  $[\mathbf{V}]_{i,j,a,b}$  since it no longer depends on the location within the image. We have made significant progress!

notice that we are describing the invariance of a single layer, considering large complex topologies, they are not invariant to translation, but they can learn to be  
<https://arxiv.org/abs/2110.05861>

## Locality

Now let us invoke the second principle: locality. As motivated above, we believe that we should not have to look very far away from location  $(i, j)$  in order to glean relevant information to assess what is going on at  $[\mathbf{H}]_{i,j}$ . This means that outside some range  $|a| > \Delta$  or  $|b| > \Delta$ , we should set  $[\mathbf{V}]_{a,b} = 0$ . Equivalently, we can rewrite  $[\mathbf{H}]_{i,j}$  as

$$[\mathbf{H}]_{i,j} = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [\mathbf{V}]_{a,b} [\mathbf{X}]_{i+a,j+b},$$

in a nutshell, is a *convolutional layer*. Convolutional neural networks (CNNs) are a special family of neural networks that contain convolutional layers. In the deep learning research community,  $V$  is referred to as a *convolution kernel*, a *filter*, or simply the layer's *weights* that are often learnable parameters. When the local region is small, the difference as compared with a fully-connected network can be dramatic. While previously, we might have required billions of parameters to represent just a single layer in an image-processing network, we now typically need just a few hundred, without altering the dimensionality of either the inputs or the hidden representations. The price paid for this drastic reduction in parameters is that our features are now translation invariant and that our layer can only incorporate local information, when determining the value of each hidden activation. All learning depends on imposing inductive bias. When that bias agrees with reality, we get sample-efficient models that generalize well to unseen data. But of course, if those biases do not agree with reality, e.g., if images turned out not to be translation invariant, our models might struggle even to fit our training data.

Returning to our Waldo detector, let us see what this looks like. The convolutional layer picks windows of a given size and weighs intensities according to the filter  $V$

We might aim to learn a model so that wherever the "waldoness" is highest, we should find a peak in the hidden layer representations.



Detect Waldo.

# *Activation function*

In MLP networks, the (historically) most used activation function is the sigmoid. In deep networks, the use of the sigmoid is problematic for the retro propagation of the gradient (vanishing gradient problem): The derivative of the sigmoid is typically less than 1 and the application of the chain derivation rule leads to multiplying many terms less than 1 with the consequence of greatly reducing the gradient values in the levels far from the output. For further information and examples:

<http://neuralnetworksanddeeplearning.com/chap5.html>

The sigmoid has a saturating behavior (moving away from 0). In the saturation regions the derivative is 0 and therefore the gradient vanishes.

A large number of activation functions are here described and compared:

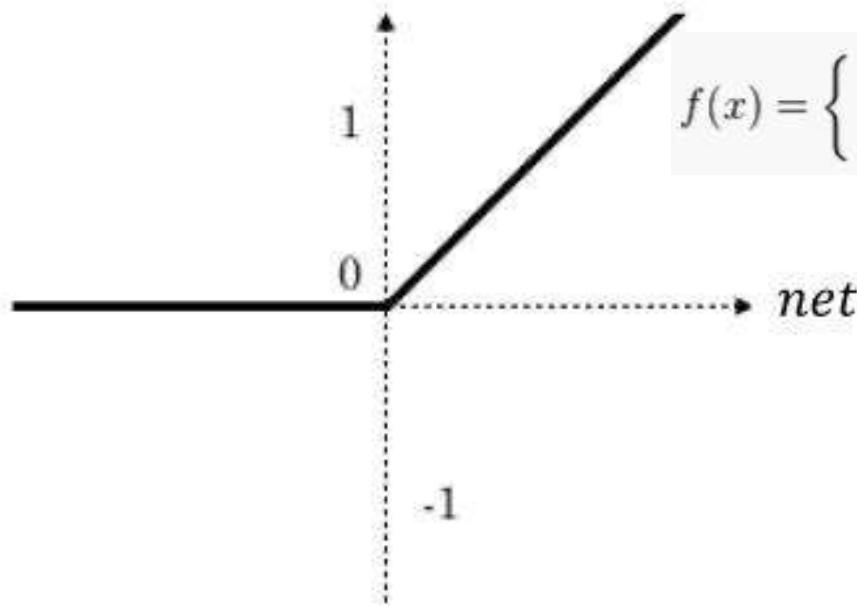
<https://www.mdpi.com/1424-8220/22/16/6129>



# ReLU (Rectified Linear Unit) activation function

$$f(\text{net}) = \max(0, \text{net})$$

The derivative is zero for negative or zero values of net and 1 for positive values



$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

$$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$$

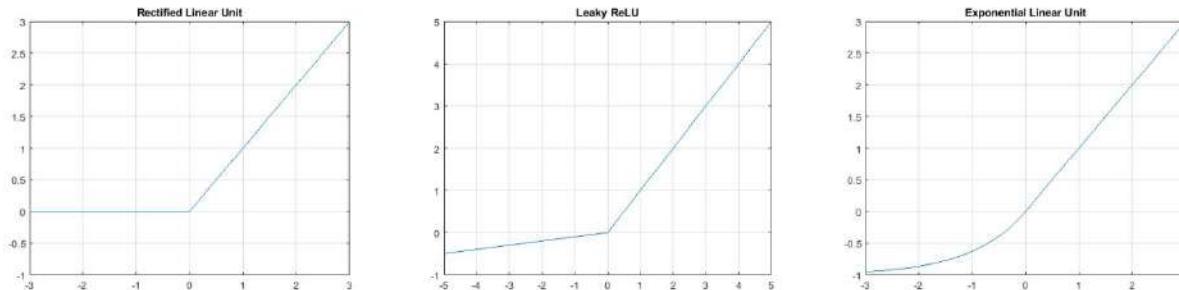
For positive values no saturation

It leads to sparse activations (part of the neurons are switched off) which can confer greater robustness

In reality, the choice of the activation function is still a research area and many variants have been proposed and others are under study



# LeakyReLu / ELU



Leaky ReLU

Figure 1. ReLu (left), Leaky ReLu (center), and ELU (right).

Leaky ReLU (see Figure 1) is defined as

$$y_i = f(x_i) = \begin{cases} ax_i, & x_i < 0 \\ x_i, & x_i \geq 0 \end{cases}$$

where  $a$  is a small real number ( $a = 0.01$  here).

no point with a null gradient:

$$\frac{dy_i}{dx_i} = f'(x_i) = \begin{cases} a, & x_i < 0 \\ 1, & x_i \geq 0 \end{cases}$$

ELU

Exponential Linear Unit (ELU) (see Figure 1) is defined as

$$y_i = f(x_i) = \begin{cases} a(\exp(x_i) - 1), & x_i < 0 \\ x_i, & x_i \geq 0 \end{cases}$$

where  $a$  is a real number ( $a = 1$  here).

ELU is differentiable and has a gradient that is always positive (as with Leaky ReLU) and bounded

from below by  $-a$ ; The gradient of ELU is given by

ELU value is bounded by

$-a$ , not the gradient

$$\frac{dy_i}{dx_i} = f'(x_i) = \begin{cases} a \exp(x_i), & x_i < 0 \\ 1, & x_i \geq 0 \end{cases}$$

Leaky ReLU solves the Dying ReLU problem i.e for the inputs 0 or negative the gradient of ReLU becomes zero and thus the network cannot make backpropagation for that neuron.

The goal of any activation function is to avoid:  
 vanishing gradient;  
 exploding gradient;  
 over fitting;  
 underfitting.

In the literature many properties of the activation functions are considered useful for avoiding the above cited problems.



# PRelu

## PRelu

Parametric ReLU (PreLU) is defined as

$$y_i = f(x_i) = \begin{cases} a_c x_i, & x_i < 0 \\ x_i, & x_i \geq 0 \end{cases}$$

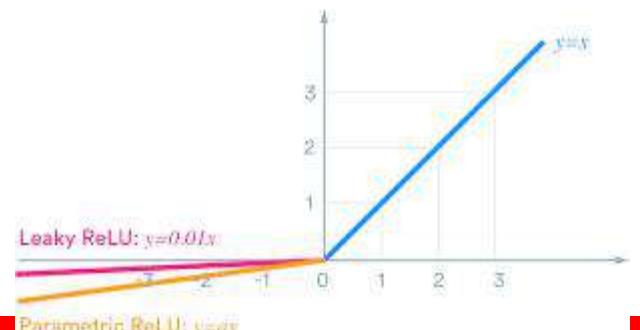
where  $a_c$  are real numbers that are different for every channel of the input. PreLU is the same as Leaky ReLU except that the parameters  $a_c$  are learnable.

The gradients of PreLU are

$$\frac{dy_i}{dx_i} = f'(x_i) = \begin{cases} a_c, & x_i < 0 \\ 1, & x_i \geq 0 \end{cases} \quad \text{and} \quad \frac{dy_i}{da_c} = \begin{cases} x_i, & x_i < 0 \\ 0, & x_i \geq 0 \end{cases}$$

Note that the slopes of the left-hand sides are all initialized at 0.

This means that the parameters 'a' are initialized to zero



# SReLU

S-Shaped ReLU (SReLU) is composed of three piecewise linear functions that are expressed by four learnable parameters, thus

$$h(x_i) = \begin{cases} t_i^r + a_i^r(x_i - t_i^r), & x_i \geq t_i^r \\ x_i, & t_i^r > x_i > t_i^l \\ t_i^l + a_i^l(x_i - t_i^l), & x_i \leq t_i^l \end{cases}$$

where  $t^l, t^r, a^l$ , and  $a^r$  are the four learnable real number parameters (initialized as  $a^l = 0, t^l = 0, t^r = maxInput$ , where  $maxInput$  is a hyperparameter). SReLU's rather large number of parameters results in high representation power.

The gradients of SReLU are given by

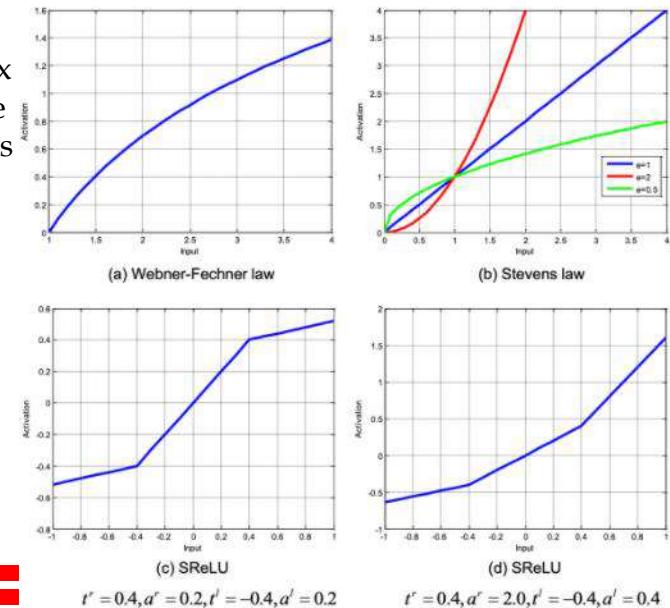
$$\begin{aligned}\frac{\partial h(x_i)}{\partial t_i^r} &= I\{x_i \geq t_i^r\}(1 - a_i^r) \\ \frac{\partial h(x_i)}{\partial a_i^r} &= I\{x_i \geq t_i^r\}(x_i - t_i^r) \\ \frac{\partial h(x_i)}{\partial t_i^l} &= I\{x_i \leq t_i^l\}(1 - a_i^l) \\ \frac{\partial h(x_i)}{\partial a_i^l} &= I\{x_i \leq t_i^l\}(x_i - t_i^l)\end{aligned}$$

where  $I\{\cdot\}$  is an indicator function and  $I\{\cdot\} = 1$  when the expression inside holds true, otherwise  $I\{\cdot\} = 0$ . By this way, the gradient of the input is

the gradient of  $h(x_i)$  with respect to  $x_i$

$$= \begin{cases} a_i^r, & x_i \geq t_i^r \\ 1, & t_i^r > x_i > t_i^l \\ a_i^l, & x_i \leq t_i^l. \end{cases}$$

SReLU learns both convex and non-convex functions, imitating the multiple function forms given by the two fundamental laws, namely the Webner-Fechner law and the Stevens law, in psychophysics and neural sciences.

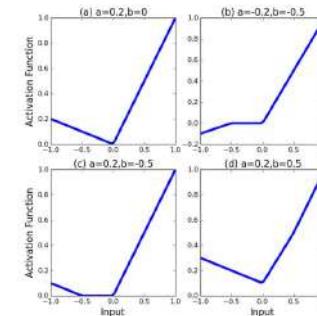


# APLU

Adaptive Piecewise Linear Unit (APLU) is defined as

$$y_i = \text{ReLU}(x_i) + \sum_{c=1}^n a_c \max(0, -x_i + b_c)$$

where  $a_c$  and  $b_c$  are real numbers, each different for each channel of the input. APLU is a linear piecewise function able to approximate any continuous function on a compact set.



S=1

<https://justinmeiners.github.io/calculus-on-manifolds-site/lectures/lecture2/html/compact.htm>

<https://www.youmath.it/lezioni/analisi-matematica/premesse-per-lanalisi-infinitesimale/3493-insieme-compatto-teorema-di-heine-borel.html>

The gradient of APLU is given by the sum of the gradients of ReLU and of the functions contained

in the sum. With respect to the parameters,  $a_c$  and  $b_c$ , the gradients are

$$\frac{df(x,a)}{da_c} = \begin{cases} -x + b_c, & x < b_c \\ 0, & x \geq b_c \end{cases} \text{ and } \frac{df(x,a)}{db_c} = \begin{cases} -a_c, & x < b_c \\ 0, & x \geq b_c \end{cases}.$$

the parameters  $a_c$  were initialized to 0, while the points were randomly initialized.

Added was a 0.001  $L^2$ -penalty on the norm of the parameters  $a_c$ . Thus, another term was included in the loss

function, which is defined as

$$L^{\text{reg}} = \sum_{c=1}^n |a_c|^2.$$

⇒ this mean that  $L^{\text{reg}}$  is multiplied by 0.001. The same process is applied also to  $b_c$

Without this penalty, the optimizer is free to choose very large values of  $a_c$  and  $b_c$  balanced by very small weights, which would lead to numerical instability (exploding gradient). Notice that high values means a large update of parameters, so we could jump over minimum.



# Mexican ReLu

To understand the Mexican ReLU (MeLU) let  $\phi^{a,\lambda}(x) = \max(\lambda - |x - a|, 0)$  be a “Mexican hat type”

function, where  $a$  and  $\lambda$  are real numbers. When  $|x - a| > \lambda$ , the function  $\phi^{a,\lambda}(x)$  is null but constantly increases with a derivative of 1 between  $a - \lambda$  and  $a$  and decreases with a derivative of  $-1$  between  $a$  and  $a + \lambda$ .

<https://arxiv.org/ftp/arxiv/papers/1905/1905.02473.pdf>

The aforementioned functions serve as the building blocks of MeLU, which is defined as

$$y_i = \text{MeLU}(x_i) = \text{PReLU}^{c_0}(x_i) + \sum_{j=1}^{k-1} c_j \phi^{a_j, \lambda_j}(x_i),$$

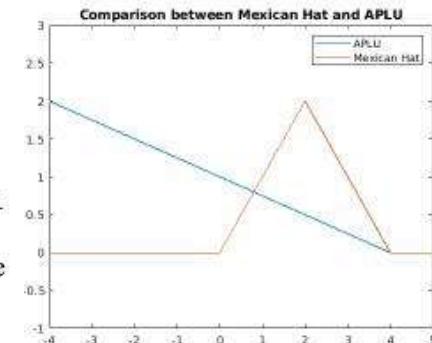
where  $k$  is the number of learnable parameters for each channel (one parameter for PReLU and  $k - 1$  parameters for the coefficients in the sum of the Mexican hat functions;  $k = 4, 8$  here),  $c_j$  are the learnable parameters,  $c_0$  is the vector of parameters in PReLU, and  $a_j$  and  $\lambda_j$  are both fixed and chosen recursively.

Initially, the parameter  $\text{maxInput}$  is set. The first Mexican hat function has its maximum in  $2 \cdot \text{maxInput}$  and is equal to zero in  $0$  and  $4 \cdot \text{maxInput}$ . The next two functions are chosen to be zero outside the interval  $[0, 2 \cdot \text{maxInput}]$  and  $[2 \cdot \text{maxInput}, 4 \cdot \text{maxInput}]$ , with the requirement being they have their maximum in  $\text{maxInput}$  and  $3 \cdot \text{maxInput}$ .

value of the first seven Mexican hat functions

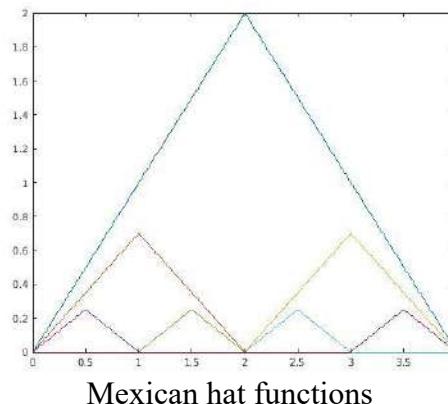
j	1	2	3	4	5	6	7
$a_j$	512	256	768	128	384	640	896
$\lambda_j$	512	256	256	128	128	128	128

Fixed parameters of MeLU with  $\text{maxInput} = 256$ .



Partial derivatives of MeLU and APLU.

Since the gradient can be zero, we sum Mexican Hat with PReLU for avoiding vanishing gradient



# Mexican ReLu

Since Mexican hat functions are continuous and piecewise differentiable, so is MeLU. If all the  $c_i$  learnable parameters are initialized to zero, MeLU coincides with ReLU, a property that is helpful in transfer learning when the network is pretrained with ReLU (in which case MeLU can simply be substituted; Of course, MeLU can also be substituted for networks trained with Leaky ReLU and PReLU).

MeLU is similar to SReLU and APLU in having multiple learnable parameters, but this number is larger in APLU. MeLU is also similar to APLU in that it can approximate the same set of piecewise linear functions equal to identity when  $x$  is large enough, but how this is done is different for each function. Given the right choice of the parameters, APLU can be equal to any piecewise linear function because the points of non-differentiability are learnable, but MeLU is more efficient because it can represent every piecewise linear function by exploiting only the joint optimization of the weight matrix and the biases. Moreover, the gradients of the two activations with respect to the learnable parameters are different. The optimization of a neural network depends in large part on two factors: the output of every hidden layer and the gradient of that output with respect to the parameters.

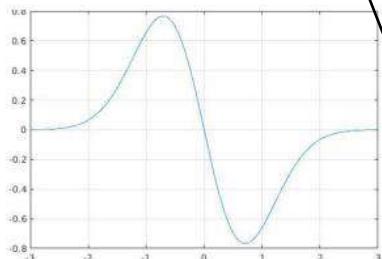
the learnable parameters in MeLU are initialized to zero, so the activation starts off as ReLU, thereby exploiting all the nice properties of ReLU at the beginning.



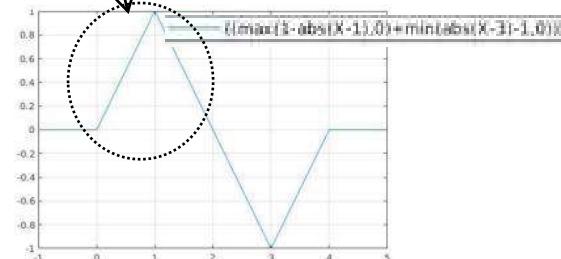
# GaLU

Gaussian ReLU (GaLU)

is based on MeLU and possesses the same desirable properties. To define GaLU, let  $\phi_g^{a,\lambda}(x) = \max(\lambda - |x - a|, 0) + \min(|x - a - 2\lambda| - \lambda, 0)$  be a Gaussian type function, where  $a$  and  $\lambda$  are real numbers. In a standard gaussian wavelet function is compared with a Gaussian type with  $a = 1, \lambda = 1$ .



mexican hat



Gaussian Wavelet (left) and our Gaussian Type (right)

GaLU is defined, similarly to MeLU, as

$$y_i = \text{GaLU}(x_i) = PReLU^{c_0}(x_i) + \sum_{j=1}^{k-1} c_j \phi_g^{a_j, \lambda_j}(x_i).$$

piecewise linear odd functions constructed of many linear pieces approximate nonlinear functions better than ReLU does. The rationale behind GaLU is to add more linear pieces than is the case with MeLU.

Like MeLU, GaLU has a set of fixed parameters

	j	1	2	3	4	5	6	7
MeLU	$a_j$	2	1	3	0.5	1.5	2.5	3.5
	$\lambda_j$	2	1	1	0.5	0.5	0.5	0.5

	j	1	2	3	4	5	6	7
GaLU	$a_j$	1	0.5	2.5	0.25	1.25	2.25	3.25
	$\lambda_j$	1	0.5	0.5	0.25	0.25	0.25	0.25

Comparison of the Fixed parameters of GaLU and MeLU with  $\maxInput = 1$ .

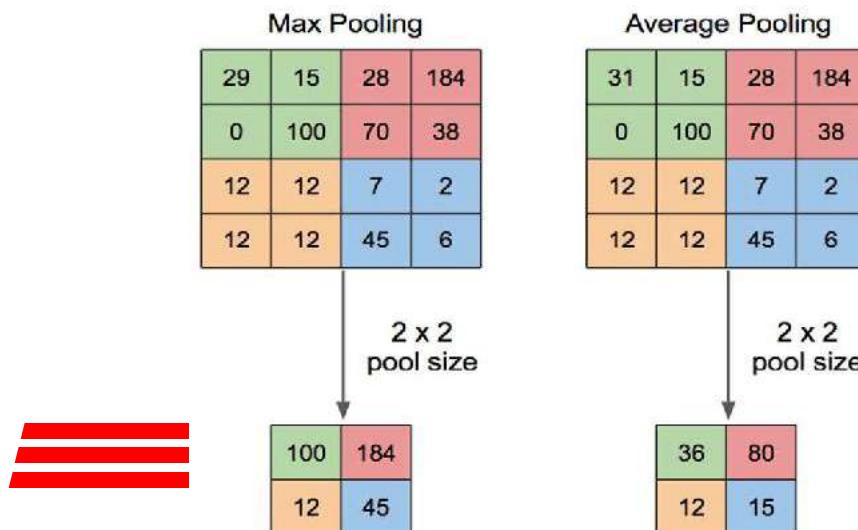
<https://www.taylorfrancis.com/chapters/edit/10.1201/9781315101323-5/ensembles-convolutional-neural-networks-different-activation-functions-small-medium-sized-biomedical-datasets-filippo-berno-loris-nanni-gianluca-maguolo-sheryl-brahnam>



# Pooling

A pooling level aggregates information in the input volume, resulting in smaller feature maps. The goal is to confer invariance with respect to simple transformations of the input while maintaining significant information for pattern discrimination.

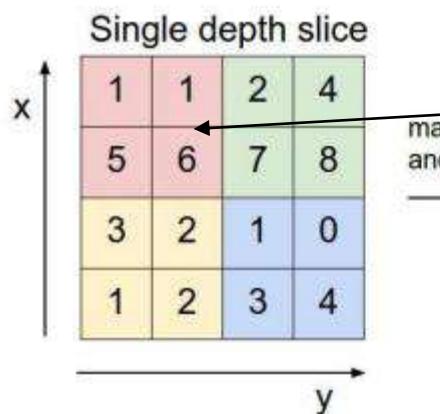
Aggregation operates (generally) within each feature map, so that the number of feature maps in the input and output volume is the same. The most used aggregation operators are the average (Avg) and the maximum (Max): both “rather” invariant under small translations. This type of aggregation has no parameters / weights to learn.



# Pooling

In this example a max-pooling with  $2 \times 2$  filters and Stride = 2 is reported.

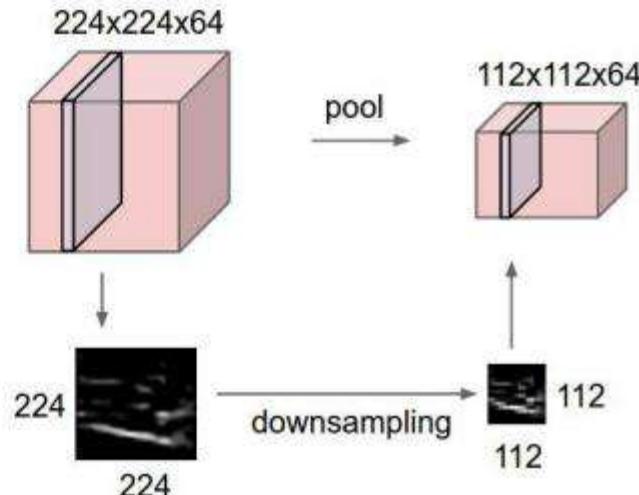
The previous equation for the calculation of  $W_{out}$  is also valid for pooling (considering Padding = 0)



max pool with  $2 \times 2$  filters and stride 2

6	8
3	4

6 is the max among  
 $\{1, 1, 5, 6\}$



# SoftMax

Other pretty common choices in modern CNNs (when used as classifiers) are:

a) the use of a final SoftMax activation function:

It consists of  $s$  neurons (one for each class) fully-connected with respect to the neurons of the previous level. The  $\text{net}_k$  activation level of the single neurons is calculated in the usual way, but as the activation function for the  $k$ -th neuron (instead of Sigmoid or Relu) is:

$$z_k = f(\text{net}_k) = \frac{e^{\text{net}_k}}{\sum_{c=1 \dots s} e^{\text{net}_c}}$$

where the values  $z_k$  produced can be interpreted as probabilities: they belong to  $[0 \dots 1]$  and their sum is 1.



# Cross Entropy - Loss function

b) the use of Cross-Entropy as a loss function (instead of Sum of Squared Error already introduced for MLP): the cross-entropy between two discrete distributions  $p$  and  $q$  (which fixed  $p$  measures how much  $q$  differs from  $p$ ) is defined by:

$$H(p, q) = - \sum_v p(v) \cdot \log(q(v))$$

Using the form  $t = [0, 0 \dots 1 \dots 0]$  to encode the desired vector for the pattern  $x$ , where  $g$  is the correct class, then:

$$J(\mathbf{w}, \mathbf{x}) \equiv H(\mathbf{t}, \mathbf{z}) = -\log(z_g) =$$

$$= -\log \left( \frac{e^{net^g}}{\sum_{c=1 \dots S} e^{net_c}} \right)$$



# Initialization

- Random initialization for weights
  - Gaussian or uniform distribution (scale is important)
  - Large weights have a stronger symmetry breaking effect but..
  - Exploding values or gradient, saturation
  - Gradient descent + early stopping is like imposing a gaussian prior around the initialization weights → close to  $\mathbf{0}$

- Some heuristics for a FC layers with  $m$  inputs and  $n$  outputs:

$$W_{i,j} \sim U\left(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right)$$

- Or Xavier Glorot (and Bengio)  $W_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$
  - If possible, the scale can be treated as a hyper-parameter
- Pre-defined constant for biases (usually 0 or 0.1 for ReLU to avoid saturation)

- the variance of the outputs of each layer should be equal to the variance of its inputs
- the same should hold for the gradient
- trade-off: for each unit normalize the initial weights with respect to the number of incoming and outgoing connections

*Examples of distributions (He initialization) for initialization according to the type of activation function (Normal distribution: mean = 0)*

Activation function	Uniform distribution $[-r, r]$	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

## Pre-training

- Unsupervised pre-training:
  - Train an unsupervised model on the training data, use the resulting model as initialization
- Supervised pre-training (transfer learning):
  - Initialize the weights with a model trained on a related task
  - Especially effective with CNNs



# To calculate the gradient of this loss function:

*"without understanding the underlying math and calculations behind each line of code, we cannot truly understand what "creating a neural network" really means or appreciate the complex intricacies that support each function that we write."*

$$C(\mathbf{y}, \mathbf{w}, \mathbf{X}, b) = \frac{1}{N} \sum_{i=1}^N (y_i - \max(0, \mathbf{w} \cdot \mathbf{X}_i + b))^2$$

Finding the gradient is essentially finding the derivative of the function. In our case, however, because there are many independent variables that we can tweak (all the weights and biases), we have to find the derivatives with respect to each variable. This is known as the partial derivative, with the symbol  $\partial$ .

Computing the partial derivative of simple functions is easy: simply treat every other variable in the equation as a constant and find the usual scalar derivative.

Here are some scalar derivative rules as a reminder:

Rule	$f(x)$	Scalar derivative notation with respect to $x$	Example
Constant	$c$	0	$\frac{d}{dx} 99 = 0$
Multiplication by constant	$cf$	$c \frac{df}{dx}$	$\frac{d}{dx} 3x = 3$
Power Rule	$x^n$	$nx^{n-1}$	$\frac{d}{dx} x^3 = 3x^2$
Sum Rule	$f + g$	$\frac{df}{dx} + \frac{dg}{dx}$	$\frac{d}{dx} (x^2 + 3x) = 2x + 3$
Difference Rule	$f - g$	$\frac{df}{dx} - \frac{dg}{dx}$	$\frac{d}{dx} (x^2 - 3x) = 2x - 3$
Product Rule	$fg$	$f \frac{dg}{dx} + g \frac{df}{dx}$	$\frac{d}{dx} x^2 x = x^2 + x2x = 3x^2$
Chain Rule	$f(g(x))$	$\frac{df(u)}{du} \frac{du}{dx}$ , let $u = g(x)$	$\frac{d}{dx} \ln(x^2) = \frac{1}{x^2} 2x = \frac{2}{x}$

# To calculate the gradient of this loss function

Consider the partial derivative with respect to  $x$  (i.e. how  $y$  changes as  $x$  changes) in the function  $f(x,y) = 3x^2y$ . Treating  $y$  as a constant, we can find partial of  $x$ :

$$\frac{\partial}{\partial x} 3yx^2 = 3y \frac{\partial}{\partial x} x^2 = 3y \cdot 2x = 6yx$$

Similarly, we can find the partial of  $y$ :

$$\frac{\partial}{\partial y} 3yx^2 = 3x^2 \frac{\partial}{\partial y} y = 3x^2 \times 1 = 3x^2$$

The gradient of the function  $f(x,y) = 3x^2y$  is a horizontal vector, composed of the two partials:

$$\nabla f(x,y) = \left[ \frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y} \right] = [6yx, 3x^2]$$

This should be pretty clear: since the partial with respect to  $x$  is the gradient of the function in the  $x$ -direction, and the partial with respect to  $y$  is the gradient of the function in the  $y$ -direction, the overall gradient is a vector composed of the two partials.

This Khan Academy video offers a pretty neat graphical explanation of partial derivatives  
<https://youtu.be/dfvnCHqzK54>

(Partial Derivative). For a function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ ,  $\mathbf{x} \mapsto f(\mathbf{x})$ ,  $\mathbf{x} \in \mathbb{R}^n$  of  $n$  variables  $x_1, \dots, x_n$  we define the *partial derivatives* as

$$\frac{\partial f}{\partial x_1} = \lim_{h \rightarrow 0} \frac{f(x_1 + h, x_2, \dots, x_n) - f(\mathbf{x})}{h}$$

⋮

$$\frac{\partial f}{\partial x_n} = \lim_{h \rightarrow 0} \frac{f(x_1, \dots, x_{n-1}, x_n + h) - f(\mathbf{x})}{h}$$

and collect them in the row vector

$$\nabla_{\mathbf{x}} f = \text{grad } f = \frac{df}{d\mathbf{x}} = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} & \frac{\partial f(\mathbf{x})}{\partial x_2} & \dots & \frac{\partial f(\mathbf{x})}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{1 \times n},$$

Consider a function  $f : \mathbb{R}^2 \rightarrow \mathbb{R}$  of two variables  $x_1, x_2$ . Furthermore,  $x_1(t)$  and  $x_2(t)$  are themselves functions of  $t$ . To compute the gradient of  $f$  with respect to  $t$ , we need to apply the chain rule for multivariate functions as

$$\frac{df}{dt} = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} \end{bmatrix} \begin{bmatrix} \frac{\partial x_1(t)}{\partial t} \\ \frac{\partial x_2(t)}{\partial t} \end{bmatrix} = \frac{\partial f}{\partial x_1} \frac{\partial x_1}{\partial t} + \frac{\partial f}{\partial x_2} \frac{\partial x_2}{\partial t},$$

where  $d$  denotes the gradient and  $\partial$  partial derivatives.

Consider  $f(x_1, x_2) = x_1^2 + 2x_2$ , where  $x_1 = \sin t$  and  $x_2 = \cos t$ , then

$$\begin{aligned} \frac{df}{dt} &= \frac{\partial f}{\partial x_1} \frac{\partial x_1}{\partial t} + \frac{\partial f}{\partial x_2} \frac{\partial x_2}{\partial t} \\ &= 2 \sin t \frac{\partial \sin t}{\partial t} + 2 \cos t \frac{\partial \cos t}{\partial t} \\ &= 2 \sin t \cos t - 2 \sin t = 2 \sin t (\cos t - 1) \end{aligned}$$

is the corresponding derivative of  $f$  with respect to  $t$ .

# To calculate the gradient of this loss function

## Chain Rules:

For simple functions like  $f(x,y) = 3x^2y$ , that is all we need to know. However, if we want to compute partial derivatives of more complicated functions — such as those with nested expressions like  $\max(0, w \cdot X + b)$  — we need to be able to utilize the multivariate chain rule, known as the *single variable total-derivative chain rule*

## Single Variable Chain Rule

Let's first review the single variable chain rule. Consider the function  $y=f(g(x))=\sin(x^2)$ . To get the derivative of this expression, we multiply the derivative of the outer expression with the derivative of the inner expression or ‘chain the pieces together’. In other words:

$$\frac{dy}{dx} = \frac{dy}{du} \frac{du}{dx}$$

For our example,  $u=x^2$  and  $y=\sin(u)$ . Hence:

$$\begin{aligned}\frac{du}{dx} &= 2x && \text{(Take derivative with respect to } x\text{)} \\ \frac{dy}{du} &= \cos(u) && \text{(Take derivative with respect to } u \text{ not } x\text{)}\end{aligned}$$

and

$$\frac{dy}{dx} = \frac{dy}{du} \frac{du}{dx} = \cos(u)2x = 2x\cos(x^2)$$

It's nice to think about the single-variable chain rule as a diagram of operations that  $x$  goes through, like so:

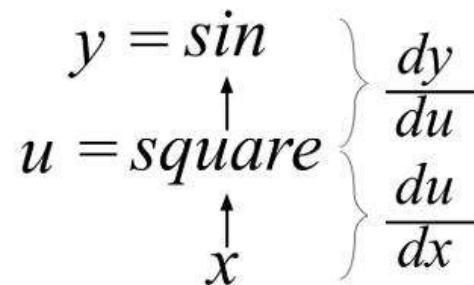


Diagram of chain of operations for  $y=\sin(x^2)$



# Single variable total derivative chain rule

Unlike what

its name suggests, it can be applied to expressions with only a single variable.

However, the expression should have multiple intermediate variables.

To illustrate this point, let us consider the equation  $y=f(x)=x+x^2$ . Using the scalar additional derivative rule, we can immediately calculate the derivative:

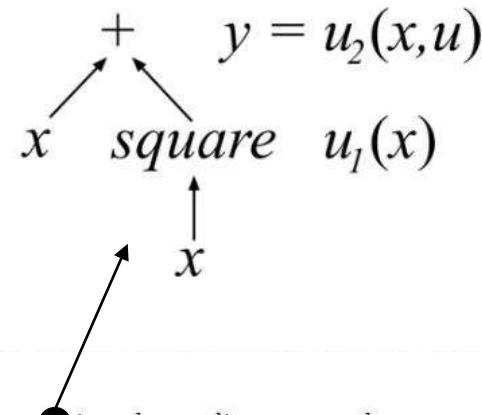
$$\frac{dy}{dx} = \frac{d}{dx}x + \frac{d}{dx}x^2 = 1 + 2x$$

Let's try doing it with the chain rule. First, we introduce intermediate variables:  $u_1(x) = x^2$  and  $u_2(x, u_1) = x + u_1$ . If we apply the single-variable chain rule, we get:

$$\frac{dy}{dx} = \frac{du_2}{dx} = \frac{du_2}{du_1} \frac{du_1}{dx} = 2x$$

Using the single-variable chain rule

Obviously,  $2x \neq 1 + 2x$ , so something is wrong here. Let's draw out the graph of our equation:



The diagram in Image is no longer linear, so we have to consider all the pathways in the diagram that lead to the final result. Since  $u_2$  has two parameters, partial derivatives come into play. To calculate the derivative of this function, we have to calculate partial derivative with respect to  $x$  of  $u_2(x, u_1)$ . Here, a change in  $x$  is reflected in  $u_2$  in two ways: as an operand of the addition and as an operand of the square operator. In symbols,  $\hat{y} = (x+\Delta x) + (x+\Delta x)^2$  and  $\Delta y = \hat{y} - y$  and where  $\hat{y}$  is the  $y$ -value at a tweaked  $x$ .



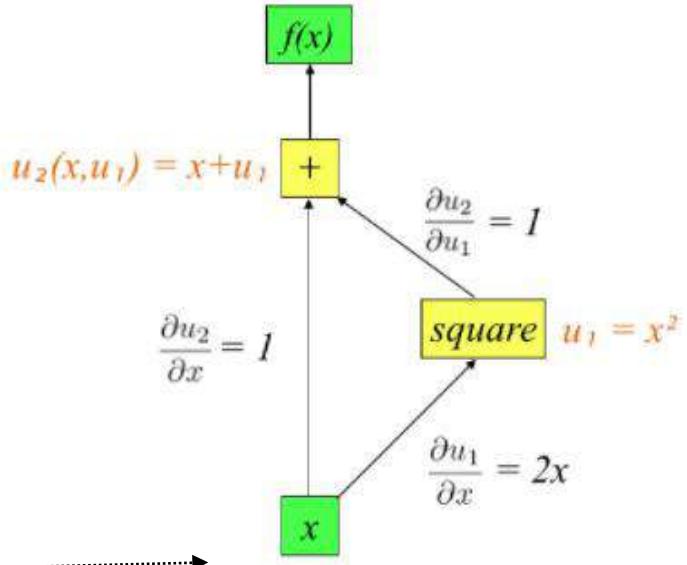
# Total derivative

Hence, to compute the partial of  $u_2(x, u_1)$ , we need to sum up all possible contributions from changes in  $x$  to the change in  $y$ . The total derivative of  $u_2(x, u_1)$  is given by:

$$\frac{dy}{dx} = \frac{\partial f(x)}{\partial x} = \frac{\partial u_2(x, u_1)}{\partial x} = \frac{\partial u_2}{\partial x} + \frac{\partial u_2}{\partial u_1} \frac{\partial u_1}{\partial x}$$

Derivative of  $y = x+x^2$

In simpler terms, you **add up** the effect of a change in  $x$  directly to  $u_2$  and the effect of a change in  $x$  through  $u_1$  to  $u_2$ . I find it easier to visualize it through a graph:



$$\text{Right-side path: } \frac{\partial u_2}{\partial u_1} \frac{\partial u_1}{\partial x} = 1 \cdot 2x = 2x$$

$$\text{Left-side path: } \frac{\partial u_2}{\partial x} = 1$$

$$\text{Total contribution of a change in } x \text{ towards a change in } y \text{ (or } f(x)) : \frac{\partial u_2}{\partial x} + \frac{\partial u_2}{\partial u_1} \frac{\partial u_1}{\partial x} = 1 + 2x$$



# To calculate the gradient of this loss function

And that's it! We got the correct answer:  $1+2x$ . We can now sum that process up in a single rule, the multivariable chain rule (or the single-variable total-derivative chain rule):

$$\frac{\partial f(x, u_1, \dots, u_n)}{\partial x} = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial u_1} \frac{\partial u_1}{\partial x} + \frac{\partial f}{\partial u_2} \frac{\partial u_2}{\partial x} + \dots + \frac{\partial f}{\partial u_n} \frac{\partial u_n}{\partial x} = \frac{\partial f}{\partial x} + \sum_{i=1}^n \frac{\partial f}{\partial u_i} \frac{\partial u_i}{\partial x}$$

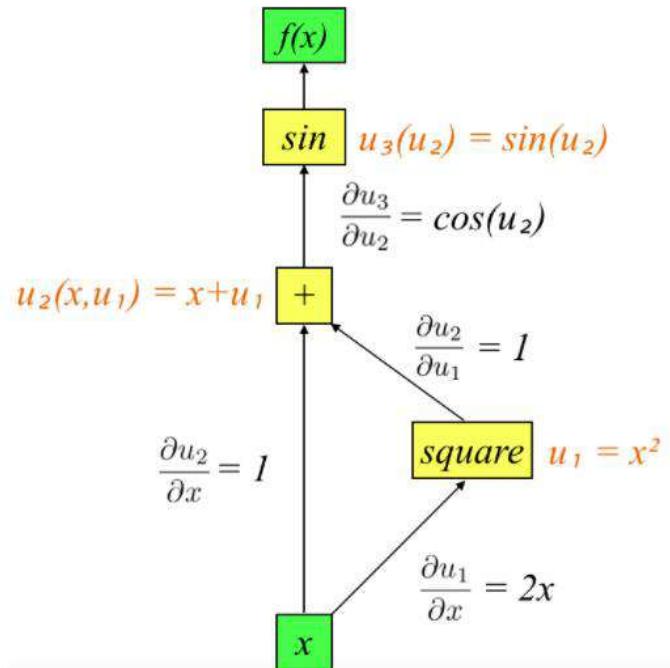
If we introduce an alias for  $x$  as  $x=u(n+1)$ , then we can rewrite that formula into its final form, which look slightly neater:

$$\frac{\partial f(u_1, \dots, u_{n+1})}{\partial x} = \sum_{i=1}^{n+1} \frac{\partial f}{\partial u_i} \frac{\partial u_i}{\partial x}$$

Multivariable chain rule

That's all to it! To review, let's do another example:  $f(x)=\sin(x+x^2)$ . Our 3 intermediate variables are:  $u_1(x) = x^2$ ,  $u_2(x, u_1) = x+u_1$ , and  $u_3(u_2) = \sin(u_2)$ .

Once again, we can draw our graph:



# To calculate the gradient of this loss function

$u_1(x) = x^2$ ,  $u_2(x, u_1) = x + u_1$ , and  $u_3(u_2) = \sin(u_2)$ .

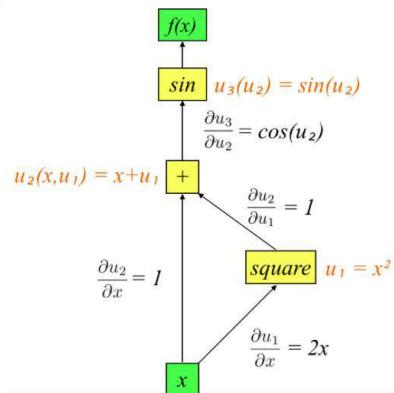
and calculate our partials:

$$\text{Right-side path: } \frac{\partial u_3}{\partial u_2} \frac{\partial u_2}{\partial u_1} \frac{\partial u_1}{\partial x} = \cos(x+x^2) \cdot 1 \cdot 2x = 2x \cos(x+x^2)$$

$$\text{Left-side path: } \frac{\partial u_3}{\partial u_2} \frac{\partial u_2}{\partial x} = \cos(x+x^2) \cdot 1 = \cos(x+x^2)$$

$$\begin{aligned} \text{Total contribution of a change in } x \text{ towards a change in } y \text{ (or } f(x)\text{)}: &= 2x \cos(x+x^2) + \cos(x+x^2) \\ &= \cos(x+x^2)(1+2x) \end{aligned}$$

Therefore, the derivative of  $f(x) = \sin(x+x^2)$  is  $\cos(x+x^2)(1+2x)$ .



How does this relate back to our problem? Remember, we need to find the partial derivative of our loss function with respect to both  $w$  (the vector of all our weights) and  $b$  (the bias). However, our loss function is not that simple — there are multiple nested subexpressions (i.e. multiple intermediate variables) which will require us to use the chain rule.

$$C(\mathbf{y}, \mathbf{w}, \mathbf{X}, b) = \frac{1}{N} \sum_{i=1}^N (y_i - \max(0, \mathbf{w} \cdot \mathbf{X}_i + b))^2$$

There's one more problem left. As you can see, our loss function doesn't just take in scalars as inputs, it takes in vectors as well. How can we compute the partial derivatives of vector equations, and what does a vector chain rule look like?



# To calculate the gradient of this loss function

## Gradient of a Scalar Function

Say that we have a function,  $f(x,y) = 3x^2y$ . Our partial derivatives are:

$$\frac{\partial f(x,y)}{\partial x} = \frac{\partial}{\partial x} 3x^2y = 6yx$$

$$\frac{\partial f(x,y)}{\partial y} = \frac{\partial}{\partial y} 3x^2y = 3x^2$$

If we organize these partials into a horizontal vector, we get the gradient of  $f(x,y)$ , or  $\nabla f(x,y)$ :

$$[\frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y}] = [6yx, 3x^2]$$

What happens when we have two functions? Let's add another function,  $g(x,y) = 2x + y^8$ . The partial derivatives are:

$$\frac{\partial g(x,y)}{\partial x} = \frac{\partial}{\partial x} (2x + y^8) = 2$$

$$\frac{\partial g(x,y)}{\partial y} = \frac{\partial}{\partial y} (2x + y^8) = 8y^7$$

So the gradient of  $g(x,y)$  is:

$$\nabla g(x,y) = [\frac{\partial g(x,y)}{\partial x}, \frac{\partial g(x,y)}{\partial y}] = [2, 8y^7]$$

$6yx$  is the change in  $f(x,y)$  with respect to a change in  $x$ , while  $3x^2$  is the change in  $f(x,y)$  with respect to a change in  $y$ .



# Gradient of a vector function

## Representing Functions

When we have multiple functions with multiple parameters, it's often useful to represent them in a simpler way. We can combine multiple parameters of functions into a single vector argument,  $\mathbf{x}$ , that looks as follows:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

Therefore,  $f(x,y,z)$  will become  $f(x_1, x_2, x_3)$  which becomes  $f(\mathbf{x})$ .

We can also combine multiple functions into a vector, like so:

$$\begin{aligned} y_1 &= f_1(\mathbf{x}) \\ y_2 &= f_2(\mathbf{x}) \\ &\vdots \\ y_m &= f_m(\mathbf{x}) \end{aligned}$$

Now,  $\mathbf{y} = f(\mathbf{x})$  where  $f(\mathbf{x})$  is a vector of  $[f_1(\mathbf{x}), f_2(\mathbf{x}), f_3(\mathbf{x}), \dots, f_n(\mathbf{x})]$

For our previous example with two functions,  $f(x,y) \Rightarrow f(\mathbf{x})$  and  $g(x,y) \Rightarrow g(\mathbf{x})$ .

Here, vector  $\mathbf{x} = [x_1, x_2]$ , where  $x_1=x$ , and  $x_2=y$ . To simplify it even further, we can combine our functions:  $[f(\mathbf{x}), g(\mathbf{x})] = [f_1(\mathbf{x}), f_2(\mathbf{x})] = f(\mathbf{x}) = \mathbf{y}$ .

$$\begin{aligned} y_1 &= f_1(\mathbf{x}) = 3x_1^2 x_2 \\ y_2 &= f_2(\mathbf{x}) = 2x_1 + x_2^8 \end{aligned}$$

Often, the number of functions and the number of variables will be the same, so that a solution exists for each variable.

## Gradient of a Vector Function

Now that we have two functions, how can we find the gradient of both functions? If we organize both of their gradients into a single matrix, we move from vector calculus into matrix calculus. This matrix, and organization of the gradients of multiple functions with multiple variables, is known as the **Jacobian matrix**.

$$J = \begin{bmatrix} \nabla f(x, y) \\ \nabla g(x, y) \end{bmatrix} = \begin{bmatrix} \frac{\partial f(x, y)}{\partial x} & \frac{\partial f(x, y)}{\partial y} \\ \frac{\partial g(x, y)}{\partial x} & \frac{\partial g(x, y)}{\partial y} \end{bmatrix} = \begin{bmatrix} 6yx & 3x^2 \\ 2 & 8y^7 \end{bmatrix}$$

There are multiple ways of representing the Jacobian. This layout, where we stack the gradients vertically, is known as the **numerator layout**, but other papers will use the **denominator layout**, which simply flips it diagonally:

$$\begin{bmatrix} 6yx & 2 \\ 3x^2 & 8y^7 \end{bmatrix}$$

# To calculate the gradient of this loss function

## Gradient of the Identity Function

Let's take the identity function,  $y = f(\mathbf{x}) = \mathbf{x}$ , where  $f_i(\mathbf{x}) = x_i$ , and find its gradient:

$$\begin{aligned} y_1 &= f_1(\mathbf{x}) &= x_1 \\ y_2 &= f_2(\mathbf{x}) &= x_2 \\ &\vdots \\ y_n &= f_n(\mathbf{x}) &= x_n \end{aligned}$$

Since it's an identity function,  $f_1(\mathbf{x}) = x_1$ ,  $f_2(\mathbf{x}) = x_2$ , and so on. Therefore,

$$\begin{aligned} \frac{\partial \mathbf{y}}{\partial \mathbf{x}} &= \begin{bmatrix} \frac{\partial}{\partial \mathbf{x}} f_1(\mathbf{x}) \\ \frac{\partial}{\partial \mathbf{x}} f_2(\mathbf{x}) \\ \vdots \\ \frac{\partial}{\partial \mathbf{x}} f_m(\mathbf{x}) \end{bmatrix} &= \begin{bmatrix} \frac{\partial}{\partial x_1} f_1(\mathbf{x}) & \frac{\partial}{\partial x_2} f_1(\mathbf{x}) & \dots & \frac{\partial}{\partial x_n} f_1(\mathbf{x}) \\ \frac{\partial}{\partial x_1} f_2(\mathbf{x}) & \frac{\partial}{\partial x_2} f_2(\mathbf{x}) & \dots & \frac{\partial}{\partial x_n} f_2(\mathbf{x}) \\ \vdots \\ \frac{\partial}{\partial x_1} f_m(\mathbf{x}) & \frac{\partial}{\partial x_2} f_m(\mathbf{x}) & \dots & \frac{\partial}{\partial x_n} f_m(\mathbf{x}) \end{bmatrix} \\ &= \begin{bmatrix} \frac{\partial}{\partial x_1} x_1 & \frac{\partial}{\partial x_2} x_1 & \dots & \frac{\partial}{\partial x_n} x_1 \\ \frac{\partial}{\partial x_1} x_2 & \frac{\partial}{\partial x_2} x_2 & \dots & \frac{\partial}{\partial x_n} x_2 \\ \vdots \\ \frac{\partial}{\partial x_1} x_n & \frac{\partial}{\partial x_2} x_n & \dots & \frac{\partial}{\partial x_n} x_n \end{bmatrix} \end{aligned}$$

Just as we created our previous Jacobian, we can find the gradients of each scalar function and stack them up vertically to create the Jacobian of the identity function:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \nabla f_1(\mathbf{x}) \\ \nabla f_2(\mathbf{x}) \\ \vdots \\ \nabla f_m(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial \mathbf{x}} f_1(\mathbf{x}) \\ \frac{\partial}{\partial \mathbf{x}} f_2(\mathbf{x}) \\ \vdots \\ \frac{\partial}{\partial \mathbf{x}} f_m(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x_1} f_1(\mathbf{x}) & \frac{\partial}{\partial x_2} f_1(\mathbf{x}) & \dots & \frac{\partial}{\partial x_n} f_1(\mathbf{x}) \\ \frac{\partial}{\partial x_1} f_2(\mathbf{x}) & \frac{\partial}{\partial x_2} f_2(\mathbf{x}) & \dots & \frac{\partial}{\partial x_n} f_2(\mathbf{x}) \\ \vdots \\ \frac{\partial}{\partial x_1} f_m(\mathbf{x}) & \frac{\partial}{\partial x_2} f_m(\mathbf{x}) & \dots & \frac{\partial}{\partial x_n} f_m(\mathbf{x}) \end{bmatrix}$$

The partial derivative of a function with respect to a variable that's not in the function is zero. For example, the partial derivative of  $2x^2$  with respect to  $y$  is 0. In other words,

$$\frac{\partial}{\partial x_j} x_i = 0 \text{ for } j \neq i$$



# To calculate the gradient of this loss function

Therefore, everything not on the diagonal of the Jacobian becomes zero.

Meanwhile, the partial derivative of any variable with respect to itself is 1. For example, the partial derivative of  $x$  with respect to  $x$  is 1. Therefore, the Jacobian becomes:

$$\begin{aligned}
 &= \begin{bmatrix} \frac{\partial}{\partial x_1} x_1 & 0 & \dots & 0 \\ 0 & \frac{\partial}{\partial x_2} x_2 & \dots & 0 \\ & \ddots & & \\ 0 & 0 & \dots & \frac{\partial}{\partial x_n} x_n \end{bmatrix} \\
 &= \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ & \ddots & & \\ 0 & 0 & \dots & 1 \end{bmatrix} \\
 &= I \quad (I \text{ is the identity matrix with ones down the diagonal})
 \end{aligned}$$

## Gradient of Element-Wise Vector Function Combinations

Element-wise binary operators are operations (such as addition  $w+x$  or  $w>x$  which returns a vector of ones and zeros) that applies an operator consecutively, from the first item of both vectors to get the first item of output, then the second item of both vectors to get the second item of output...and so forth.

$$\mathbf{y} = \mathbf{f}(\mathbf{w}) \bigcirc \mathbf{g}(\mathbf{x})$$

Here, the  $\bigcirc$  means any element-wise operator (such as  $+$ ), and not a composition of functions.

So how do you find the gradient of an element-wise operation of two vectors?

Since we have two sets of functions, we need two Jacobians, one representing the gradient with respect to  $\mathbf{x}$  and one with respect to  $\mathbf{w}$ :

$$J_{\mathbf{w}} = \frac{\partial \mathbf{y}}{\partial \mathbf{w}} = \begin{bmatrix} \frac{\partial}{\partial w_1} (f_1(\mathbf{w}) \bigcirc g_1(\mathbf{x})) & \frac{\partial}{\partial w_2} (f_1(\mathbf{w}) \bigcirc g_1(\mathbf{x})) & \dots & \frac{\partial}{\partial w_n} (f_1(\mathbf{w}) \bigcirc g_1(\mathbf{x})) \\ \frac{\partial}{\partial w_1} (f_2(\mathbf{w}) \bigcirc g_2(\mathbf{x})) & \frac{\partial}{\partial w_2} (f_2(\mathbf{w}) \bigcirc g_2(\mathbf{x})) & \dots & \frac{\partial}{\partial w_n} (f_2(\mathbf{w}) \bigcirc g_2(\mathbf{x})) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial}{\partial w_1} (f_n(\mathbf{w}) \bigcirc g_n(\mathbf{x})) & \frac{\partial}{\partial w_2} (f_n(\mathbf{w}) \bigcirc g_n(\mathbf{x})) & \dots & \frac{\partial}{\partial w_n} (f_n(\mathbf{w}) \bigcirc g_n(\mathbf{x})) \end{bmatrix}$$

$$J_{\mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial}{\partial x_1} (f_1(\mathbf{w}) \bigcirc g_1(\mathbf{x})) & \frac{\partial}{\partial x_2} (f_1(\mathbf{w}) \bigcirc g_1(\mathbf{x})) & \dots & \frac{\partial}{\partial x_n} (f_1(\mathbf{w}) \bigcirc g_1(\mathbf{x})) \\ \frac{\partial}{\partial x_1} (f_2(\mathbf{w}) \bigcirc g_2(\mathbf{x})) & \frac{\partial}{\partial x_2} (f_2(\mathbf{w}) \bigcirc g_2(\mathbf{x})) & \dots & \frac{\partial}{\partial x_n} (f_2(\mathbf{w}) \bigcirc g_2(\mathbf{x})) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial}{\partial x_1} (f_n(\mathbf{w}) \bigcirc g_n(\mathbf{x})) & \frac{\partial}{\partial x_2} (f_n(\mathbf{w}) \bigcirc g_n(\mathbf{x})) & \dots & \frac{\partial}{\partial x_n} (f_n(\mathbf{w}) \bigcirc g_n(\mathbf{x})) \end{bmatrix}$$

# To calculate the gradient of this loss function

Most arithmetic operations we'll need are simple ones, so  $f(\mathbf{w})$  is often simply the vector  $\mathbf{w}$ . In other words,  $f_i(w_i) = w_i$ . For example, the operation  $\mathbf{w} + \mathbf{x}$  fits this category as it can be represented as  $f(\mathbf{w}) + g(\mathbf{x})$  where  $f_i(w_i) + g_i(x_i) = w_i + x_i$ .

Under this condition, every element in the two Jacobians simplifies to:

$$J_w : \frac{\partial}{\partial w_j} (f_i(w_i) \circ g_i(x_i)) = \frac{\partial}{\partial w_j} (w_i \circ x_i)$$

$$J_x : \frac{\partial}{\partial x_j} (f_i(w_i) \circ g_i(x_i)) = \frac{\partial}{\partial x_j} (w_i \circ x_i)$$

On the diagonal,  $i=j$ , so a value exists for the partial derivative. Off the diagonal, however,  $i \neq j$ , so the partial derivatives become zero:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{w}} = \begin{bmatrix} \frac{\partial}{\partial w_1} (f_1(w_1) \circ g_1(x_1)) & & & 0 \\ & \frac{\partial}{\partial w_2} (f_2(w_2) \circ g_2(x_2)) & & \\ & & \ddots & \\ 0 & & & \frac{\partial}{\partial w_n} (f_n(w_n) \circ g_n(x_n)) \end{bmatrix}$$

We can represent this more succinctly as:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{w}} = \text{diag} \left( \frac{\partial}{\partial w_1} (f_1(w_1) \circ g_1(x_1)), \frac{\partial}{\partial w_2} (f_2(w_2) \circ g_2(x_2)), \dots, \frac{\partial}{\partial w_n} (f_n(w_n) \circ g_n(x_n)) \right)$$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \text{diag} \left( \frac{\partial}{\partial x_1} (f_1(w_1) \circ g_1(x_1)), \frac{\partial}{\partial x_2} (f_2(w_2) \circ g_2(x_2)), \dots, \frac{\partial}{\partial x_n} (f_n(w_n) \circ g_n(x_n)) \right)$$

Let's try to find the gradient of the function  $\mathbf{w} + \mathbf{x}$ . We know that everything off the diagonal is 0. The values of the partials on the diagonal with respect to  $\mathbf{w}$  and  $\mathbf{x}$  are:

$$\frac{\partial}{\partial w_i} (f_i(w_i) + g_i(x_i)) = \frac{\partial}{\partial w_i} (w_i + x_i) = 1 + 0 = 1$$

$$\frac{\partial}{\partial x_i} (f_i(w_i) + g_i(x_i)) = \frac{\partial}{\partial x_i} (w_i + x_i) = 0 + 1 = 1$$

So both Jacobians have a diagonal of 1. This looks familiar...it's the identity matrix!

# To calculate the gradient of this loss function

Let's try it with multiplication:  $w^*x$ . The values of the partials on the diagonal with respect to  $w$  and  $x$  are:

$$\frac{\partial}{\partial x_i} (f_i(w_i)g_i(x_i)) = \frac{\partial}{\partial x_i} (w_i x_i) = w_i = w$$

$$\frac{\partial}{\partial w_i} (f_i(w_i)g_i(x_i)) = \frac{\partial}{\partial w_i} (w_i x_i) = x_i = x$$

Therefore, the gradient with respect to  $w$  of  $w^*x$  is  $\text{diag}(x)$ , while the gradient with respect to  $x$  of  $w^*x$  is  $\text{diag}(w)$ .

Applying the same steps for subtraction and division, we can sum it all up:

Op	Partial with respect to w
+	$\frac{\partial(w+x)}{\partial w} = \text{diag}(\dots \frac{\partial(w_i+x_i)}{\partial w_i} \dots) = \text{diag}(\vec{1}) = I$
-	$\frac{\partial(w-x)}{\partial w} = \text{diag}(\dots \frac{\partial(w_i-x_i)}{\partial w_i} \dots) = \text{diag}(\vec{1}) = I$
$\otimes$	$\frac{\partial(w \otimes x)}{\partial w} = \text{diag}(\dots \frac{\partial(w_i \times x_i)}{\partial w_i} \dots) = \text{diag}(x)$
$\oslash$	$\frac{\partial(w \oslash x)}{\partial w} = \text{diag}(\dots \frac{\partial(w_i/x_i)}{\partial w_i} \dots) = \text{diag}(\dots \frac{1}{x_i} \dots)$

Op	Partial with respect to x
+	$\frac{\partial(w+x)}{\partial x} = \text{diag}(\dots \frac{\partial(w_i+x_i)}{\partial x_i} \dots) = \text{diag}(-\vec{1}) = -I$
-	$\frac{\partial(w-x)}{\partial x} = \text{diag}(\dots \frac{\partial(w_i-x_i)}{\partial x_i} \dots) = \text{diag}(\vec{1}) = I$
$\otimes$	$\frac{\partial(w \otimes x)}{\partial x} = \text{diag}(\dots \frac{\partial(w_i \times x_i)}{\partial x_i} \dots)$
$\oslash$	$\frac{\partial(w \oslash x)}{\partial x} = \text{diag}(\dots \frac{\partial(w_i/x_i)}{\partial x_i} \dots)$

remember that:  $D \frac{1}{x^n} = Dx^{-n} = -n x^{-n-1} = -\frac{n}{x^{n+1}}$

## Gradient of Vector Sums

One of the most common operations in deep learning is the summation operation. How can we find the gradient of the function  $y=\text{sum}(x)$ ?

$y=\text{sum}(x)$  can also be represented as:

$$y = \sum_i x_i$$

$y=\text{sum}(x)$

Therefore, the gradient can be represented as:

$$\left[ \sum_i \frac{\partial x_i}{\partial x_1}, \sum_i \frac{\partial x_i}{\partial x_2}, \dots, \sum_i \frac{\partial x_i}{\partial x_n} \right]$$

# To calculate the gradient of this loss function

And since the partial derivative of a function with respect to a variable that's not in the function is zero, it can be further simplified as:

$$\nabla y = \left[ \frac{\partial x_1}{\partial x_1}, \frac{\partial x_2}{\partial x_2}, \dots, \frac{\partial x_n}{\partial x_n} \right] = [1, 1, \dots, 1] = \vec{1}^T$$

Gradient of  $y=\text{sum}(x)$

While that is the derivative with respect to  $x$ , the derivative with respect to the scalar  $z$  is simply a number:

$$\begin{aligned}\frac{\partial y}{\partial z} &= \frac{\partial}{\partial z} \sum_{i=1}^n x_i z \\ &= \sum_i \frac{\partial}{\partial z} x_i z \\ &= \sum_i x_i \\ &= \text{sum}(x)\end{aligned}$$

Note that the result is a horizontal vector.

What about the gradient of  $y=\text{sum}(xz)$ ? The only difference is that we multiply every partial with a constant,  $z$ :

$$\begin{aligned}\frac{\partial y}{\partial x} &= \left[ \sum_i \frac{\partial}{\partial x_1} x_i z, \sum_i \frac{\partial}{\partial x_2} x_i z, \dots, \sum_i \frac{\partial}{\partial x_n} x_i z \right] \\ &= \left[ \frac{\partial}{\partial x_1} x_1 z, \frac{\partial}{\partial x_2} x_2 z, \dots, \frac{\partial}{\partial x_n} x_n z \right] \\ &= [z, z, \dots, z]\end{aligned}$$

Gradient of  $y=\text{sum}(xz)$  with respect to  $x$

Gradient of  $y=\text{sum}(xz)$  with respect to  $z$



# To calculate the gradient of this loss function

## Gradient of Chain Rule Vector Function Combinations

Let us take a vector function,  $y = f(x)$ , and find its gradient. Let us define the function as:

$$\begin{bmatrix} y_1(x) \\ y_2(x) \end{bmatrix} = \begin{bmatrix} f_1(x) \\ f_2(x) \end{bmatrix} = \begin{bmatrix} \ln(x^2) \\ \sin(3x) \end{bmatrix}$$

$y = f(x)$

Both  $f_1(x)$  and  $f_2(x)$  are composite functions. Let us introduce intermediate variables for  $f_1(x)$  and  $f_2(x)$ , and rewrite our function:

$$\begin{bmatrix} g_1(x) \\ g_2(x) \end{bmatrix} = \begin{bmatrix} x^2 \\ 3x \end{bmatrix}$$

$$\begin{bmatrix} f_1(g) \\ f_2(g) \end{bmatrix} = \begin{bmatrix} \ln(g_1) \\ \sin(g_2) \end{bmatrix}$$

Now, we can use our multivariable chain rule to compute the derivative of vector  $y$ . Simply compute the derivative of  $f_1(x)$  and  $f_2(x)$ , and place them one above another:

$$\frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial f_1(g)}{\partial x} \\ \frac{\partial f_2(g)}{\partial x} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial g_1} \frac{\partial g_1}{\partial x} + \frac{\partial f_1}{\partial g_2} \frac{\partial g_2}{\partial x} \\ \frac{\partial f_2}{\partial g_1} \frac{\partial g_1}{\partial x} + \frac{\partial f_2}{\partial g_2} \frac{\partial g_2}{\partial x} \end{bmatrix} = \begin{bmatrix} \frac{1}{g_1} 2x + 0 \\ 0 + \cos(g_2) 3 \end{bmatrix} = \begin{bmatrix} \frac{2x}{x^2} \\ 3\cos(3x) \end{bmatrix} = \begin{bmatrix} \frac{2}{x} \\ 3\cos(3x) \end{bmatrix}$$

Voila! We have our gradient. However, we've come to our solution with scalar rules, merely grouping the numbers together into a vector. Is there a way to represent the multivariable chain rule for vectors?

Right now, our gradient is computed with:

$$\begin{bmatrix} \frac{\partial f_1}{\partial g_1} \frac{\partial g_1}{\partial x} + \frac{\partial f_1}{\partial g_2} \frac{\partial g_2}{\partial x} \\ \frac{\partial f_2}{\partial g_1} \frac{\partial g_1}{\partial x} + \frac{\partial f_2}{\partial g_2} \frac{\partial g_2}{\partial x} \end{bmatrix}$$



# To calculate the gradient of this loss function

Notice that the first term of the gradients of both  $f_1(x)$  and  $f_2(x)$  include the partial of  $g_1$  over  $x$ , and the second term of the gradients of both  $f_1(x)$  and  $f_2(x)$  include the partial of  $g_2$  over  $x$ . This is just like matrix multiplication! We can therefore represent it as:

$$\begin{bmatrix} \frac{\partial f_1}{\partial g_1} \frac{\partial g_1}{\partial x} + \frac{\partial f_1}{\partial g_2} \frac{\partial g_2}{\partial x} \\ \frac{\partial f_2}{\partial g_1} \frac{\partial g_1}{\partial x} + \frac{\partial f_2}{\partial g_2} \frac{\partial g_2}{\partial x} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial g_1} & \frac{\partial f_1}{\partial g_2} \\ \frac{\partial f_2}{\partial g_1} & \frac{\partial f_2}{\partial g_2} \end{bmatrix} \begin{bmatrix} \frac{\partial g_1}{\partial x} \\ \frac{\partial g_2}{\partial x} \end{bmatrix} = \frac{\partial \mathbf{f}}{\partial \mathbf{g}} \frac{\partial \mathbf{g}}{\partial x}$$

Let's test our this new representation of the vector chain rule:

$$\frac{\partial \mathbf{f}}{\partial \mathbf{g}} \frac{\partial \mathbf{g}}{\partial x} = \begin{bmatrix} \frac{1}{g_1} & 0 \\ 0 & \cos(g_2) \end{bmatrix} \begin{bmatrix} 2x \\ 3 \end{bmatrix} = \begin{bmatrix} \frac{1}{g_1} 2x + 0 \\ 0 + \cos(g_2) 3 \end{bmatrix} = \begin{bmatrix} \frac{2}{x} \\ 3\cos(3x) \end{bmatrix}$$

We get the same answer as the scalar approach! If instead of a single parameter  $x$  we have a vector parameter  $\mathbf{x}$ , we just have to alter our rule a little to get the complete vector chain rule:

$$\frac{\partial}{\partial \mathbf{x}} \mathbf{f}(\mathbf{g}(\mathbf{x})) = \frac{\partial \mathbf{f}}{\partial \mathbf{g}} \frac{\partial \mathbf{g}}{\partial \mathbf{x}}$$

In other words:

$$\frac{\partial}{\partial \mathbf{x}} \mathbf{f}(\mathbf{g}(\mathbf{x})) = \begin{bmatrix} \frac{\partial f_1}{\partial g_1} & \frac{\partial f_1}{\partial g_2} & \cdots & \frac{\partial f_1}{\partial g_k} \\ \frac{\partial f_2}{\partial g_1} & \frac{\partial f_2}{\partial g_2} & \cdots & \frac{\partial f_2}{\partial g_k} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial g_1} & \frac{\partial f_m}{\partial g_2} & \cdots & \frac{\partial f_m}{\partial g_k} \end{bmatrix} \begin{bmatrix} \frac{\partial g_1}{\partial x_1} & \frac{\partial g_1}{\partial x_2} & \cdots & \frac{\partial g_1}{\partial x_n} \\ \frac{\partial g_2}{\partial x_1} & \frac{\partial g_2}{\partial x_2} & \cdots & \frac{\partial g_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial g_k}{\partial x_1} & \frac{\partial g_k}{\partial x_2} & \cdots & \frac{\partial g_k}{\partial x_n} \end{bmatrix}$$

In our example above,  $f$  is purely a function of  $g$ ; that is,  $f_i$  is a function of  $g_i$  but not  $g_j$  (each function  $f$  matches with exactly 1 function  $g$ ). In this case, everything off the diagonal becomes zero, and:

$$\begin{aligned} \frac{\partial \mathbf{f}}{\partial \mathbf{g}} &= \text{diag}\left(\frac{\partial f_i}{\partial g_i}\right) \\ \frac{\partial \mathbf{g}}{\partial \mathbf{x}} &= \text{diag}\left(\frac{\partial g_i}{\partial x_i}\right) \end{aligned}$$

Now we have all the pieces we find the gradient of the neural network we started

$$C(\mathbf{y}, \mathbf{w}, \mathbf{X}, b) = \frac{1}{N} \sum_{i=1}^N (y_i - \max(0, \mathbf{w} \cdot \mathbf{X}_i + b))^2$$

# To calculate the gradient of this loss function

## Gradient of A Neuron

We need to approach this problem step by step. Let's first find the gradient of a single neuron with respect to the weights and biases.

The function of our neuron (complete with an activation) is:

$$\text{neuron}(\mathbf{x}) = \max(0, \mathbf{w} \cdot \mathbf{x} + b)$$

Where it takes  $\mathbf{x}$  as an input, multiplies it with weight  $\mathbf{w}$ , and adds a bias  $b$ .

This function is really a composition of other functions. If we let  $f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b$ , and  $g(x) = \max(0, x)$ , then our function is  $\text{neuron}(\mathbf{x}) = g(f(\mathbf{x}))$ . We can use the vector chain rule to find the derivative of this composition of functions!

The derivative of our neuron is simply:

$$\frac{\partial \text{neuron}}{\partial \mathbf{w}} = \frac{\partial \text{neuron}}{\partial z} \frac{\partial z}{\partial \mathbf{w}}$$

Where  $z = f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b$ .

There are two parts to this derivative: the partial of  $z$  with respect to  $\mathbf{w}$ , and the partial of  $\text{neuron}(z)$  with respect to  $z$ .

## What is the partial derivative of $z$ with respect to $\mathbf{w}$ ?

There are two parts to  $z$ :  $\mathbf{w} \cdot \mathbf{x}$  and  $+b$ . Let's look at  $\mathbf{w} \cdot \mathbf{x}$  first.

$\mathbf{w} \cdot \mathbf{x}$ , or the dot product, is really just a summation of the element-wise multiplication of every element in the vector. In other words:

$$\sum_i^n (w_i x_i) = \text{sum}(\mathbf{w} \otimes \mathbf{x})$$

This is once again a composition of functions, so we can write  $\mathbf{v} = \mathbf{w} \otimes \mathbf{x}$  and  $u = \text{sum}(\mathbf{v})$ . We're trying to find the derivative of  $u$  with respect to  $\mathbf{w}$ . We've learned about both of these functions — element-wise multiplication and summation.

$$\frac{\partial \mathbf{v}}{\partial \mathbf{w}} = \frac{\partial}{\partial \mathbf{w}} (\mathbf{w} \otimes \mathbf{x}) = \text{diag}(\mathbf{x})$$

$$\frac{\partial u}{\partial \mathbf{v}} = \frac{\partial}{\partial \mathbf{v}} \text{sum}(\mathbf{v}) = \mathbf{1}^{T \rightarrow}$$

see page 219

see page 220

# To calculate the gradient of this loss function

Therefore, by the vector chain rule:

$$\frac{\partial u}{\partial \mathbf{w}} = \frac{\partial u}{\partial \mathbf{v}} \frac{\partial \mathbf{v}}{\partial \mathbf{w}} = \mathbf{1}^T \text{diag}(\mathbf{x}) = \mathbf{x}^T$$

That's it! Now, let's find the derivative of  $z = u + b$  where  $u = \mathbf{w} \cdot \mathbf{x}$  with respect to both the weights  $\mathbf{w}$  and the bias  $b$ . Remember that the derivative of a function with respect to a variable not in that function is zero, so:

$$\frac{\partial z}{\partial \mathbf{w}} = \frac{\partial}{\partial \mathbf{w}} u + \frac{\partial}{\partial \mathbf{w}} b = \mathbf{x}^T + \mathbf{0}^T = \mathbf{x}^T$$

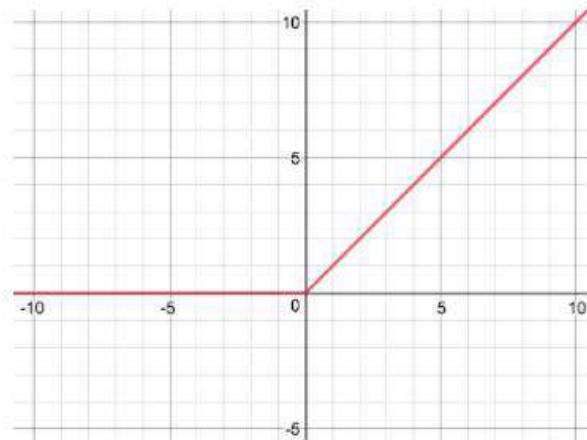
$$\frac{\partial z}{\partial b} = \frac{\partial}{\partial b} u + \frac{\partial}{\partial b} b = \frac{\partial}{\partial b} \mathbf{w} \cdot \mathbf{x} + \frac{\partial}{\partial b} b = \mathbf{0} + 1 = 1$$

That's it! Those two are the derivatives of  $u$  with respect to both the weights and biases.

What is the partial derivative of  $\text{neuron}(z)$  with respect to  $z$ ?

$$\text{Neuron}(z) = \max(0, z) = \max(0, \sum(\mathbf{w} \otimes \mathbf{x}) + b).$$

The  $\max(0, z)$  function simply treats all negative values as 0. The graph would thus look something like this:



Looking at that graph, we can immediately see that the derivative is a piecewise function: it's 0 for all values of  $z$  less than or equal to 0, and 1 for all values of  $z$  greater than 0, or:

$$\text{neuron}(\mathbf{x}) = \max(0, \mathbf{w} \cdot \mathbf{x} + b)$$

$$\frac{\partial}{\partial z} \max(0, z) = \begin{cases} 0 & z \leq 0 \\ \frac{dz}{dz} = 1 & z > 0 \end{cases}$$

# To calculate the gradient of this loss function

Now that we have both parts, we can multiply them together to get the derivative of our neuron:

$$\frac{\partial \text{neuron}}{\partial \mathbf{w}} = \frac{\partial \text{neuron}}{\partial z} \frac{\partial z}{\partial \mathbf{w}}$$

$$= \begin{cases} 0 \frac{\partial z}{\partial \mathbf{w}} = \vec{0}^T & z \leq 0 \\ 1 \frac{\partial z}{\partial \mathbf{w}} = \frac{\partial z}{\partial \mathbf{w}} = \mathbf{x}^T & z > 0 \end{cases}$$

And substitute  $z = \mathbf{w} \cdot \mathbf{x} + b$  back in:

$$= \begin{cases} \vec{0}^T & \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ \mathbf{x}^T & \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

$$\frac{\partial \text{neuron}}{\partial b} = \frac{\partial \text{neuron}}{\partial z} \frac{\partial z}{\partial b}$$

Voilà! We have our derivative for a neuron with respect to its weights! Similarly, we can use the same steps for the bias:

$$\frac{\partial \text{neuron}}{\partial b} = \begin{cases} 0 \frac{\partial z}{\partial b} = 0 & \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1 \frac{\partial z}{\partial b} = 1 & \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

And there you go! We now have the gradient of a neuron in our neural network!

## Gradient of Loss Function

$$C(\mathbf{y}, \mathbf{w}, \mathbf{X}, b) = \frac{1}{N} \sum_{i=1}^N (y_i - \max(0, \mathbf{w} \cdot \mathbf{X}_i + b))^2$$

We can immediately identify this as a composition of functions, which require the chain rule. We'll define our intermediate variables as:

$$\begin{aligned} u(\mathbf{w}, b, \mathbf{x}) &= \max(0, \mathbf{w} \cdot \mathbf{x} + b) \\ v(y, u) &= y - u \\ C(v) &= \frac{1}{N} \sum_{i=1}^N v^2 \end{aligned}$$

# To calculate the gradient of this loss function

Let's compute the gradient with respect to the weights  $w$  first.

## Gradient With Respect to Weights

$u$  is simply our neuron function, which we solved earlier. Therefore:

$$\frac{\partial}{\partial w} u(w, b, x) = \begin{cases} \vec{0}^T & w \cdot x + b \leq 0 \\ x^T & w \cdot x + b > 0 \end{cases}$$

$v(y, u)$  is simply  $y - u$ . Therefore, we can find its derivative (with respect to  $w$ ) using the distributive property and substituting in the derivative of  $u$ :

$$\frac{\partial v(y, u)}{\partial w} = \frac{\partial}{\partial w} (y - u) = \vec{0}^T - \frac{\partial u}{\partial w} = -\frac{\partial u}{\partial w} = \begin{cases} \vec{0}^T & w \cdot x + b \leq 0 \\ -x^T & w \cdot x + b > 0 \end{cases}$$

Finally, we need to find the derivative of the whole cost function with respect to  $w$ . Using the chain rule, we know that:

$$\frac{\partial C(v)}{\partial w} = \frac{\partial C}{\partial v} \frac{\partial v}{\partial w}$$

Let's find the first part of that equation, the partial of  $C(v)$  with respect to  $v$  first:

$$\frac{\partial C(v)}{\partial v} = \frac{\partial}{\partial v} \frac{1}{N} \sum_{i=1}^N v^2 = \frac{1}{N} \sum_{i=1}^N \frac{\partial}{\partial v} v^2 = \frac{1}{N} \sum_{i=1}^N 2v$$

we know the derivative of  $v$  with respect to  $w$ . To find the partial of  $C(v)$ , we multiply the two derivatives together:

$$\frac{1}{N} \sum_{i=1}^N \begin{cases} 2v \vec{0}^T = \vec{0}^T & w \cdot x_i + b \leq 0 \\ -2v x^T & w \cdot x_i + b > 0 \end{cases}$$

Now, substitute  $y - u$  for  $v$ , and  $\max(0, w \cdot x + b)$  for  $u$ :

$$\begin{aligned} &= \frac{1}{N} \sum_{i=1}^N \begin{cases} \vec{0}^T & w \cdot x_i + b \leq 0 \\ -2(y_i - u)x_i^T & w \cdot x_i + b > 0 \end{cases} \\ &= \frac{1}{N} \sum_{i=1}^N \begin{cases} \vec{0}^T & w \cdot x_i + b \leq 0 \\ -2(y_i - \max(0, w \cdot x_i + b))x_i^T & w \cdot x_i + b > 0 \end{cases} \end{aligned}$$

# To calculate the gradient of this loss function

Since the `max` function is on the second line of our piecewise function, where  $\mathbf{w} \cdot \mathbf{x} + b$  is greater than 0, the `max` function will always simply output the value of  $\mathbf{w} \cdot \mathbf{x} + b$ :

$$= \frac{1}{N} \sum_{i=1}^N \begin{cases} \vec{0}^T & \mathbf{w} \cdot \mathbf{x}_i + b \leq 0 \\ -2(y_i - \max(0, \mathbf{w} \cdot \mathbf{x}_i + b))\mathbf{x}_i^T & \mathbf{w} \cdot \mathbf{x}_i + b > 0 \end{cases}$$

$$= \frac{1}{N} \sum_{i=1}^N \begin{cases} \vec{0}^T & \mathbf{w} \cdot \mathbf{x}_i + b \leq 0 \\ -2(y_i - (\mathbf{w} \cdot \mathbf{x}_i + b))\mathbf{x}_i^T & \mathbf{w} \cdot \mathbf{x}_i + b > 0 \end{cases}$$

Finally, we can move the summation inside our piecewise function and tidy it up a little:

$$= \begin{cases} \vec{0}^T & \mathbf{w} \cdot \mathbf{x}_i + b \leq 0 \\ \frac{-2}{N} \sum_{i=1}^N (y_i - (\mathbf{w} \cdot \mathbf{x}_i + b))\mathbf{x}_i^T & \mathbf{w} \cdot \mathbf{x}_i + b > 0 \end{cases}$$

$$= \begin{cases} \vec{0}^T & \mathbf{w} \cdot \mathbf{x}_i + b \leq 0 \\ \frac{2}{N} \sum_{i=1}^N (\mathbf{w} \cdot \mathbf{x}_i + b - y_i)\mathbf{x}_i^T & \mathbf{w} \cdot \mathbf{x}_i + b > 0 \end{cases}$$

That's it! We have our derivative with respect to the weights! However, what does this mean?

$\mathbf{w} \cdot \mathbf{x} + b - y$  can be interpreted as an error term — the difference between the predicted output of the neural network and the actual output. If we call this error term  $e_i$ , our final derivative is:

$$\frac{\partial C}{\partial \mathbf{w}} = \frac{2}{N} \sum_{i=1}^N e_i \mathbf{x}_i^T$$

Here, the greater the error, the higher the derivative. In other words, the derivative represents the slope, or how much we have to move our weights by in order to minimize our error. If our neural network has just begun training, and has a very low accuracy, the error will be high and thus the derivative will be large as well. Therefore, we will have to take a big step in order to minimize our error.

You might notice that this gradient is pointing in the direction of higher cost, meaning we cannot add the gradient to our current weights — that will only increase the error and take us a step away from the local minimum. Therefore, we must subtract our current weights with the derivative in order to get one step closer to minimizing our loss function:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \frac{\partial C}{\partial \mathbf{w}}$$

Here,  $\eta$  represents the learning rate, which we as programmers can set. The larger the learning rate, the bigger the step. However, setting a too-large learning rate may result in taking too big a step and spiraling out of the local minimum

# To calculate the gradient of this loss function

## Gradient With Respect to Bias

Once again, we have our intermediate variables:

$$\begin{aligned} u(\mathbf{w}, b, \mathbf{x}) &= \max(0, \mathbf{w} \cdot \mathbf{x} + b) \\ v(y, u) &= y - u \\ C(v) &= \frac{1}{N} \sum_{i=1}^N v^2 \end{aligned}$$

We also have the value of the derivative of  $u$  with respect to the bias that we calculated previously:

$$\frac{\partial u}{\partial b} = \begin{cases} 0 & \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1 & \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

Similarly, we can find the derivative of  $v$  with respect to  $b$  using the distributive property and substituting in the derivative of  $u$ :

$$\frac{\partial v(y, u)}{\partial b} = \frac{\partial}{\partial b}(y - u) = 0 - \frac{\partial u}{\partial b} = -\frac{\partial u}{\partial b} = \begin{cases} 0 & \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ -1 & \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

Again, we can use the vector chain rule to find the derivative of  $C$ :

$$\frac{\partial C(v)}{\partial b} = \frac{\partial C}{\partial v} \frac{\partial v}{\partial b}$$

The derivative of  $C$  with respect to  $v$  is identical to the one we calculated for the weights:

$$\frac{\partial C(v)}{\partial v} = \frac{\partial}{\partial v} \frac{1}{N} \sum_{i=1}^N v^2 = \frac{1}{N} \sum_{i=1}^N \frac{\partial}{\partial v} v^2 = \frac{1}{N} \sum_{i=1}^N 2v$$

Multiplying the two together to find the derivative of  $C$  with respect to  $b$ , and substituting in  $y - u$  for  $v$ , and  $\max(0, \mathbf{w} \cdot \mathbf{x} + b)$  for  $u$ , we get:

$$\begin{aligned} &= \frac{1}{N} \sum_{i=1}^N \begin{cases} 0 & \mathbf{w} \cdot \mathbf{x}_i + b \leq 0 \\ -2v_i & \mathbf{w} \cdot \mathbf{x}_i + b > 0 \end{cases} \\ &= \frac{1}{N} \sum_{i=1}^N \begin{cases} 0 & \mathbf{w} \cdot \mathbf{x}_i + b \leq 0 \\ -2(y_i - \max(0, \mathbf{w} \cdot \mathbf{x}_i + b)) & \mathbf{w} \cdot \mathbf{x}_i + b > 0 \end{cases} \end{aligned}$$

# To calculate the gradient of this loss function

Once again, because the second line explicitly states that  $\mathbf{w} \cdot \mathbf{x} + b > 0$ , the *max* function will always simply be the value of  $\mathbf{w} \cdot \mathbf{x} + b$ .

$$\begin{aligned} &= \frac{1}{N} \sum_{i=1}^N \begin{cases} 0 & \mathbf{w} \cdot \mathbf{x}_i + b \leq 0 \\ 2(\mathbf{w} \cdot \mathbf{x}_i + b - y_i) & \mathbf{w} \cdot \mathbf{x}_i + b > 0 \end{cases} \\ &= \begin{cases} 0 & \mathbf{w} \cdot \mathbf{x}_i + b \leq 0 \\ \frac{2}{N} \sum_{i=1}^N (\mathbf{w} \cdot \mathbf{x}_i + b - y_i) & \mathbf{w} \cdot \mathbf{x}_i + b > 0 \end{cases} \end{aligned}$$

Just like the derivative with respect to the weights, the magnitude of this gradient is also proportional to the error: the bigger the error, the larger step towards the local minimum we have to take. It is also pointing towards the direction of higher cost, meaning that we have to subtract the gradient from our current value to get one step closer to the local minimum:

$$b_{t+1} = b_t - \eta \frac{\partial C}{\partial b}$$

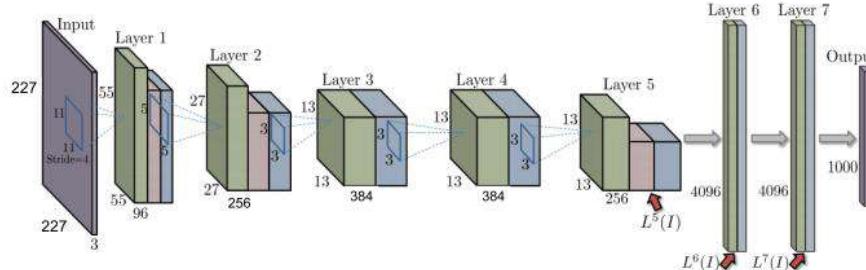
Just like before, we can substitute in an error term,  $e = \mathbf{w} \cdot \mathbf{x} + b - y$ :

$$\frac{\partial C}{\partial b} = \frac{2}{N} \sum_{i=1}^N e_i$$



# CaffeNet. CaffeNet is a variant of AlexNet.

[1] Babenko et al., Neural Codes for Image Retrieval, 2014.



Color codes

Purple: Input (227x227x3 images) and Output (it has been trained using the 1000 ImageNet classes)

Green: Convolution

Red: Pooling (max)

Blue: Relu activation

Layer 6, 7 and 8 (named output in the figure): Fully connected

Layer 8: Softmax

Stride: 4 for the first level of convolution, then always 1

Filters: Scaled dimensions: from 11x11 to 3x3

L5, L6, L7, denote reusable features for other problems (see transfer-learning and [1]).

Total number of parameters: about 60M

Some links for visual / demo explanations:

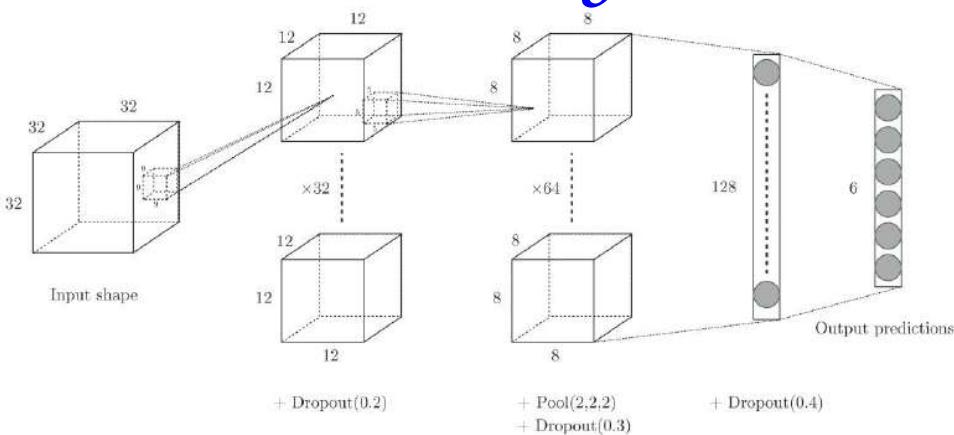
<https://hackernoon.com/visualizing-parts-of-convolutional-neural-networks-using-keras-and-cats-5cc01b214e59>

<https://poloclub.github.io/cnn-explainer/>

<http://cs231n.github.io/convolutional-networks>

input volumes of size  $32 \times 32 \times 32$ , containing the structural coordinates of the enzymes encoded in voxels, first go through a convolutional layer of 32 filters of size  $9 \times 9 \times 9$  with stride 2. Then, a second convolutional layer of 64 filters of size  $5 \times 5 \times 5$  with stride 1 is used, followed by a max-pooling layer of size  $2 \times 2 \times 2$  with stride 2. Finally, there are two fully-connected layers of 128 and six (the number of classes) hidden units respectively, concluded by a softmax layer that outputs class probabilities

Stride also exists along the z axis, the z dimension of the filter may not necessarily be the same as the input.



# GoogleNet

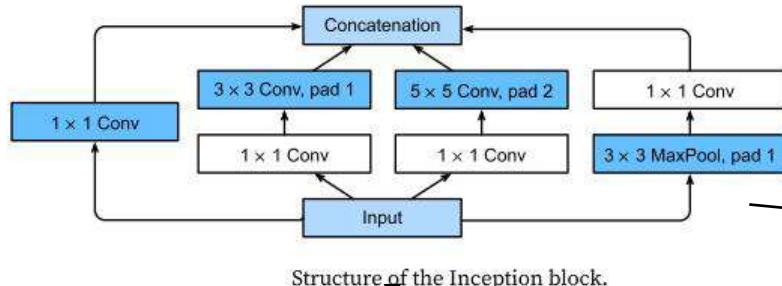
In 2014, GoogLeNet won the ImageNet Challenge, proposing a structure that combined the strengths of NiN and paradigms of repeated blocks. One focus of the paper was to address the question of which size convolution kernels are best. After all, previous popular networks employed choices as small as  $1 \times 1$  and as large as  $11 \times 11$ . One insight in this paper was that sometimes it can be advantageous to employ a combination of variously-sized kernels. In this section, we will introduce GoogLeNet, presenting a slightly simplified version of the original model: we omit a few ad-hoc features that were added to stabilize training but are unnecessary now with better training algorithms available.

## Inception Blocks

<https://arxiv.org/abs/1409.4842>

Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. CoRR, abs/1312.4400, 2013

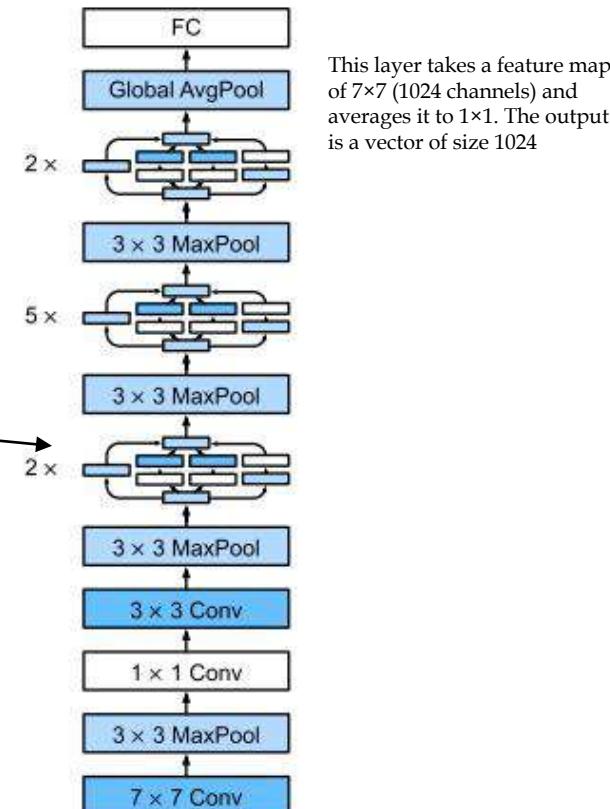
The basic convolutional block in GoogLeNet is called an *Inception block*, likely named due to a quote from the movie *Inception* ("We need to go deeper"), which launched a viral meme.



the inception block consists of four parallel paths. The first three paths use convolutional layers with window sizes of  $1 \times 1$ ,  $3 \times 3$ , and  $5 \times 5$  to extract information from different spatial sizes. The middle two paths perform a  $1 \times 1$  convolution on the input to reduce the number of channels, reducing the model's complexity. The fourth path uses a  $3 \times 3$  maximum pooling layer, followed by a  $1 \times 1$  convolutional layer to change the number of channels. The four paths all use appropriate padding to give the input and output the same height and width. Finally, the outputs along each path are concatenated along the channel dimension and comprise the block's output.

$1 \times 1$  convolutional layer is often called feature map pooling or projection layer.  
<https://machinelearningmastery.com/introduction-to-1x1-convolutions-to-reduce-the-complexity-of-convolutional-neural-networks/>

To gain some intuition for why this network works so well, consider the combination of the filters. They explore the image in a variety of filter sizes. This means that details at different extents can be recognized efficiently by filters of different sizes. At the same time, we can allocate different amounts of parameters for different filters.



The GoogLeNet architecture.

# ResNet - Batch Normalization

To motivate batch normalization, let us review a few practical challenges that arise when training machine learning models and neural networks in particular.

for a typical MLP or CNN, as we train, the variables (e.g., affine transformation outputs in MLP) in intermediate layers may take values with widely varying magnitudes: both along the layers from the input to the output, across units in the same layer, and over time due to our updates to the model parameters. The inventors of batch normalization postulated informally that this drift in the distribution of such variables could hamper the convergence of the network. Intuitively, we might conjecture that if one layer has variable values that are 100 times that of another layer, this might necessitate compensatory adjustments in the learning rates.

deeper networks are complex and easily capable of overfitting. This means that regularization becomes more critical.

Batch normalization is applied to individual layers (optionally, to all of them) and works as follows: In each training iteration, we first normalize the inputs (of batch normalization) by subtracting their mean and dividing by their standard deviation, where both are estimated based on the statistics of the current minibatch. Next, we apply a scale coefficient and a scale offset. It is precisely due to this *normalization* based on *batch* statistics that *batch normalization* derives its name.

Note that if we tried to apply batch normalization with minibatches of size 1, we would not be able to learn anything. That is because after subtracting the means, each hidden unit would take value 0! As you might guess, since we are devoting a whole section to batch normalization, with large enough minibatches, the approach proves effective and stable. One takeaway here is that when applying batch normalization, the choice of batch size may be even more significant than without batch normalization.



# ResNet

Formally, denoting by  $\mathbf{x} \in \mathcal{B}$  an input to batch normalization (BN) that is from a minibatch  $\mathcal{B}$ , batch normalization transforms  $\mathbf{x}$  according to the following expression:

$$\text{BN}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \hat{\mu}_{\mathcal{B}}}{\hat{\sigma}_{\mathcal{B}}} + \beta.$$

$\hat{\mu}_{\mathcal{B}}$  is the sample mean and  $\hat{\sigma}_{\mathcal{B}}$  is the sample standard deviation of the minibatch  $\mathcal{B}$ . After applying standardization, the resulting minibatch has zero mean and unit variance. Because the choice of unit variance (vs. some other magic number) is an arbitrary choice, we commonly include elementwise *scale parameter*  $\gamma$  and *shift parameter*  $\beta$  that have the same shape as  $\mathbf{x}$ . Note that  $\gamma$  and  $\beta$  are parameters that need to be learned jointly with the other model parameters.

Consequently, the variable magnitudes for intermediate layers cannot diverge during training because batch normalization actively centers and rescales them back to a given mean and size (via  $\hat{\mu}_{\mathcal{B}}$  and  $\hat{\sigma}_{\mathcal{B}}$ ). One piece of practitioner's intuition or wisdom is that batch normalization seems to allow for more aggressive learning rates.

Formally, we calculate  $\hat{\mu}_{\mathcal{B}}$  and  $\hat{\sigma}_{\mathcal{B}}$  in as follows:

$$\begin{aligned}\hat{\mu}_{\mathcal{B}} &= \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} \mathbf{x}, \\ \hat{\sigma}_{\mathcal{B}}^2 &= \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} (\mathbf{x} - \hat{\mu}_{\mathcal{B}})^2 + \epsilon.\end{aligned}$$

the parameters are chosen for each tensor channel, shared among all the minibatches

Note that we add a small constant  $\epsilon > 0$  to the variance estimate to ensure that we never attempt division by zero, even in cases where the empirical variance estimate might vanish. The estimates  $\hat{\mu}_{\mathcal{B}}$  and  $\hat{\sigma}_{\mathcal{B}}$  counteract the scaling issue by using noisy estimates of mean and variance. You might think that this noisiness should be a problem. As it turns out, this is actually beneficial.



# ResNet

This turns out to be a recurring theme in deep learning. For reasons that are not yet well-characterized theoretically, various sources of noise in optimization often lead to faster training and less overfitting: this variation appears to act as a form of regularization.

Fixing a trained model, you might think that we would prefer using the entire dataset to estimate the mean and variance. Once training is complete, why would we want the same image to be classified differently, depending on the batch in which it happens to reside? During training, such exact calculation is infeasible because the intermediate variables for all data examples change every time we update our model. However, once the model is trained, we can calculate the means and variances of each layer's variables based on the entire dataset. Indeed this is standard practice for models employing batch normalization and thus batch normalization layers function differently in *training mode* (normalizing by minibatch statistics) and in *prediction mode* (normalizing by dataset statistics).



# ResNet

## Batch Normalization Layers

Batch normalization implementations for fully-connected layers and convolutional layers are slightly different. We discuss both cases below. Recall that one key difference between batch normalization and other layers is that because batch normalization operates on a full minibatch at a time, we cannot just ignore the batch dimension as we did before when introducing other layers.

### Fully-Connected Layers

When applying batch normalization to fully-connected layers, the original paper inserts batch normalization after the affine transformation and before the nonlinear activation function (later applications may insert batch normalization right after activation functions)

Denoting the input to the fully-connected layer by  $\mathbf{x}$ , the affine transformation by  $\mathbf{Wx} + \mathbf{b}$  (with the weight parameter  $\mathbf{W}$  and the bias parameter  $\mathbf{b}$ ), and the activation function by  $\phi$ , we can express the computation of a batch-normalization-enabled, fully-connected layer output  $\mathbf{h}$  as follows:

$$\mathbf{h} = \phi(\text{BN}(\mathbf{Wx} + \mathbf{b})).$$

Recall that mean and variance are computed on the *same* minibatch on which the transformation is applied.

### Convolutional Layers

Similarly, with convolutional layers, we can apply batch normalization after the convolution and before the nonlinear activation function. When the convolution has multiple output channels, we need to carry out batch normalization for *each* of the outputs of these channels, and each channel has its own scale and shift parameters, both of which are scalars. Assume that our minibatches contain  $m$  examples and that for each channel, the output of the convolution has height  $p$  and width  $q$ . For convolutional layers, we carry out each batch normalization over the  $m \cdot p \cdot q$  elements per output channel simultaneously. Thus, we collect the values over all spatial locations when computing the mean and variance and consequently apply the same mean and variance within a given channel to normalize the value at each spatial location.

## Batch Normalization During Prediction

As we mentioned earlier, batch normalization typically behaves differently in training mode and prediction mode. First, the noise in the sample mean and the sample variance arising from estimating each on minibatches are no longer desirable once we have trained the model. Second, we might not have the luxury of computing per-batch normalization statistics. For example, we might need to apply our model to make one prediction at a time.

Typically, after training, we use the entire dataset to compute stable estimates of the variable statistics and then fix them at prediction time. Consequently, batch normalization behaves differently during training and at test time. Recall that dropout also exhibits this characteristic.

In the prediction step (when the test set is classified), the normalization is performed by estimating the mean and the variance using the whole training data.



# ResNet

Intuitively, batch normalization is thought to make the optimization landscape smoother. However, we must be careful to distinguish between speculative intuitions and true explanations for the phenomena that we observe when training deep models. Recall that we do not even know why simpler deep neural networks (MLPs and conventional CNNs) generalize well in the first place. Even with dropout and weight decay, they remain so flexible that their ability to generalize to unseen data cannot be explained via conventional learning-theoretic generalization guarantees.

In the original paper proposing batch normalization, the authors, in addition to introducing a powerful and useful tool, offered an explanation for why it works: by reducing *internal covariate shift*. Presumably by *internal covariate shift* the authors meant something like the intuition expressed above—the notion that the distribution of variable values changes over the course of training. However, there were two problems with this explanation: i) This drift is very different from *covariate shift*, rendering the name a misnomer. ii) The explanation offers an under-specified intuition but leaves the question of *why precisely this technique works* an open question wanting for a rigorous explanation.

<https://arxiv.org/abs/1807.03341>

Following the success of batch normalization, its explanation in terms of *internal covariate shift* has repeatedly surfaced in debates in the technical literature and broader discourse about how to present machine learning research. In a memorable speech given while accepting a Test of Time Award at the 2017 NeurIPS conference, Ali Rahimi used *internal covariate shift* as a focal point in an argument likening the modern practice of deep learning to alchemy. Subsequently, the example was revisited in detail in a position paper outlining troubling trends in machine learning (Lipton & Steinhardt, 2018). Other authors have proposed alternative explanations for the success of batch normalization, some claiming that batch normalization’s success comes despite exhibiting behavior that is in some ways opposite to those claimed in the original paper (Santurkar et al., 2018) ➔ <https://arxiv.org/abs/1805.11604>

We note that the *internal covariate shift* is no more worthy of criticism than any of thousands of similarly vague claims made every year in the technical machine learning literature. Likely, its resonance as a focal point of these debates owes to its broad recognizability to the target audience. Batch normalization has proven an indispensable method, applied in nearly all deployed image classifiers, earning the paper that introduced the technique tens of thousands of citations.

As just pointed out, it is not entirely clear, from a theoretical point of view, why batch normalization is useful for training a CNN. Here some heuristic explanations are detailed, but none have a strong mathematical proof



# ResNet

As we design increasingly deeper networks it becomes imperative to understand how adding layers can increase the complexity and expressiveness of the network. Even more important is the ability to design networks where adding layers makes networks strictly more expressive rather than just different. To make some progress we need a bit of mathematics.

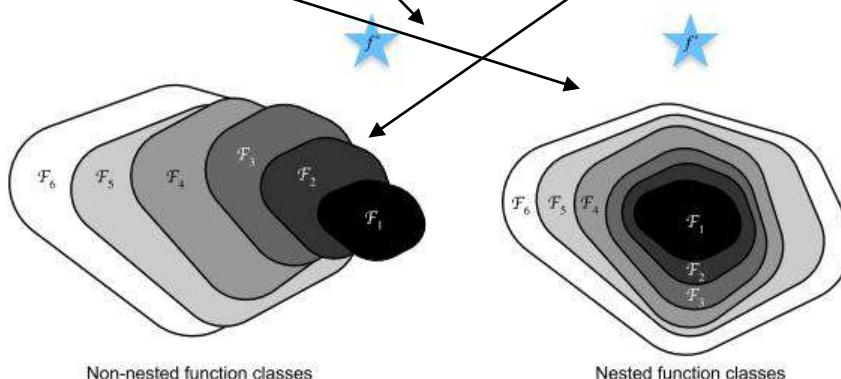
## Function Classes

Consider  $\mathcal{F}$ , the class of functions that a specific network architecture (together with learning rates and other hyperparameter settings) can reach. That is, for all  $f \in \mathcal{F}$  there exists some set of parameters (e.g., weights and biases) that can be obtained through training on a suitable dataset. Let us assume that  $f^*$  is the “truth” function that we really would like to find. If it is in  $\mathcal{F}$ , we are in good shape but typically we will not be quite so lucky. Instead, we will try to find some  $f_{\mathcal{F}}^*$  which is our best bet within  $\mathcal{F}$ . For instance, given a dataset with features  $\mathbf{X}$  and labels  $\mathbf{y}$ , we might try finding it by solving the following optimization problem:

$$f_{\mathcal{F}}^* \stackrel{\text{def}}{=} \underset{f}{\operatorname{argmin}} L(\mathbf{X}, \mathbf{y}, f) \text{ subject to } f \in \mathcal{F}.$$

It is only reasonable to assume that if we design a different and more powerful architecture  $\mathcal{F}'$  we should arrive at a better outcome. In other words, we would expect that  $f_{\mathcal{F}'}^*$  is “better” than  $f_{\mathcal{F}}^*$ . However, if  $\mathcal{F} \not\subseteq \mathcal{F}'$  there is no guarantee that this should even happen. In fact,  $f_{\mathcal{F}'}^*$  might well

be worse. As illustrated by the diagram, for non-nested function classes, a larger function class does not always move closer to the “truth” function  $f^*$ . For instance, on the left of the diagram, though  $\mathcal{F}_3$  is closer to  $f^*$  than  $\mathcal{F}_1$ ,  $\mathcal{F}_6$  moves away and there is no guarantee that further increasing the complexity can reduce the distance from  $f^*$ . With nested function classes where  $\mathcal{F}_1 \subseteq \dots \subseteq \mathcal{F}_6$  on the right of the diagram, we can avoid the aforementioned issue from the non-nested function classes.



For non-nested function classes, a larger (indicated by area) function class does not guarantee to get closer to the “truth” function ( $f^*$ ). This does not happen in nested function classes.



# Resnet

<https://arxiv.org/abs/1512.03385>

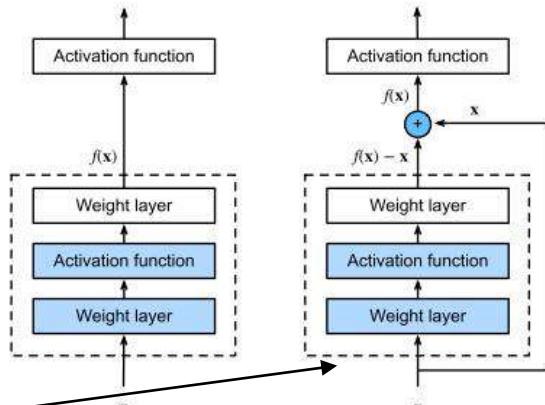
Thus, only if larger function classes contain the smaller ones are we guaranteed that increasing them strictly increases the expressive power of the network. For deep neural networks, if we can train the newly-added layer into an identity function  $f(\mathbf{x}) = \mathbf{x}$ , the new model will be as effective as the original model. As the new model may get a better solution to fit the training dataset, the added layer might make it easier to reduce training errors.

This is the question that He et al. considered when working on very deep computer vision models (He et al., 2016a). At the heart of their proposed *residual network (ResNet)* is the idea that every additional layer should more easily contain the identity function as one of its elements. These considerations are rather profound but they led to a surprisingly simple solution, a *residual block*. With it, ResNet won the ImageNet Large Scale Visual Recognition Challenge in 2015. The design had a profound influence on how to build deep neural networks.

## Residual Blocks

Let us focus on a local part of a neural network, as depicted in the figure. Denote the input by  $\mathbf{x}$ . We assume that the desired underlying mapping we want to obtain by learning is  $f(\mathbf{x})$ , to be used as the input to the activation function on the top. On the left of the dotted-line box, the portion within the dotted-line box must directly learn the mapping  $f(\mathbf{x})$ . On the right, the portion within the dotted-line box needs to learn the *residual mapping*  $f(\mathbf{x}) - \mathbf{x}$ , which is how the residual block derives its name. If the identity mapping  $f(\mathbf{x}) = \mathbf{x}$  is the desired underlying mapping, the residual mapping is easier to learn: we only need to push the weights and biases of the upper weight layer (e.g., fully-connected layer and convolutional layer) within the dotted-line box to zero. The right figure in the figure illustrates the *residual block* of ResNet, where the solid line carrying the layer input  $\mathbf{x}$  to

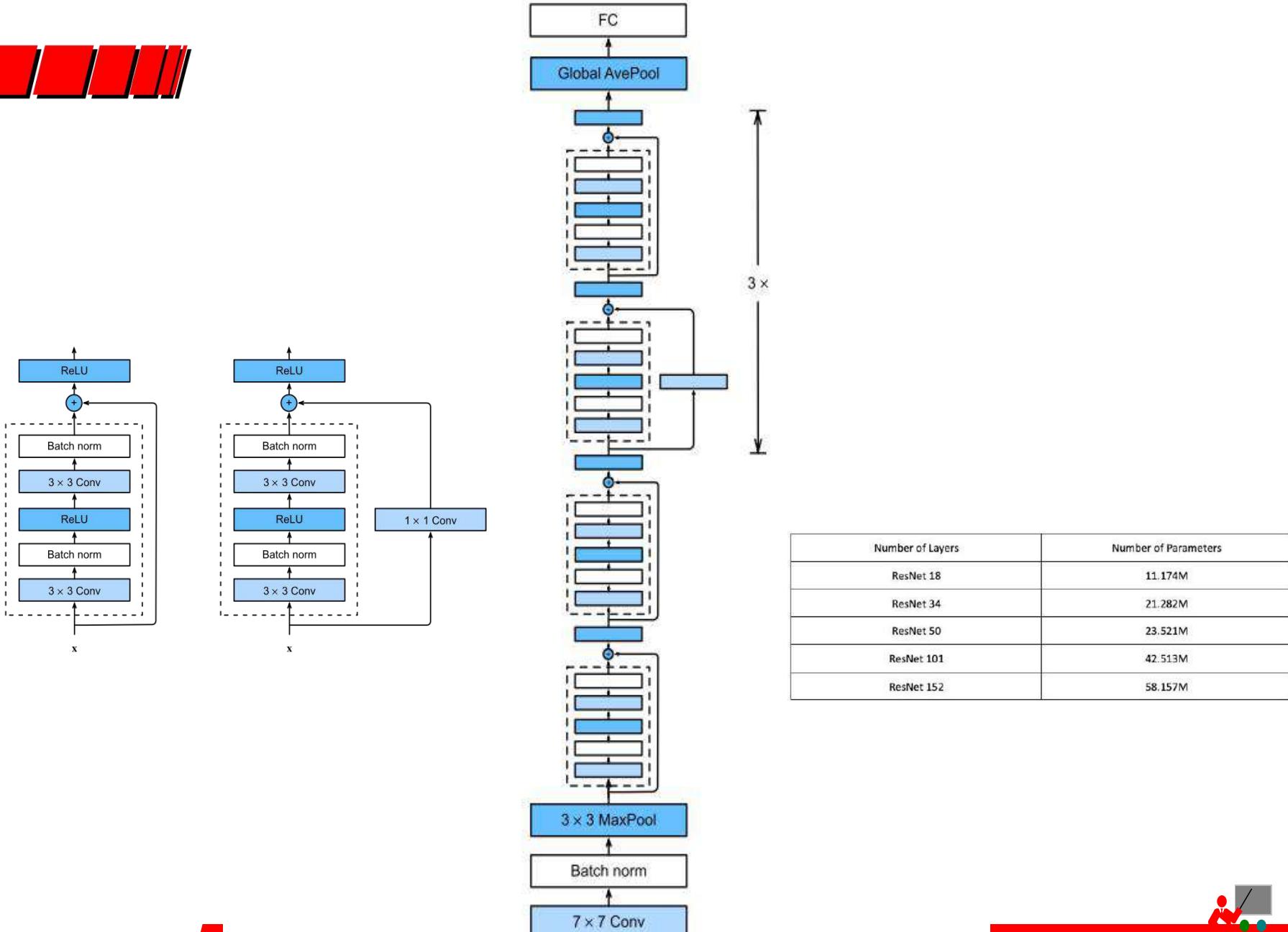
the addition operator is called a *residual connection* (or *shortcut connection*). With residual blocks, inputs can forward propagate faster through the residual connections across layers.



A regular block (left) and a residual block (right).

ResNet follows VGG's full  $3 \times 3$  convolutional layer design. The residual block has two  $3 \times 3$  convolutional layers with the same number of output channels. Each convolutional layer is followed by a batch normalization layer and a ReLU activation function. Then, we skip these two convolution operations and add the input directly before the final ReLU activation function. This kind of design requires that the output of the two convolutional layers has to be of the same shape as the input, so that they can be added together. If we want to change the number of channels, we need to introduce an additional  $1 \times 1$  convolutional layer to transform the input into the desired shape for the addition operation.





Number of Layers	Number of Parameters
ResNet 18	11.174M
ResNet 34	21.282M
ResNet 50	23.521M
ResNet 101	42.513M
ResNet 152	58.157M

The ResNet-18 architecture.

# DenseNet

ResNet significantly changed the view of how to parametrize the functions in deep networks. DenseNet (dense convolutional network) is to some extent the logical extension of this (Huang et al., 2017). To understand how to arrive at it, let us take a small detour to mathematics.

## From ResNet to DenseNet

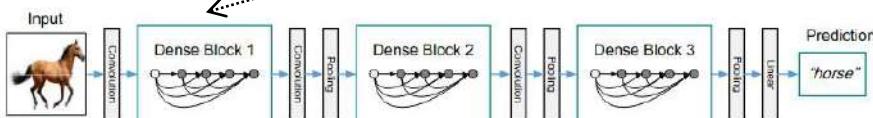
Recall the Taylor expansion for functions. For the point  $x = 0$  it can be written as

$$f(x) = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \frac{f'''(0)}{3!}x^3 + \dots$$

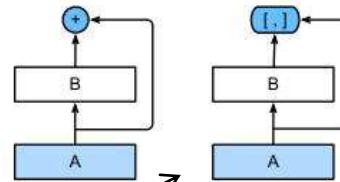
The key point is that it decomposes a function into increasingly higher order terms. In a similar vein, ResNet decomposes functions into

$$f(\mathbf{x}) = \mathbf{x} + g(\mathbf{x}).$$

[https://openaccess.thecvf.com/content\\_cvpr\\_2017/papers/Huang\\_Densely\\_Connected\\_Convolutional\\_CVPR\\_2017\\_paper.pdf](https://openaccess.thecvf.com/content_cvpr_2017/papers/Huang_Densely_Connected_Convolutional_CVPR_2017_paper.pdf)



That is, ResNet decomposes  $f$  into a simple linear term and a more complex nonlinear one. What if we want to capture (not necessarily add) information beyond two terms? One solution was DenseNet (Huang et al., 2017).

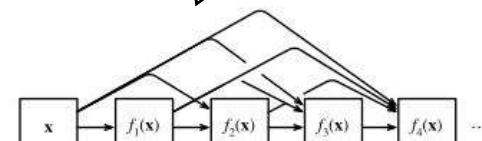


The main difference between ResNet (left) and DenseNet (right) in cross-layer connections: use of addition and use of concatenation.

As shown in the diagram, the key difference between ResNet and DenseNet is that in the latter case outputs are *concatenated* (denoted by  $[,]$ ) rather than added. As a result, we perform a mapping from  $\mathbf{x}$  to its values after applying an increasingly complex sequence of functions:

$$\mathbf{x} \rightarrow [\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})]), f_3([\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})])]), \dots].$$

In the end, all these functions are combined in MLP to reduce the number of features again. In terms of implementation this is quite simple: rather than adding terms, we concatenate them. The name DenseNet arises from the fact that the dependency graph between variables becomes quite dense. The last layer of such a chain is densely connected to all previous layers. The dense connections are shown in



Dense connections in DenseNet.

The main components that compose a DenseNet are *dense blocks* and *transition layers*. The former define how the inputs and outputs are concatenated, while the latter control the number of channels so that it is not too large.

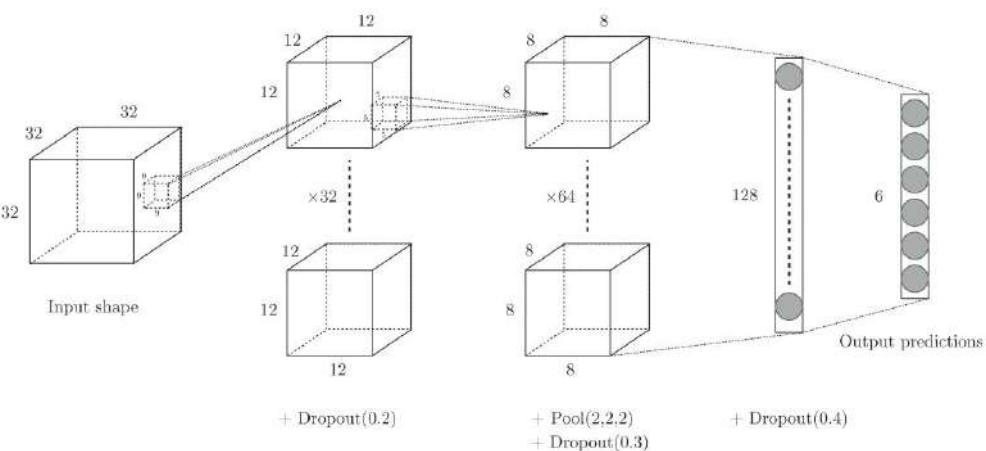




*Let's see an example:*  
*Matlab\_Examples\CNNImageClassification*



# EnzyNet



Accordingly, enzymes are represented as a binary volumetric shape with volume elements (voxels) fitted in a cube  $V$  of a fixed grid size  $l$  with respect to the three dimensions. Continuity between the voxels is achieved by nearest neighbor interpolation, such that for  $(i, j, k) \in [0, l-1]^3$  a voxel of vertices

$$(i + \delta x, j + \delta y, k + \delta z) \quad | \quad (\delta x, \delta y, \delta z) \in \{0, 1\}^3$$

takes the value 1 if the backbone of the enzyme passes through the voxel, and 0 otherwise.

[https://en.wikipedia.org/wiki/Protein\\_Data\\_Bank](https://en.wikipedia.org/wiki/Protein_Data_Bank)

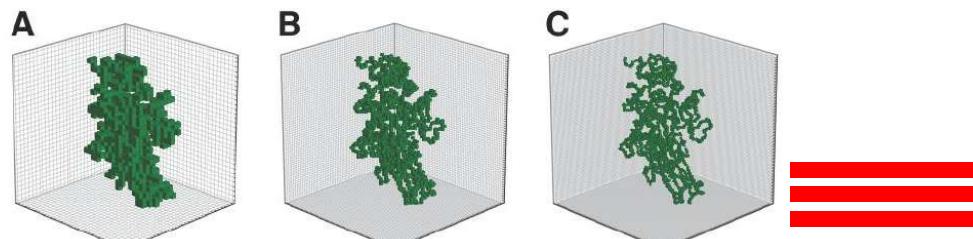
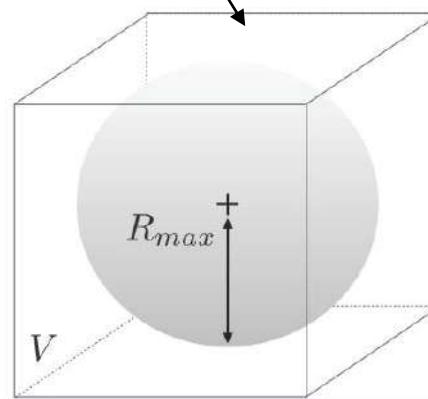


Illustration of enzyme 2Q3Z for grid sizes  $l = 32$  (A),  $l = 64$  (B), and  $l = 96$  (C).

To construct this shape representation, some preprocessing steps are necessary. First, protein structure is mapped to a grid of a predefined resolution. The selection of grid resolution determines the level of complexity/scale retained for the enzymatic structures. A full resolution is not preferred due to high data dimensionality and because fine local details are less relevant in characterizing enzymes' chemical reactions. Thus, in order to avoid getting trapped into local minima, side chains are ignored and enzymes are represented exclusively through their "backbone" atoms that are carbon, nitrogen, and calcium.

Additionally, we note that enzymes do not possess any absolute spatial orientation. Unlike objects such as chairs or boats that appear usually with a specific orientation, proteins can have any orientation in 3D conformational space, thus the Cartesian coordinates defined by the model stored in PDB represent only a frozen in space and time snapshot of an overall highly dynamic structural diversity. The orientation is irrelevant to the properties of the protein. This observation underlines the need of either a rotation invariant representation or of a convention that makes output structures comparable one to another based on the definition of an intrinsic coordinate system. We define as origin of this intrinsic coordinate system the consensus barycenter of the protein as it is defined by taking into account only the four atoms of the backbone for each residue, and as axes the principal directions of each enzyme calculated by principal component analysis. Each structure is rotated around its center and the three principal directions of the enzyme aligned with the three axes of the Cartesian coordinate system of the enzymes at a same scale. After all, proteins are comprised of various combinations of equally sized amino acids. This scaling issue is therefore equivalent to determining a maximum radius  $R_{max}$  so that the atom occupancy information contained in the sphere centered on the barycenter of the enzyme and of radius  $R_{max}$  fits into  $V$ . This situation is illustrated in



Principal directions obtained by PCA

# EnzyNet

$R_{\max}$  has to be large enough so that a sufficient number of enzymes fit the most of their volume inside  $V$ . Conversely, it also has to be small enough so that most enzymes are represented at a satisfactory resolution.

As a result, a homothetic transformation with center  $S$  and ratio  $\lambda$  defined by

$$S \text{ center of } V \text{ and } \lambda = \left\lfloor \frac{l}{2} - 1 \right\rfloor \times \frac{1}{R_{\max}} \quad \text{https://en.wikipedia.org/wiki/Homothety}$$

is performed on all enzymes to scale them to the desired size.

When  $l$  is low, the grid is coarse enough so that the voxels of the structure have a contiguous shape. Conversely, big volumes tend to separate voxels, which engender “holes.” In that case, consecutive backbone atoms  $(\vec{A}_i, \vec{A}_{i+1})$  are interpolated by  $p$  regularly spaced new points computed by

$$\frac{(p - k + 1) \times \vec{A}_i + k \times \vec{A}_{i+1}}{p + 1}$$

where  $k$  varies from 1 to  $p$ .

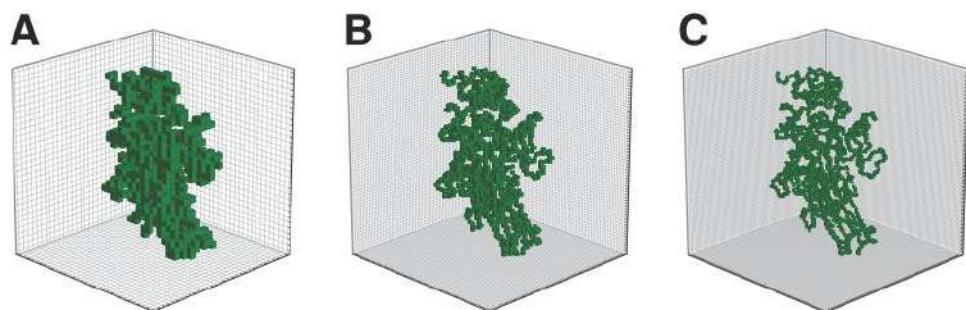
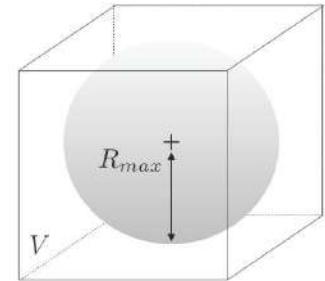


Illustration of enzyme 2Q3Z for grid sizes  $l = 32$  (A),  $l = 64$  (B), and  $l = 96$  (C).



# EnzyNet

## Algorithm 1 Summary of the preprocessing steps done to each enzyme at training time

**Data:**  $N$  training enzymes, grid size of  $l$ , homothetic transformation ratio  $\lambda$ ,  $p$  interpolations, probability  $p_{\text{flip}}$  to flip with respect to each axis.

**Input:** Raw coordinates contained in PDB files

**Output:** Volumes of binary voxels representing backbone atoms occupancy

```
1 foreach of the  $N$  enzymes of the training set do
2   Step 1: structural information extraction
3     Extract coordinates of backbone atoms from its PDB file
4   Step 2: holes completion
5     Interpolate consecutive backbone atoms by  $p$  new points
6   Step 3: size adjustment
7     Center barycenter  $S$  of the coordinates on  $(0, 0, 0)$ 
8     Homothetic transformation of each point with center  $S$  and ratio  $\lambda$ 
9   Step 4: enzyme orientation
10  Principal component analysis (PCA) transformation
11  Step 5: random augmentation
12  if True with probability  $p_{\text{flip}}$  then
13    Flip coordinates with respect to the origin along  $x$ -axis
14  if True with probability  $p_{\text{flip}}$  then
15    Flip coordinates with respect to the origin along  $y$ -axis
16  if True with probability  $p_{\text{flip}}$  then
17    Flip coordinates with respect to the origin along  $z$ -axis
18  Step 6: voxelization
19  Center barycenter  $S$  of the coordinates on  $(\frac{l}{2}, \frac{l}{2}, \frac{l}{2})$ 
20  Transform coordinate points into binary voxels
```



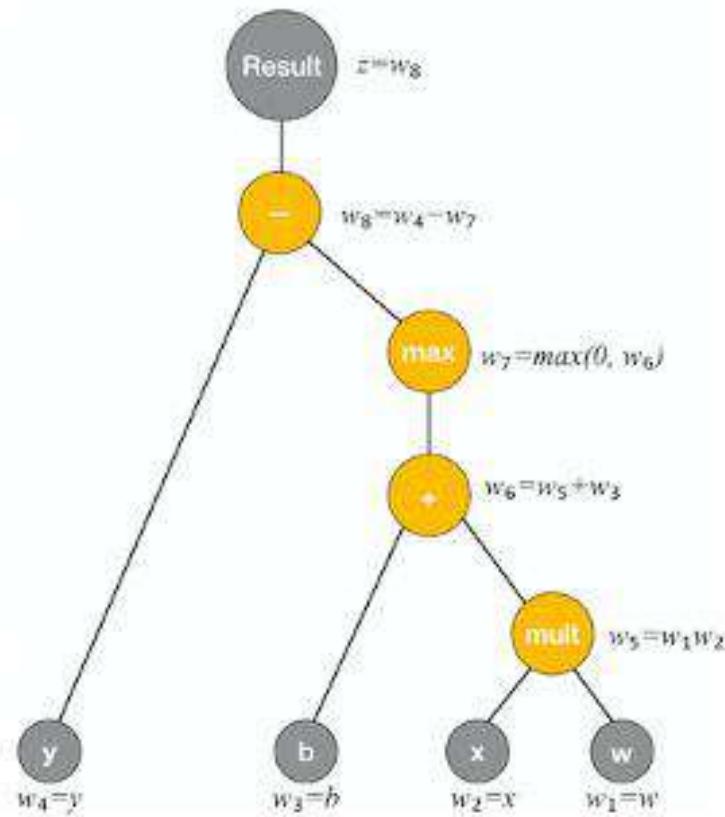
# AutoDiff

However, how do neural networks — computers — calculate the partial derivatives of an expression? The answer lies in a process known as **automatic differentiation**. Let me illustrate it to you using the cost function

$$C(y, w, x, b) = y - \max(0, w \cdot x + b)$$

In addition, because automatic differentiation can only calculate the partial derivative of an expression on a certain point, we have to assign initial values to each of the variables. Let us say:  $y=5$ ;  $w=2$ ;  $x=1$ ; and  $b=1$ .

Let's find the derivative of the function!



Before we can begin deriving the expression, it must be converted into a computational graph. A computational graph simply turns each operation into a **node** and connects them through lines, called **edges**. The computation graph for our example function is shown below.



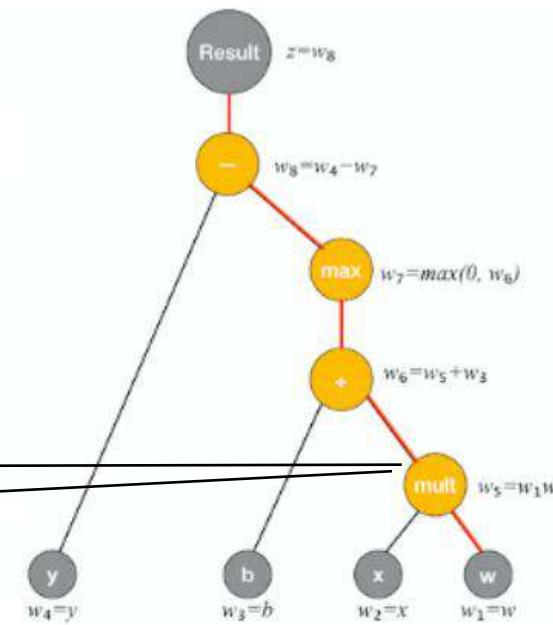
# AutoDiff

First, let us calculate the values of each node, propagating from the bottom (the input variables) to the top (the output function). This is what we get:

Node	Expression	Value
$w_1$	$w$	2
$w_2$	$x$	1
$w_3$	$b$	1
$w_4$	$y$	5
$w_5$	$w_1 \cdot w_2$	2
$w_6$	$w_5 + w_3$	3
$w_7$	$\max(0, w_6)$	3
$w_8$	$w_4 - w_7$	2
$z$	$w_8$	2

In addition, because automatic differentiation can only calculate the partial derivative of an expression on a certain point, we have to assign initial values to each of the variables. Let us say:  $y=5$ ;  $w=2$ ;  $x=1$ ; and  $b=1$ .

$$C(y, w, x, b) = y - \max(0, w \cdot x + b)$$



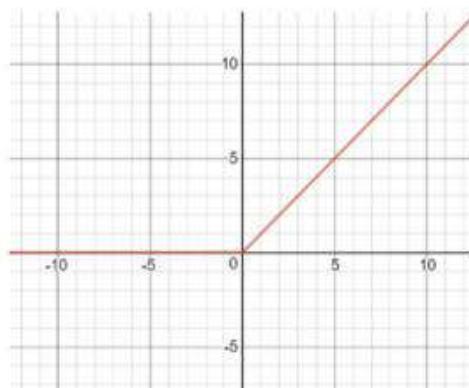
Next, we need to calculate the partial derivatives of each connection between operations, represented by the edges. These are the calculations of the partials of each edge:

$\frac{\partial w_7}{\partial w_6}$  is 1 when  $x > -1/2$  and 0 when  $x < -1/2$ .  
 $\max(0, w_6)$  is  $w_6$  when  $w_6 > 0$ , i.e.  $(w_5 + w_3) > 0$  that is  $w_1 \cdot w_2 + w_3 > 0$ , when we put in the values, we get  $2x+1 > 0$

$\frac{\delta w_5}{\delta w_1} = \frac{\delta w_1 w_2}{\delta w_1} = w_2$	$\frac{\delta w_5}{\delta w_2} = \frac{\delta w_1 w_2}{\delta w_2} = w_1$
$\frac{\delta w_6}{\delta w_5} = \frac{\delta w_5 + w_3}{\delta w_5} = 1$	$\frac{\delta w_6}{\delta w_3} = \frac{\delta w_5 + w_3}{\delta w_3} = 1$
$\frac{\delta w_7}{\delta w_6} = \frac{\delta \max(0, w_6)}{\delta w_6} = \begin{cases} 0, x \leq -1/2 \\ 1, x > -1/2 \end{cases}$	$\frac{\delta w_8}{\delta w_7} = \frac{\delta w_4 - w_7}{\delta w_7} = -1$
$\frac{\delta w_8}{\delta w_4} = \frac{\delta w_4 - w_7}{\delta w_4} = 1$	$\frac{\delta z}{\delta w_8} = 1$

# AutoDiff

Notice how the partial of the  $\max(0, x)$  piece-wise function is also a piece-wise function. The function converts all negative values to zero, and keeps all positive values as they are. The partial of the function (or graphically, its slope), should be clear on its graph:

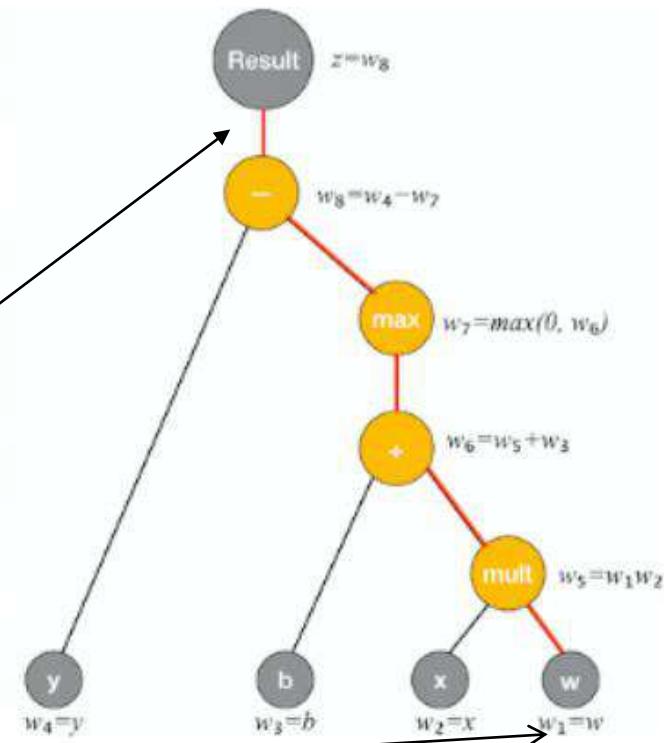


$\max(0, x)$  function. As seen, there is a slope of 1 when  $x>0$ , and a slope of 0 when  $x<0$ . The slope is undefined when  $x=0$ .

Now we can move on to calculating the partials! Let's find the partial of  $z = w_1 \cdot x + b$  with respect to the weights. As seen in Figure 1, there is just one line that connects the result to the weights.

the red path connects the result to the weight  $w_1$ .

- there is not always only one path between "Result" and the parameter, in the exercise handout we will also see another approach based on autodiff, for instance, try to create the graph related to ' $x/(1+|x|)$ '



# AutoDiff

Now we simply multiply up the edges:

$$\frac{\delta z}{\delta w_1} = \frac{\delta z}{\delta w_8} \times \frac{\delta w_8}{\delta w_7} \times \frac{\delta w_7}{\delta w_6} \times \frac{\delta w_6}{\delta w_5} \times \frac{\delta w_5}{\delta w_1} = 1 \times (-1) \times \begin{cases} 0, x \leq -1/2 \\ 1, x > -1/2 \end{cases} \times 1 \times w_2 = -1$$

And that's our partial!

$\frac{\delta w_5}{\delta w_1} = \frac{\delta w_1 w_2}{\delta w_1} = w_2$	$\frac{\delta w_5}{\delta w_2} = \frac{\delta w_1 w_2}{\delta w_2} = w_1$
$\frac{\delta w_6}{\delta w_5} = \frac{\delta w_5 + w_3}{\delta w_5} = 1$	$\frac{\delta w_6}{\delta w_3} = \frac{\delta w_5 + w_3}{\delta w_3} = 1$
$\frac{\delta w_7}{\delta w_6} = \frac{\delta \max(0, w_6)}{\delta w_6} = \begin{cases} 0, x \leq -1/2 \\ 1, x > -1/2 \end{cases}$	$\frac{\delta w_8}{\delta w_7} = \frac{\delta w_4 - w_7}{\delta w_7} = -1$
$\frac{\delta w_8}{\delta w_4} = \frac{\delta w_4 - w_7}{\delta w_4} = 1$	$\frac{\delta z}{\delta w_8} = 1$

Node	Expression	Value
$w_1$	$w$	2
$w_2$	$x$	1
$w_3$	$b$	1
$w_4$	$y$	5

This whole process can be completed automatically, and allows computers to compute the partial derivative of a value of a function accurately and quickly. It is this process that allows AI to be as efficient as it is today.

useful for practicing <https://www.symbolab.com/solver/gradient-calculator/gradient%20%5Csqrt%7Bx%5E%7B2%7D%2By%5E%7B2%7D%7D%2C%20%5Cat%282%2C2%29?or=ex>



# Transfer learning

Training complex CNNs on large datasets (eg ImageNet) can take days / weeks of machine time, even if performed on GPU. Fortunately, once the network has been trained, the time required for classifying a new pattern (forward propagation) is usually fast. In addition, training a CNN on a new problem requires a large labeled training set (often not available). As an alternative to training from scratch, we can pursue two transfer learning approaches:

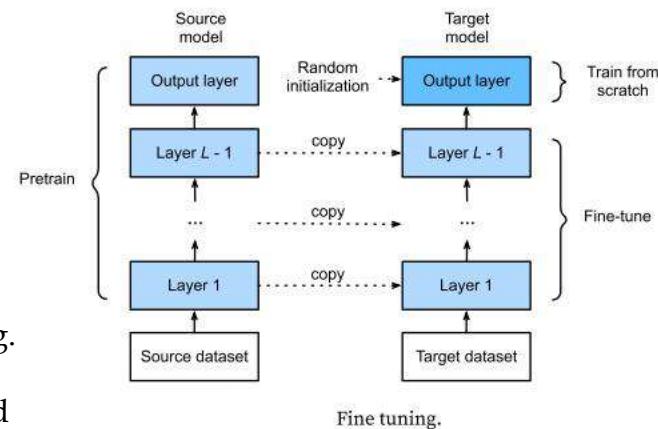
Fine-Tuning;

Deep Features.

Fine-Tuning: we start with a pre-trained network trained on a similar problem and:

1. replace the output level with a new softmax output level (adjusting the number of classes)
2. the initial values of the weights are used those of the pre-trained network, except for the connections between the penultimate and last level whose weights are initialized at random.
3. new training iterations (e.g. SGD) are performed to optimize the weights with respect to the peculiarities of the new dataset  
(it does not need to be very large).

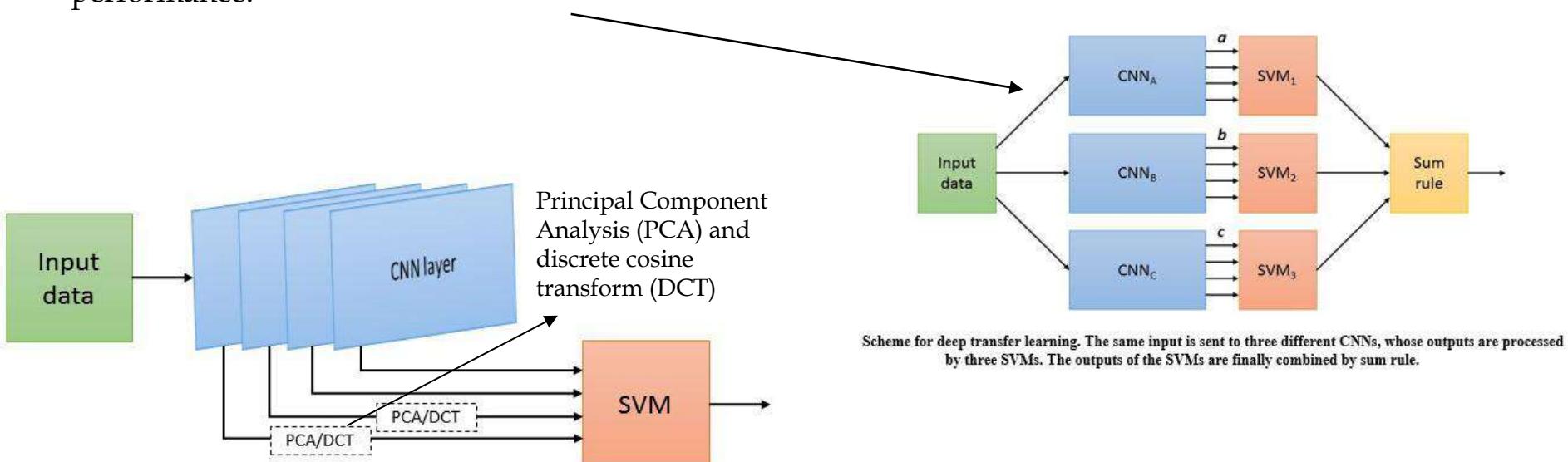
Deep Features: An existing (pre-trained) network is used without further fine-tuning. The features generated by the network during the forward step are extracted (at intermediate levels) (see L5, L6, L7 in the CaffeNet example). These features are used to train an external classifier (e.g. Support Vector Machine) to classify the patterns of the new application domain.



# Transfer Learning - Deep Features

Even the innermost layers of a deep network can be used to train a classifier. The problem is the very high dimensionality of these levels, a possible way is to reduce this dimensionality (using computationally manageable methods), e.g., using PCA or discrete cosine transform (DCT). Instead of concatenating the various feature vectors extracted from different layers and from different networks (e.g. different topologies), for each single feature extraction you can train a different SVM and then the results are combined (e.g. by sum rule).

It is already shown in the literature that in many applications different topologies carry partially independent information, therefore combining different CNNs allows increased performance.



Further details on deep features extracted from innermost layers:  
<https://www.mdpi.com/2313-433X/7/9/177>



# AutoEncoder

The aim of an AutoEncoder (AE) is to learn a representation (encoding) for a set of data, typically for dimensionality reduction, by training the network to ignore signal "noise".

AEs transform the input into a new representation (called code or latent-space representation) and then reconstruct the output from this representation.

An AE is composed by:

- an encoding function  $E(x): \mathbb{R}^n \rightarrow \mathbb{R}^k$  outputting a latent representation  $s$ ;
- a decoding function  $D(s): \mathbb{R}^k \rightarrow \mathbb{R}^n$  computing the reconstructed output  $o$ .



The simplest form of an AE is a feed-forward neural network having an input layer and an output layer with the same number of neurons and one or more hidden layers connecting them.

The purpose is to minimize the difference between the input and the output.

An AE consists of two parts:

- the encoder ( $E$ ) – it compresses the input ( $x$ ) and produces the code ( $s$ );
- the decoder ( $D$ ) – it reconstructs the input ( $o$ ) starting from the code ( $s$ ).

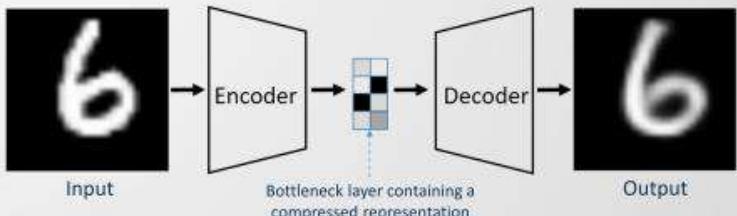
An AE can be trained by minimizing the reconstruction error,  $\mathcal{L}(x, o)$ , which measures the difference between the input and its reconstruction.

If the only purpose of AEs is to copy the input to the output, they would be useless.

The hope is that during training the latent representation will take on useful properties.

The risk is the AE could learn the so-called identity function, so the output equals the input, and does not perform any useful representation learning or dimensionality reduction.

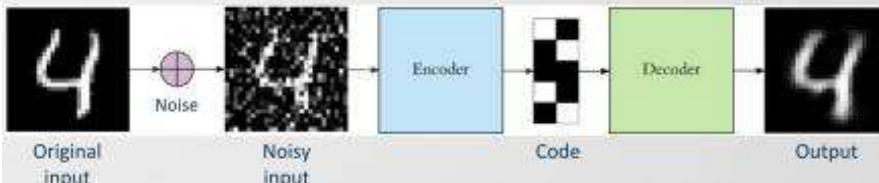
To avoid this risk, the simplest solution is to use a bottleneck layer which forces a compressed knowledge representation of the original input constraining the amount of information that can traverse the full network ( $k < n$ ).



Denoising AEs prevent the network learning the identity function by corrupting the input data on purpose (adding noise or masking some of the input values) and making it recover the original noise-free data.

The AE cannot simply copy the input to its output, but it is forced to extract useful features that constitute better higher-level representations of the input.

The input corruption is performed only during the training phase. Once the model has learnt the optimal parameters, in order to extract the representations from the original data no corruption is added.



Sparse AEs represent an alternative method to avoid that the model learns the identity function, without a reduction in the number of neurons, by using a sparsity constraint.

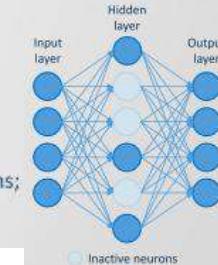
A penalty term is added to the loss function such that only a fraction of the neurons become active:

$$\mathcal{L}(x, o) + \Omega(h)$$

where  $\Omega$  is a penalty function on hidden layer activations ( $h$ ).

This forces the AE:

- to represent each input as a combination of small number of neurons;
- to discover interesting structure in the data.



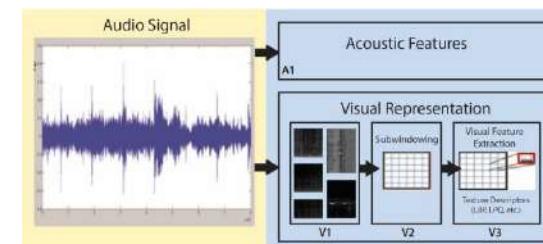
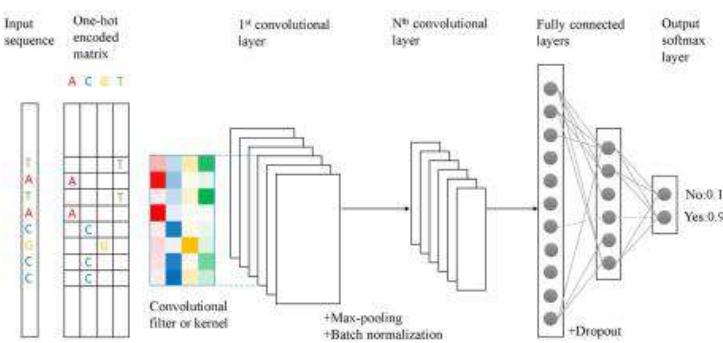
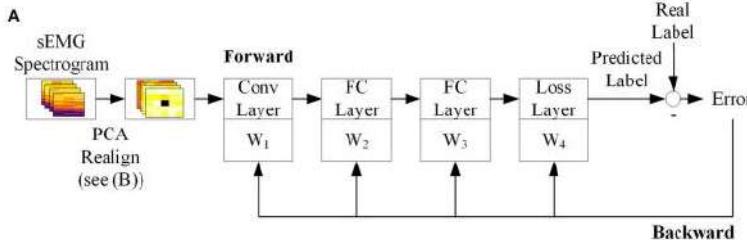
*Let's see an example:*  
*Matlab\_Examples\TransferLearn  
ingSVM*



# Application examples

- PS: the PAP SMEAR dataset contains 917 images acquired during Pap tests to identify cervical cancer diagnosis.
- VI: the VIRUS dataset contains 1500 images of viruses, divided into 10 classes. Images were acquired by negative stain transmission electron microscopy. VI also includes the masks for background removal, which are not used in our tests.
- CH: the CHINESE HAMSTER OVARY CELLS dataset contains 327 fluorescent microscopy images, divided into 5 classes.
- SM: the SMOKE dataset contains 1383 images of smoke, divided into 2 classes used for a challenge of intelligent video surveillance systems.
- HI: the HISTOPATHOLOGY dataset contains 2828 images of connective, epithelial, muscular, and nervous tissue classes.
- BR: the BREAST CANCER dataset contains 1394 images divided into the control, malignant cancer, and benign cancer classes.
- PR: the DNA dataset contains 329 proteins, divided into the DNA-binding and non-DNA-binding classes.
- HE: the 2D HELA dataset contains 862 images of HeLa cells acquired by fluorescence microscope and divided into 10 classes.
- LO: the LOCATE ENDOGENOUS dataset contains 502 images of mouse sub-cellular images showing endogenous proteins or specific organelle features. The images are unevenly divided into 10 classes.
- TR: the LOCATE TRANSFECTED dataset contains 553 mouse sub-cellular images showing fluorescence-tagged or epitope-tagged proteins transiently expressed in specific organelles. The images are unevenly divided into 11 classes.
- PI: the HOLY BIBLE dataset contains images extracted from digitalized pages of ancient editions of the Holy Bible (1450 - 1471 A.D.). The images are divided into 13 classes.
- RN: the FLY CELL dataset contains 200 images of fly cells acquired by fluorescence microscopy and divided into 10 classes.
- PA: the PAINTING dataset contains 2338 paintings by 50 artists divided into 13 painting styles.
- LE: the BRAZILIAN FLORA dataset contains 400 images of several species of Brazilian flora evenly divided into 20 classes. Each image was manually split into three windows of size 128×128 pixels leading to 1200 texture samples.
- KU: the BUTTERFLY dataset contains 140 butterfly images divided into 14 classes of different butterfly species of Styridae family. Each image was cropped to a 256 × 256 pixel image before processing. Each image was split into two non-overlapping equal regions (an upper and lower region). For each region a set of descriptors were extracted.
- FM: the FLICKR MATERIAL dataset contains 1000 images of 10 different materials classes: fabric, foliage, glass, leather, metal, paper, plastic, stone, water, and wood. For each class, 50 images are close-up views, and the other 50 are views at object-scale. The location of the object is defined in each image by a binary, manually-labeled mask.
- SR: the SCENE dataset contains 8100 images divided into 15 classes scene. The testing protocol requires five experiments, each using 100 randomly selected images per class for training; the remaining images are used for testing. Each image was split into four non-overlapping equal regions and a central region half the size of the original image. For each region a set of descriptors were extracted.
- FR: the OXFORD FLOWERS 102 dataset contains 8189 images divided into 102 categories, containing 40 to 250 images each.

## Self-Recalibrating Surface EMG Pattern Recognition for Neuroprosthesis Control Based on Convolutional Neural Network



the audio signal is converted into a spectrogram image that shows the spectrum of frequencies (vertical axis) according to time (horizontal axis). The intensity of each point in the image represents the signal's amplitude.



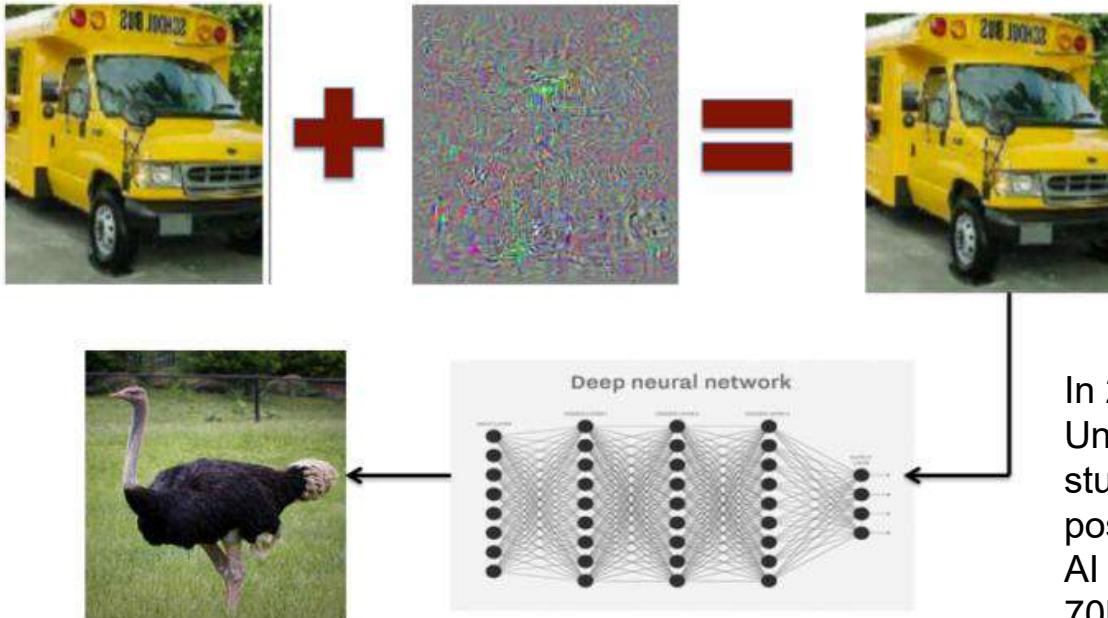
*Let's see an example:*  
*Matlab\_Examples\AudioClassification&*  
*Augmentation*



# Adversarial examples

- They can be easily fooled with **adversarial examples**
- The human vision system suffers from allucinations/illusions too...

[Szegedy et al., Intriguing properties of neural networks, 2014]



In 2017, Dawn Song of the University of California, Berkeley, stuck some stickers in strategic positions on a stop sign and the AI mistook it for a speed limit of 70km/h. <https://www.tomshw.it/gannare-ia-facilissimo-allarme-degli-scienti-96229>



# Adversarial examples, review (2021):

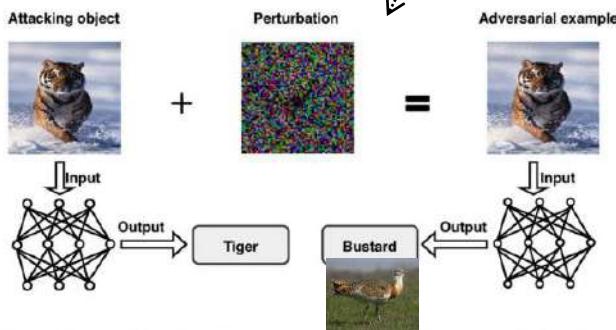
<https://link.springer.com/article/10.1007/s13042-020-01242-z>

Szegedy et al. first proposed the concept of adversarial example. The study found two “unconventional” phenomena of neural networks. Among them, one is the vulnerability of neural networks in image classification. The attacker generates examples with offensive characteristics by adding carefully crafted, small-magnitude, and imperceptible perturbations to the clean samples. Since the added perturbation is very small, it will not significantly affect the data distribution. Such examples can make the neural network with high classification confidence and accuracy output the wrong results. The research calls this kind of sample added with attack perturbations as adversarial examples. Figure shows the generation pipeline of adversarial examples. Its mathematical model is shown in

such that

$$\min \|\delta\|_2, \text{ s.t. } f(x + \delta) \neq l, x + \delta \in [0, 1]^m, \quad (1)$$

where  $x + \delta$  is an adversarial example composed of clean samples  $x$  and perturbations  $\delta$ , and  $l$  is the true label of  $x$ .  $f$  represents the classification model,  $f(\cdot)$  which is a mapping function that maps the input image to a label output. To solve the above-constrained optimization problem efficiently, some work reformulated it in the Lagrangian-relaxed form



General pipeline for generating adversarial examples in digital world. Adversarial examples can mislead the model into predicting a tiger image as a bustard

Sun et al. summarized its mathematical model into the following formula.

$$\arg \min_{\delta} L(f(x + \delta), l) + \text{QualityPenalty}(\delta). \quad (2)$$

Here  $L(\cdot, \cdot)$  is the loss function measuring the difference between the model's prediction and the target label  $l$ . QualityPenalty is usually a kind of norm of  $\delta$ .

## Non-target attacks

The model output is any category other than the real category, so the attack is successful. In (1),  $f(x + \delta) \neq l$ .

## Targeted attacks

The model not only outputs an error, but the output is a category specified by the attacker. Targeted attacks are harder to achieve than non-targeted attacks. In (1),  $f(x + \delta) = y$  and  $y \neq l$ , where  $y$  is the prediction category that the attacker wants the model output.

## White-box attacks

The attacker knows the internal structure, weight parameters, and training algorithm of the attack model.

## Black-box attacks

In contrast to white-box attacks, the attacker knows nothing about the attack model.

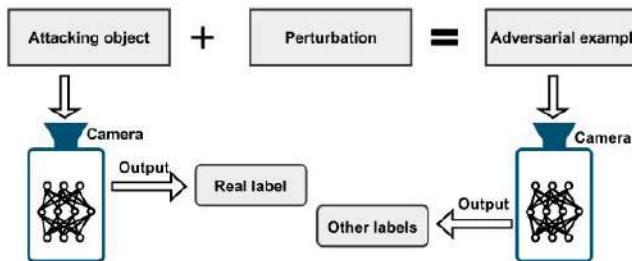
In a query-based attack, the attacker estimates the gradient by querying the mode multiple times.

Sun L, Tan M, Zhou Z (2018) A survey of practical adversarial example attacks. Cybersecurity 1(1):9



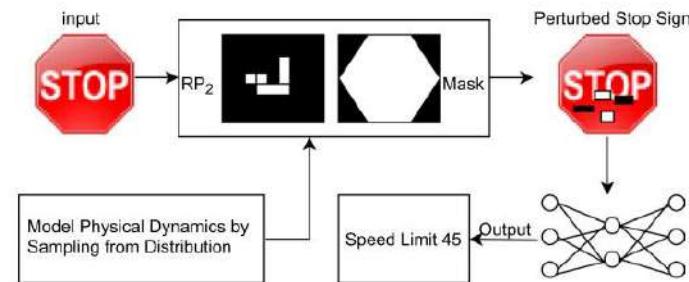
# Adversarial examples - physical world

In the digital world, the attacker has “digital level” access to the input. For example, the attacker can make arbitrary pixel-level modifications to the input image of the classifier. However, in real applications, attackers cannot control the sensors and data pipes of the system. At the same time, the perturbation must survive in various environmental transformations. This part mainly introduces the challenge of adversarial attacks in the physical world and the mitigation methods proposed by some work.



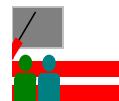
Practical adversarial examples attack. The attack object with added perturbation is captured by the sensor (such as camera, LiDAR), and then passed into the model to be predicted as other labels instead of the real label

Overview of the adversarial examples generation process in LiDAR attack



Black and white stickers of road signs attack the pipeline. Adversarial examples can mislead the model to recognize the stop sign as a speed limit sign of 45

<https://arxiv.org/abs/1907.07174> is shown that some architectural changes can enhance robustness to natural adversarial examples. Currently, defense methods mainly add adversarial images in the training set.



# Distribution shift

Many failed machine learning deployments can be traced back to this pattern. Sometimes models appear to perform marvelously as measured by test set accuracy but fail catastrophically in deployment when the distribution of data suddenly shifts. More insidiously, sometimes the very deployment of a model can be the catalyst that perturbs the data distribution. Say, for example, that we trained a model to predict who will repay vs. default on a loan, finding that an applicant's choice of footwear was associated with the risk of default (Oxfords indicate repayment, sneakers indicate default). We might be inclined to thereafter grant loans to all applicants wearing Oxfords and to deny all applicants wearing sneakers.

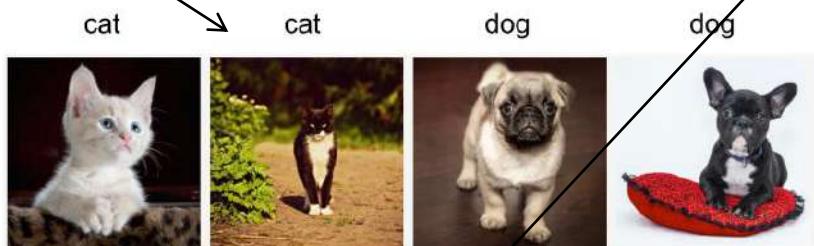
Consider a binary classification problem, where we wish to distinguish between dogs and cats. If the distribution can shift in arbitrary ways, then our setup permits the pathological case in which the distribution over inputs remains constant:  $p_S(\mathbf{x}) = p_T(\mathbf{x})$ , but the labels are all flipped:  $p_S(y|\mathbf{x}) = 1 - p_T(y|\mathbf{x})$ . In other words, if God can suddenly decide that in the future all "cats" are now dogs and what we previously called "dogs" are now cats—without any change in the distribution of inputs  $p(\mathbf{x})$ , then we cannot possibly distinguish this setting from one in which the distribution did not change at all.

Fortunately, under some restricted assumptions on the ways our data might change in the future, principled algorithms can detect shift and sometimes even adapt on the fly, improving on the accuracy of the original classifier.

## Covariate Shift

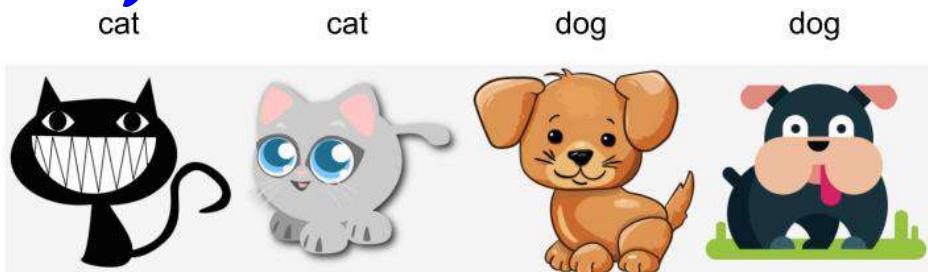
Among categories of distribution shift, covariate shift may be the most widely studied. Here, we assume that while the distribution of inputs may change over time, the labeling function, i.e., the conditional distribution  $P(y | \mathbf{x})$  does not change. Statisticians call this *covariate shift* because the problem arises due to a shift in the distribution of the covariates (features). While we can sometimes reason about distribution shift without invoking causality, we note that covariate shift is the natural assumption to invoke in settings where we believe that  $\mathbf{x}$  causes  $y$ .

Consider the challenge of distinguishing cats and dogs. Our training data might consist of images of the kind in



Training data for distinguishing cats and dogs.

At test time we are asked to classify the images in



Test data for distinguishing cats and dogs.

The training set consists of photos, while the test set contains only cartoons. Training on a dataset with substantially different characteristics from the test set can spell trouble absent a coherent plan for how to adapt to the new domain.

## Label Shift

*Label shift* describes the converse problem. Here, we assume that the label marginal  $P(y)$  can change but the class-conditional distribution  $P(\mathbf{x} | y)$  remains fixed across domains. Label shift is a reasonable assumption to make when we believe that  $y$  causes  $\mathbf{x}$ . For example, we may want to predict diagnoses given their symptoms (or other manifestations), even as the relative prevalence of diagnoses are changing over time. Label shift is the appropriate assumption here because diseases cause symptoms. In some degenerate cases the label shift and covariate shift assumptions can hold simultaneously. For example, when the label is deterministic, the covariate shift assumption will be satisfied, even when  $y$  causes  $\mathbf{x}$ . Interestingly, in these cases, it is often advantageous to work with methods that flow from the label shift assumption. That is because these methods tend to involve manipulating objects that look like labels (often low-dimensional), as opposed to objects that look like inputs, which tend to be high-dimensional in deep learning.

## Concept Shift

We may also encounter the related problem of *concept shift*, which arises when the very definitions of labels can change. This sounds weird—a *cat* is a *cat*, no? However, other categories are subject to changes in usage over time. Diagnostic criteria for mental illness, what passes for fashionable, and job titles, are all subject to considerable amounts of concept shift. It turns out that if we navigate around the United States, shifting the source of our data by geography, we will find considerable concept shift regarding the distribution of names for *soft drinks*.

# Reinforcement Learning



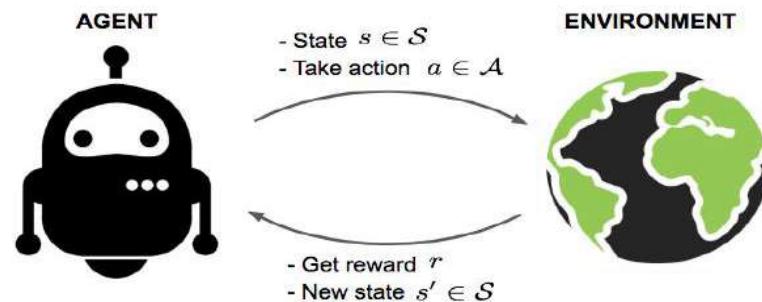
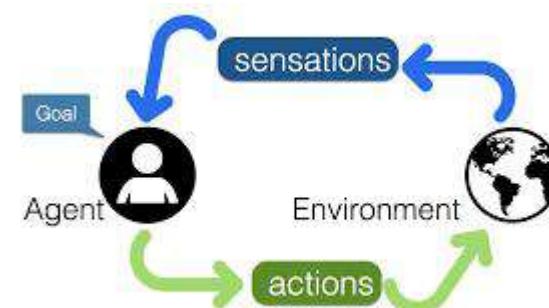
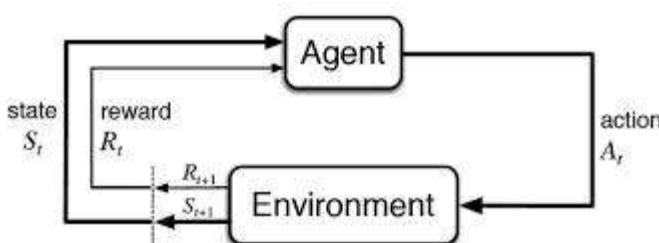
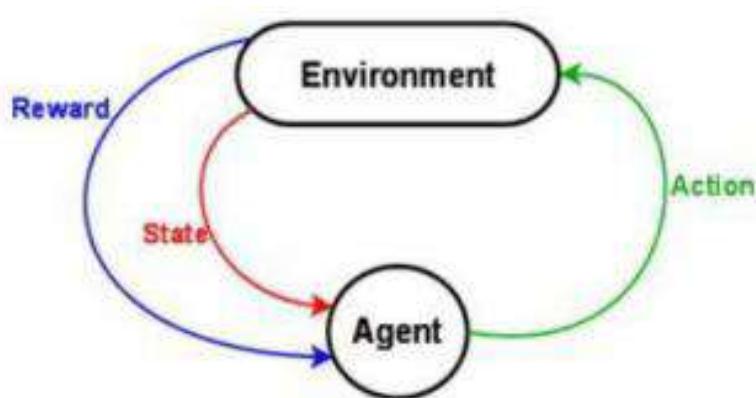
*Reinforcement learning is the science of decision making. It is about learning the optimal behavior in an environment to obtain maximum reward.*



# Reinforcement Learning

The goal is to learn optimal behavior from past experiences.

An agent performs actions ( $a$ ) that modify the environment, causing changes from one state ( $s$ ) to another. When the agent obtains positive results, it receives a reward ( $r$ ) which, however, may be temporally delayed with respect to the action, or the sequence of actions, which determined it.



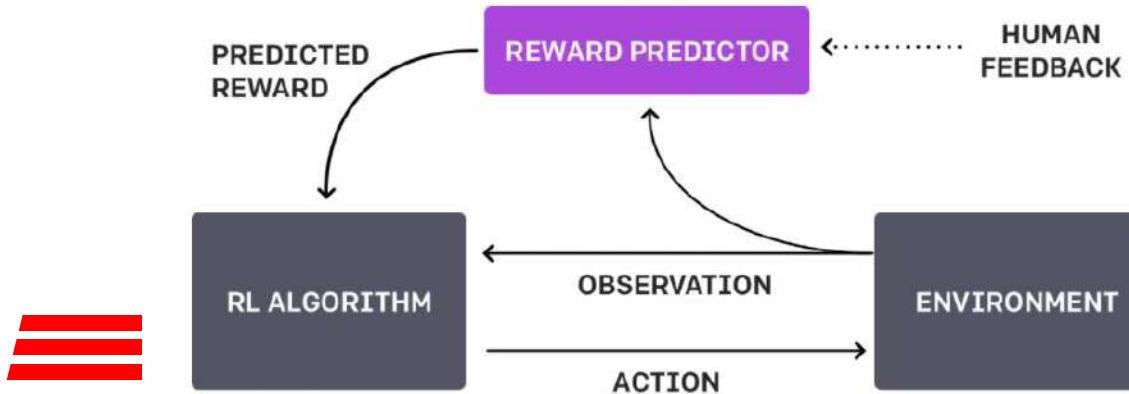
# *Deep reinforcement learning from human preferences (named also with human feedback)*

<https://openai.com/blog/deep-reinforcement-learning-from-human-preferences/>

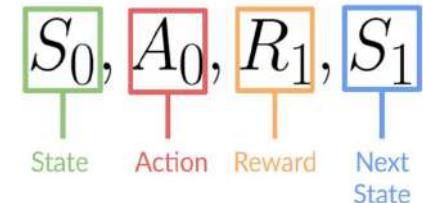
The overall training process is a 3-step feedback cycle between the human, the agent's understanding of the goal, and the RL training. Our AI agent starts by acting randomly in the environment. Periodically, two video clips (two different actions) of its behavior are given to a human, and the human decides which of the two clips is closest to fulfilling its goal. The AI gradually builds a model of the goal of the task by finding the reward function that best explains the human's judgments. It then uses RL to learn how to achieve that goal. As its behavior improves, it continues to ask for human feedback on trajectory pairs where it's most uncertain about which is better, and further refines its understanding of the goal.

In

[https://cdn.openai.com/papers/Training\\_language\\_models\\_to\\_follow\\_instructions\\_with\\_human\\_feedback.pdf](https://cdn.openai.com/papers/Training_language_models_to_follow_instructions_with_human_feedback.pdf) is shown an avenue for aligning language models with user intent on a wide range of tasks by fine-tuning with human feedback.



# Reinforcement Learning (RL)



An episode (or game) is a finite sequence of states, actions, rewards:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2 \dots, S_{n-1}, A_{n-1}, R_n, S_n$$

In each state  $s_{t-1}$  the goal is to choose the optimal action  $a_{t-1}$ , that is the one that maximizes the future reward  $R_t$ :

$$R_t = r_t + r_{t+1} + \dots + r_n$$

Value of the Environment state is the reward we are likely to receive in the future ( $t+1\dots n$ ).

In many real applications the environment is stochastic (ie, it is not certain that the same action always determines the same sequence of states and rewards), therefore applying the logic of "better an egg today than a hen tomorrow" the RL approaches weigh more the rewards temporally close (discounted future reward):

$$R_t = r_t + \gamma \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + \dots + \gamma^{n-t} \cdot r_n \quad 0 \leq \gamma \leq 1$$

The "Discounted sum of future rewards" using discount factor  $\gamma$  is

$$\begin{aligned} & (\text{reward now}) + \\ & \gamma (\text{reward in 1 time step}) + \\ & \gamma^2 (\text{reward in 2 time steps}) + \\ & \gamma^3 (\text{reward in 3 time steps}) + \\ & \dots \quad (\text{infinite sum}) \end{aligned}$$



# *Q-Learning*

The discounted future reward can be defined recursively:

$$R_t = r_t + \gamma \cdot R_{t+1}$$

In Q-learning, the  $Q(s, a)$  function indicates the optimality (or quality) of the action  $a$  when it is applied at state  $s$ . The objective is to maximize the discounted future reward:

$$Q(s_t, a_t) = \max R_{t+1}$$

Assuming that the function  $Q(s, a)$  is known, when it is in state  $s$ , it can be shown that the optimal policy is the one that chooses action  $a^*$  such that:

$$a^* = \arg \max_a Q(s, a)$$



# *Q-learning*

■ The crucial point is therefore learning the Q function.

Given a transition (quatern)  $\langle s_t, a_t, r_{t+1}, s_{t+1} \rangle$  we can write:

$$Q(s_t, a_t) = \max R_{t+1} = \max(r_{t+1} + \gamma \cdot R_{t+2}) =$$

$$Q(s_t, a_t) = r_{t+1} + \gamma \cdot \max R_{t+2} = r_{t+1} + \gamma \cdot Q(s_{t+1}, a_{t+1})$$

The action  $a_{t+1}$  (which is not part of the quatern) will be chosen with the previous policy, obtaining:

$$Q(s_t, a_t) = r_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a)$$

It is known as Bellman's equation.



# *Q-learning*

Q learning algorithm based on Bellman's equation:

Set values for learning rate  $\alpha$ , discount rate  $\gamma$ , reward matrix  $R$

Initialize  $Q(s,a)$  to zeros

Repeat for each episode, do

Select state  $s$  randomly

Repeat for each step of episode, do

Choose  $a$  from  $s$

Take action  $a$  obtain reward  $r$  from  $R$ , and next state  $s'$

Update  $\hat{Q}(s,a) \leftarrow \hat{Q}(s,a) + \alpha[r + \gamma \max_{a'} \hat{Q}(s',a') - \hat{Q}(s,a)]$

Set  $s = s'$

## Estimate on the future

Until  $s$  is the terminal state optimal value at time

End do

End do

$O((H^3SAm)^{1/2})$   
 S and A are the  
 numbers of states and  
 actions, H is the  
 number of steps per  
 episode, and m is the  
 total number of steps.

<https://papers.nips.cc/paper/2018/file/d3b1fb02964aa64e257f9f26a31f72cf-Paper.pdf>



# Q-learning

$\alpha$  is the learning rate: if  $\alpha = 1$  the update of Q ( $s_t, a_t$ ) is performed exactly with the Bellman equation, if  $\alpha < 1$ , the modification goes in the direction suggested by the Bellman equation (but with smaller steps).

Typical initial values:  $\gamma = 0.9$ ,  $\alpha = 0.5$  (it is, generally, progressively reduced during learning)

Problem (practical): how big is Q?

All possible combinations “states”  $\times$  “possible actions”.

A robotic arm with three joints and gripper, must take a ball placed on a pedestal of random height (Y) and position (X).

State: coded with  $\Delta x$  and  $\Delta y$  of the gripper with respect to the ball + status of the gripper (open / closed).

Actions: rotate right / left each of the three joints, invert gripper status (close  $\rightarrow$  open and vice versa)  $\rightarrow$  7 actions.

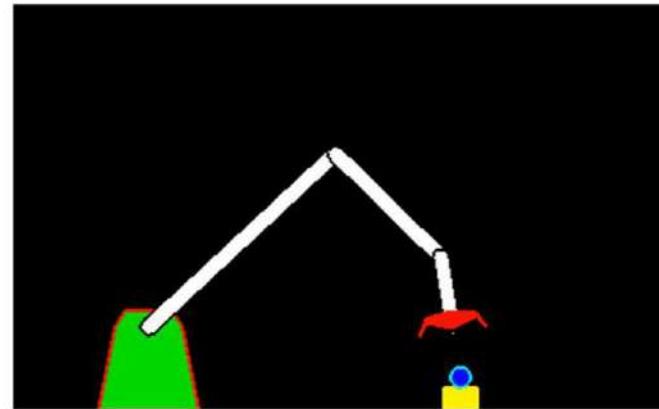
Reward:

Approach to the ball: +1

Move away from the ball: -1

movement with closed gripper = -0.2

Ball capture = +100



RL can be easily explained using the game of PacMan as [example](#).

- The [goal](#) of PacMan (the agent) is to [eat the food](#) in the grid while [avoiding the ghosts](#) on its way.
- The [grid](#) is the interactive [environment](#) for PacMan.
- PacMan can make [four different actions](#): up, down, left and right.
- PacMan receives:
  - [rewards](#) for [eating](#) food;
  - [punishments](#) if it gets killed by a [ghost](#) (loses the game).
- The [state](#) is represented by the [locations](#) of PacMan, ghosts and food in the [grid](#) world.
- The [total cumulative reward](#) is PacMan [winning](#) the game.



# Deep Q-Network (DQN)

In 2013/2014, researchers from the Deep Mind company (immediately acquired by Google) published the article “Playing Atari with Deep Reinforcement Learning” <https://arxiv.org/pdf/1312.5602.pdf> where reinforcement learning algorithms are successfully used to train a computer to play (at a super-human level) numerous Atari console games.

This in itself would not be extraordinary, except for the fact that the state  $s$  observed by the agent does not consist of game-specific numerical features (e.g. the position of the spacecraft, its speed), but of simple screen images (raw pixel).

Among other things, this allows the same algorithm to learn different games simply by playing.

The raw frames are preprocessed by first converting their RGB representation to gray-scale and down-sampling it to a  $110 \times 84$  image. The final input representation is obtained by cropping an  $84 \times 84$  region of the image that roughly captures the playing area, so the number of states is  $256^{84 \times 84} \times 4 \sim 10^{67970}$ , more than the number of atoms in the known universe! It is impossible to explicitly handle a Q table of this size.

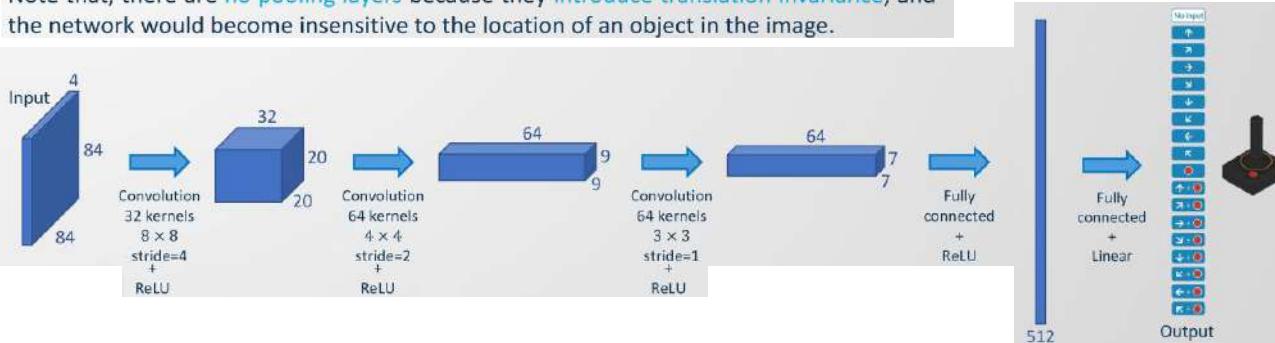
To solve the problem of **temporal limitation** and give the network the **sense of motion**, DQN takes a **stack** of four frames as input.



The DQN consists of:

- three **convolutional layers**;
- two **fully-connected layers**.

Note that, there are **no pooling layers** because they **introduce translation invariance**, and the network would become insensitive to the location of an object in the image.



**examples of DQN:**  
[https://www.youtube.com/watch?v=V1eYniJ0Rnk&ab\\_channel=TwoMinutePapers](https://www.youtube.com/watch?v=V1eYniJ0Rnk&ab_channel=TwoMinutePapers)

[https://www.youtube.com/watch?v=4MIZncshy1Q&ab\\_channel=benbuc](https://www.youtube.com/watch?v=4MIZncshy1Q&ab_channel=benbuc)

```
initialize network  $Q$  with random weights
for  $m$  episodes
     $t = 0$ 
    repeat
        with probability  $\epsilon$  select a random action  $a_t$  otherwise select  $a_t = \text{argmax}_a Q(s_t, a)$ 
        execute action  $a_t$  and observe reward  $r_{t+1}$  and new state  $s_{t+1}$ 
        estimate the target value  $y_t = r_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a)$ 
        perform a gradient descent step to update  $Q$  weights by minimizing  $l = (y_t - Q(s_t, a_t))^2$ 
         $t = t + 1$ 
    until terminated
end for
```

## Deep Q-network – training algorithm

# Deep Q-Network (DQN)

There is an issue when using neural network as Q approximator: the transitions are very correlated since they are all extracted from the same episode reducing the overall variance.

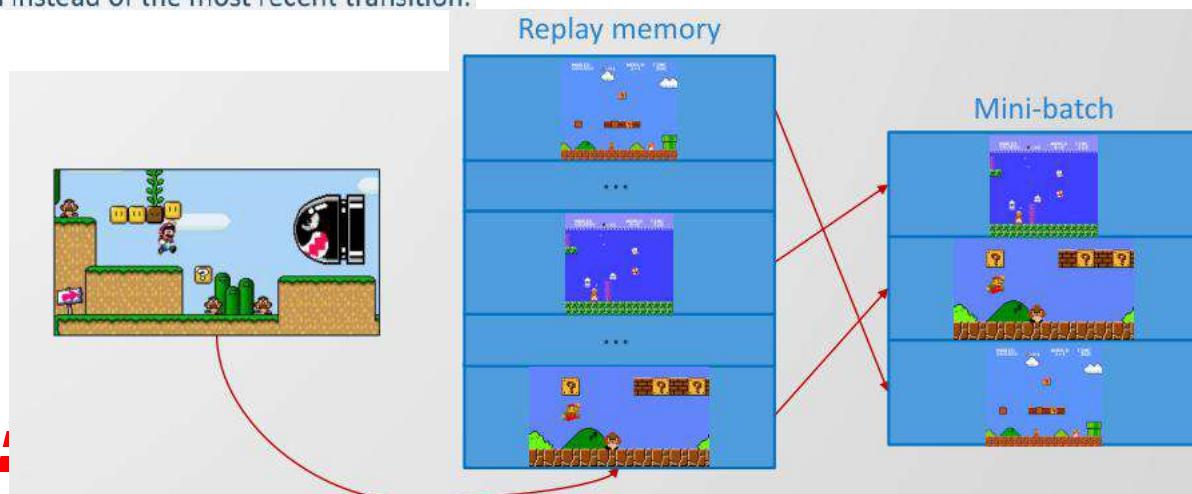
As a result, the network tends to forget the previous transitions as it overwrites them with new ones resulting in a network overfitted on the current episode.



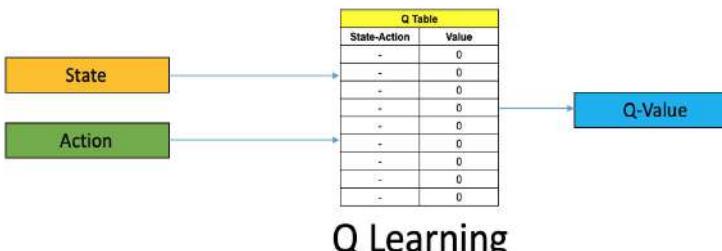
For instance, if we are in the first level and then in the second (which is totally different), the Mario agent can forget how to behave in the first level.

To remove correlations and make the DQN training more stable, the *experience replay* technique can be used:

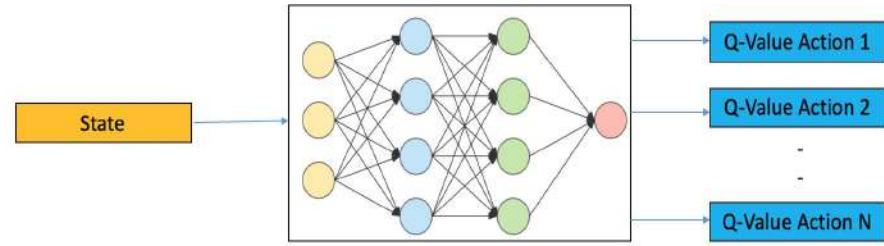
- during training, all transitions are stored in a replay memory;
- when updating the network, mini-batches are randomly sampled from the replay memory and used instead of the most recent transition.



# Deep Q-Network



**Q Learning**



**Deep Q Learning**

The idea consists in approximating the Q function with a deep neural network (CNN) which, for each input state, provides a quality value for each possible action.

For more details see the excellent introduction by T. Matiisen:

<http://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/>

<https://www.nervanasys.com/demystifying-deep-reinforcement-learning/>

Further refinements led to the development of AlphaGo which in 2016 beat human champion Lee Sedol to Go.

AlphaGo - The Movie | Full award-winning documentary

[https://www.youtube.com/watch?v=WXuK6gekU1Y&ab\\_channel=DeepMind](https://www.youtube.com/watch?v=WXuK6gekU1Y&ab_channel=DeepMind)

Q Learning with a Neural Network - Pong:

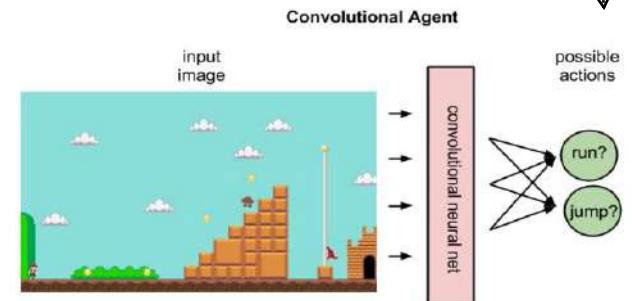
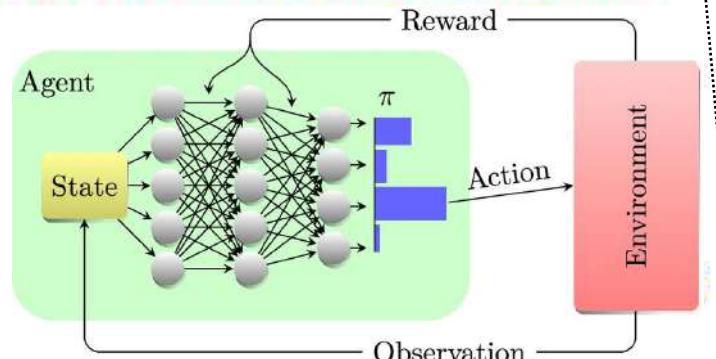
<https://www.youtube.com/watch?v=bPkWMIcq2tc>

Another example (source code in Python): Deep Reinforcement Learning: Pong from Pixels

<http://karpathy.github.io/2016/05/31/rl/>

<https://www.deepmind.com/blog/alphadev-discovers-faster-sorting-algorithms>

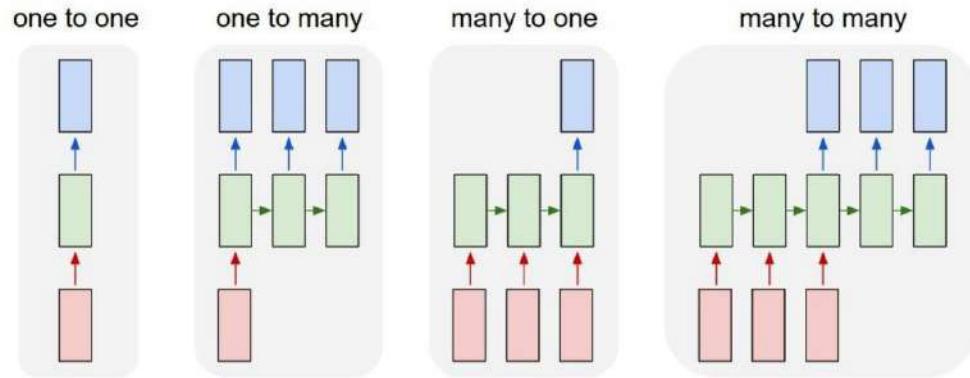
AlphaDev, an artificial intelligence (AI) system that uses reinforcement learning to discover enhanced computer science algorithms – surpassing those honed by scientists and engineers over decades. AlphaDev uncovered a faster algorithm for sorting



# *Sequence based neural networks*



# Sequence to sequence



The feedforward networks (eg MLP, CNN) studied so far operate on input of a predetermined size. For example, the input is an image vector and the output a probability vector of the classes. This case is represented by the first column of the figure as a “one to one sequence”.

Different applications require that the input and / or output can be sequences (even of variable length).

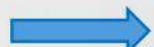
**One to many.** Eg image captioning where the input is an image and the output a sentence (sequence of words) in natural language that describes it.

**Many to one.** Eg sentiment analysis where the input is a text (eg a product review) and the output a continuous value that expresses the sentiment or judgment (positive or negative).

**Many to many.** Eg. Language translation where the input is a sentence in English and the output is its translation in Italian.

Many-to-one  
 $T_x > 1, T_y = 1$

*"This café is great, the staff is really friendly and the coffee is delicious"*



Positive

One-to-many  
 $T_x = 1, T_y > 1$



*"Dog is running through the water"*

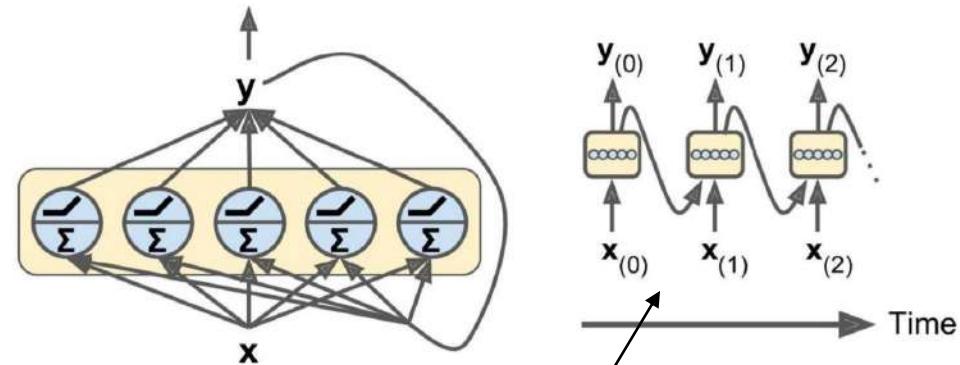
Many-to-many  
 $T_x = T_y > 1$

*"Luke joined Google as a data scientists in Mountain view"*



*"Luke<sup>PERSON</sup> joined Google<sup>ORG</sup> as a data scientists in Mountain view<sup>PLACE</sup>"*

# Recurrent Neural Networks (RNN)



In recurring networks there are also backward connections or to the same level. The most common and widespread models (eg LSTM, GRU) have connections towards the same level.

At each step of the sequence (for example at each time instant  $t$ ) in addition to the input  $x_t$ , the level also receives the output from the previous step  $y_{(t-1)}$ . This allows the network to make its decisions on past history (memory effect) or on all the elements of a sequence and on their reciprocal position.

In a sentence, it is not only the presence of specific words that is relevant but it is also important how the words are linked together (reciprocal position).

The classification of videos (image sequences) is not necessarily based on the interrelationships of the individual frames.

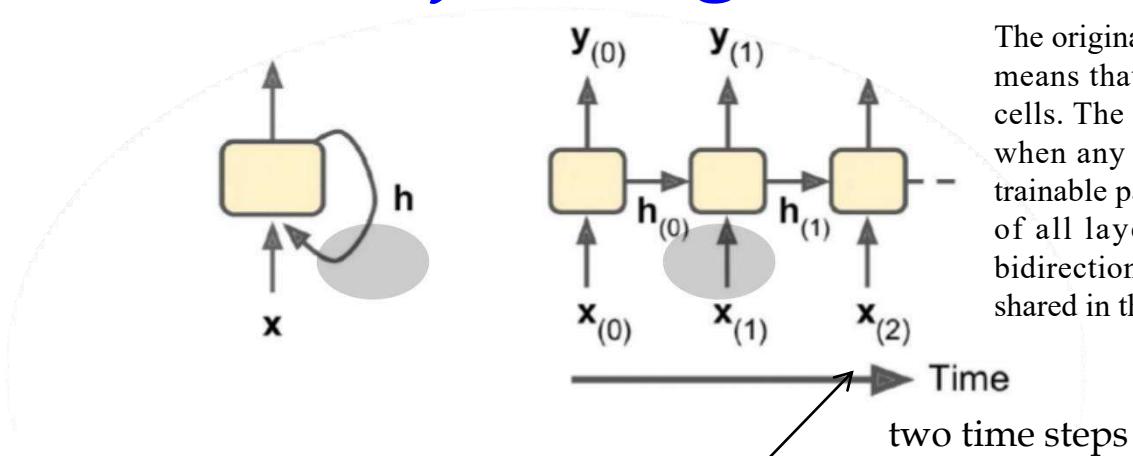
Example:

to understand if a video is a documentary on animals, the detection of animals in a few frames is sufficient (no RNN is required).

to understand sign language it is necessary to analyze the interrelationships of hand movements in the frames (It is useful to use a RNN).



# Unfolding in time



The original RNNs and LSTMs share weights over time, which means that, for example, the  $W$  weights are the same for all cells. The decision to share parameters in an RNN was made when any serious computation was a problem (1980s). The trainable parameters of an RNN include the weights and biases of all layers, which are replicated at each time step. In bidirectional networks (see page 279), however, they are not shared in the forward and backward steps.

A cell is a part of a recurring network that preserves an internal state (or memory)  $h_t$  for each instant in time. It consists of a predetermined number of neurons (it can be seen as a layer).  $h_t$  depends on the input  $x_t$  and the previous state  $h_{(t-1)}$

$$h_{(t)} = f(h_{(t-1)}, x_{(t)})$$

In order to train (with backpropagation) an RNN it is necessary to perform the so-called unfolding or unrolling in time (right part of the figure) and establishing a priori the number of time steps on which to perform the analysis.

An unfolded RNN on 20 steps is equivalent to a feedforward DNN with 20 layers. Therefore, training RNNs, that appear relatively simple, can be very expensive and critical for convergence problem (vanishing gradient). This product of matrices is the source of exploding and vanishing gradients. Gradient clipping is a technique that tackles exploding gradients. The idea of gradient clipping is very simple: If the gradient gets too large, we rescale it to keep it small. More precisely, if  $\|g\| \geq c$ , then

$$g = c \cdot g / \|g\|$$

where  $c$  is a hyperparameter,  $g$  is the gradient, and  $\|g\|$  is the norm of  $g$ . Since  $g/\|g\|$  is a unit vector, after rescaling the new  $g$  will have norm  $c$ . Note that if  $\|g\| < c$ , then we don't need to do anything. Example:  $g = [7 \quad 9 \quad 12]$ ;  $\text{norm}(g) = 16.5529$ ;  $c=5$ ;  $g1=(c*g)/\text{norm}(g)$ ;  $g1 = [2.1144 \quad 2.7185 \quad 3.6247]$ ;  $\text{norm}(g1) = 5$  ( $\text{norm}(g1)$  is equal to  $c$ )

# RNN

In a base cell of RNN:

The state  $h_t$  depends on the input  $x_t$  and the previous state  $h_{(t-1)}$

$$h_{(t)} = \phi(x_{(t)})^T \cdot W_x + h_{(t-1)}^T \cdot W_h + b$$

where:

$W_x$  and  $W_h$  are the weight vectors and  $b$  the bias to be learned.

$\phi$  is the activation function (eg Relu).

and the output  $y_t$  corresponds to the state  $h_t$ :

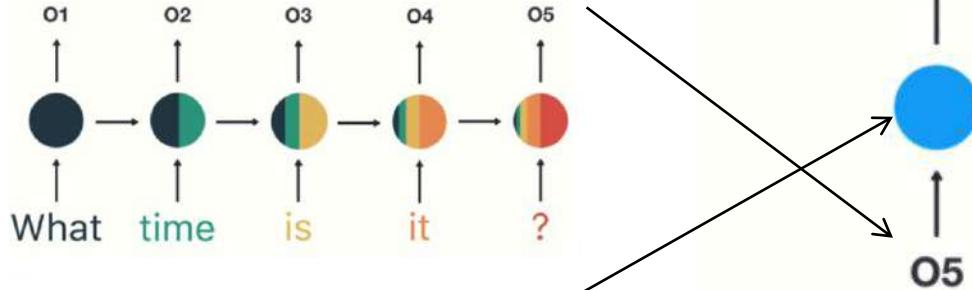
$$y_{(t)} = h_{(t)}$$

The base cells have difficulty remembering / exploiting inputs from distant steps: the memory of the first inputs tends to vanish. On the other hand, we know that in a sentence even the first words can have a very relevant importance.

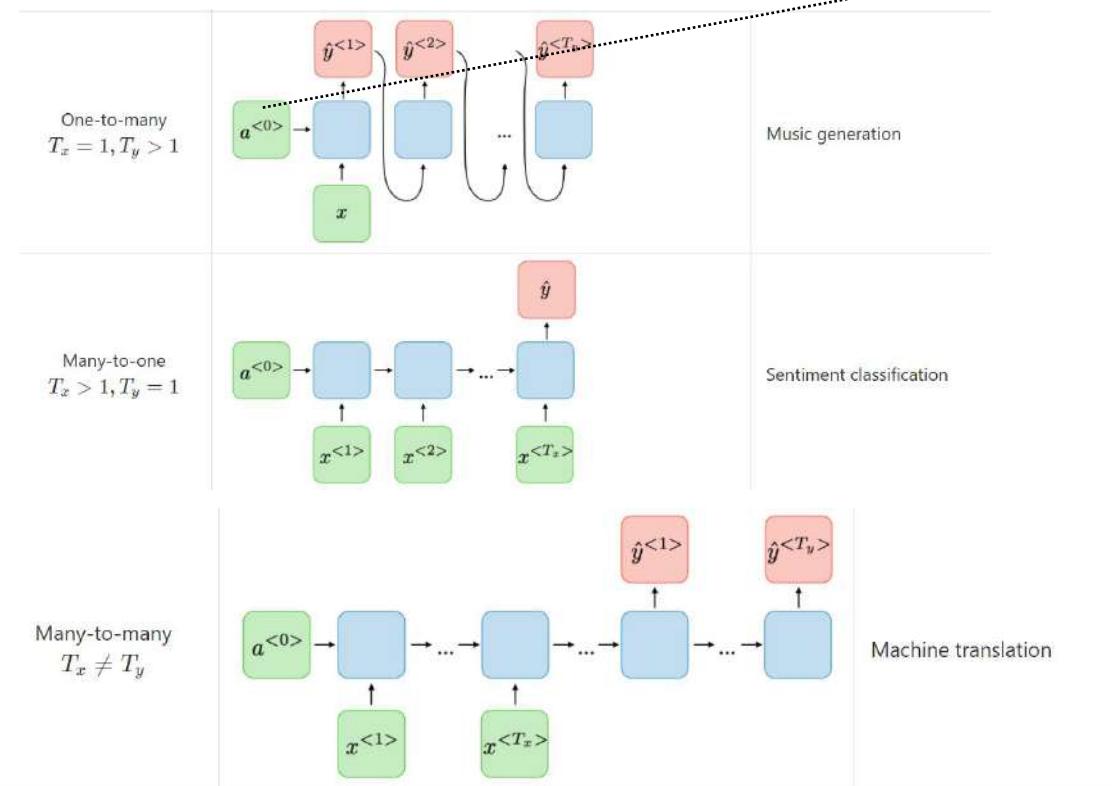
To solve this problem and facilitate convergence in complex applications, more advanced cells, with a long-term memory effect, have been proposed: LSTM and GRU are the best known topologies.



# RNN



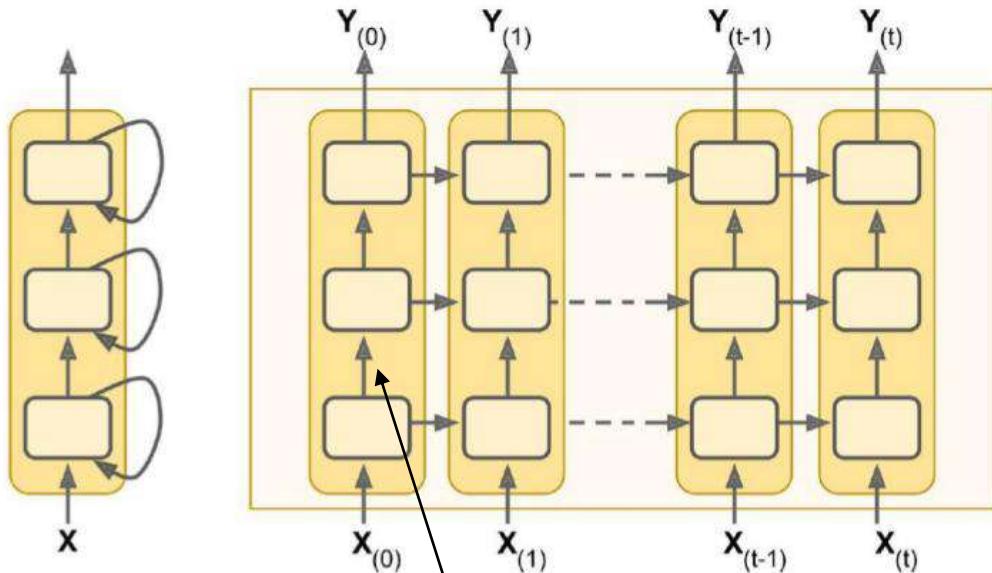
Since the final output was created from the rest of the sequence, we should be able to take the final output and pass it to the feed-forward layer to classify an intent.



Instead of randomly (or setting 0) initializing the hidden state  $a^{<0>}$ , there are smarter approaches: according to this article [Non-Zero Initial States for Recurrent Neural Networks](#), learning the initial state can speed up training and improve generalization.



# Deep RNN



In the figure an RNN consisting of three layers (stacked): on the right the unfolded version (on  $t+1$  step), the outputs of one level are the inputs of the next level.  
with more levels, more complex data relationships can be learned.  
(supervised) training always consists in providing the (desired) inputs and outputs for the last level only.

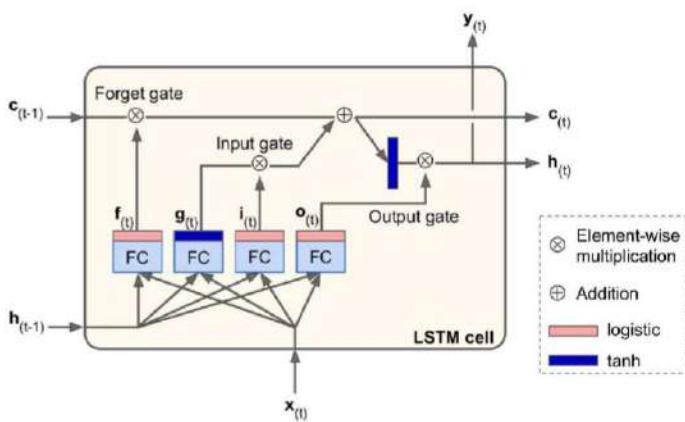
if the input are images, the  $x_t$  vectors generally do not correspond to pixels but to high-level features extracted from, e.g, a CNN or a handcrafted feature extractor.

if the input are words, the  $x_t$  vectors could be the result of word embedding  
(<https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa>).

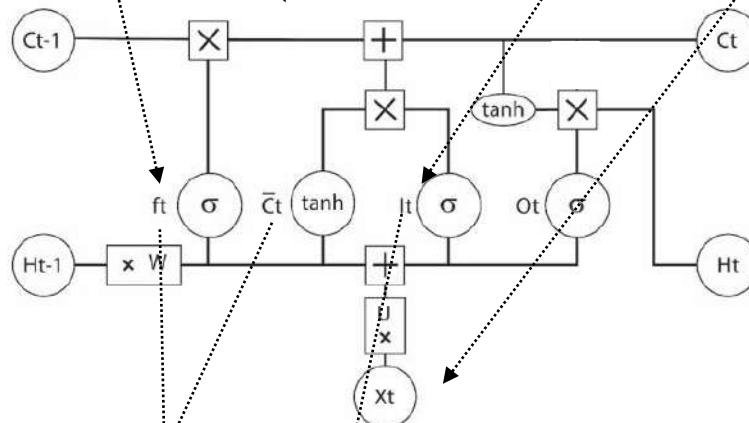


# Long Short-Term Memory

LSTM is a Recurrent Neural Network that makes a decision for what to remember at every time step. As illustrated in Figure 1, this network contains three gates: 1) input gate  $I_t$ , 2) output gate  $O_t$ , and 3) forget gate  $f_t$ , each of which consist of one layer with the sigmoid ( $\sigma$ ) activation function. LSTM also contains a specialized single layer network candidate  $\bar{C}_t$ , which has a  $Tanh$  activation function. In addition, there are four state vectors: 1) memory state  $C_t$  with 2) its previous memory state  $C_{t-1}$  and 3) hidden state  $H_t$  with 4) its previous state  $H_{t-1}$ . The variable  $X$  in Figure 1 represents the current input at time step  $t$ .



Instead of randomly (or setting 0) initializing the hidden state  $h_0$ , there are smarter approaches: according to this article [Non-Zero Initial States for Recurrent Neural Networks](#), learning the initial state can speed up training and improve generalization.



The process for updating LSTM at time  $t$  is as follows. Given  $X_t$  and  $H_{t-1}$  and letting  $U, W, b$  be the learnable weights of the network (each independent of  $t$ ), the candidate layer  $\bar{C}_t$  is

$$\bar{C}_t = \Tanh(U_c X_t + W_c H_{t-1} + b_c).$$

The next memory cell is

$$C_t = f_t * C_{t-1} + I_t * \bar{C}_t,$$

where  $*$  is element-wise multiplication.

# LSTM

The gates are defined as

$$f_t = \sigma(U_f X_t + W_f H_{t-1} + b_f),$$

$$I_t = \sigma(U_i X_t + W_i H_{t-1} + b_i);$$

$$O_t = \sigma(U_o X_t + W_o H_{t-1} + b_o).$$

The output is  $H_t = O_t * \text{Tanh}(C_t)$ .

Regarding input, all sequences for this task are of the same length, so sorting input by length is not required.

An LSTM that has two stacked layers trained on the same set of samples is called a Bidirectional LSTM (BiLSTM). The second LSTM connects to the end of the first sequence and runs in reverse. BiLSTM is best used to train data not related to time.

Illustrated Guide to LSTM's and GRU's: A step by step explanation

<https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21>

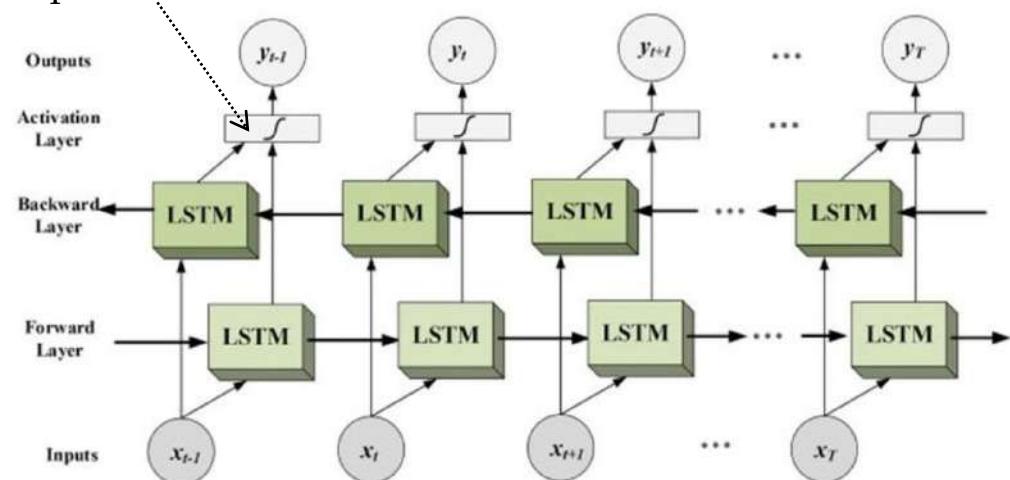
# Bi-LSTM

Bidirectional long-short term memory(bi-lstm) is the process of making any neural network have the sequence information in both directions backwards(future to past) or forward(past to future).

In bidirectional, our input flows in two directions, making a bi-lstm different from the regular LSTM. With the regular LSTM, we can make input flow in one direction, either backwards or forward. However, in bi-directional, we can make the input flow in both directions to preserve the future and the past information. For a better explanation, let's have an example.

In the sentence “boys go to .....” we can not fill the blank space. Still, when we have a future sentence “boys come out of school”, we can easily predict the past blank space the similar thing we want to perform by our model and bidirectional LSTM allows the neural network to perform this.

we can combine (before activation function) the outputs from both LSTM layers in several ways, such as average, sum, multiplication, or concatenation.



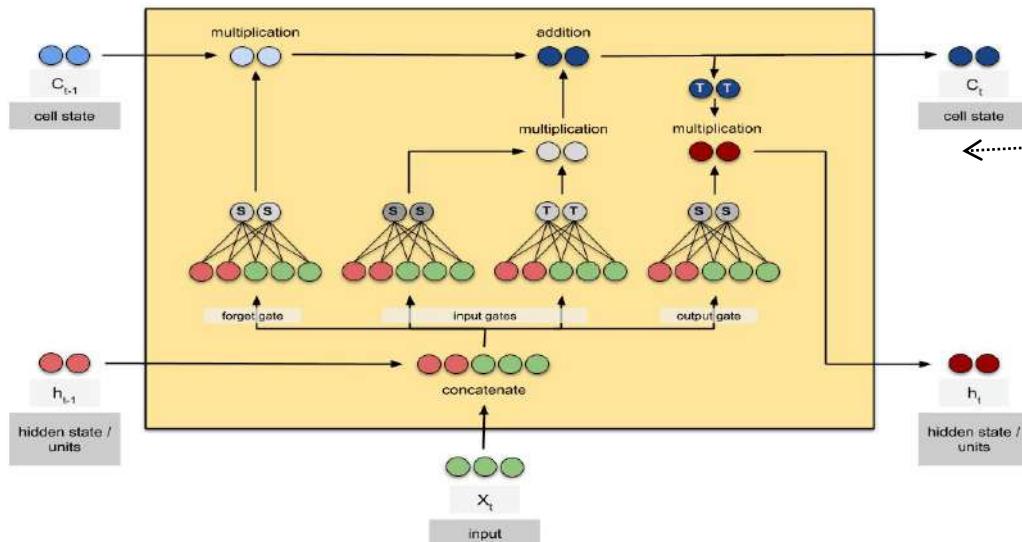
In the diagram, we can see the flow of information from backward and forward layers. BI-LSTM is usually employed where the sequence to sequence tasks are needed. This kind of network can be used in text classification, speech recognition and forecasting models.



# Classify Text Data Using LSTM

1	Sequence Input	Sequence input with 1 dimensions
2	Word Embedding Layer	Word embedding layer with 50 dimensions and 423 unique words
3	LSTM	LSTM with 80 hidden units
4	Fully Connected	4 fully connected layer
5	Softmax	softmax
6	Classification Output	crossentropyex

This example shows how to classify text data using a deep learning long short-term memory (LSTM) network. Text data is naturally sequential. A piece of text is a sequence of words, which might have dependencies between them. To learn and use long-term dependencies to classify sequence data, use an LSTM neural network. A word embedding layer maps word indices to vectors. A word embedding layer maps a sequence of word indices to embedding vectors and learns the word embedding during training.



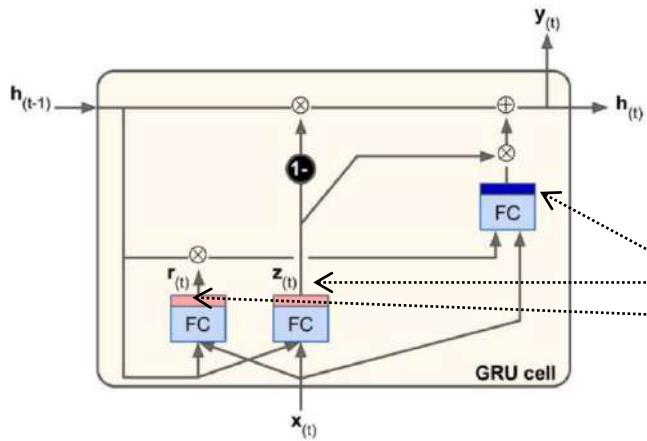
The number of units actually is the dimension of the hidden state (or the output).

For example, in the image., the hidden state (the red circles) has length 2. The number of units is the number of neurons connected to the layer holding the concatenated vector of hidden state and input. In this example, there are 2 neurons connected to that layer.

The number of units defines the dimension of hidden states (or outputs) and the number of params in the LSTM layer. We can see that basically, the LSTM layer is a feed-forward network with some more Hadamard products and matrix operations.



# GRU



GRU (Gated Recurrent Unit) is a simplified version of LSTM, which tries to maintain its advantages, reducing parameters and complexity. The main simplifications are:  
only one state of memory  $h_t$ ;  
a single gate (with inverted logic) to quantify how much to forget and how much to "add"

GRU can be considered as a simple Long-Short Term Memory (LSTM) variant. Differently from LSTM, GRU has a gate that lets the network decide which part of the old information is relevant to understand the new information. GRU has also fewer parameters and generally has a better performance on smaller data sets.

The basic components of a GRU are a reset gate and an update gate: the first determines how much old information to forget, the second determines what information should forget and what information should pass to the output. Let  $x_t$  be the input sequence and initialize  $h_0 = 0$ . Then we define the update gate vector  $z_t$  and the reset gate vector  $r_t$  as:

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z)$$

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$$

where  $W_z, U_z, b_z, W_r, U_r, b_r$  are matrices and vectors and  $\sigma$  is the sigmoid function.

Then we define:

$$\hat{h}_t = \phi(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h)$$

to be the candidate activation vector, where  $\phi$  is the tanh activation and  $\odot$  is the Hadamard (component-wise) product. Notice that the term  $r_t$  determines the amount of past information that is relevant for the candidate activation vector. Then

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

is the output vector. The update gate vector  $z_t$  measures the amount of old and new information to keep.

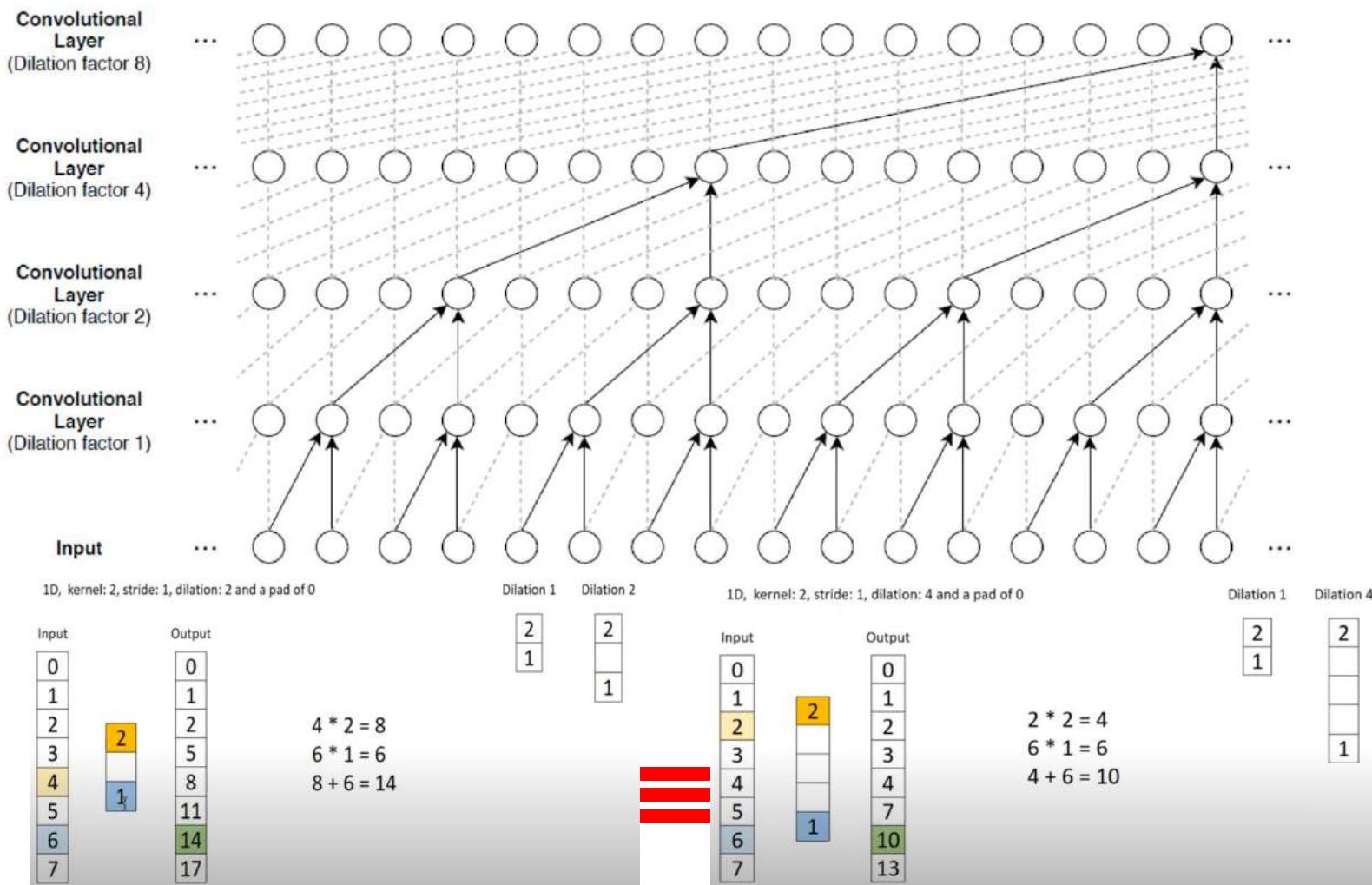
here: <http://dprogrammer.org/rnn-lstm-gru> Backpropagation for RNN, LSTM and GRU is reported



# Temporal convolutional network (TCN)

The main building block of a TCN is a dilated causal convolution layer, which operates over the time steps of each sequence. In this context, "causal" means that the activations computed for a particular time step cannot depend on activations from future time steps.

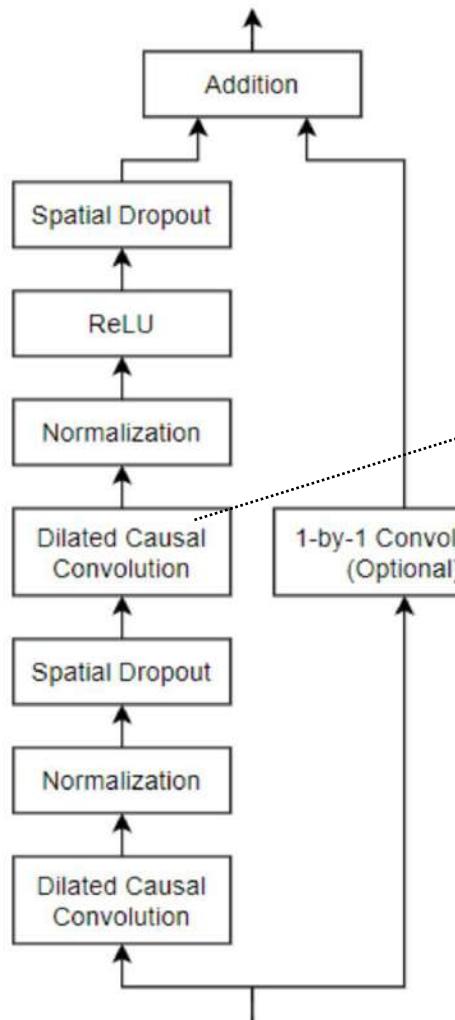
To build up context from previous time steps, multiple convolutional layers are typically stacked on top of each other. To achieve large receptive field sizes, the dilation factor of subsequent convolution layers is increased exponentially, as shown in the following image. Assuming that the dilation factor of the  $k$ -th convolutional layer is  $2^{(k-1)}$  and the stride is 1, then the receptive field size of such a network can be computed as  $R = (f - 1)(2^K - 1) + 1$ , where  $f$  is the filter size and  $K$  is the number of convolutional layers. Change the filter size and number of layers to easily adjust the receptive field size and the number of learnable parameters as necessary for the data and task at hand.



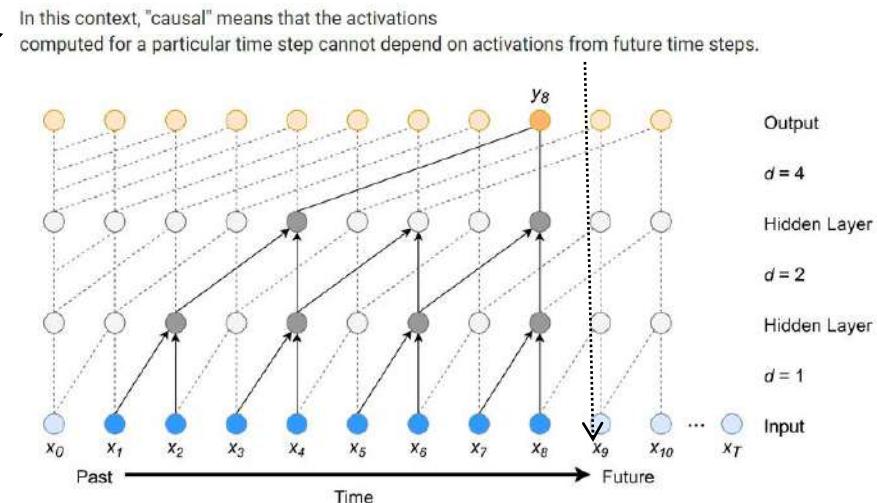
# Temporal convolutional network

One of the disadvantages of TCNs compared to recurrent networks is that they have a larger memory footprint during inference. The entire raw sequence is required to compute the next time step. To reduce inference time and memory consumption, especially for step-ahead predictions, train with the smallest sensible receptive field size  $R$  and perform prediction only with the last  $R$  time steps of the input sequence.

The general TCN architecture (as described in [\[●\]](#)) consists of multiple residual blocks, each containing two sets of dilated causal convolution layers with the same dilation factor, followed by normalization, ReLU activation, and spatial dropout layers. The network adds the input of each block to the output of the block (including a 1-by-1 convolution on the input when the number of channels between the input and output do not match) and applies a final activation function.



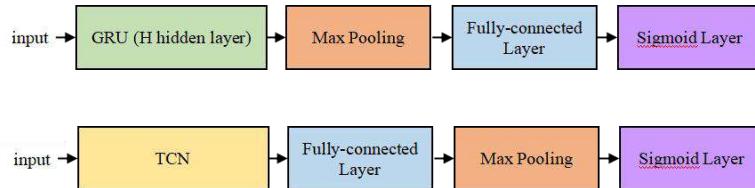
<https://arxiv.org/abs/1803.01271>



A *causal convolution* is a convolution layer to ensure there is no information "leakage" from future into past. In other words, given an time-series input  $x_0, \dots, x_T$ , the predicted output  $\hat{y}_t$  at a time instant  $t$  depends only on the inputs at time  $t$  and earlier  $x_t, x_{t-1}, \dots, x_{t-r+1}$ .

# Recurrent DNN for multilabel classification <https://www.mdpi.com/2624-6120/3/4/54>

We create a Deep Neural Network (DNN) architecture based on Gated Recurrent Unit (GRU) and Temporal Convolutional Neural Network (TCN) both adapted to a multilabel classification problem. The base schema of the model is a GRU with  $N$  hidden units (e.g. number  $N$  of hidden units is set to 50), followed by a max pooling layer and a fully connected layer. The output layer is a sigmoid that provides multiclass classification. The TCN base approach has a similar architecture but with max pooling following the fully connected layer.



Schema of multilabel recurrent DNN.

The loss function is the Binary Cross-Entropy loss between the outputs (predicted labels) and the actual (target) labels. The multilabel Binary Cross-Entropy loss calculates the loss in the following way:

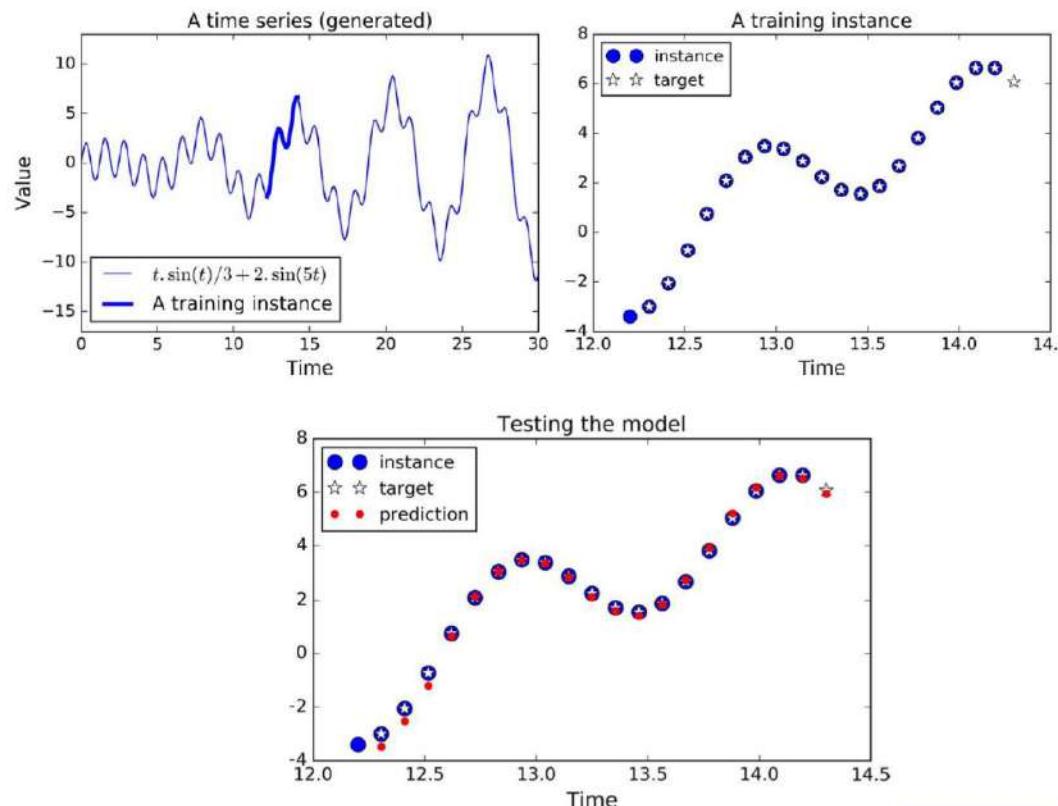
$$\text{CELoss} = -\frac{1}{m} \sum_{i=1}^m \sum_{j=1}^l y_i(j) \cdot \log(h_i(j)) + (1 - y_i(j)) \cdot \log(1 - h_i(j))$$

where  $y_i \in \{0, 1\}^l$  and  $h_i \in \{0, 1\}^l$  are the actual and predicted label vectors of each sample ( $i \in 1 \dots m$ ), respectively.



# Temporale series, examples

A time series can be related to the trend over time, e.g:  
a stock exchange, the temperature of an environment, the energy consumption of a system, etc.  
We can consider a time series as a function sampled in several time instants. Knowing the values up to instant  $t$ , we are interested in predicting the value at  $t + 1$ . Please note: not only 1D input.



# Let's see an example: Matlab\_Examples\MultilabelGR U

the dataset used in the example is the following:

Scene |: this is a collection of 2407 color images (divided into training and testing images) of different scenes grouped into six base categories *beach* (369), *sunset* (364), *fall foliage* (360), *field* (327), *mountain* (223), and *urban* (210), with sixty-three images belonging to two categories and only one to three categories. Taking into account the joint categories, the total number of labels is fifteen. Each image in this data set went through a four-step preprocessing procedure. In step 1, an image was converted to the CIE Luv space since this space is perceptually uniform (i.e., close to Euclidean distances). In step 2, the image was divided into 49 blocks with a  $7 \times 7$  grid. In step 3, the mean and variance of each band were computed. The mean is equivalent to a low-resolution image, whereas the variance corresponds to computationally inexpensive texture features. Finally, in step 4, the image was transformed into a 294-dimensional feature vector ( $49 \times 3 \times 2$ ).



# Generative Adversarial Network



<https://impakter.com/art-made-by-ai-wins-fine-arts-competition/>  
An artwork made by Artificial Intelligence (AI) won first place at the Colorado State Fair's fine arts competition last week, sparking controversy about whether AI-generated art can be used to compete in competitions.

Sony World Photography Award 2023: Winner refuses award after revealing AI creation. The winner of a major photography award has refused his prize after revealing his work was in fact an AI creation.

<https://www.bbc.com/news/entertainment-arts-65296763>

Generative models are a class of machine learning models that can generate new data that is similar to the data they were trained on. These models are trained on a dataset and learn to understand the underlying patterns and structure of the data. Once trained, they can generate new data that is similar to the training data.

There are many types of generative models, such as generative adversarial networks (GANs), variational autoencoders (VAEs), and normalizing flow models. These models can be used to generate a wide variety of data, including images, text, and audio.

Current (February 2024) state of the art - image generation:

<https://www.tomsguide.com/news/midjourney-v6-released-this-new-ai-model-brings-photorealism-to-image-generation>

<https://venturebeat.com/ai/midjourney-v6-is-here-with-in-image-text-and-completely-overhauled-prompting>

Current (February 2024) state of the art - video generation:

<https://venturebeat.com/ai/runways-gen-2-update-is-blowing-peoples-minds-with-incredible-ai-video/>

How to Use Midjourney:

[https://www.youtube.com/watch?app=desktop&v=ABWYANWWExE&ab\\_channel=HitPaw](https://www.youtube.com/watch?app=desktop&v=ABWYANWWExE&ab_channel=HitPaw)

Stability AI has introduced Stable Audio - a generative model designed to generate music and sounds based on user-provided text prompts <https://neurohive.io/en/ai-apps/stable-audio/>

Midjourney Version Comparison, the same prompt is used



**V1**

**V2**

**V3**

**V4**

**V5.0**

**V5.1**

**V5.2**

**V6**

# Generative Adversarial Network

Yann LeCun described it as “the most interesting idea in the last 10 years in Machine Learning”. Of course, such a compliment coming from such a prominent researcher in the deep learning area is always a great advertisement for the subject we are talking about! And, indeed, Generative Adversarial Networks (GANs for short) have had a huge success since they were introduced in 2014 by Ian J. Goodfellow and co-authors in the article [Generative Adversarial Nets](#).

Before going into the details, let's give a quick overview of what GANs are made for. Generative Adversarial Networks belong to the set of generative models. It means that they are able to produce / to generate (we'll see how) new content. To illustrate this notion of “generative models”, we can take a look at some well known examples of results obtained with GANs.

Naturally, this ability to generate new content makes GANs look a little bit “magic”, at least at first sight. In the following parts, we will overcome the apparent magic of GANs in order to dive into ideas, maths and modelling behind these models. Not only we will discuss the fundamental notions Generative Adversarial Networks rely on but, more, we will build step by step and starting from the very beginning the reasoning that leads to these ideas.

- GAN are the networks used for the artificial generation of images / videos: from the photorealistic faces of NVIDIA <https://thispersondoesnotexist.com/> deep fake <https://it.wikipedia.org/wiki/Deepfake>
- FaceApp, whose AI technology can age your appearance <https://www.youtube.com/watch?v=-8XIR3OwKKo>

Applications – image inpainting

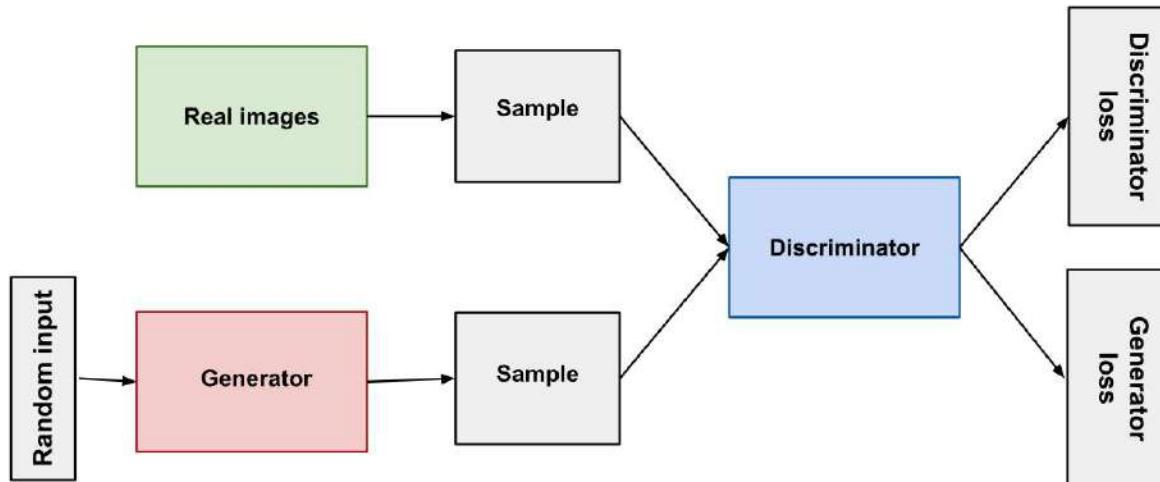


# Overview of GAN

A generative adversarial network (GAN) has two parts:

- The **generator** learns to generate plausible data. The generated instances become negative training examples for the discriminator.
- The **discriminator** learns to distinguish the generator's fake data from real data. The discriminator penalizes the generator for producing implausible results.

When training begins, the generator produces obviously fake data, and the discriminator quickly learns to tell that it's fake:



Both the generator and the discriminator are neural networks. The generator output is connected directly to the discriminator input. Through [backpropagation](#), the discriminator's classification provides a signal that the generator uses to update its weights.





# GAN

## Generative models

**We try to generate very complex random variables...**

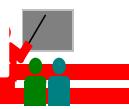
Suppose that we are interested in generating black and white square images of dogs with a size of  $n$  by  $n$  pixels. We can reshape each data as a  $N=n \times n$  dimensional vector (by stacking columns on top of each others) such that an image of dog can then be represented by a vector. However, it doesn't mean that all vectors represent a dog once shaped back to a square! So, we can say that the  $N$  dimensional vectors that effectively give something that look like a dog are distributed according to a very specific probability distribution over the entire  $N$  dimensional vector space (some points of that space are very likely to represent dogs whereas it is highly unlikely for some others). In the same spirit, there exists, over this  $N$  dimensional vector space, probability distributions for images of cats, birds and so on.

Then, the problem of generating a new image of dog is equivalent to the problem of generating a new vector following the "dog probability distribution" over the  $N$  dimensional vector space. So we are, in fact, facing a problem of generating a random variable images that looks like dog and others that doesn't) we obviously don't know how to express explicitly this distribution. Both previous points make the process of generating random variables from this distribution pretty difficult. Let's then try to tackle these two problems in the following.

**... so let's use transform method with a neural network as function!**

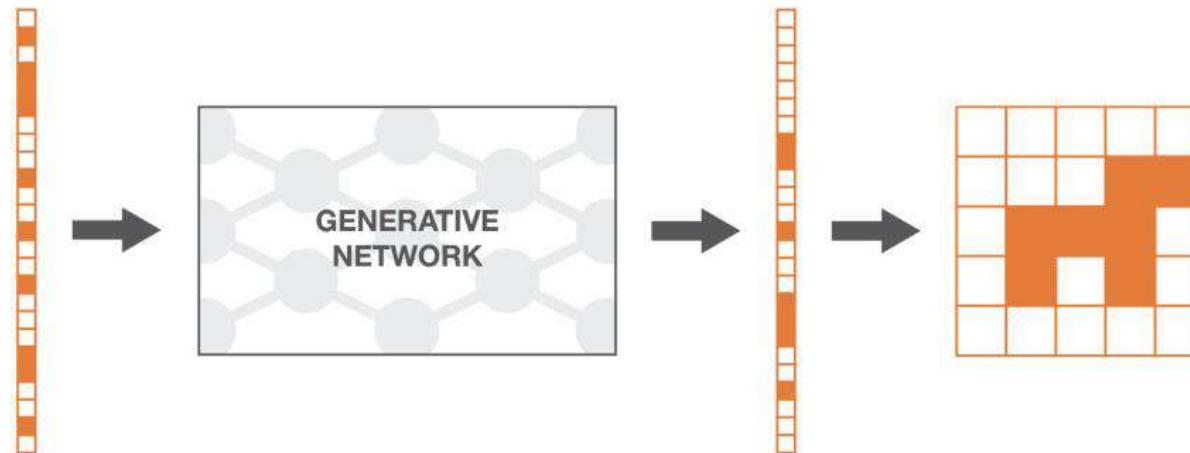
Our first problem when trying to generate our new image of dog is that the "dog probability distribution" over the  $N$  dimensional vector space is a very complex one and we don't know how to directly generate complex random variables. However, as we know pretty well how to generate  $N$  uncorrelated uniform random variables, we could make use of the transform method. To do so, we need to express our  $N$  dimensional random variable as the result of a very complex function applied to a simple  $N$  dimensional random variable!

The transform function can't be explicitly expressed and, then, we have to learn it from data.



# GAN

As most of the time in these cases, very complex function naturally implies neural network modelling. Then, the idea is to model the transform function by a neural network that takes as input a simple N dimensional uniform random variable and that returns as output another N dimensional random variable that should follow, after training, the right “dog probability distribution”. Once the architecture of the network has been designed, we still need to train it.



GMN (generative matching networks, a simplified version of GAN based only on a generator network)

---

Input random variable  
(drawn from a simple  
distribution, for  
example uniform).

The generative network  
transforms the simple  
random variable into  
a more complex one.

Output random variable  
(should follow the targeted  
distribution, after training  
the generative network).

The output of the  
generative network  
once reshaped.

Illustration of the notion of generative models using neural networks. Obviously, the dimensionality we are really talking about are much higher than represented here.



# GAN

## Training generative models

So far, we have shown that our problem of generating a new image of dog can be rephrased into a problem of generating a random vector in the N dimensional vector space that follows the “dog probability distribution” and we have suggested to use a transform generated probability distributions and backpropagating the difference (the error) through the network. This is the idea that rules Generative Matching Networks (GMNs). For the indirect training method, we do not directly compare the true and generated distributions. Instead, we train the generative network by making these two distributions go through a downstream task chosen such that the optimisation process of the generative network with respect to the downstream task will enforce the generated distribution to be close to the true distribution. **it will be better explained in the next pages**

## Comparing two probability distributions based on samples

As mentioned, the idea of GMNs is to train the generative network by directly comparing the generated distribution to the true one. However, we do not know how to express explicitly the true “dog probability distribution” and we can also say that the generated distribution is far too complex to be expressed explicitly. So, comparisons based on explicit expressions are not possible. However, if we have a way to compare probability distributions based on samples, we can use it to train the network. Indeed, we have a sample of true data and we can, at each iteration of the training process, produce a sample of generated data.

Although, in theory, any distance (or similarity measure) able to compare effectively two distributions based on samples can be used, we can mention in particular the Maximum Mean Discrepancy (MMD) approach. The MMD defines a distance between two probability distributions that can be computed (estimated) based on samples of these distributions.

<https://stats.stackexchange.com/questions/276497/maximum-mean-discrepancy-distance-distribution>



# GAN

## Backpropagation of the distribution matching error

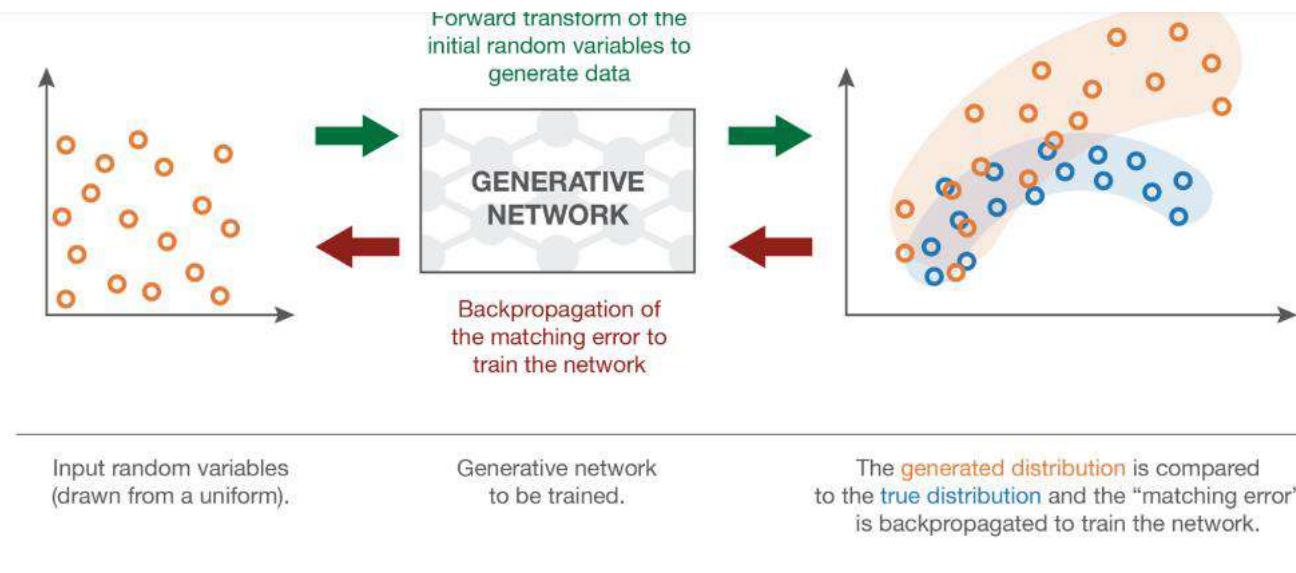
So, once we have defined a way to compare two distributions based on samples, we can define the training process of the generative network in GMNs. Given a random variable with uniform probability distribution as input, we want the probability distribution of the generated output to be the “dog probability distribution”. The idea of GMNs is then to optimise the network by repeating the following steps:

- generate some uniform inputs
- make these inputs go through the network and collect the generated outputs
- compare the true “dog probability distribution” and the generated one based on the available samples (for example compute the MMD distance between the sample of true dog images and the sample of generated ones)

MMD:maximum-mean-discrepancy-distance

- use backpropagation to make one step of gradient descent to lower the distance (for example MMD) between true and generated distributions

As written above, when following these steps we are applying a gradient descent over the network with a loss function that is the distance between the true and the generated distributions at the current iteration.



# GAN

## The “indirect” training method

The “direct” approach presented above compare directly the generated distribution to the true one when training the generative network. The brilliant idea that rules GANs consists in replacing this direct comparison by an indirect one that takes the form of a downstream task over these two distributions. The training of the generative network is then done with respect to this task such that it forces the generated distribution to get closer and closer to the true distribution.

that in the end replaces the need for MMD distance

The downstream task of GANs is a discrimination task between true and generated samples. Or we could say a “non-discrimination” task as we want the discrimination to fail as much as possible. So, in a GAN architecture, we have a discriminator, that takes samples of true and generated data and that try to classify them as well as possible, and a generator that is trained to fool the discriminator as much as possible. Let’s see on a simple example why the direct and indirect approaches we mentioned should, in theory, lead to the same optimal generator.

the two classes are real and generated

## The ideal case: perfect generator and discriminator

In order to better understand why training a generator to fool a discriminator will lead to the same result as training directly the generator to match the target distribution, let’s take a simple one dimensional example. We forget, for the moment, how both generator and discriminator are represented and consider them as abstract notions

Moreover, both are supposed “perfect” (with infinite capacities) in the sense that they are not constrained by any kind of (parametrised) model.

Suppose that we have a true distribution, for example a one dimensional gaussian, and that we want a generator that samples from this probability distribution. What we called “direct” training method would then consist in adjusting iteratively the generator (gradient descent iterations) to correct the measured difference/error between true and generated distributions. Finally, assuming the optimisation process perfect, we should end up with the generated distribution that matches exactly the true distribution.



# GAN

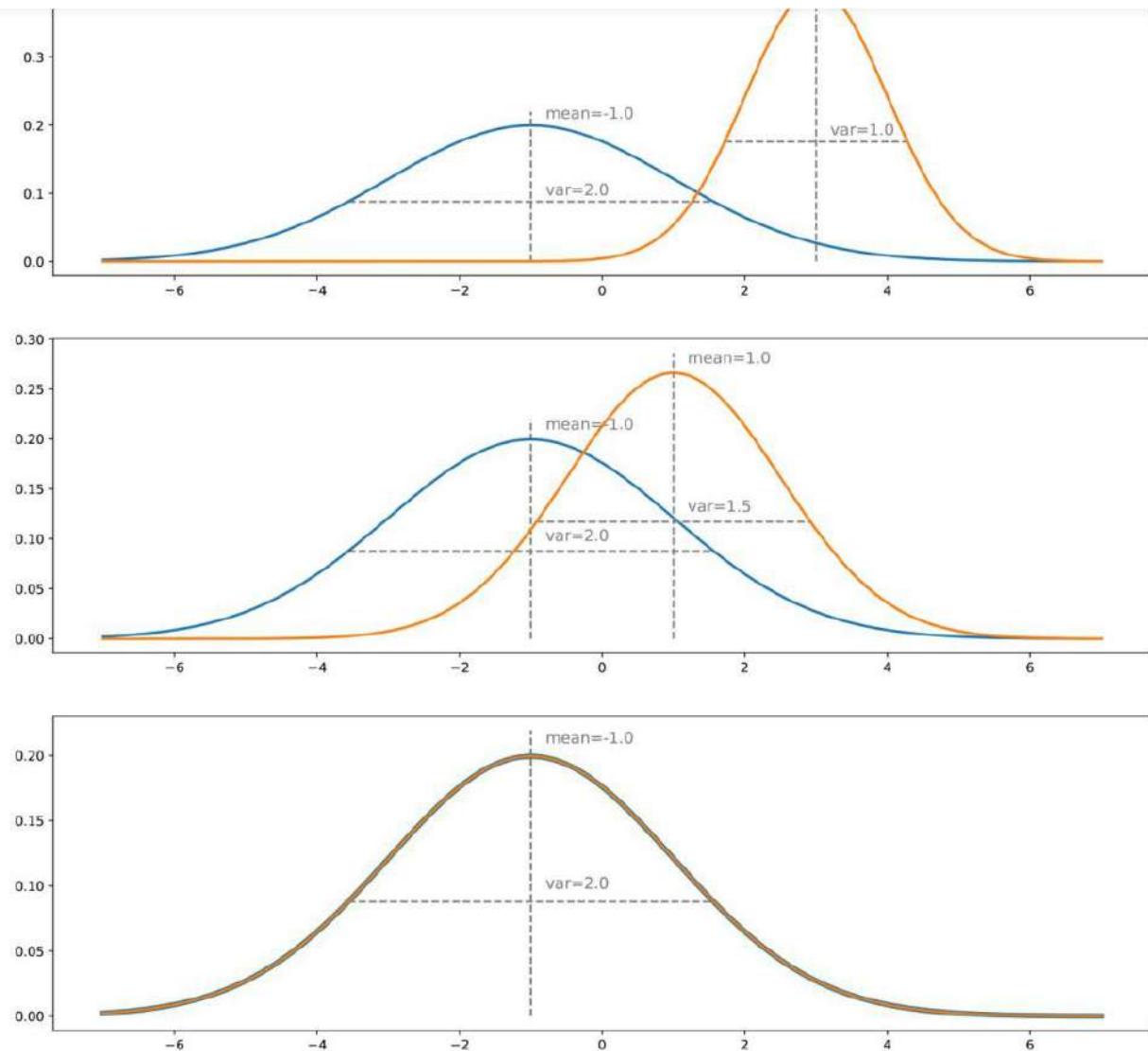
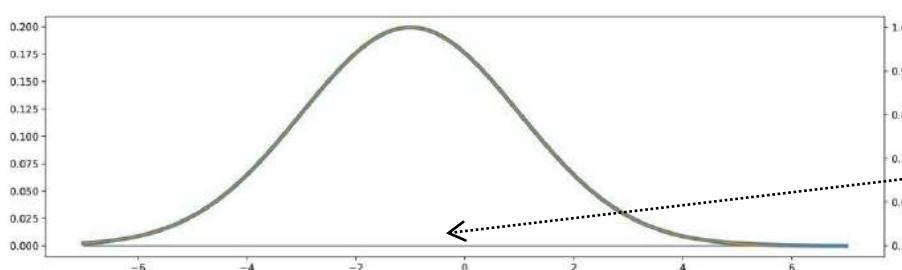
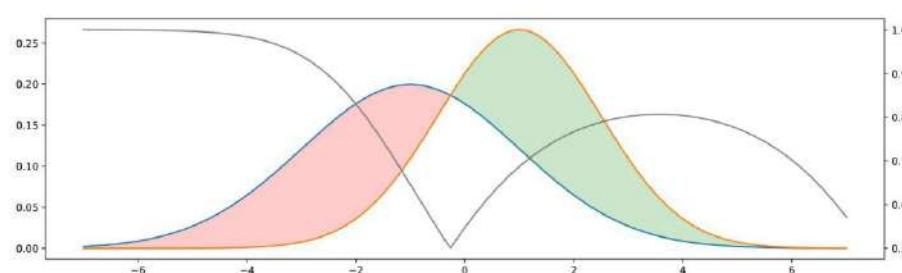
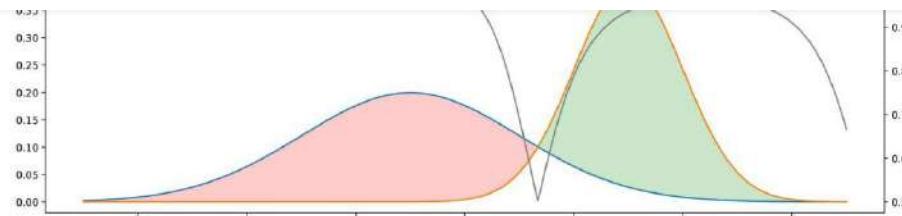


Illustration of the concept of direct matching method. The distribution in blue is the true one while the generated distribution is depicted in orange. Iteration by iteration, we compare the two distributions and adjust the networks weights through gradient descent steps. Here the comparison is done over the mean and the variance (similar to a truncated moments matching method). Notice that (obviously) this example is so simple that it doesn't require an iterative approach: the purpose is only to illustrate the intuition given above.

# GAN

For the “indirect” approach, we have to consider also a discriminator. We assume for now that this discriminator is a kind of oracle that knows exactly what are the true and generated distribution and that is able, based on this information, to predict a class (“true” or “generated”) for any given point. If the two distributions are far apart, the discriminator will be able to classify easily and with a high level of confidence most of the points we present to it. If we want to fool the discriminator, we have to bring the generated distribution close to the true one. The discriminator will have the most difficulty to predict the class when the two distributions will be equal in all points: in this case, for each point there are equal chances for it to be “true” or “generated” and then the discriminator can’t do better than being true in one case out of two in average.

At this point, it seems legit to wonder whether this indirect method is really a good idea. Indeed, it seems to be more complicated (we have to optimise the generator based on a downstream task instead of directly based on the distributions) and it requires a discriminator that we consider here as a given oracle but that is, in reality, neither known nor perfect. For the first point, the difficulty of directly comparing two probability distributions based on samples counterbalances the apparent higher complexity of indirect method. For the second point, it is obvious that the discriminator is not known. However, it can be learned!



Intuition for the adversarial method. The blue distribution is the true one, the orange is the generated one. In grey, with corresponding y-axis on the right, we displayed the probability to be true for the discriminator if it chooses the class with the higher density in each point (assuming “true” and “generated” data are in equal proportions). The closer the two distributions are, the more often the discriminator is wrong. When training, the goal is to “move the green area” (generated distribution is too high) towards the red area (generated distribution is too low).

the light gray line is always 0.5, as the two distributions are completely overlapping, so each pattern has the same probability for both distributions.

# GAN

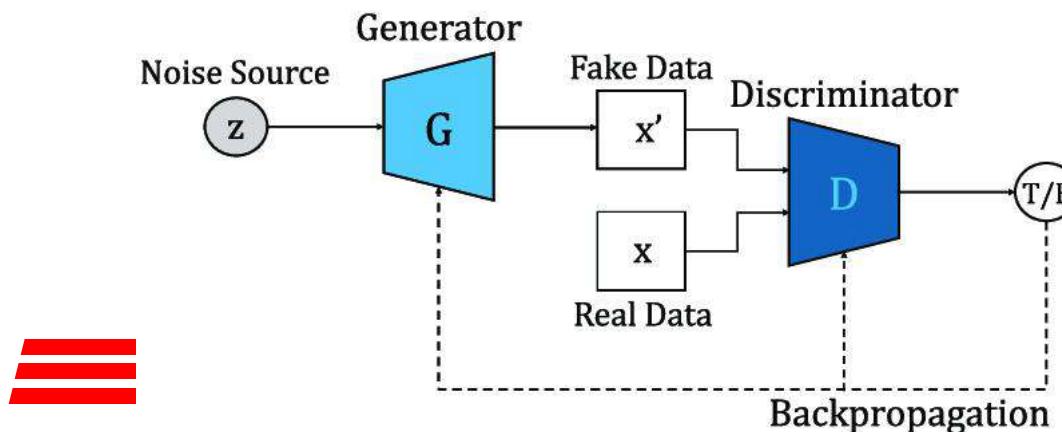
## The approximation: adversarial neural networks

Let's now describe the specific form that take the generator and the discriminator in the GANs architecture. The generator is a neural network that models a transform function. It takes as input a simple random variable and must return, once trained, a random variable that follows the targeted distribution. As it is very complicated and unknown, we decide to model the discriminator with another neural network. This neural network models a discriminative function. It takes as input a point (in our dog example a N dimensional vector) and returns as output the probability of this point to be a "true" one.

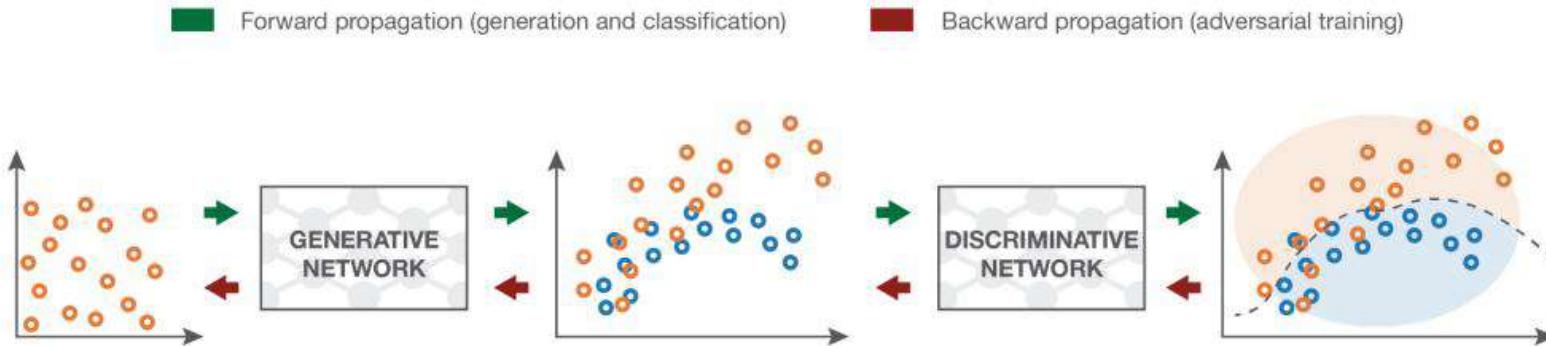
Notice that the fact that we impose now a parametrised model to express both the generator and the discriminator (instead of the idealised versions in the previous subsection) has, in practice, not a huge impact on the theoretical argument/intuition given above:

- the goal of the generator is to fool the discriminator, so the generative neural network is trained to maximise the final classification error (between true and generated data)
- the goal of the discriminator is to detect fake generated data, so the discriminative neural network is trained to minimise the final classification error

So, at each iteration of the training process, the weights of the generative network are updated in order to increase the classification error (error gradient ascent over the generator's parameters) whereas the weights of the discriminative network are updated so that to decrease this error (error gradient descent over the discriminator's parameters).



# GAN



Input random variables.

The generative network is trained to **maximise** the final classification error.

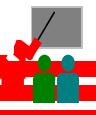
The **generated distribution** and the **true distribution** are not compared directly.

The discriminative network is trained to **minimise** the final classification error.

The classification error is the basis metric for the training of both networks.

Generative Adversarial Networks representation. The generator takes simple random variables as inputs and generate new data. The discriminator takes “true” and “generated” data and try to discriminate them, building a classifier. The goal of the generator is to fool the discriminator (increase the classification error by mixing up as much as possible generated data with true data) and the goal of the discriminator is to distinguish between true and generated data.

These opposite goals and the implied notion of adversarial training of the two networks explains the name of “adversarial networks”: both networks try to beat each other and, doing so, they are both becoming better and better. The competition between them makes these two networks “progress” with respect to their respective goals.



# Mathematical details about GANs

The adversarial modeling framework is most straightforward to apply when the models are both multilayer perceptrons. To learn the generator's distribution  $p_g$  over data  $\mathbf{x}$ , we define a prior on input noise variables  $p_z(\mathbf{z})$ , then represent a mapping to data space as  $G(\mathbf{z}; \theta_g)$ , where  $G$  is a differentiable function represented by a multilayer perceptron with parameters  $\theta_g$ . We also define a second multilayer perceptron  $D(\mathbf{x}; \theta_d)$  that outputs a single scalar.  $D(\mathbf{x})$  represents the probability that  $\mathbf{x}$  came from the data rather than  $p_g$ . We train  $D$  to maximize the probability of assigning the correct label to both training examples and samples from  $G$ . We simultaneously train  $G$  to minimize  $\log(1 - D(G(\mathbf{z})))$ :

$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_z(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))].$$

D is the discriminator; G is the generator

- Sample minibatch of  $m$  noise samples  $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$  from noise prior  $p_g(\mathbf{z})$ .
- Sample minibatch of  $m$  examples  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  from data generating distribution  $p_{\text{data}}(\mathbf{x})$ .

In practice, optimizing D upon completion in the inner loop of training is computationally prohibitive, and on finite datasets would result in overfitting. Instead, we alternate between k steps of optimizing D and one step of optimizing G. This results in D being maintained near its optimal solution, as long as G changes slowly enough.

We want G to minimise the output of D whilst D is trying to maximise the same thing. See pseudo code in the next page.

$$L^{(D)} = \max[\log(D(x)) + \log(1 - D(G(z)))]$$

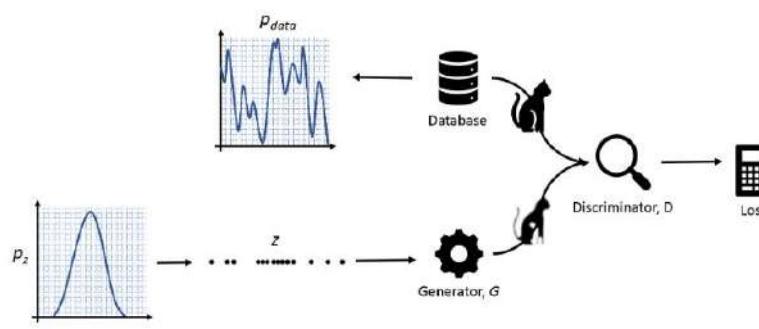
$$L^{(G)} = \min[\log(D(x)) + \log(1 - D(G(z)))]$$

notice that the component  $\log(D(x))$  is not related to the weights of G, so in the backward step its value is zero.

$$L = \min_G \max_D [\log(D(x)) + \log(1 - D(G(z)))]$$

Remember that the loss function is valid only for a single data point, to consider entire dataset we need to take the expectation of the equation (the one shown above)

# Pseudo Code



Generative Adversarial Network concept. Simple, known distribution  $p_z$  from which the vector  $z$  is drawn. Generator  $G(z)$  generates an image. Discriminator tries to determine if image came from  $G$  or from the true, unknown distribution  $p_{data}$ .

**for** number of training iterations **do**

**for**  $k$  steps **do**

- Sample minibatch of  $m$  noise samples  $\{z^{(1)}, \dots, z^{(m)}\}$  from noise prior  $p_z(z)$ .
- Sample minibatch of  $m$  examples  $\{x^{(1)}, \dots, x^{(m)}\}$  from data, generating distribution  $p_{data}(x)$ .
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m [\log D(x^{(i)}) + \log (1 - D(G(z^{(i)})))] .$$

Assume the label for the true data is 1 and 0 for the fake data.

That's the negative of the binary cross-entropy, which is a measure of classification quality; if  $D$  is 1 on the  $x$  samples and 0 on the  $G$  samples then its value is 0. If it's 1/2 on everything then the value is  $-2\log 2$  ( $= \log(0.5)+\log(0.5)$ ). So maximizing it gives a better classifier.  $D()$  is the output of the network.

**end for**

- Sample minibatch of  $m$  noise samples  $\{z^{(1)}, \dots, z^{(m)}\}$  from noise prior  $p_z(z)$ .
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(z^{(i)}))) .$$

notice that  $\log D(x)$  is not related to the generator,  $p_{data}$  is the training set with the real images

**end for**

The gradient-based updates can use any standard gradient-based learning rule.

To sum up,  $D$  and  $G$  are playing a “minimax” game with the comprehensive objective function:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)} [\log D(x)] + \mathbb{E}_{z \sim p_z(z)} [\log (1 - D(G(z)))] .$$

[https://en.wikipedia.org/wiki/Expected\\_value](https://en.wikipedia.org/wiki/Expected_value) this means that the variable 'x' or 'z' are generated from the related distribution

# beyond GAN: MvM

The capabilities of generative-adversarial neural networks (GAN) in computer vision tasks are limited by two main factors. Firstly, these neural networks model distributions using statistical characteristics, such as mean and moments, rather than geometric characteristics. Secondly, traditional GANs represent the loss of the discriminator network only in the form of a one-dimensional scalar value corresponding to the Euclidean distance between the real and fake data distributions. Because of these two problems, it is impossible to directly apply metric training methods, as well as apply new loss functions and training methods.

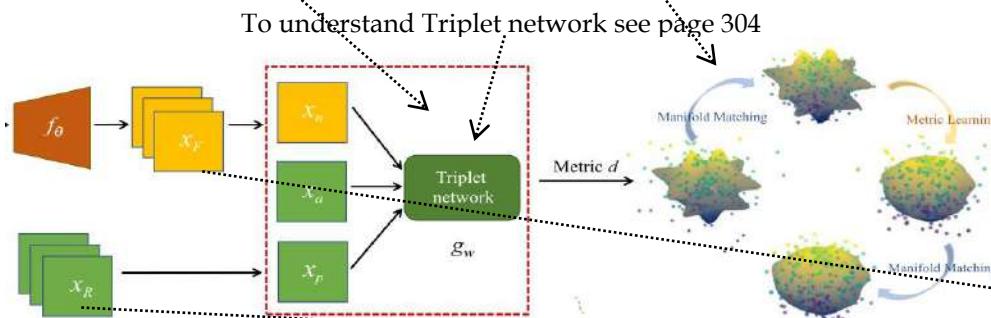
In MvM (Manifold Matching via Metric Learning), two networks are trained against each other. The metric generator network learns to determine the best metric for the distribution generator network, and the distribution generator network learns to create negative examples for the metric generator network.

Through competitive learning, MvM creates a distribution generator network that can create a fake data distribution close to the real data distribution, and a metric generator network that can provide an effective metric to capture the internal geometric structure of the data distribution.

Unlike GAN, MvM forms a multidimensional representation of images. This fact will accelerate the study of generative models and open up new potential directions that were previously considered impossible.

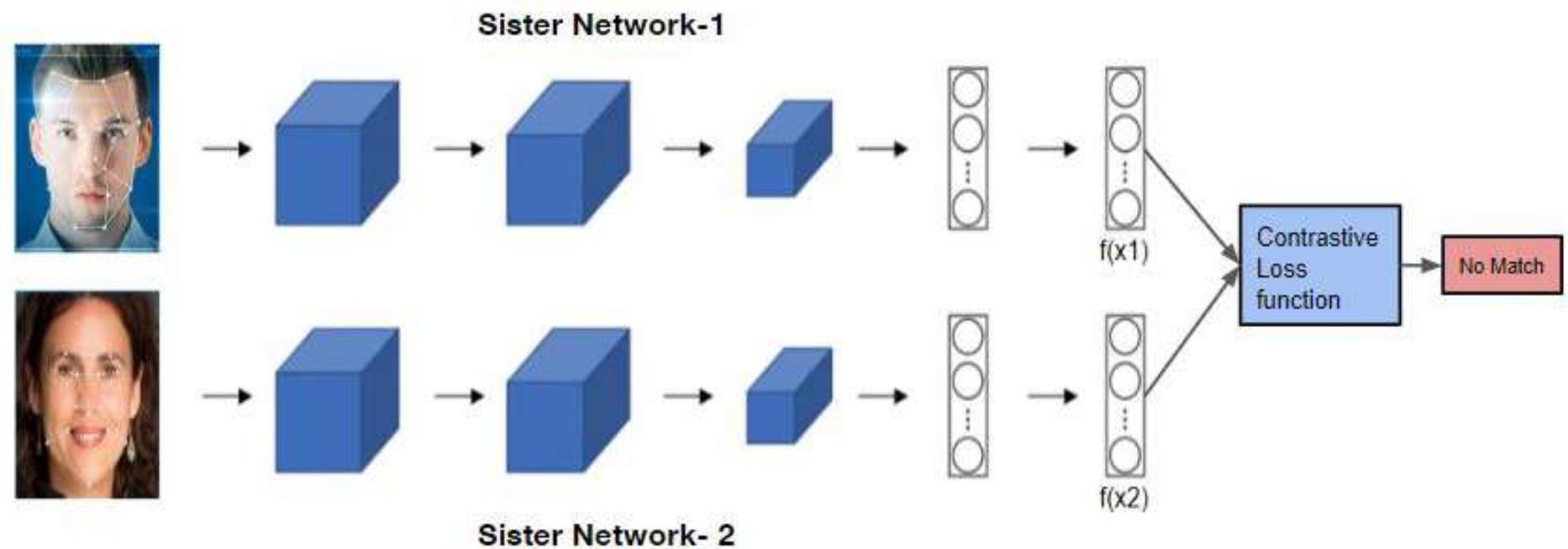
<https://arxiv.org/pdf/2106.10777.pdf>

Metric learning is an approach based directly on a distance metric that aims to estimate similarity or dissimilarity between images.  
 $dist(a, b) = \|\mathbf{G}a - \mathbf{G}b\|_2$   $\mathbf{G}$  is learnable



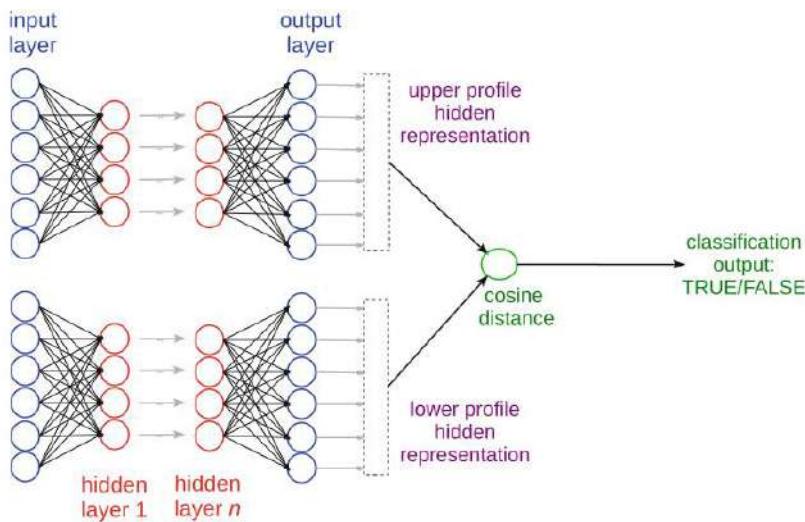
$x_R$  and  $x_F$  represent samples in real data set  $S_R$  and generated data set  $S_F$ , respectively. The distribution generator  $f_\theta$  outputs samples  $x_F \in S_F$  based on manifold matching criteria under learned metric. For Triplet metric learning, anchor samples  $x_a$  and positive samples  $x_p$  are randomly selected from  $S_R$ , and negative samples  $x_n$  are randomly selected from  $S_F$ , without labelled data involved in this step. Learned distance metric is then used for manifold matching. Manifold matching step makes the fake samples (dots) condense around the real data manifold (surface), while metric learning step tries to “straighten” or “flatten” the real data manifold. These two steps goes interchangeably until convergence.

# *Siamese Network*



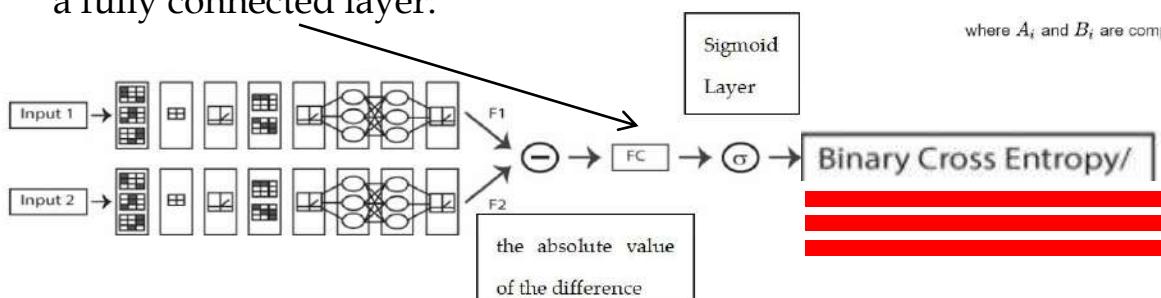
# Siamese Network

A Siamese Neural Network is a class of neural network architectures that contain two or more identical subnetworks. ‘identical’ here means, they have the same configuration with the same parameters and weights (usually, the inputs of the two networks are different). Parameter updating is mirrored across both sub-networks. It is used to find the similarity of the inputs by comparing its feature vectors, so these networks are used in many applications



Representation of the structure of the siamese neural network model. The data are processed from left to right. The value of the cosine distance is a measure of the similarity between the input pair of data instances, as final output

Another siamese topology, that uses also a fully connected layer:



The algorithm compares the output of the upper neural network and the output of the lower neural network through a distance metric

Through this similarity measure, the neural network states that the two profiles are different (cosine similarity value in the  $[-1, 0]$  interval) or similar (cosine similarity value in the  $[0, +1]$  range). The algorithm then labels the data instance as *positive* if in the former case, or as *negative* if in the latter case. The final output value can finally be compared with its corresponding ground truth value; all the classified outputs can be employed to generate a confusion matrix.

The cosine of two non-zero vectors can be derived by using the Euclidean dot product formula:

$$\mathbf{A} \cdot \mathbf{B} = \|\mathbf{A}\| \|\mathbf{B}\| \cos \theta$$

Given two vectors of attributes,  $A$  and  $B$ , the cosine similarity,  $\cos(\theta)$ , is represented using a dot product and magnitude as

$$\text{cosine similarity} = S_C(A, B) := \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = \frac{\sum_{i=1}^n A_i B_i}{\sqrt{\sum_{i=1}^n A_i^2} \sqrt{\sum_{i=1}^n B_i^2}},$$

where  $A_i$  and  $B_i$  are components of vector  $A$  and  $B$  respectively.



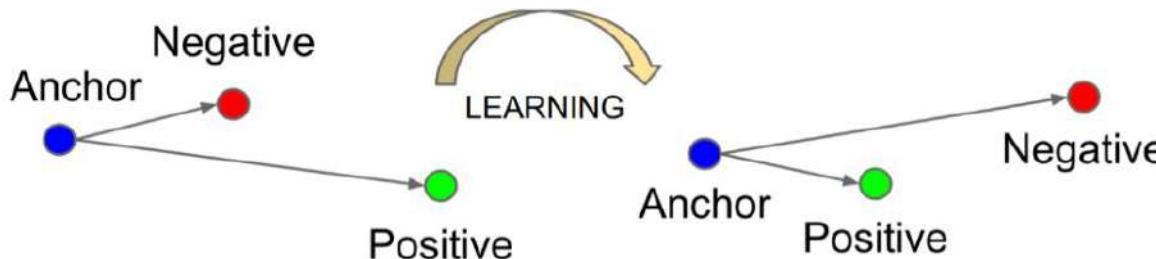
# Siamese network - Triplet loss

With Triplet, we take three images as the inputs, labelled A, P, and N. It is assumed that A and P have the same label and A and N different labels.

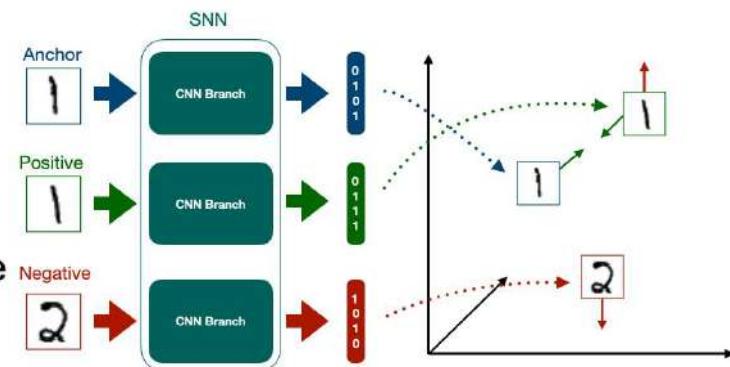
In the training phase, for every Triplet in the training set, feature vectors  $F_A, F_P, F_N$  are computed then passed through a sigmoid to obtain  $Y_A, Y_P, Y_N$ . At that point, the loss function is:

$$L = \max(|Y_A - Y_P|_2 - |Y_A - Y_N|_2, -\xi)$$

where  $\xi$  is a positive number, and  $|x|_2$  is the Euclidean norm of the vector. In other words, the loss function encourages the network to create similar representations for samples in the same class and different representations for samples in different classes.  $\xi$  is the margin, the value used is 1 because in the fixed margin tests carried out it was the one that returned the best results.



The Triplet Loss minimizes the distance between an anchor and a positive, both of which have the same identity, and maximizes the distance between the Anchor and a negative of a different identity.



# Siamese network - Contrastive loss

$$LossC = Y \frac{1}{2} D_w^2 + (1 - Y) \frac{1}{2} \{ \max(0, m - D_w) \}^2$$

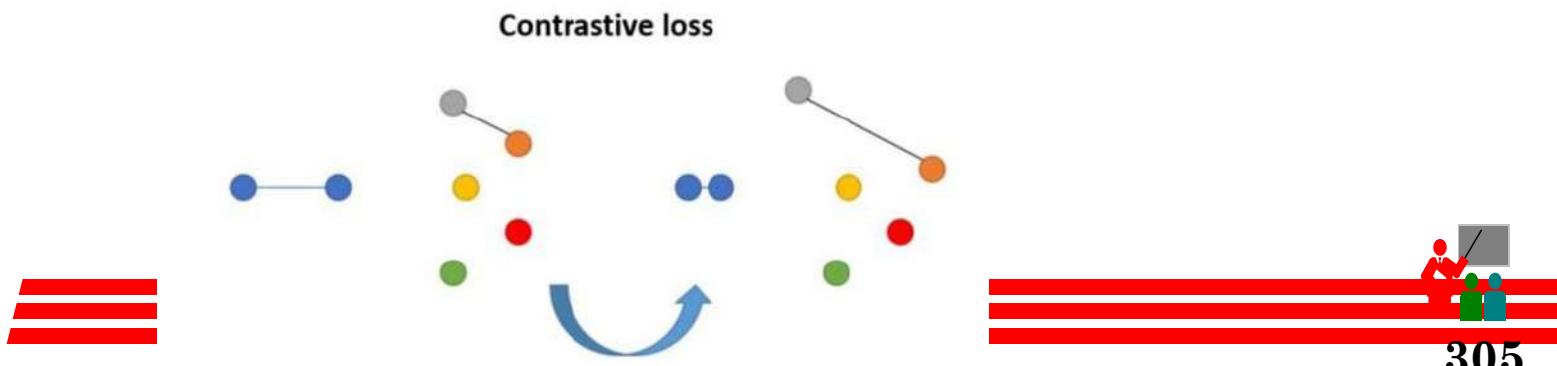
The input of this formula is  $D_w$  that is the distance between the couple of patterns

With contrastive loss, the objective to be pursued is twofold: for each class we want both to try to approach the patterns of the same class and to try to distance the patterns of the other class. The first part of the objective therefore corresponds to trying to minimize intra-class distances, while the second part corresponds to trying to maximize the interclass distances.

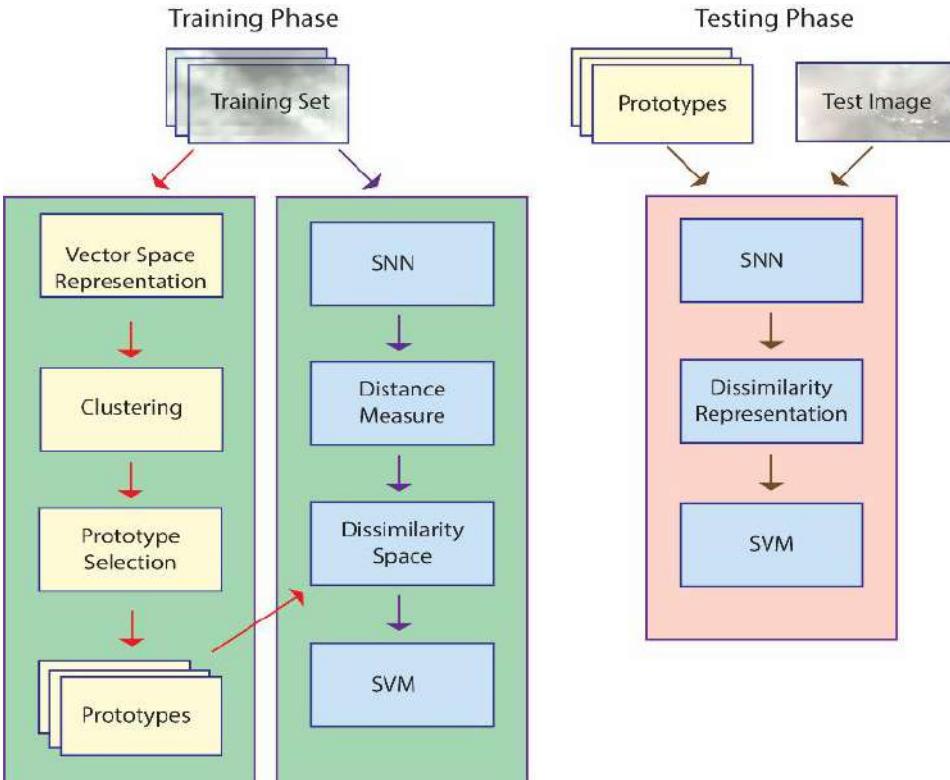
The  $D_w$  in the formula corresponds to the used distance metric, e.g. Euclidean distance of the feature vector. The "Y" in the formula changes its value depending on whether the two considered patterns belong to the same class or not:

1 if their class is the same, 0 otherwise. Consequently, if Y is 1 then only the first part of the equation is different from 0, vice versa two cases can arise. The first case is that in which the factor  $(m - D_w)$  is less than zero; the parameter  $m$  is the margin, that is the radius of the circumference within which all the patterns of the same class are to be found. If  $(m - D_w)$  is less than 0 then the two patterns are at a greater distance than the margin and therefore all the loss is equal to 0 (thanks to the "max" function). If instead  $(m - D_w)$  is greater than 0 then it means that the two patterns belong to different classes, but their distance lies within the circumference of radius  $m$ :

the loss will therefore take on a value other than 0; thanks to this the weights of the network will be modified to try to avoid this situation from happening again for the next patterns.



# Dissimilarity space using Siamese



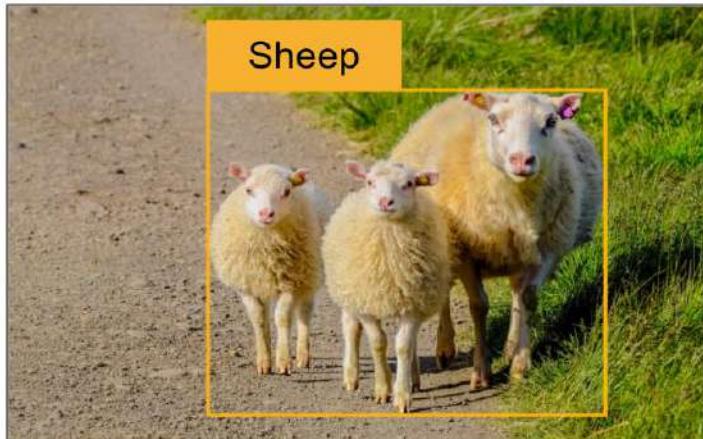
Starting with the vector space representations, step 1 of the training process begins by generating a set of clusters that produce a set of prototypes. The prototypes are centroids generated by k-means on the vector space representations. In step 2, a dissimilarity space is generated by an SNN that learns a distance measure from the prototypes that maximizes differences between pairs within class while also minimizing differences of pairs between other classes, a process that produces a feature vector that is trained on an Support Vector Machine (SVM). In the testing stage, an unknown pattern is projected onto the dissimilarity space that was learned by the SNN, which generates the feature vector that is then fed into the trained SVM for a decision.

Dissimilarity space is explained at page 47.

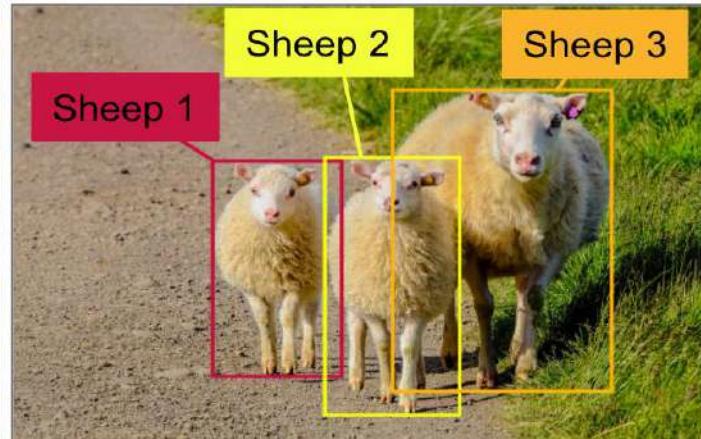
For further details: <https://www.mdpi.com/1424-8220/21/17/5809>



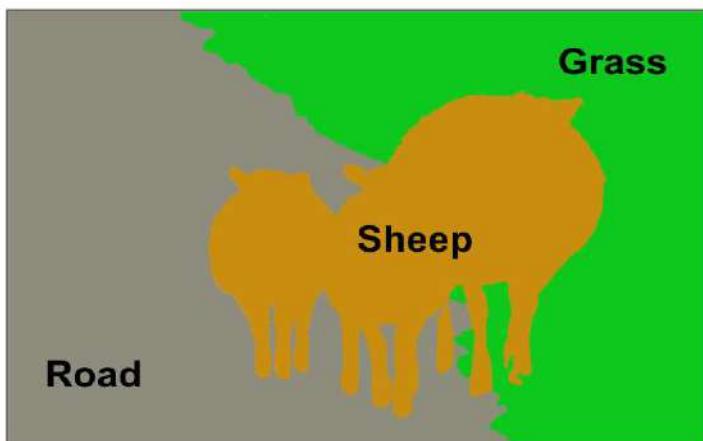
# Segmentation



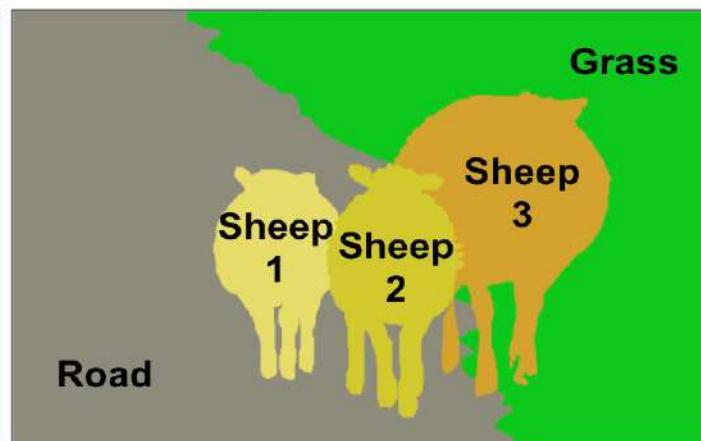
Classification + Localization



Object Detection



Semantic Segmentation



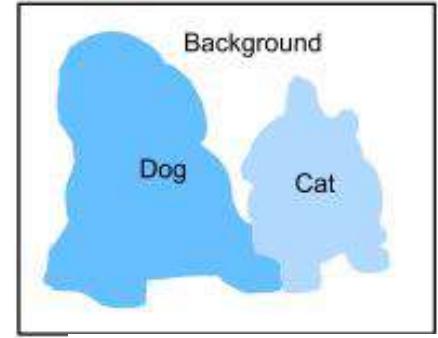
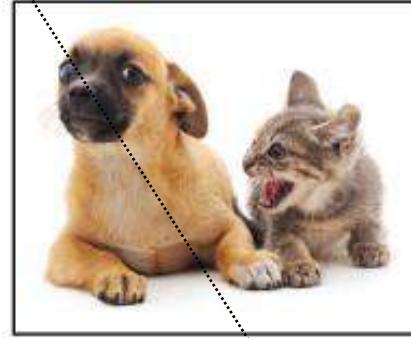
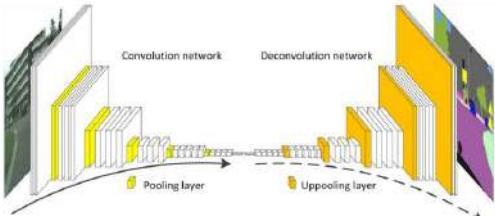
Panoptic Segmentation



# *Image/Instance Segmentation*



- E.g. output a tensor with the same shape of the input, with a class label for each pixel to produce object detection masks
  - No pooling/ padding same
  - Convolution / Deconvolution

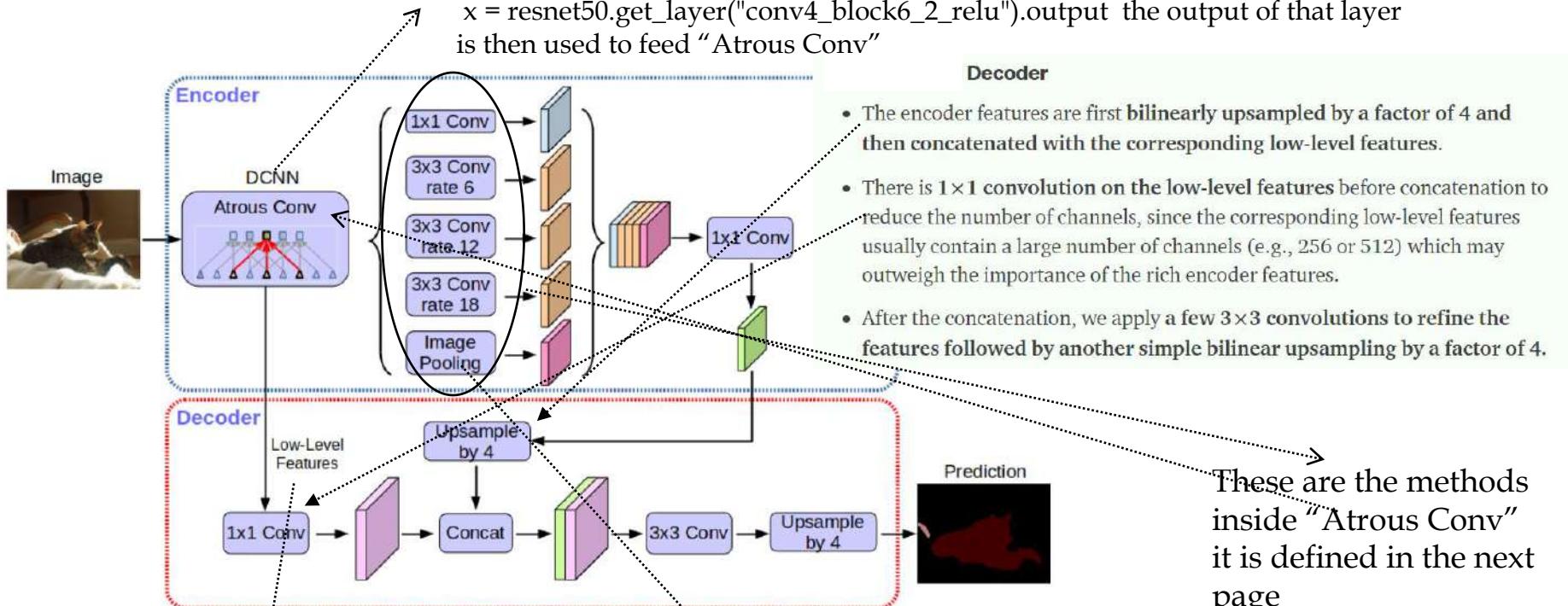


- *Image segmentation* divides an image into several constituent regions. The methods for this type of problem usually make use of the correlation between pixels in the image. It does not need label information about image pixels during training, and it cannot guarantee that the segmented regions will have the semantics that we hope to obtain during prediction. Taking the image in Fig. [ ] as input, image segmentation may divide the dog into two regions: one covers the mouth and eyes which are mainly black, and the other covers the rest of the body which is mainly yellow.
- *Instance segmentation* is also called *simultaneous detection and segmentation*. It studies how to recognize the pixel-level regions of each object instance in an image. Different from semantic segmentation, instance segmentation needs to distinguish not only semantics, but also different object instances. For example, if there are two dogs in the image, instance segmentation needs to distinguish which of the two dogs a pixel belongs to.

Suppose, you have an input image of a street view consisting of several people, cars, buildings etc. If you only want to group objects belonging to the same category, say distinguish all cars from all buildings, it is the task of semantic segmentation. Within each category say, people, if you want to distinguish each individual person, that will be the task of instance segmentation. Whereas if you want both category-wise as well as instance-wise division, it will be a panoptic segmentation task  
<https://encord.com/blog/panoptic-segmentation-guide/>.

# DeepLabV3+ - segmentation network

This is the CNN backbone, you use a CNN for representing an image, e.g.  
`x = resnet50.get_layer("conv4_block6_2_relu").output` the output of that layer  
is then used to feed "Atrous Conv"



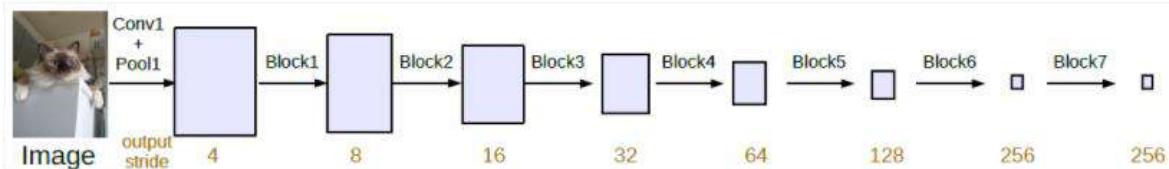
Typically, the encoder-decoder networks contain (1) an encoder module that gradually reduces the feature maps and captures higher semantic information, and (2) a decoder module that gradually recovers the spatial information.

the low level features are the output of a given layer of CNN backbone, e.g.  
`input_b = resnet50.get_layer("conv2_block3_2_relu").output`

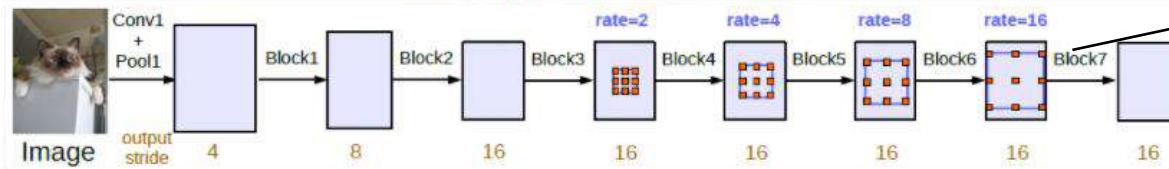
Image Pooling is composed by: average pooling layer, **1x1 convolution block**, upsampling layer for resizing to the right size for the following concatenation

# DeepLabV3+

**Atrous convolution:** Atrous convolution, a powerful tool that allows us to explicitly control the resolution of features computed by deep convolutional neural networks and adjust filter's field-of-view in order to capture multi-scale information, generalizes standard convolution operation. In the case of two-dimensional signals, for each location  $i$  on the output feature map  $y$  and a convolution filter  $w$ , atrous convolution is applied over the input feature map  $x$  as follows:

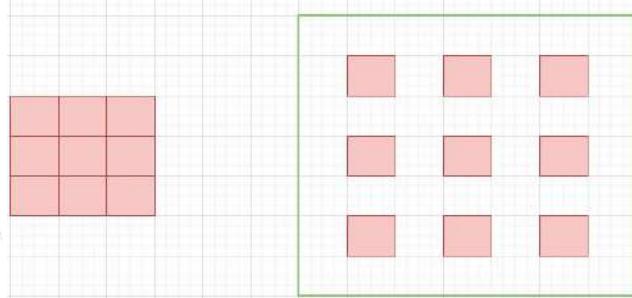


(a) Going deeper without atrous convolution.



(b) Going deeper with atrous convolution. Atrous convolution with  $rate > 1$  is applied after block3 when  $output\_stride = 16$ .

- **(a) Without Atrous Conv:** Standard conv and pooling are performed which makes the output stride increasing, i.e. the output feature map smaller, when going deeper. However, consecutive striding is harmful for semantic segmentation because location/spatial information is lost at the deeper layers.
- **(b) With Atrous Conv:** With atrous conv, we can keep the stride constant but with larger field-of-view without increasing the number of parameters or the amount of computation. And finally, we can have larger output feature map which is good for semantic segmentation.



(a) 3x3 Kernel

(a) 3x3 Dilated Kernel with  $r=2$

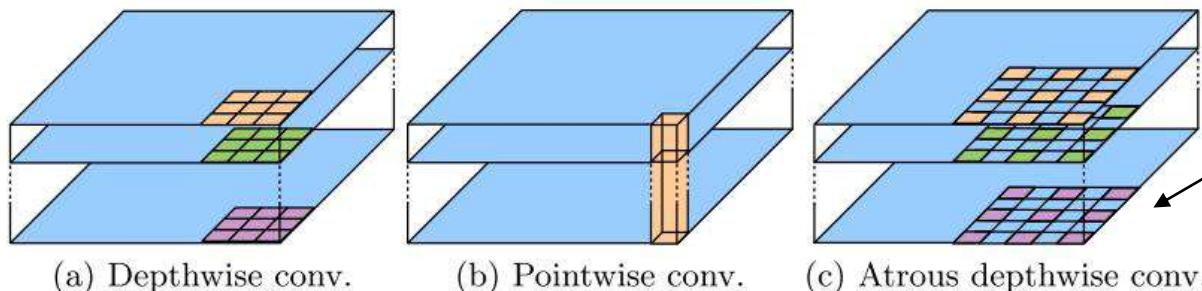
Standard vs Dilated Kernel. If  $r==1$  dilated kernel collapses into the standard, see the formula reported in the next page. A 3x3 kernel with a dilation rate of 2 has the same field of view as a 5x5 kernel

Padding is applied in atrous convolution

Output stride explains the ratio of the input image size to the output feature map size. It defines how much signal decimation the input vector suffers as it passes the network.



# DeepLabV3+



$3 \times 3$  Depthwise separable convolution decomposes a standard convolution into (a) a depthwise convolution (applying a single filter for each input channel) and (b) a pointwise convolution (combining the outputs from depthwise convolution across channels).

Atrous convolution with rate=2

Depthwise separable convolution is applied to reduce the computation time, this means that the convolution is applied separately to each channel, then these outputs are combined using a pointwise convolution.

$$y[i] = \sum_k x[i + r \cdot k] w[k]$$

<https://arxiv.org/abs/1802.02611v3>

where the atrous rate  $r$  determines the stride with which we sample the input signal. We refer interested readers to [for more details](#). Note that standard convolution is a special case in which rate  $r = 1$ . The filter's field-of-view is adaptively modified by changing the rate value.

**Depthwise separable convolution:** Depthwise separable convolution, factorizing a standard convolution into a *depthwise convolution* followed by a *pointwise convolution* (*i.e.*,  $1 \times 1$  convolution), drastically reduces computation complexity. Specifically, the depthwise convolution performs a spatial convolution independently for each input channel, while the pointwise convolution is employed to combine the output from the depthwise convolution.



# DeepLabV3+ - matlab

```
layerGraph = deeplabv3plusLayers([350 350],3,'resnet50');  
%it does not follow exactly the original DeepLabV3+ approach  
>> analyzeNetwork(layerGraph)
```

E.g. this implementation of DeepLab v3+ does not include an average pooling layer in the encoding step.

## *Region growing*

Region growing is a region-based image segmentation method, it involves the selection of initial seed points.

This approach to segmentation examines neighboring pixels of initial seed points and determines whether the pixel neighbors should be added to the region. The process is iterated on, in the same manner as general data clustering algorithms.

Current (April 2023) state of the art: <https://segment-anything.com/>



# Segmentation - loss

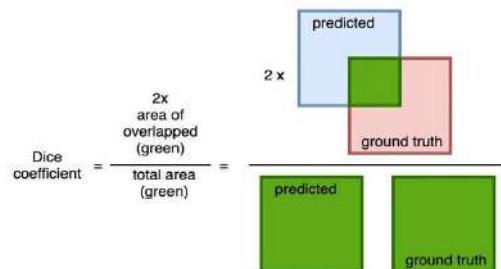
Dice Loss comes from Sørensen-Dice coefficient, a metric widely used to estimate the performance of semantic segmentation models. The Sereneness-Dice coefficient, also called dice coefficient, shows how similar two images are to each other. Its value is greater than 0 and less than 1; it can also be equal to 0, case in which no pixel is correctly classified by the model, or equal to 1, case in which all the pixels are correctly classified.

In order to apply dice loss to multiclass problems, the generalized dice loss was proposed. The formula of the generalized dice loss between the predictions  $Y$  and the training targets  $T$  is:

$$DL(Y, T) = 1 - \frac{2 * \sum_{k=1}^K w_k * \sum_{m=1}^M Y_{km} * T_{km}}{\sum_{k=1}^K w_k * \sum_{m=1}^M Y_{km} + T_{km}}$$

$$w_k = \frac{1}{(\sum_{m=1}^M T_{km})^2}$$

where  $K$  is the number of classes and  $M$  is the number of pixels. The weighting factor  $w_k$  is introduced to help the network focuses on a small region. Indeed, it is inversely proportional to the frequency of the labels of a given class  $k$ .



$$\text{Dice} = \frac{2 \times TP}{(TP + FP) + (TP + FN)}$$

standard 2-class dice index. TP stands for True Positive, FP stands for False Positive and FN stands for False Negatives.

for example, if there is no overlap between "predicted" and "ground truth" we have: TP=0 (no overlap between "predicted" and "ground truth"); FP='area of predicted'; FN='area of ground truth'

# Segmentation - loss

A recurring problem in several image segmentation problems is the predominance of one class over another. In the interest of dealing with this issue, Tversky Loss was introduced. Tversky Loss derives from Tversky Index, this can be seen as an extension of the dice similarity coefficient. Tversky Index has two weighting factors,  $\alpha$  and  $\beta$ , in order to manage trade-off between false positives and false negatives. A special case is when Tversky Index has  $\alpha = 0.5$  and  $\beta = 0.5$ . It degenerates into dice similarity coefficient. The Tversky Index is given by:

$$TI_c = \frac{TP}{TP + \alpha FN + \beta FP}$$

“positive” are the pixels  
that belong to the c-th class  
(if we are managing a multiclass dataset)

The formula of the Tversky Loss is:

$$TL = \sum_{c=1}^C 1 - TI_c$$

where  $C$  is the number of classes.

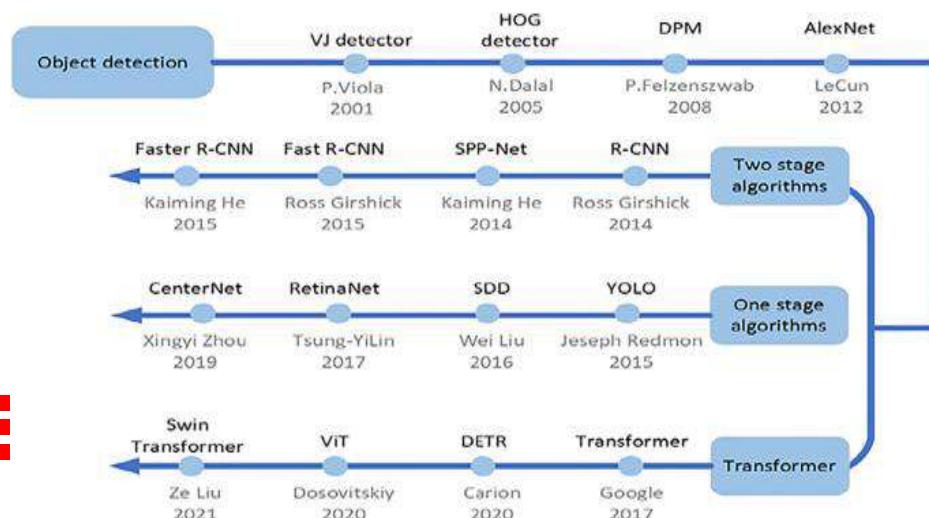
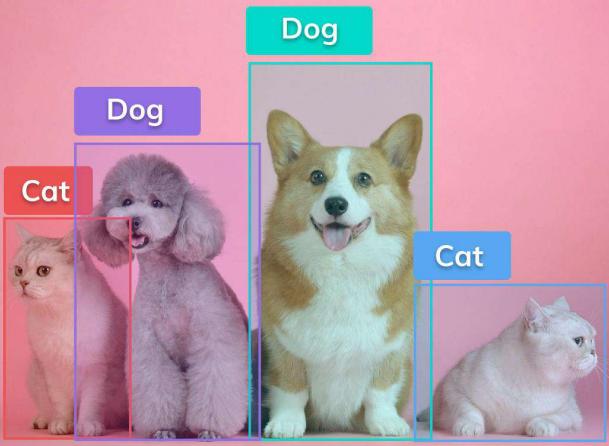
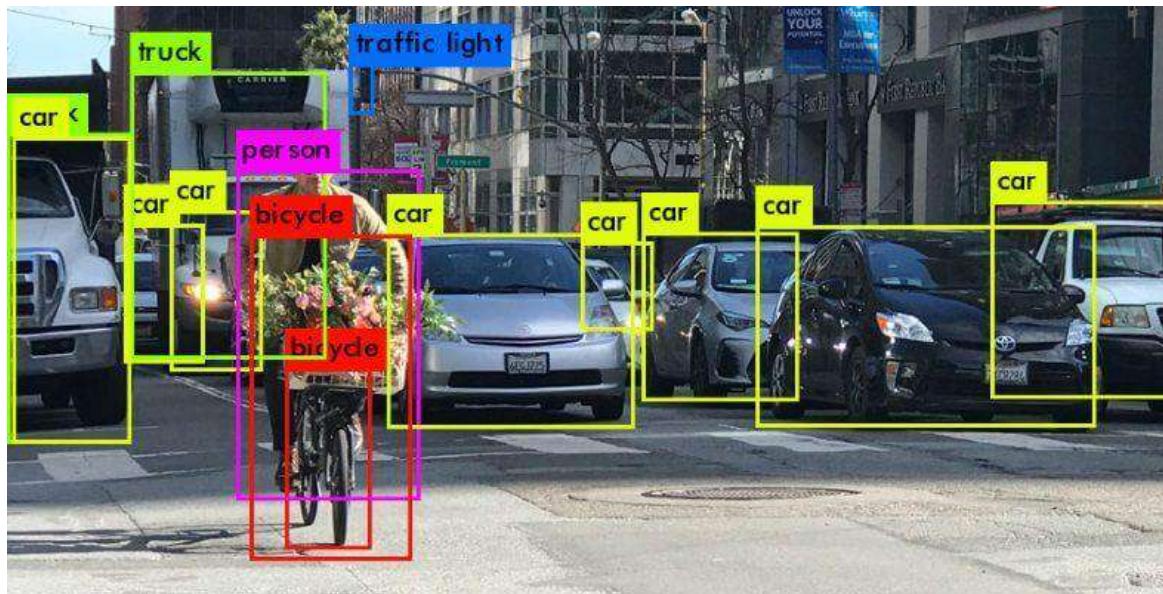


*Let's see an example:*  
*Matlab\_Examples\Segmentation*



# Image Detection

A survey: object detection methods from CNN to transformer (2022)  
<https://link.springer.com/article/10.1007/s11042-022-13801-3>

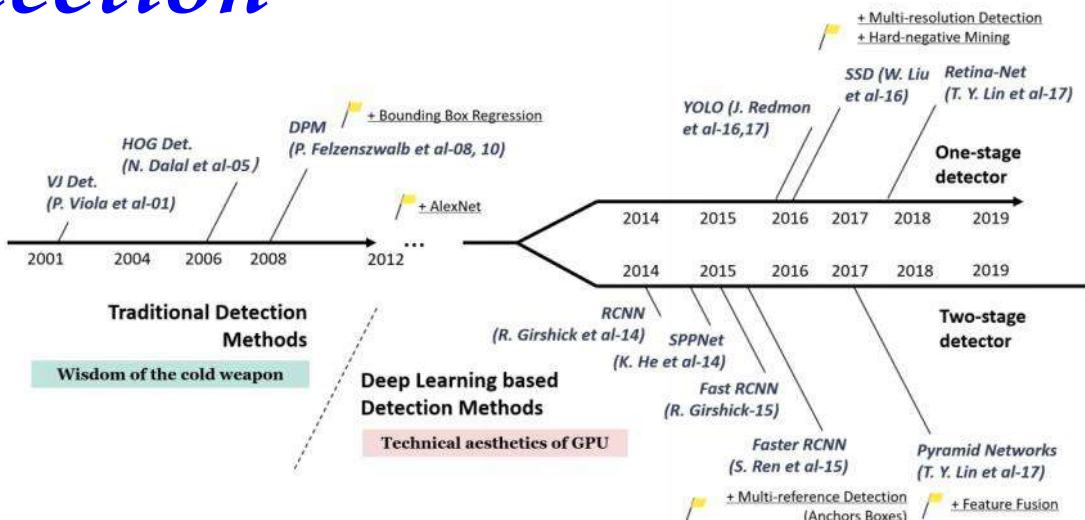


# Image detection

Object detection is an important problem which consists in identifying instances of objects within an image and in classifying them as belonging to a certain class (e.g. as humans, animals or cars), i.e.: the answer to the question "what objects are here?".

From an application point of view, it is possible to group object detection into two categories:

"general object detection" and "detection applications". For the first, the goal is to investigate methods for identifying different types of objects using a single framework, in order to simulate human vision and cognition. In the second case we refer to the recognition, under specific application scenarios, of objects of a certain class: this is the case of applications for pedestrian detection, face detection or for text detection.



The timeline of the evolution of object detection models in recent years.

Currently the models for object detection can be divided into two macro-categories two stage and one stage detectors. In the first case, we are talking about models that divide the task of identifying objects into several phases, following a "coarse to fine" policy.

In the second we talk about models whose process tries to complete the recognition in a single step with the use of a single network.



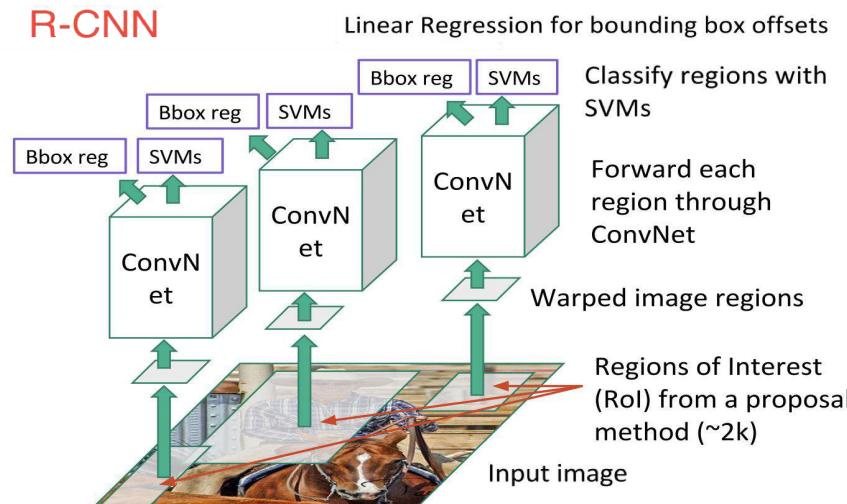
# Region based CNN (R-CNN)

The R-CNN first extracts many (e.g., 2000) *region proposals* from the input image (e.g., anchor boxes can also be considered as region proposals), labeling their classes and bounding boxes (e.g., offsets). Anchor boxes are a set of predefined bounding boxes of a certain height and width. These boxes are defined to capture the scale and aspect ratio of specific object classes you want to detect and are typically chosen based on object sizes in your training datasets. During detection, the predefined anchor boxes are tiled across the image.

Then a CNN is used to perform forward propagation on each region proposal to extract its features. Next, features of each region proposal are used for predicting the class and bounding box of this region proposal.

i.e. only inference step is applied to thousands of sub-images

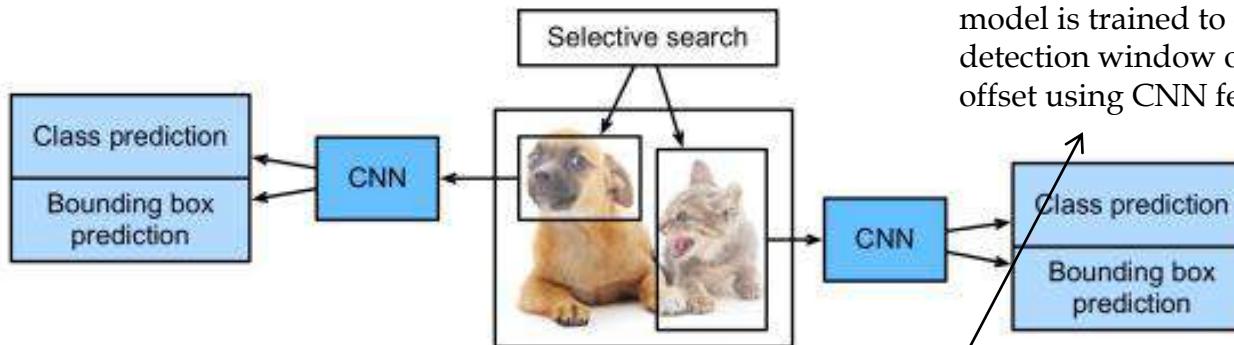
The main performance bottleneck of an R-CNN lies in the independent CNN forward propagation for each region proposal, without sharing computation. Since these regions usually have overlaps, independent feature extractions lead to much repeated computation. One of the major improvements of the *fast R-CNN* from the R-CNN is that the CNN forward propagation is only performed on the entire image



The R-CNN detector first generates region proposals using an algorithm such as Edge Boxes. The proposal regions are cropped out of the image and resized.



To reduce the localization errors, a regression model is trained to correct the predicted detection window on bounding box correction offset using CNN features.



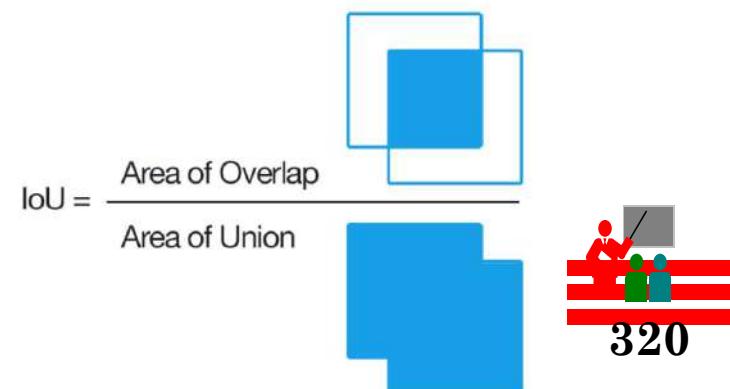
More concretely, the R-CNN consists of the following four steps:

1. Perform *selective search* to extract multiple high-quality region proposals on the input image. These proposed regions are usually selected at multiple scales with different shapes and sizes. Each region proposal will be labeled with a class and a ground-truth bounding box. <https://ivi.fnwi.uva.nl/isis/publications/2013/UijlingsIJCV2013/UijlingsIJCV2013.pdf>
2. Choose a pretrained CNN and truncate it before the output layer. Resize each region proposal to the input size required by the network, and output the extracted features for the region proposal through forward propagation. → pre-train a convolutional neural network on image classification tasks.
3. Take the extracted features and labeled class of each region proposal as an example. Train multiple support vector machines to classify objects, where each support vector machine individually determines whether the example contains a specific class.
4. Take the extracted features and labeled bounding box of each region proposal as an example. Train a linear regression model to predict the ground-truth bounding box.

Although the R-CNN model uses pretrained CNNs to effectively extract image features, it is slow. Imagine that we select thousands of region proposals from a single input image: this requires thousands of CNN forward propagations to perform object detection. This massive computing load makes it infeasible to widely use R-CNNs in real-world applications.

# R-CNN

- To train the CNN for feature extraction, an architecture is initialized with the pre-trained weights from imangenet data. The output layer having 1000 classes (of the ImageNet dataset) is chopped off. So when a region proposal image is passed to the network we get a 4096-dimensional feature vector (an internal layer of the CNN is used to extract the features). In this way, each region proposal is represented by a 4096-dimensional feature vector.
- The next step is to fine-tune the weights of the network with the region proposal images. To measure the performance of a classification model, we generally use metrics like accuracy, recall, precision etc. But how to measure the performance of object detection? In object detection we have to evaluate two things:
  - How well the bounding box can locate the object in the image. In other words, how close the predicted bounding box is to the ground truth.
  - Whether the bounding box is classifying the enclosed object correctly
- The Intersection over Union (IoU) score measures how close the predicted box is to the ground truth. It is the ratio of the overlap between the predicted box and the ground truth and the total area enclosed by both the boxes (union between predicted and ground truth).
- To fine-tune the model the output layer having 1000 classes (if we are using a CNN pretrained using the ImageNet) is replaced with N+1 classes(softmax layers) and the rest of the model is kept unchanged. N is the number of the distinct classes the objects be classified and plus 1 for the background class.
- Next, data is required for fine-tuning. The region proposals whose IoU > 50% are considered a positive class for that object and rest are considered as background. These images (region proposals) are warped (resized) to the size compatible with CNN. Using these images the weights of the network are fine-tuned.

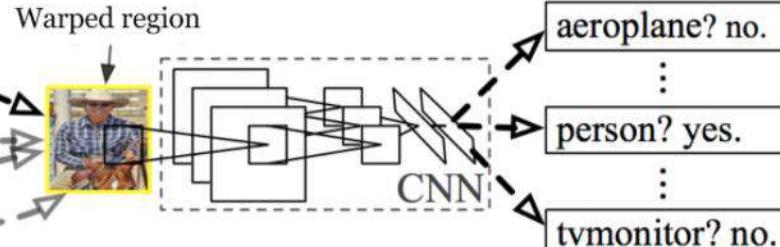
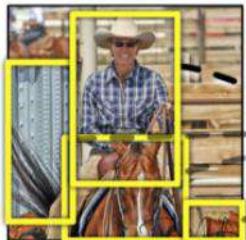


# R-CNN

<https://sahiltinky94.medium.com/object-detection-r-cnn-aa2b180bfb49>



1. Input images

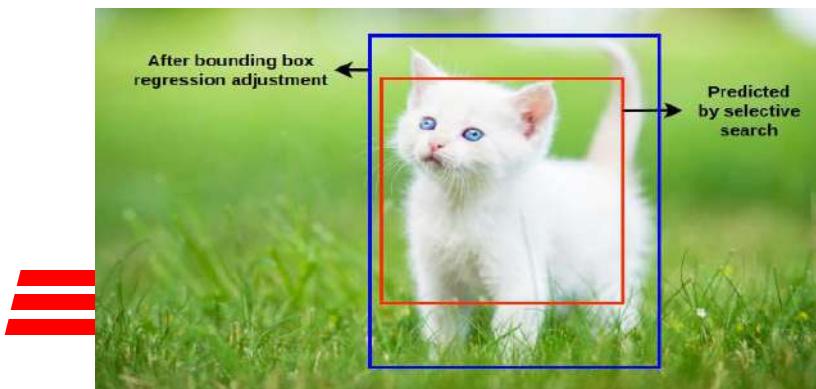


4. Classify regions

Scheme of RCNN



Region selection:  
first step the image  
is segmented, then  
a set of different  
subwindows is  
extracted



Bounding box regression:  
Given a subwindow we optimize  
the detection, applying spatial  
transformation which parameters are  
obtained by a regression approach  
trained given the predicted bounding box  
coordinate and its corresponding ground truth  
box coordinates

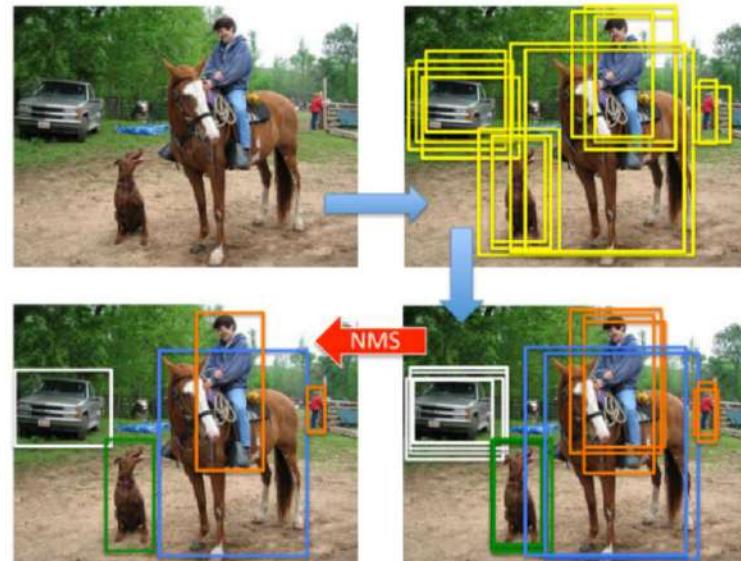
# Simple “trick”: Non-maximum suppression

Out of those many bounding boxes, a lot of them would be redundant and overlapping bounding boxes which need to be removed. To accomplish that non-maximum suppression algorithm is used. Non-maximum suppression is a greedy algorithm. It works on one class at a time. For a particular class, it picks the box with the maximum score obtained using SVM. Then it calculates IoU score with all other bounding boxes belonging to that class. The boxes having IoU score greater than 70% are removed. In other words, the bounding boxes which have very high overlap are removed. Then the next highest score box is chosen and so on until all the overlapping bounding boxes are removed for that class. This is done for all classes to obtain the result as shown above.

NMS is commonly used in RCNN and other detection models.

Once the labelled bounding boxes are obtained the next task is to fine-tune the location of the boxes using regression.

Non-maximum suppression algorithm:



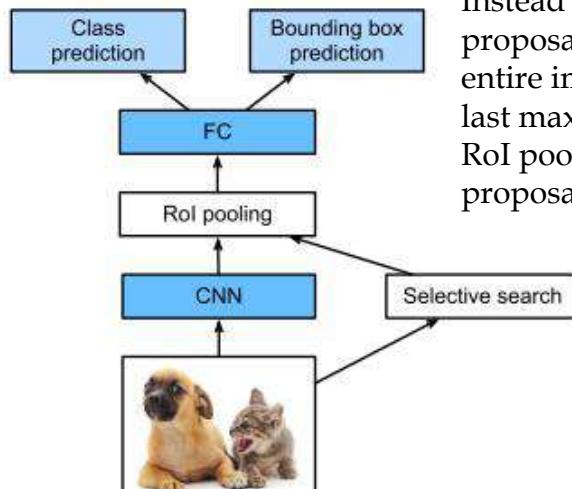
Input: A list of Proposal boxes B, corresponding confidence scores S and overlap threshold N.

Output: A list of filtered proposals D.

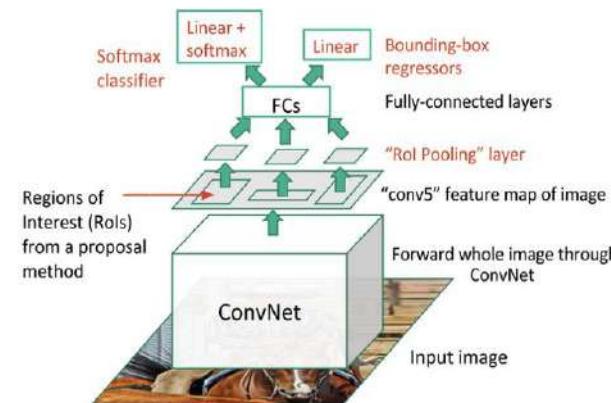
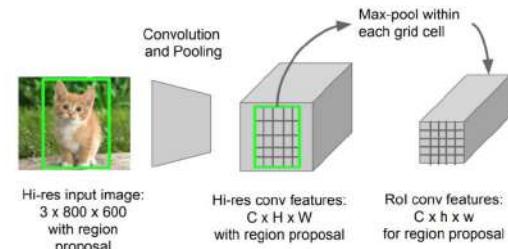
Algorithm:

- Select the proposal with highest confidence score, remove it from B and add it to the final proposal list D. (Initially D is empty).
- Now compare this proposal with all the proposals – calculate the IOU (Intersection over Union) of this proposal with every other proposal. If the IOU is greater than the threshold N, remove that proposal from B.
- Again take the proposal with the highest confidence from the remaining proposals in B and remove it from B and add it to D.
- Once again calculate the IOU of this proposal with all the proposals in B and eliminate the boxes which have high IOU than threshold.
- This process is repeated until there are no more proposals left in B.

# fast R-CNN



Instead of extracting CNN feature vectors independently for each region proposal, this model aggregates them into one CNN forward pass over the entire image and the region proposals share this feature matrix. Replace the last max pooling layer of the pre-trained CNN with a ROI pooling layer. The ROI pooling layer outputs fixed-length feature vectors for each region proposals. The ROI pooling layer is applied on the output of a chosen internal layer of the CNN.

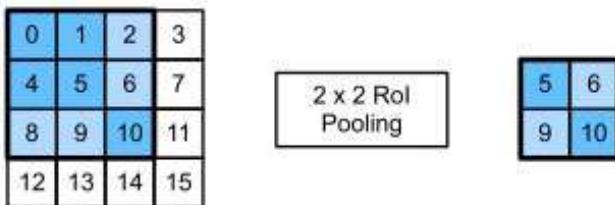


- Compared with the R-CNN, in the fast R-CNN the input of the CNN for feature extraction is the entire image, rather than individual region proposals. Moreover, this CNN is trainable. Given an input image, let the shape of the CNN output be  $1 \times c \times h_1 \times w_1$ .
- Suppose that selective search generates  $n$  region proposals. These region proposals (of different shapes) mark regions of interest (of different shapes) on the CNN output. Then these regions of interest further extract features of the same shape (say height  $h_2$  and width  $w_2$  are specified) in order to be easily concatenated. To achieve this, the fast R-CNN introduces the *region of interest (ROI) pooling* layer: the CNN output and region proposals are input into this layer, outputting concatenated features of shape  $n \times c \times h_2 \times w_2$  that are further extracted for all the region proposals.
- Using a fully-connected layer, transform the concatenated features into an output of shape  $n \times d$ , where  $d$  depends on the model design.
- Predict the class and bounding box for each of the  $n$  region proposals. More concretely, in class and bounding box prediction, transform the fully-connected layer output into an output of shape  $n \times q$  ( $q$  is the number of classes) and an output of shape  $n \times 4$ , respectively. The class prediction uses softmax regression.



# Fast R-CNN: RoI Pooling layer

the upper-left  $3 \times 3$  region of interest is selected on a  $4 \times 4$  input. For this region of interest, we use a  $2 \times 2$  region of interest pooling layer to obtain a  $2 \times 2$  output. Note that each of the four divided subwindows contains elements 0, 1, 4, and 5 (5 is the maximum); 2 and 6 (6 is the maximum); 8 and 9 (9 is the maximum); and 10.



A  $2 \times 2$  region of interest pooling layer.

The region of interest pooling layer proposed in the fast R-CNN is different from the pooling layer

In the pooling layer, we indirectly control the output shape by specifying sizes of the pooling window, padding, and stride. In contrast, we can directly specify the output shape in the region of interest pooling layer.

For example, let us specify the output height and width for each region as  $h_2$  and  $w_2$ , respectively. For any region of interest window of shape  $h \times w$ , this window is divided into a  $h_2 \times w_2$  grid of subwindows, where the shape of each subwindow is approximately  $(h/h_2) \times (w/w_2)$ . In practice, the height and width of any subwindow shall be rounded up, and the largest element shall be used as the output of the subwindow. Therefore, the region of interest pooling layer can extract features of the same shape even when regions of interest have different shapes.

region proposal								
0.88	0.44	0.14	0.16	0.37	0.77	0.96	0.27	
0.19	0.45	0.57	0.16	0.63	0.29	0.71	0.70	
0.66	0.26	0.82	0.64	0.54	0.73	0.59	0.26	
0.85	0.34	0.76	0.84	0.29	0.75	0.62	0.25	
0.32	0.74	0.21	0.39	0.34	0.03	0.33	0.48	
0.20	0.14	0.16	0.13	0.73	0.65	0.96	0.32	
0.19	0.69	0.09	0.86	0.88	0.07	0.01	0.48	
0.83	0.24	0.97	0.04	0.24	0.35	0.50	0.91	

By dividing it into  $(2 \times 2)$  sections (because the output size is  $2 \times 2$ ) we get:

pooling sections								
0.88	0.44	0.14	0.16	0.37	0.77	0.96	0.27	
0.19	0.45	0.57	0.16	0.63	0.29	0.71	0.70	
0.66	0.26	0.82	0.64	0.54	0.73	0.59	0.26	
0.85	0.34	0.76	0.84	0.29	0.75	0.62	0.25	
0.32	0.74	0.21	0.39	0.34	0.03	0.33	0.48	
0.20	0.14	0.16	0.13	0.73	0.65	0.96	0.32	
0.19	0.69	0.09	0.86	0.88	0.07	0.01	0.48	
0.83	0.24	0.97	0.04	0.24	0.35	0.50	0.91	

Notice that the size of the region of interest doesn't have to be perfectly divisible by the number of pooling sections (in this case our RoI is  $7 \times 5$  and we have  $2 \times 2$  pooling sections).

The max values in each of the sections are:

0.85	0.84
0.97	0.96

# Fast R-CNN, CNN training / Beyond Fast RCNN

1. First, pre-train a convolutional neural network on image classification tasks.
2. Propose regions by selective search (~2k candidates per image).
3. Alter the pre-trained CNN:
  - o Replace the last max pooling layer of the pre-trained CNN with a RoI pooling layer. The RoI pooling layer outputs fixed-length feature vectors of region proposals. Sharing the CNN computation makes a lot of sense, as many region proposals of the same images are highly overlapped.
  - o Replace the last fully connected layer and the last softmax layer ( $K$  classes) with a fully connected layer and softmax over  $K + 1$  classes.
4. Finally the model branches into two output layers:
  - o A softmax estimator of  $K + 1$  classes (same as in R-CNN,  $+1$  is the "background" class), outputting a discrete probability distribution per RoI.
  - o A bounding-box regression model which predicts offsets relative to the original RoI for each of  $K$  classes.

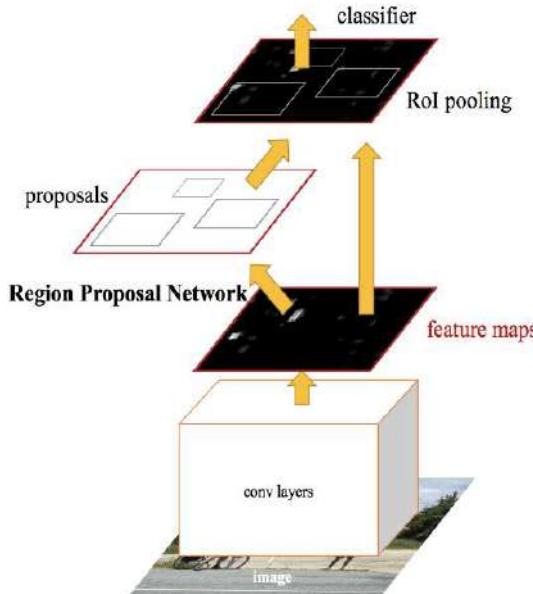
**Loss Function** i.e. this loss is used to train the CNN detailed above

The model is optimized for a loss combining two tasks (classification + localization):

## BEYOND FAST R-CNN

There also new methods related to the R-CNN family,  
e.g. Faster R-CNN.

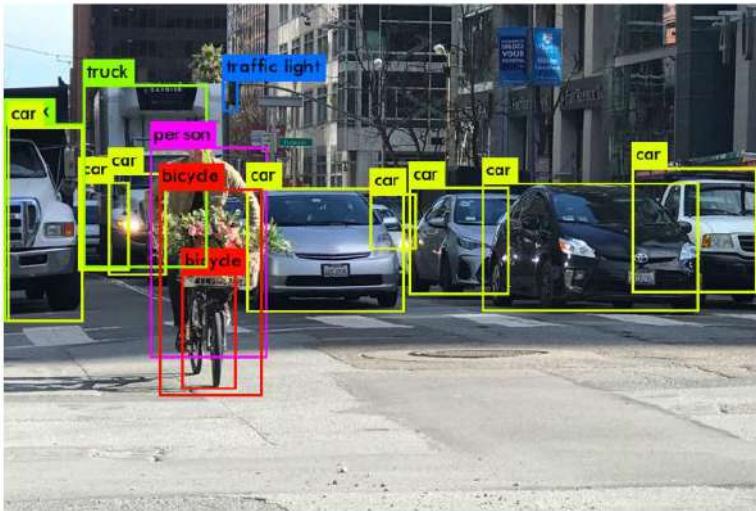
Faster R-CNN fixes the problem of selective search by replacing it with Region Proposal Network (RPN). We first extract feature maps from the input image using ConvNet and then pass those maps through a RPN which returns object proposals. Finally, these maps are classified and the bounding boxes are predicted.



To be more accurate in object detection, the fast R-CNN model usually has to generate a lot of region proposals in selective search. To reduce region proposals without loss of accuracy, the *faster R-CNN* proposes to replace selective search with a *region proposal network* <https://arxiv.org/abs/1506.01497>



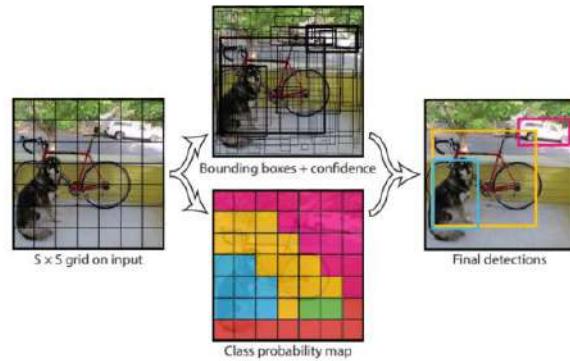
# You only look once YOLO



Compared to pre-existing object detection models, YOLO is significantly faster.

This is mainly possible due to the fact that YOLO does not divide the recognition into several phases, but predicts bounding boxes, probabilities and classes of the objects present in the input image in a single phase.

Compared to other object detection systems, YOLO makes more localization errors, but at the same time it is less likely to recognize false-positives in the background of the image, as well as being considerably faster.



Phases of YOLO execution, the model divides the image into  $S \times S$  cells, for each cell it predicts  $B$  bounding box, each with a confidence score and  $C$  classes with relative probabilities. In Non Maximal Suppression, YOLO suppresses all bounding boxes that have lower probability scores.

YOLO is the first "one-stage" object detection system. J. Redmon et al. present an innovative solution to the problem of object detection, unifying the different components of the Object Detection systems in a single network. This forces the network to reason over the entire image at the same time, rather than specific regions, providing a greater understanding of the environment in which the different classes of objects are located. Furthermore, by doing so, it creates an implicit link between effectively related classes of objects.



# ||||| YOLO - Bounding box

- To understand the YOLO algorithm, it is necessary to establish what is actually being predicted. Ultimately, we aim to predict a class of an object and the bounding box specifying object location. Each bounding box can be described using four descriptors:

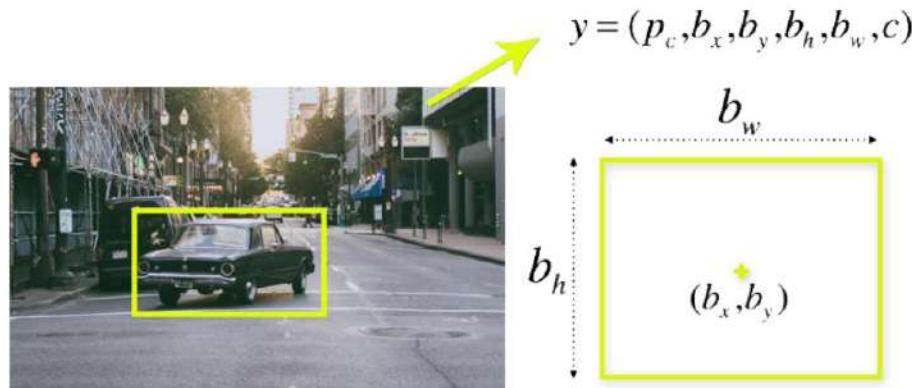
center of a bounding box ( $b_x, b_y$ )

width ( $b_w$ )

height ( $b_h$ )

value  $c$  is corresponding to a class of an object (e.g., car, traffic lights, etc.)

In addition, we have to predict the  $p_c$  value, which is the probability that there is an object in the bounding box.



As we mentioned above, when working with the YOLO algorithm we are not searching for interesting regions in our image that could potentially contain an object. The final level of the network predicts both class probabilities and bounding box coordinates for recognized objects

Instead, we are splitting our image into cells, now typically using a  $19 \times 19$  grid (original YOLO  $7 \times 7$ ). Each cell is responsible for predicting 5 bounding boxes (in case there is more than one object in this cell). Therefore, we arrive at a large number of 1805 bounding boxes for one image.



# How to train YOLO

Fixed S and B, the model divides the image into a grid of  $S \times S$  cells. The grid cell containing the center of an object is defined as the cell responsible for it (see figure below). Each cell of the grid predicts the B bounding boxes and assigns each of them a score called "Confidence Score". This score indicates how likely it is that the bounding box contains an object ( $\text{Pr}(\text{Object})$ ) and also how accurate the size and location of the bounding box with respect to the object (IOU) is believed to be.

Each bounding box (of each cell) predicted by the model with five values:

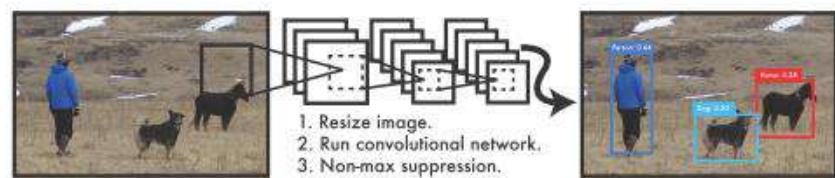
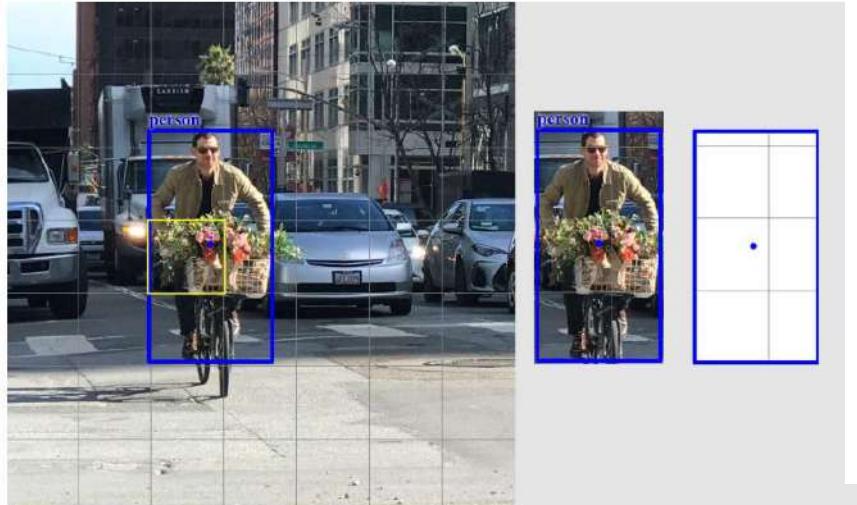
x, y, w, h and the confidence score " $\text{Pr}(\text{Object}) * \text{IOU}$ " where (x, y) are the coordinates of the center of the bounding box with respect to the edges of the grid cell, while w and h represent respectively the width and height of the bounding box, however, relatively to the whole image. Each cell of the grid, regardless of the number B of bounding boxes generated, predicts C classes with their conditional probability ( $\text{Pr}(\text{Class}_i | \text{Object})$ ).

At this point the model, for each bounding box, multiplies the conditional probabilities of the classes (of the cell) with the confidence score of the bounding box, thus obtaining a specific confidence score of each class for each bounding box:

$$\text{Pr}(\text{Class}_i | \text{Object}) * \text{Pr}(\text{Object}) * \text{IOU}_{\text{pred}}^{\text{truth}} = \text{Pr}(\text{Class}_i) * \text{IOU}_{\text{pred}}$$

The value thus obtained encodes both the probabilities that an object of a given class appears in the bounding box under examination, and the precision with which the predicted bounding box delimits the spatial boundaries of the object.

We will see on pages 331-334 how the loss is implemented starting from the idea of the above formula.



**The YOLO Detection System.**

(1) resizes the input image to  $448 \times 448$ , (2) runs a single convolutional network on the image, and (3) thresholds the resulting detections by the model's confidence.

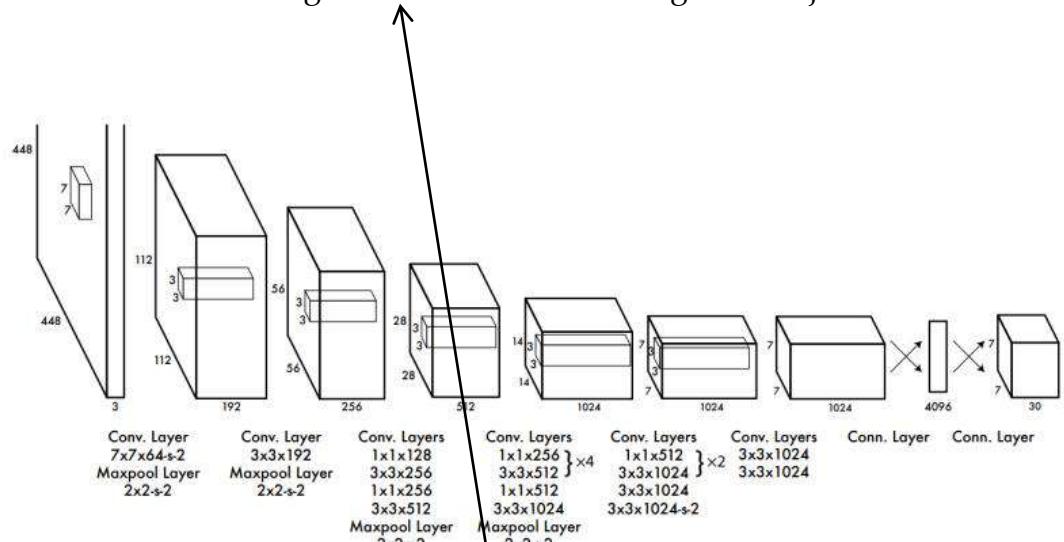
# YOLO - architecture

Name	Filters	Output Dimension
conv 1	$7 \times 7 \times 64$ , stride=2	$224 \times 224 \times 64$
Conv 1	$7 \times 7 \times 64$ , stride=2	$224 \times 224 \times 64$
Max Pool 1	$2 \times 2$ , stride=2	$112 \times 112 \times 64$
Conv 2	$3 \times 3 \times 192$	$112 \times 112 \times 192$
Max Pool 2	$2 \times 2$ , stride=2	$56 \times 56 \times 192$
Conv 3	$1 \times 1 \times 128$	$56 \times 56 \times 128$
Conv 4	$3 \times 3 \times 256$	$56 \times 56 \times 256$
Conv 5	$1 \times 1 \times 256$	$56 \times 56 \times 256$
Conv 6	$1 \times 1 \times 512$	$56 \times 56 \times 512$
Max Pool 3	$2 \times 2$ , stride=2	$28 \times 28 \times 512$
Conv 7	$1 \times 1 \times 256$	$28 \times 28 \times 256$
Conv 8	$3 \times 3 \times 512$	$28 \times 28 \times 512$
Conv 9	$1 \times 1 \times 256$	$28 \times 28 \times 256$
Conv 10	$3 \times 3 \times 512$	$28 \times 28 \times 512$
Conv 11	$1 \times 1 \times 256$	$28 \times 28 \times 256$
Conv 12	$3 \times 3 \times 512$	$28 \times 28 \times 512$
Conv 13	$1 \times 1 \times 256$	$28 \times 28 \times 256$
Conv 14	$3 \times 3 \times 512$	$28 \times 28 \times 512$
Conv 15	$1 \times 1 \times 512$	$28 \times 28 \times 512$
Conv 16	$3 \times 3 \times 1024$	$28 \times 28 \times 1024$
Max Pool 4	$2 \times 2$ , stride=2	$14 \times 14 \times 1024$
Conv 17	$1 \times 1 \times 512$	$14 \times 14 \times 512$
Conv 18	$3 \times 3 \times 1024$	$14 \times 14 \times 1024$
Conv 19	$1 \times 1 \times 512$	$14 \times 14 \times 512$
Conv 20	$3 \times 3 \times 1024$	$14 \times 14 \times 1024$
Conv 21	$3 \times 3 \times 1024$	$14 \times 14 \times 1024$
Conv 22	$3 \times 3 \times 1024$ , stride=2	$7 \times 7 \times 1024$
Conv 23	$3 \times 3 \times 1024$	$7 \times 7 \times 1024$
Conv 24	$3 \times 3 \times 1024$	$7 \times 7 \times 1024$
FC 1	-	4096
FC 2	-	$7 \times 7 \times 30$ (1470)

The architecture of YOLO is very much inspired by that of GoogLeNet. YOLO has 24 convolutional levels, followed by 2 entirely interconnected levels. YOLO uses a linear activation function for last level (i.e.  $f(x)=x$ , it's not possible to use backpropagation as the derivative of the function is a constant), while all other levels use the following activation function:

$$\phi(x) = \begin{cases} x & \text{if } x>0 \\ 0.1x & \text{otherwise} \end{cases}$$

“ $\times 5$ ” since each bounding box has 5 parameters (see previous page). The final level of the network predicts both class probabilities and bounding box coordinates for recognized objects



Representation of the architecture of YOLO. YOLO has 24 convolutional levels followed by 2 fully connected levels. Note the alternation of convolutional levels of dimension  $1 \times 1$ , used to reduce the space of the activations obtained from the previous levels.

YOLO has a  $7 \times 7 \times 30$  size output layer. 7 since in original YOLO the authors fixed  $S=7$ . As detailed in previous pages: “It divides the image into an  $S \times S$  grid and for each grid cell predicts  $B$  bounding boxes, confidence for those boxes, and  $C$  class probabilities. These predictions are encoded as an  $S \times S \times (B \times 5 + C)$  tensor.” In original YOLO the training dataset was the PASCAL VOC, it has 20 labelled classes so  $C = 20$ . Each cell is responsible for predicting 2 bounding boxes, so  $B=2$  in this example.

# YOLO - training

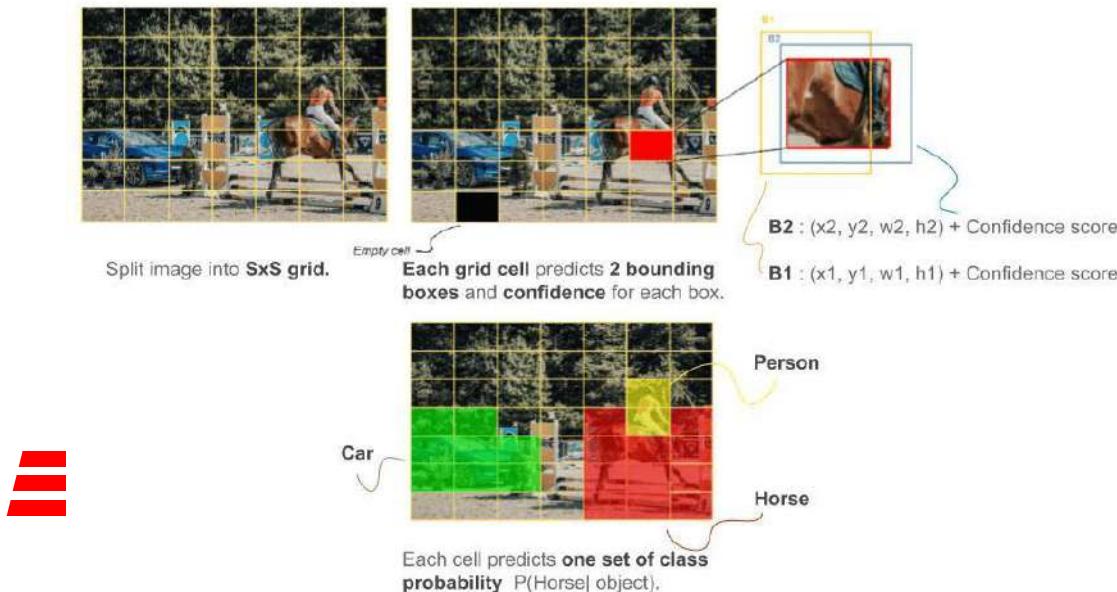
The training of YOLO, as conceived by the authors, is divided into two phases:  
a first pre-training phase, during which only the deepest levels are trained, and a second training phase that involves the entire network.

## Pre-Training

The authors of YOLO begin the training by training only the first 20 convolutional layers to which they add, an average-pooling level and one fully connected. For this purpose they use the ImageNet competition dataset (it is a classification problem) with input images of size 224x224 pixels.

## Training

At this point they have eliminated the last two levels of the network, added specifically for the first part of the training, to add another 4 convolutional levels, finally followed by 2 fully connected levels. I.e. the network detailed on the previous page.



Since object detection models require precise information to improve accuracy, the model is trained with an high resolution images (i.e. 448x448 pixels).

The second training phase, as conceived by the authors, foresees 135 epochs that use both the training and the validation set of the PASCAL VOC 2007 and PASCAL VOC 2012 competitions (object detection problems).

To reduce overfitting, the model has a dropout layer with a 50% reduction rate placed after the first fully connected layer to prevent co-adaptation between the fully connected layers. Again in order to reduce overfitting, data augmentation techniques have also been introduced: before using the images taken as input, the model processes them by adding translations and changing the aspect ratio of the image up to 20% compared to the original characteristics. In addition, the images are modified by changing exposure and saturation by a factor between 0 and 1.5.

# YOLO

The final level of the network predicts both class probabilities and bounding box coordinates for recognized objects. The model normalizes the width and height of the bounding boxes with respect to the size of the input image so that their values fall in the real range between zero and one.

The center of the bounding boxes is in turn normalized and expressed as a function of the cell (of the grid) to which it belongs so that it too belongs to the interval between zero and one.

Regarding the learning rate, the authors noted that the model often diverges towards unstable gradients when trained from the beginning with high levels of this parameter.

For this reason the learning rate starts from a low value,  $10^{-3}$ , then following this scheme (for the second training phase-> 135 epochs):

- the rate slowly increases in the first epochs (how many epochs depends on the implementation, it is not specified in the original paper) until it reaches the value of  $10^{-2}$ .
- remains constant until epoch 75;
- for the following 30 epochs a rate of  $10^{-3}$  is applied;
- for the last 30 epochs a learning rate of  $10^{-4}$  is used.

The YOLO loss function consists of:

- classification loss;
- localization loss (spatial error between the predicted and the real bounding box);
- confidence loss (general error of the bounding box, i.e. calculated confidence score error).



# *YOLO - loss function*

Classification loss: Calculate the error in the classification of a specific object; that is, it indicates whether or not the recognized class corresponds to the object identified in the image. Given a cell in the grid, if it is considered responsible for a certain object, then it is also responsible for its classification. The error in this operation is given by the quadratic function which calculates the conditional probabilities of the recognized class with respect to each class:

$$\sum_{i=0}^{S^2} \mathbb{1}_i^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2$$

[https://en.wikipedia.org/wiki/Indicator\\_function](https://en.wikipedia.org/wiki/Indicator_function)

$1^{obj}_i$  it is 1 if an object appears in the i-th cell, OR 0 otherwise:

$\hat{p}_i(c)$  denotes the conditional probability of the c-th class to appear in the i-th cell.

## Localization Loss

It measures the error in the prediction of the bounding boxes, i.e. the height, width and coordinates of the center. YOLO calculates this error only for the bounding boxes responsible for recognizing an object.

The error in the localization calculated as follows:

$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbf{1}_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] + \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbf{1}_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2]$$

Difference between «True» value  
and the value predicted by the network

$\mathbb{1}_{ij}^{obj}$  it is 1 if the j-th bounding box of the i-th cell is responsible for recognizing an object (i.e. an object is located in that bounding box), 0 otherwise.  $\lambda_{coord}=0.5$

We note that the error with respect to the size of the bounding box is calculated on the square root of the values. This is in order not to give the same weight to errors, regardless of the size of the bounding box: for example, a 2-pixel error could be negligible if found on a 500-pixel bounding box, but important on a 30-pixel bounding box.

The square root downscale high values while less affecting low values of width and height.



# Yolo - Loss function

## Confidence Loss

It is used to quantify the objectivity of the prediction made on a single bounding box. If, given a bounding box, the model recognizes an object in it, the confidence loss is calculated as follows:

$$\sum_{i=0}^{S^2} \sum_{j=0}^B \mathbf{1}_{ij}^{obj} (C_{ij} - \hat{C}_{ij})^2$$

$\mathbf{1}_{ij}^{obj}$  it is 1 if the j-th bounding box of the i-th cell is responsible for recognizing an object, or 0 otherwise;

$\hat{C}_{ij}$  confidence score of the j-th bounding box of the i-th cell, i.e. it is one of “parameter” of the bounding box, it is an output of the network. We have defined the confidence score on page 328.

Most bounding boxes contain no objects. This causes an imbalance of the model, which in practice trains more frequently to recognize backgrounds rather than objects. To remedy this, the confidence loss for bounding boxes that do not contain objects is changed by a  $\lambda$  factor (default value: 0.5). We therefore get:

$$\lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbf{1}_{ij}^{noobj} (C_{ij} - \hat{C}_{ij})^2$$

$\mathbf{1}_{ij}^{noobj}$  is the complement of  $\mathbf{1}_{ij}^{obj}$ ;

$\hat{C}_{ij}$  confidence score of the j-th bounding box of the i-th cell



# YOLO - loss function

$1$  if object appears in cell  $i$  and  $j$ -th box detects it,  $0$  otherwise

$$\sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{obj} (C_i - \hat{C}_i)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{noobj} (C_i - \hat{C}_i)^2$$

Ground truth confidence score  
 Set to 0.5 to decrease the loss for empty boxes  
 Predicted confidence score  
 For each grid cell      For each box  
 1 if there is no object in the  $i$ -th cell, 0 otherwise

Upweights the loss for boxes with objects

Down-weights the loss for empty boxes

$$\begin{aligned}
 & \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] \\
 & + \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2]
 \end{aligned}$$

Consequently, the loss function of YOLO is expressed as follows:

$$\begin{aligned}
 & + \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{obj} (C_{ij} - \hat{C}_{ij})^2 \\
 & + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{noobj} (C_{ij} - \hat{C}_{ij})^2 \\
 & + \sum_{i=0}^{S^2} \mathbb{1}_i^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2
 \end{aligned}$$

# YoloV2

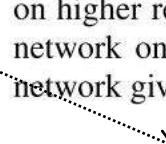
Yolo v2 strongly outperforms Yolo, it is based on Yolo adding some «tricks», here we report the most importat differences between Yolo and Yolo v2.

**Batch Normalization.** Batch normalization leads to significant improvements in convergence while eliminating the need for other forms of regularization. By adding batch normalization on all of the convolutional layers in YOLO we get more than 2% improvement in mAP. Batch normalization also helps regularize the model. With batch normalization we can remove dropout from the model without overfitting.

**High Resolution Classifier.** All state-of-the-art detection methods use classifier pre-trained on ImageNet. Starting with AlexNet most classifiers operate on input images smaller than  $256 \times 256$ . The original YOLO trains the classifier network at  $224 \times 224$  and increases the resolution to 448 for detection. This means the network has to

simultaneously switch to learning object detection and adjust to the new input resolution.

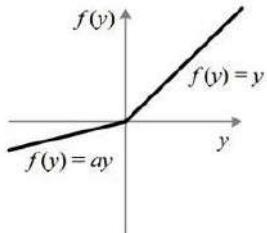
For YOLOv2 we first fine tune the classification network at the full  $448 \times 448$  resolution for 10 epochs on ImageNet. This gives the network time to adjust its filters to work better on higher resolution input. We then fine tune the resulting network on detection. This high resolution classification network gives us an increase of almost 4% mAP.

 see the ResNet section



# YOLOv2

A new net is used, it has been named DarkNet-19: 19 convolutional layers and 5 max pooling layer. After each convolutional layer there a Batch Normalization and a Leaky ReLu activation function ('a' is an hyperparameter usually 'a'=0.01).



For detection purposes, the last convolution layer of this architecture is removed and instead the authors added three  $3 \times 3$  convolution layers every 1024 filters followed by  $1 \times 1$  convolution with the number of outputs needed for detection, in a similar way of Yolo.

Type	Filters	Size/Stride	Output
Convolutional	32	$3 \times 3$	$224 \times 224$
Maxpool		$2 \times 2/2$	$112 \times 112$
Convolutional	64	$3 \times 3$	$112 \times 112$
Maxpool		$2 \times 2/2$	$56 \times 56$
Convolutional	128	$3 \times 3$	$56 \times 56$
Convolutional	64	$1 \times 1$	$56 \times 56$
Convolutional	128	$3 \times 3$	$56 \times 56$
Maxpool		$2 \times 2/2$	$28 \times 28$
Convolutional	256	$3 \times 3$	$28 \times 28$
Convolutional	128	$1 \times 1$	$28 \times 28$
Convolutional	256	$3 \times 3$	$28 \times 28$
Maxpool		$2 \times 2/2$	$14 \times 14$
Convolutional	512	$3 \times 3$	$14 \times 14$
Convolutional	256	$1 \times 1$	$14 \times 14$
Convolutional	512	$3 \times 3$	$14 \times 14$
Convolutional	256	$1 \times 1$	$14 \times 14$
Convolutional	512	$3 \times 3$	$14 \times 14$
Maxpool		$2 \times 2/2$	$7 \times 7$
Convolutional	1024	$3 \times 3$	$7 \times 7$
Convolutional	512	$1 \times 1$	$7 \times 7$
Convolutional	1024	$3 \times 3$	$7 \times 7$
Convolutional	512	$1 \times 1$	$7 \times 7$
Convolutional	1024	$3 \times 3$	$7 \times 7$
Convolutional	1000	$1 \times 1$	$7 \times 7$
Avgpool		Global	1000
Softmax			

Darknet-19.



# Yolo v3

A further performance improvement is obtained with Yolo v3, it improves Yolo v2 using some heuristics; anyway the most important difference is the new net: Darknet-53, based on 53 convolutional layers.

Moreover, YOLOv3 uses multi-label classification. For example, the output labels may be “pedestrian” and “child” which are not non-exclusive.

Type	Filters	Size	Output
1x	Convolutional	32	$3 \times 3$
	Convolutional	64	$3 \times 3 / 2$
	Convolutional	32	$1 \times 1$
	Convolutional	64	$3 \times 3$
2x	Residual		$128 \times 128$
	Convolutional	128	$3 \times 3 / 2$
	Convolutional	64	$1 \times 1$
	Convolutional	128	$3 \times 3$
8x	Residual		$64 \times 64$
	Convolutional	256	$3 \times 3 / 2$
	Convolutional	128	$1 \times 1$
	Convolutional	256	$3 \times 3$
8x	Residual		$32 \times 32$
	Convolutional	512	$3 \times 3 / 2$
	Convolutional	256	$1 \times 1$
	Convolutional	512	$3 \times 3$
4x	Residual		$16 \times 16$
	Convolutional	1024	$3 \times 3 / 2$
	Convolutional	512	$1 \times 1$
	Convolutional	1024	$3 \times 3$
Avgpool Connected Softmax	Residual		$8 \times 8$
	Avgpool		Global
	Connected		1000
	Softmax		

Darknet-53.

There are some new Yolo versions, but they are not developed by the original authors of Yolo v1/2/3, for a short review of others Yolo based approaches you could read <https://appsilon.com/object-detection-yolo-algorithm/> <https://learnopencv.com/category/yolo/>

The last (summer 2022) developed Yolo approach is Yolo v7 [https://github.com/jinfagang/yolov7\\_d2](https://github.com/jinfagang/yolov7_d2)



YOLOv3 replaces the softmax function (used in “standard” DarkNet”) with independent logistic classifiers to calculate the likeliness of the input belonging to a specific label. YOLOv3 predicts an objectness score for each bounding box using logistic regression. YOLOv3 changes the way in calculating the cost function.



*Let's see an example:  
Matlab\_Examples\Detection*



# *Neural Style Transfer*



# Neural Style Transfer

If you are a photography enthusiast, you may be familiar with the filter. It can change the color style of photos so that landscape photos become sharper or portrait photos have whitened skins. However, one filter usually only changes one aspect of the photo. To apply an ideal style to a photo, you probably need to try many different filter combinations. This process is as complex as tuning the hyperparameters of a model.

In this section, we will leverage layerwise representations of a CNN to automatically apply the style of one image to another image, i.e., *style transfer*. This task needs two input images: one is the *content image* and the other is the *style image*. We will use neural networks to modify the content image to make it close to the style image in style. For example, the content image in  is a landscape photo taken by us in Mount Rainier National Park in the suburbs of Seattle, while the style image is an oil painting with the theme of autumn oak trees. In the output synthesized image, the oil brush strokes of the style image are applied, leading to more vivid colors, while preserving the main shape of the objects in the content image.

Content image



Synthesized image



Style image

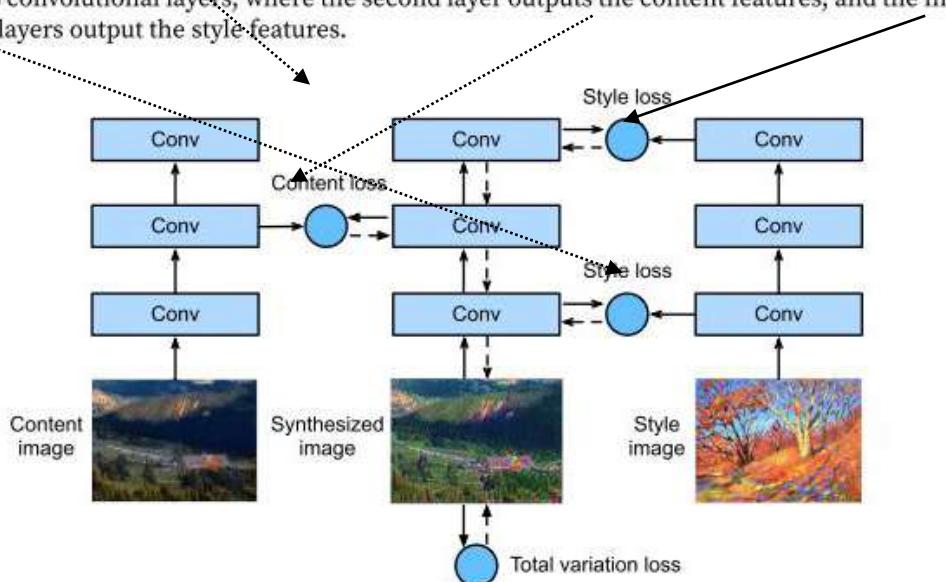


Given content and style images, style transfer outputs a synthesized image.



# Neural Style Transfer

illustrates the CNN-based style transfer method with a simplified example. First, we initialize the synthesized image, for example, into the content image. This synthesized image is the only variable that needs to be updated during the style transfer process, i.e., the model parameters to be updated during training. Then we choose a pretrained CNN to extract image features and freeze its model parameters during training. This deep CNN uses multiple layers to extract hierarchical features for images. We can choose the output of some of these layers as content features or style features. The pretrained neural network here has 3 convolutional layers, where the second layer outputs the content features, and the first and third layers output the style features.



CNN-based style transfer process. Solid lines show the direction of forward propagation and dotted lines show backward propagation.

Next, we calculate the loss function of style transfer through forward propagation (direction of solid arrows), and update the model parameters (the synthesized image for output) through back-propagation (direction of dashed arrows). The loss function commonly used in style transfer consists of three parts: (i) *content loss* makes the synthesized image and the content image close in content features; (ii) *style loss* makes the synthesized image and style image close in style features; and (iii) *total variation loss* helps to reduce the noise in the synthesized image. Finally, when the model training is over, we output the model parameters of the style transfer to generate the final synthesized image.

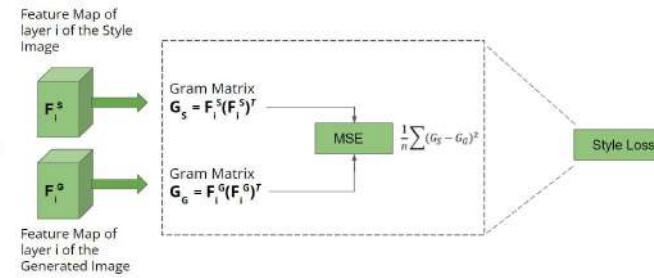
Let  $A^l_{ij}(I)$  be the activation of the  $l$ th layer,  $i$ th feature map and  $j$ th position obtained using the image  $I$ . Then the content loss is defined as,

$$L_{content} = \frac{1}{2} \sum_{i,j} (A^l_{ij}(g) - A^l_{ij}(c))^2$$

The content loss

In style transfer the only variable to be updated in the training step is the synthesized image.

Style and Total variation loss are explained in the next page.



# Neural Style Transfer - matlab example

<https://it.mathworks.com/help/images/neural-style-transfer-using-deep-learning.html>

In order to extract the content features and style features of the image, we can select the output of certain layers in the VGG network. Generally speaking, the closer to the input layer, the easier to extract details of the image, and vice versa, the easier to extract the global information of the image. In order to avoid excessively retaining the details of the content image in the synthesized image, we choose a VGG layer that is closer to the output as the *content layer* to output the content features of the image. We also select the output of different VGG layers for extracting local and global style features. These layers are also called *style layers*.

This mean to extract the values of a given layer of the CNN

Style loss, similar to content loss, also uses the squared loss function to measure the difference in style between the synthesized image and the style image. To express the style output of any style layer, we first use the `extract_features` function to compute the style layer output. Suppose that the output has 1 example,  $c$  channels, height  $h$ , and width  $w$ , we can transform this output into matrix  $\mathbf{X}$  with  $c$  rows and  $hw$  columns. This matrix can be thought of as the concatenation of  $c$  vectors  $\mathbf{x}_1, \dots, \mathbf{x}_c$ , each of which has a length of  $hw$ . Here, vector  $\mathbf{x}_i$  represents the style feature of channel  $i$ .

If you extract  $l$  style layers you have  $l$  gram matrices, the loss is the (weighted) average of all the  $l$  style loss

In the *Gram matrix* of these vectors  $\mathbf{XX}^T \in \mathbb{R}^{c \times c}$ , element  $x_{ij}$  in row  $i$  and column  $j$  is the dot product of vectors  $\mathbf{x}_i$  and  $\mathbf{x}_j$ . It represents the correlation of the style features of channels  $i$  and  $j$ . We use this Gram matrix to represent the style output of any style layer. Note that when the value of  $hw$  is larger, it likely leads to larger values in the Gram matrix. Note also that the height and width of the Gram matrix are both the number of channels  $c$ . To allow style loss not to be affected by these values, the `gram` function divides the Gram matrix by the number of its elements, i.e.,  $chw$ .

Sometimes, the learned synthesized image has a lot of high-frequency noise, i.e., particularly bright or dark pixels. One common noise reduction method is *total variation denoising*. Denote by  $x_{i,j}$  the pixel value at coordinate  $(i, j)$ . Reducing total variation loss

$$\sum_{i,j} |x_{i,j} - x_{i+1,j}| + |x_{i,j} - x_{i,j+1}|$$

makes values of neighboring pixels on the synthesized image closer.

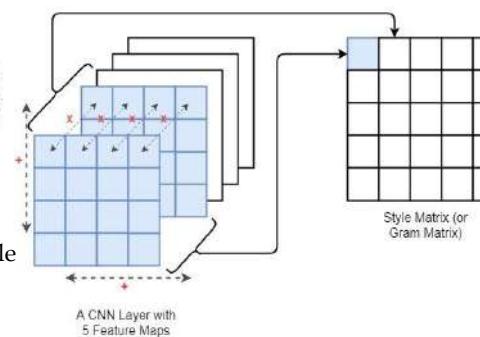
To reduce the computation time, it is assumed that the Gram matrix based on the style images of your dataset have been precomputed.

You can use also a different  
► CNN as ResNet50

After the gram matrix, the style loss is a squared distance between the gram matrix of the generated image and the gram matrix of the style image. In this case, the distance is called the Frobenius norm because it is measuring the distance between two matrices.

$$L_{style} = \sum_l \sum_{i,j} (\beta G_{i,j}^{s,l} - \beta G_{i,j}^{p,l})^2$$

Style Loss, defined as the squared-loss error between the gram matrices between  $s$  &  $p$   
 $s$  = Style Image  
 $p$  = Generated Image



[https://en.wikipedia.org/wiki/Gram\\_matrix](https://en.wikipedia.org/wiki/Gram_matrix)

How the Style Matrix is Computed for a CNN Layer with 5 Feature Maps

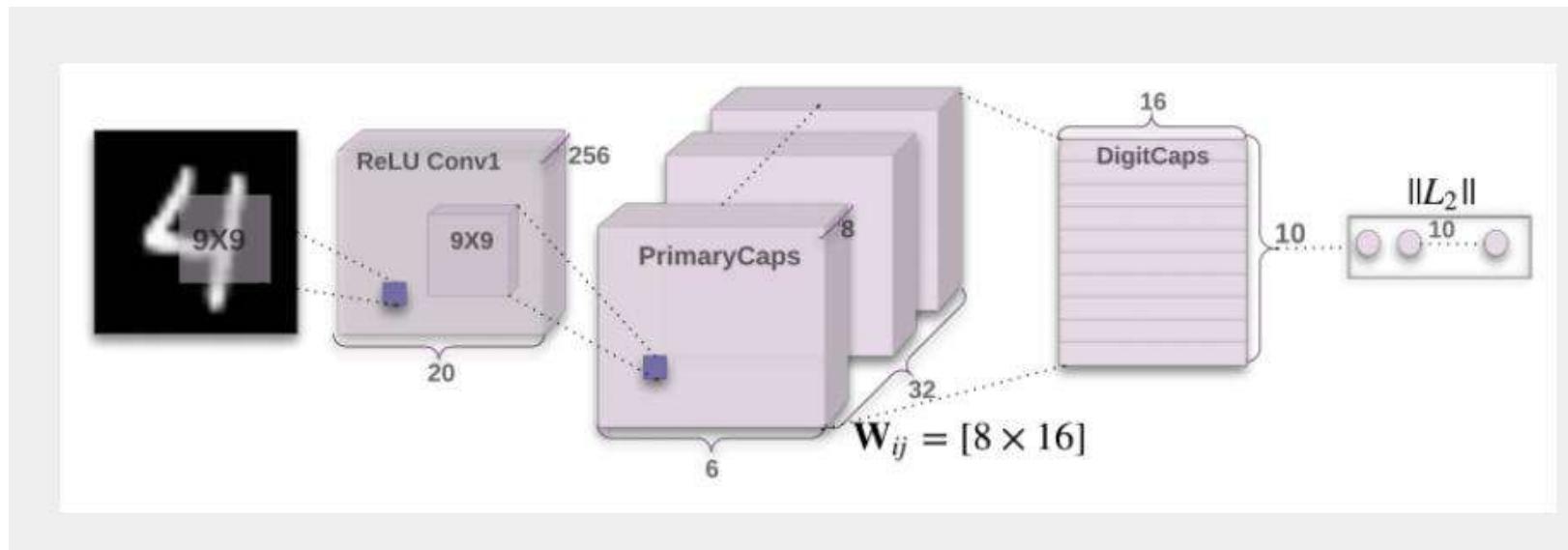
# *Capsule Network*



# Capsule network

Convolution Neural Network being computationally strong have the ability to automatically detect feature maps from the images. It's the CNN algorithm that has made it easy for a machine to do several image-related tasks whether it is about classification or detection. In CNN, the convolution layer holds a very important function that is to detect the features from an image pixel. Deeper ConvNet layers detect the simple features like edges and color. Although the performance of CNN is really good, still they have few of the drawbacks. As quoted by Hinton "[The pooling operation used in convolutional neural networks is a big mistake](#), and the fact that it works so well is a disaster. There is a loss of valuable information when we are using pooling layers.

Capsule Networks (CapsNet) are the networks that are able to fetch spatial information and more important features so as to overcome the loss of information that is seen in pooling operations. Let us see what is the difference between a capsule and a neuron. Capsule gives us a vector as an output that has a direction. For example, if you are changing the orientation of the image then the vector will also get moved in that same direction whereas the output of a neuron is a scalar quantity that does not tell anything about the direction.

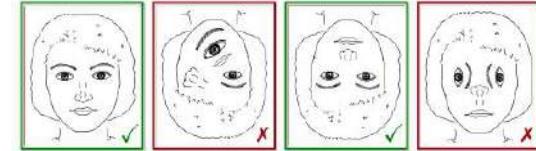
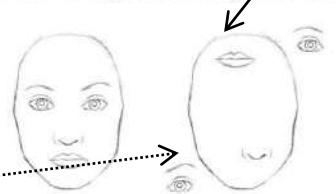
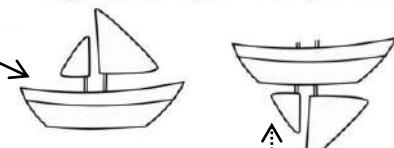


# Capsule network

Let me run through an example. Say you rotate a boat or a face upside down and then feed it to a CNN; it would not be able to identify features like the eyes, nose, or mouth. Similarly, if you reconstruct specific regions of the face (i.e., switch the positions of the nose and eyes), the network will still be able to recognize the face—even though it isn't exactly a face anymore. In short, CNNs can learn the patterns of the images statistically, but not what the actual image looks like in different orientations. In deep learning, the activation level of a neuron is often interpreted as the likelihood of detecting a specific feature.

*Internal data representation of a convolutional neural network does not take into account important spatial hierarchies between simple and complex objects.*

The following pictures may fool a *simple* CNN model in believing that this is a good sketch



If CNNs are fed with images of different sizes and orientations, they fail. The boat is NOT recognized as a boat, but the weird face is classified as a face.

They cannot extrapolate their understanding of geometric relationships to radically new viewpoints

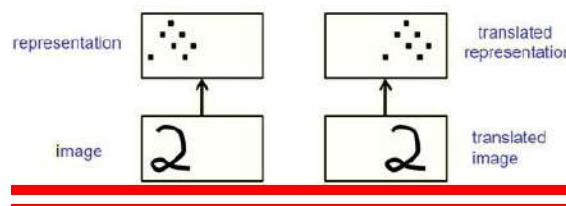


In general it is said that:

- an Operator is **invariant** with respect to a Transformation when the effect of the Transformation is not detectable in the Operator Output
- an Operator is **equivariant** with respect to a Transformation when the effect of the Transformation is detectable in the Operator Output

Sub-sampling tries to make the **neural activities** *invariant* to small changes in viewpoint.

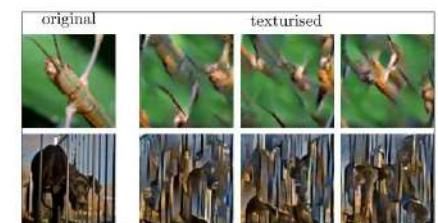
- This is the wrong goal, motivated by the fact that the final label needs to be viewpoint-invariant.
- It's better to aim for **equivariance**: we want that changes in viewpoint lead to corresponding changes in neural activities.



Without pooling, CNN give «place-coded» equivariance for discrete translation

$$\begin{array}{ccc} \boxed{\text{---}} & \star f & = \boxed{\text{---}} \\ x & R_{\text{flip}}(x) & [x * f] \\ \boxed{\text{---}} & \star f & = \boxed{\text{---}} \\ x' & & [x' * f] \end{array}$$

Example of lack of equivariance in convolution. Each filter weight was sampled as:  $f_i \sim \text{Uniform}(0, 1)$ . Unlike the translation case, the output  $[x' * f]$  is not simply a rotated version of  $[x * f]$  as convolution is not rotation equivariant.

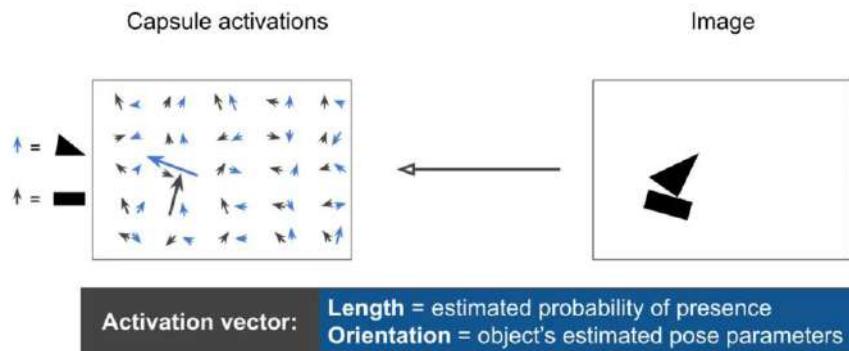


Example of texture bias in CNNs. CNNs still achieve high accuracy on the texturised images, whereas humans do not due to the loss of global shape information.

# Capsule network

- Group of neurons that perform a lot of internal computation and then encapsulate the results of these computations into a **small vector** of highly informative outputs. Inspired by mini-column in brain.
- Each capsule learns to recognize an implicitly defined visual **entity** over a **limited domain** of viewing conditions and deformations
- It outputs two things (embedded in the vector):
  1. the **probability** that the entity is present within its limited domain
  2. a set of “instantiation parameters”, the generalized **pose** of the object. That may include the precise position, lighting and deformation of the visual entity relative to an implicitly defined canonical version of that entity

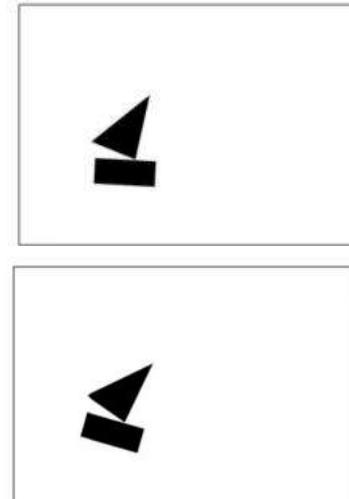
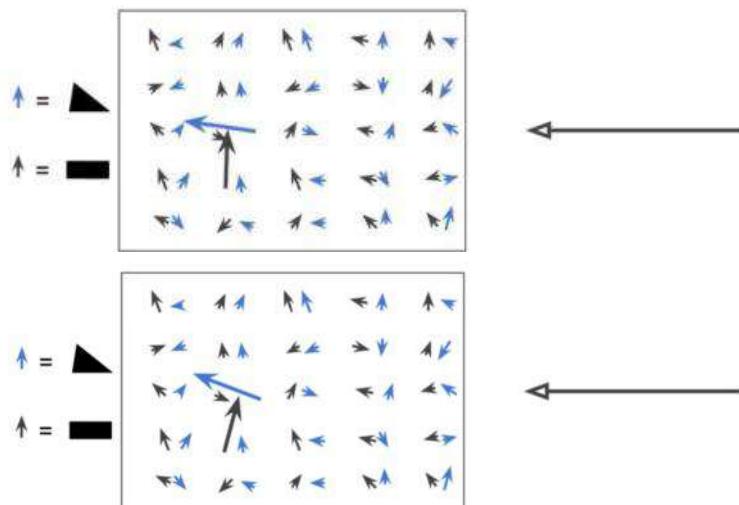
In the last paper (*Dynamic Routing Between Capsules*), Capsules encode probability of detection of a feature **as the length** of their output vector.



# Capsule network

- Capsules encode probability of detection of a feature as the length of their output vector. And the state of the detected feature is encoded as the direction in which that vector points to ("instantiation parameters").
- So when detected feature moves around the image or its state somehow changes, the probability still stays the same (length of vector does not change), but its orientation changes.
- This is what Hinton refers to as **activities equivariance**: neuronal activities will change when an object "moves over the manifold of possible appearances" in the picture. At the same time, the **probabilities of detection remain constant**, which is the form of invariance that we should aim at, and not the type offered by CNNs with max pooling.

each arrow is the output of a given capsule



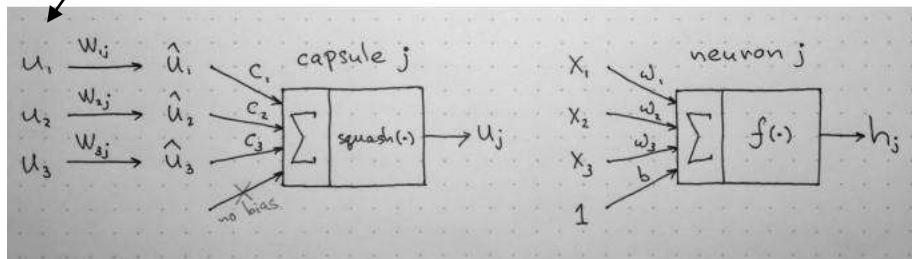
If the triangle and the rectangle rotate of some degrees, the probability (length of vector) is the same but the orientations of the vectors change. We say that it is an activities equivariance since the probability of the given capsule does not change, only the orientation changes.



# Capsule networks

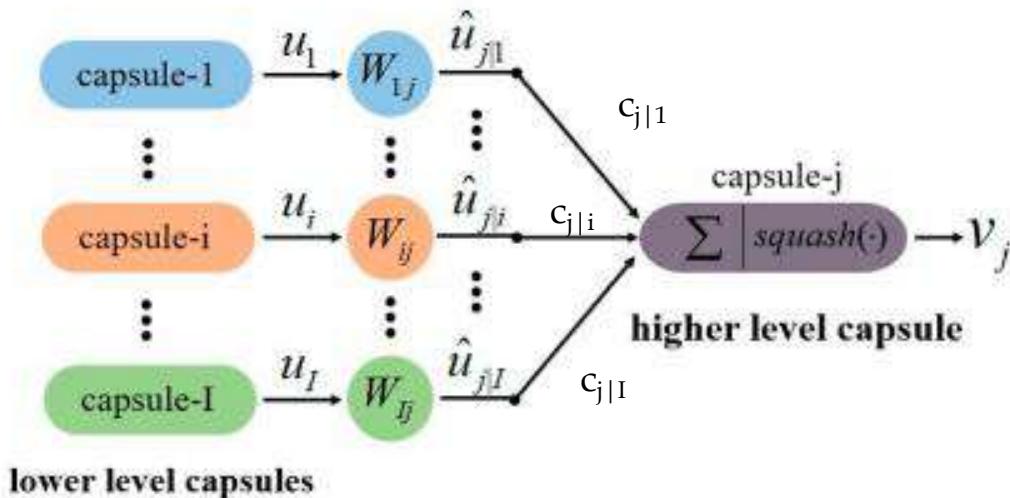
They are organized in layers.

Let  $u_1, u_2, u_3$  be the output **vectors** coming from capsules of the layer below. The vector is sent to all possible parents in the neural network.



These vectors then are multiplied by corresponding weight matrices  $W$  (**learned** during training) that encode important spatial and other relationships between lower level features (eyes, mouth and nose) and higher level feature (face).  $W$  performs an **affine transformation**

We get the predicted position of the higher level feature,  $\hat{u}_{j|i} = W_{ij}u_i$   
i.e. where the face should be according to the detected position of the eyes



# Capsule Network

There are 4 main components that are present in the CapsNet that are listed below:

Matrix Multiplication – It is applied to the image that is given as an input to the network to convert into vector values to understand the spatial part (see pg 360).

Scalar Weighting of the Input (it means  $W \times u$ , see previous figure) – It computes which higher-level capsule should receive the current capsule output.

Dynamic routing algorithm (detailed on page 356, for finding the values  $c$ , see previous figure) – It permits these different components to transfer information amongst each other. Higher-level capsules get the input from the lower level. This is a iterative process.

Squashing Function (detailed on page 358) – It is the last component that condenses the information. The squashing function takes all the information and converts it into a vector.

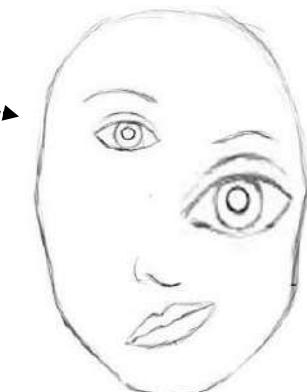
Let's take a look at capsules and how they go about solving the problem of providing spatial information.

When we look at some of the logic that's behind CNN's, we begin to notice where its architecture fails. Take a look at this picture.

It doesn't look quite right for a face, even though it has all the necessary components to make up a face. We know that this is not how faces are supposed to look, but because CNN's only look for features in images, and don't pay attention to their pose, it's hard for them to notice a difference between that face and a real face.

How capsule networks solve this problem is by implementing groups of elements that encode spatial information as well as the probability of an object being present. The length of a capsule vector is the probability of the feature existing in the image and the direction of the vector would represent its pose information.

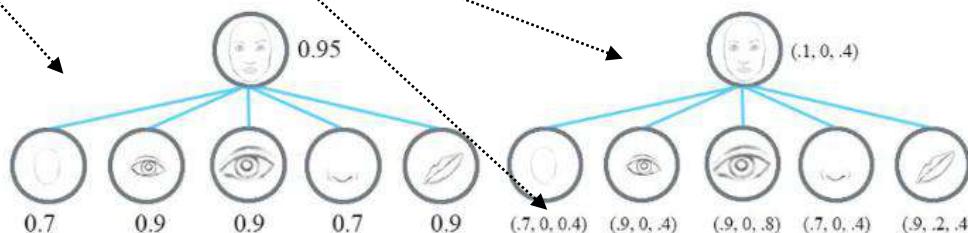
In computer graphics applications such as design and rendering, objects are often created by giving some sort of parameter which it will render from. However, in capsules networks, it's the opposite, where the network learns how to inversely render an image; looking at an image and trying to predict what the instantiation parameters for it are.



# Capsule

Vectors encode more information: relational and relative information. Imagine that instead of taking just the scalar activation of a feature, we considered its vector containing something like [probability, orientation, size]. The original scalar version might work something like the diagram on the left; it detects a face even though an eye is huge relative to the face (95% probability)! Capsules on the other hand due to their richer vector information see that the sizes of the features are different and therefore output a lower likelihood for the detection of the face! Pay special attention to the scores in the diagrams to fully understand.

With this spatial information, we can detect the in-consistency in the orientation and size among the nose, eyes and ear features and therefore output a much lower activation for the face detection.



\*Left\*: How a regular CNN would detect facial features. \*Right\*: How a capsule network would detect features.

CNNs only consider probability, the capsule uses vectors that encode [probability, orientation, size], for example, an eye is huge compared to the face (size 0.9 of the large eye with respect to 0.4 of the right size eye).

Since a capsule has vector information, it can learn a richer representation of images.

Capsules also have the advantage of being able to achieve good accuracy with far less training data. Equivariance is the detection of objects that can transform to each other. Thus, since a capsule has vector information such as orientation, size, etc, it can use that information to learn a richer representation of the features as it is trained. It doesn't need 50 examples of the same rotated dog; it just needs one with a vector representation which can easily be transformed. By forcing the model to learn the feature variant in a capsule, we may extrapolate possible variants more effectively with less training data.



# Capsule network

		capsule	VS.	traditional neuron
Input from low-level neuron/capsule		vector( $\mathbf{u}_i$ )		scalar( $x_i$ )
Operation	Affine Transformation	$\hat{\mathbf{u}}_{j i} = W_{ij}\mathbf{u}_i$		—
	Weighting	$s_j = \sum_i c_{ij} \hat{\mathbf{u}}_{j i}$		$a_j = \sum_{i=1}^3 W_i x_i + b$
	Sum			
	Non-linearity activation fun	$v_j = \frac{\ s_j\ ^2}{1 + \ s_j\ ^2} \frac{s_j}{\ s_j\ }$		$h_{w,b}(x) = f(a_j)$
output		vector( $\mathbf{v}_j$ )		scalar( $h$ )
$\mathbf{u}_1 \xrightarrow{w_{1j}} \hat{\mathbf{u}}_1$ $\mathbf{u}_2 \xrightarrow{w_{2j}} \hat{\mathbf{u}}_2$ $\mathbf{u}_3 \xrightarrow{w_{3j}} \hat{\mathbf{u}}_3$ $+ \mathbf{b}$		$\Sigma$ squash( $\cdot$ ) $\rightarrow \mathbf{v}_j$		$\Sigma$ $f(\cdot)$ $\rightarrow h_{w,b}(x)$ $f(\cdot)$ : sigmoid, tanh, ReLU, etc.
<b>Capsule = New Version Neuron!</b> <b>vector in, vector out VS. scalar in, scalar out</b>				

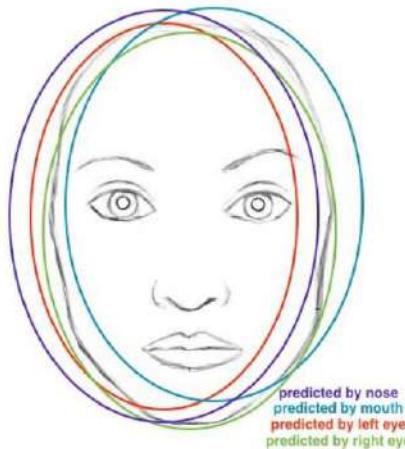
$c_{ij}$  are coupling coefficients that are determined by the iterative dynamic routing process.

$\sum_j c_{ij}$  are designed to sum to one. Conceptually,  $c_{ij}$  measures how likely capsule  $i$  may activate capsule  $j$ .



# Capsule

- Affine Transformation (see previous page): The input vectors represent either the initial input, or an input provided by an earlier layer in the network. These vectors are first multiplied by the weight matrices. The weight matrix, as described previously, captures the spatial relationships. Say that one object is centered around another, and they are equally proportioned in size. The product of the input vector and the weight matrix will signify the high-level feature. For example, if the low-level features are nose, mouth, left eye, and right eye, then if the predictions of the four low-level features point to the same orientation and size of a face, a face will be what's predicted (as shown below). This is what the "high-level" feature is:



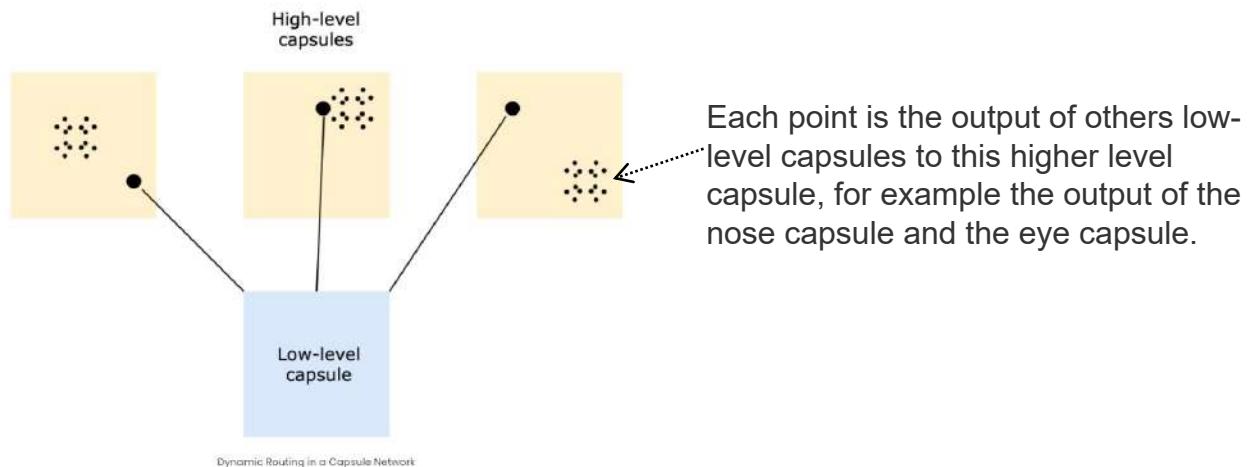
For example, matrix  $W_{1j}$  may encode relationship between nose and face: face is centered around its nose, its size is 10 times the size of the nose and its orientation in space corresponds to orientation of the nose. In other words,  $W_{1j} \times u_1$  represents where the face should be according to the detected position of the nose

- For example, matrix  $W_{1j}$  may encode relationship between nose and face: face is centered around its nose, its size is 10 times the size of the nose and its orientation in space corresponds to orientation of the nose, because they all lie on the same plane. Similar intuitions can be drawn for the other matrices. After multiplication by these matrices, what we get is the predicted position of the higher level feature. In other words,  $W_{1j} \times u_1$  represents where the face should be according to the detected position of the nose,  $W_{2j} \times u_2$  represents where the face should be according to the detected position of the mouth and  $W_{3j} \times u_3$  represents where the face should be according to the detected position of the left eye...



# Dynamic Routing

- Weighting: A capsule network adjusts the weights such that a low-level capsule is strongly associated with high-level capsules that are in its proximity. The proximity measure is determined by the affine transformation step we discussed previously. The distance between the outputs obtained from the affine transformation step and the dense clusters of the predictions of low-level capsules is computed (the dense clusters could be formed if the predictions made by the low-level capsules are similar, thus lying near to each other). The high-level capsule that has the minimum distance between the cluster of already made predictions and the newly predicted one will have a higher weight, and the remaining capsules would be assigned lower weights, based on the distance metric.



- In the above image, the weights would be assigned in the following order: middle > left > right. In a nutshell, the essence of the dynamic routing algorithm could be seen as this: the lower level capsule will send its input to the higher level capsule that “agrees” with the output of the given low-level capsule.

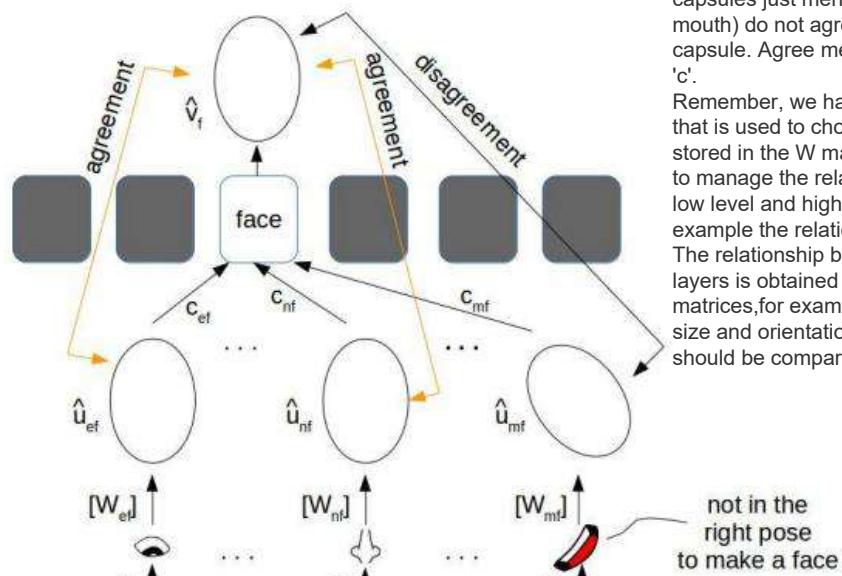
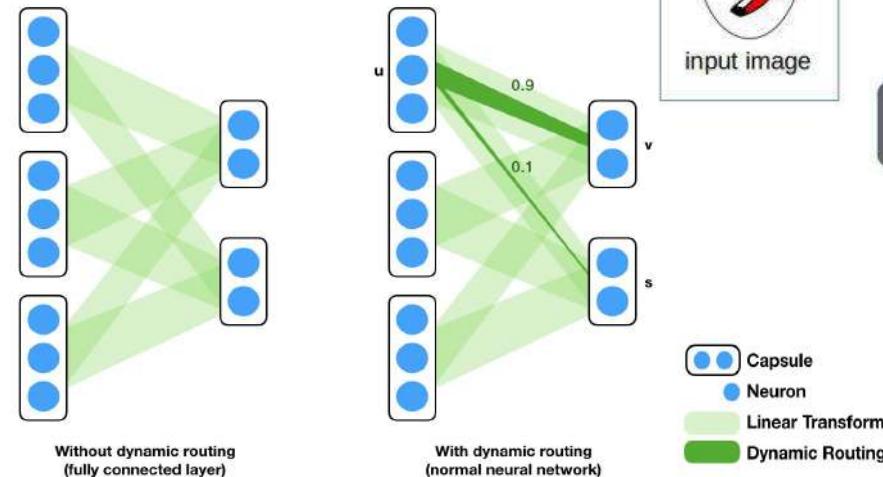


# Routing by agreement

- According to Hinton, when a visual stimulus is triggered, the brain has an inbuilt mechanism to "route" low level visual data to parts of the brain where it believes can handle it best.
- ConvNets perform routing via pooling layers, a very primitive way to do routing as it only attends to the most active neuron in the pool.

Capsule Networks is different as it tries to send the information to the capsule above it that is best at dealing with it.

The parameters  $W_{ij}$  models a "**part-whole**" relationship between the lower and higher level entities



If the capsule next to "face" represents the "dog" class, all three capsules just mentioned (nose, eye, mouth) do not agree with the dog capsule. Agree means a high value for 'c'.

Remember, we have a training set that is used to choose the weights stored in the  $W$  matrices.  $W$  are used to manage the relationship between low level and higher level capsule. For example the relationship eye-face. The relationship between two capsule layers is obtained using the  $W$  matrices, for example, to learn how the size and orientation of the nose should be compared to the usual face.



# Recap

Things to know about weights  $c_{ij}$ :

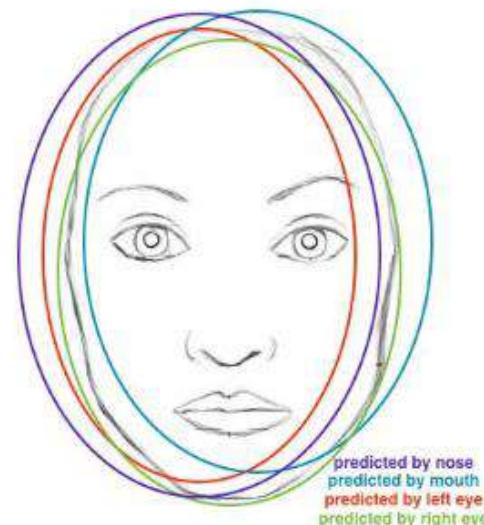
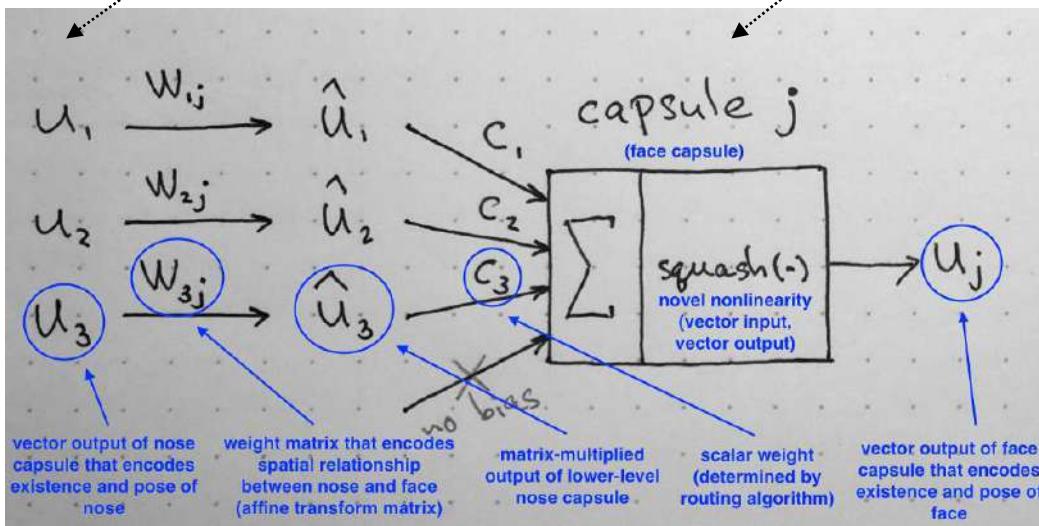
1. Each weight is a non-negative scalar
2. For each lower level capsule  $i$ , the sum of all weights  $c_{ij}$  equals to 1
3. For each lower level capsule  $i$ , the number of weights equals to the number of higher-level capsules
4. These weights are determined by the iterative dynamic routing algorithm

The first two facts allow us to interpret weights in probabilistic terms. Recall that the length a capsule's output vector is interpreted as probability of existence of the feature that this capsule has been trained to detect. Orientation of the output vector is the parametrized state of the feature. So, in a sense, for each lower level capsule  $i$ , its weights  $c_{ij}$  define a probability distribution of its output belonging to each higher level capsule  $j$ .

Input vectors that our capsule receives ( $u_1, u_2$  and  $u_3$  in the diagram) come from 3 other capsules in the layer below. Lengths of these vectors encode probabilities that lower-level capsules detected their corresponding objects and directions of the vectors encode some internal state of the detected objects. Let us assume that lower level capsules detect eyes, mouth and nose respectively and out capsule detects face.

These vectors then are multiplied by corresponding weight matrices  $W$  that encode important spatial and other relationships between lower level features (eyes, mouth and nose) and higher level feature (face). For example, matrix  $W_{2j}$  may encode relationship between nose and face: face is centered around its nose, its size is 10 times the size of the nose and its orientation in space corresponds to orientation of the nose, because they all lie on the same plane. Similar intuitions can be drawn for matrices  $W_{1j}$  and  $W_{3j}$ . After multiplication by these matrices, what we get is the predicted position of the higher level feature. In other words,  $\hat{u}_1$  represents where the face should be according to the detected position of the eyes,  $\hat{u}_2$  represents where the face should be according to the detected position of the mouth and  $\hat{u}_3$  represents where the face should be according to the detected position of the nose.

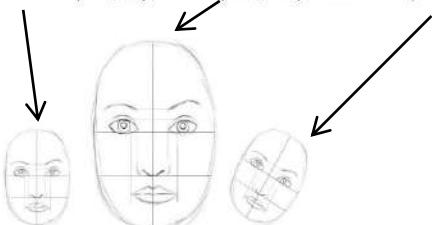
At this point your intuition should go as follows: if these 3 predictions of lower level features point at the same position and state of the face, then it must be a face there.



# Capsule network

## Intuition

We collect 3 similar sketches with different scale and orientation, and we measures the horizontal width of the mouth and the eye in pixels. They are  $s^{(1)} = (100, 66)$ ,  $s^{(2)} = (200, 131)$  and  $s^{(3)} = (50, 33)$ .



Let's assume  $W_m = 2$ ,  $W_e = 3$ , and we calculate a vote from the mouth and the eye for  $s^{(1)}$  as:

$$W_m \times \text{width}_m = 2 \times 100 = 200$$

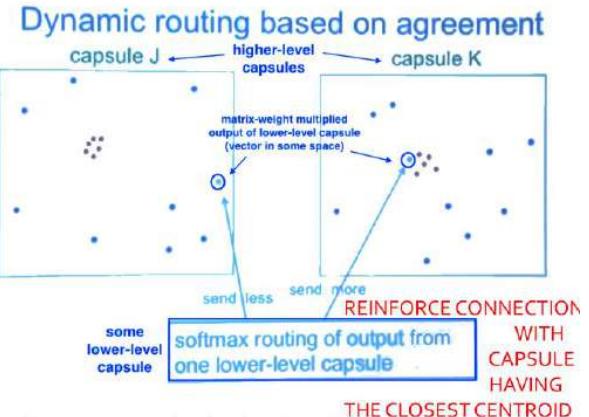
$$W_e \times \text{width}_e = 3 \times 66 = 198$$

So the mouth capsule and the eye capsule can be strongly related to a parent capsule with width approximate 200 pixels. From our experience, a face is 2 times ( $W_m = 2$ ) the width of a mouth and 3 times the width ( $W_e = 3$ ) of an eye. So the parent capsule we detected is a face capsule. Of course, we can make it more accurate by adding more properties like height or color. In dynamic routing, we transform the vectors of an input capsules with a transformation matrix  $W$  to form a vote, and group capsules with similar votes. Those votes eventually becomes the output vector of the parent capsule. So how can we know  $W$ ? Just do it in the deep learning way: backpropagation with a cost function.

Intuitively, prediction vector  $\hat{u}_{j|i}$  is the prediction (**vote**) from the capsule  $i$  on the output of the capsule  $j$  above. If the activity vector  $v_j$  has close similarity with the prediction vector, we conclude that capsule  $i$  is highly related with the capsule  $j$ . (For example, the eye capsule is highly related to the face capsule.)

Similarity measured with the “**agreement**” quantity  $a_{ij} = \langle \hat{u}_{j|i}, v_j \rangle$ .

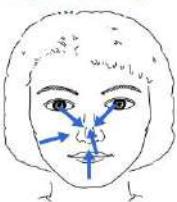
Judging by the values of  $a_{ij}$  we can then “*strengthen*” or “*weaken*” the corresponding connection strength by highering or lowering  $c_{ij}$  appropriately.



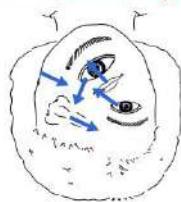
visually we can see that the output of this lower level capsule is close to the cluster center of the different inputs (I mean the output of other capsules) of the capsule K, so the weight ' $c_k$ ' will be greater than the weights ' $c_j$ '.

# Dynamic routing

Agreement ✓

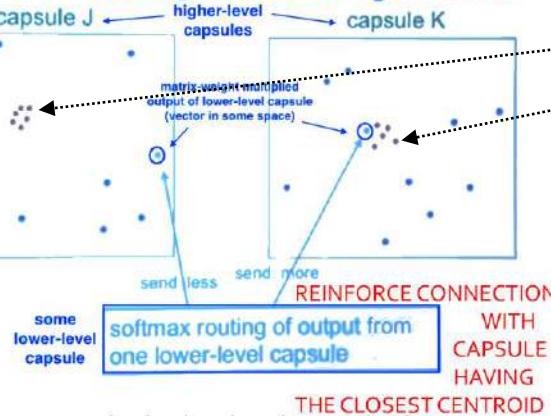


Disagreement ✗



Example of agreement and no agreement between part poses (eyes, nose, mouth etc) with respect to the object (person), where each pose is encoded by a vector. Capsule routing aims to detect objects by looking for agreement between its parts, and thereby perform equivariant inference.

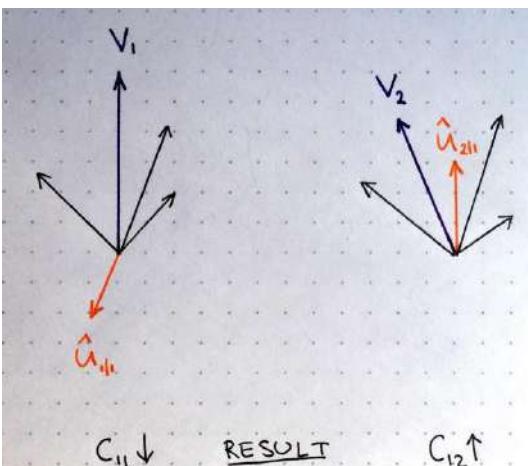
## Dynamic routing based on agreement



In the image above, we have one lower level capsule that needs to “decide” to which higher level capsule it will send its output. It will make its decision by adjusting the weights  $C$  that will multiply this capsule’s output before sending it to either left or right higher-level capsules  $J$  and  $K$ .

Now, the higher level capsules already received many input vectors from other lower-level capsules. All these inputs are represented by red and blue points. Where these points cluster together, this means that predictions of lower level capsules are close to each other. This is why, for the sake of example, there is a cluster of red points in both capsules  $J$  and  $K$ .

So, where should our lower-level capsule send its output: to capsule  $J$  or to capsule  $K$ ? The answer to this question is the essence of the dynamic routing algorithm. The output of the lower capsule, when multiplied by corresponding matrix  $W$ , lands far from the red cluster of “correct” predictions in capsule  $J$ . On the other hand, it will land very close to “true” predictions red cluster in the right capsule  $K$ . Lower level capsule has a mechanism of measuring which upper level capsule better accommodates its results and will automatically adjust its weight in such a way that weight  $C$  corresponding to capsule  $K$  will be high, and weight  $C$  corresponding to capsule  $J$  will be low.



Two higher level capsules with their outputs represented by purple vectors, and inputs represented by black and orange vectors. Lower level capsule with orange output will decrease the weight for higher level capsule 1 (left side) and increase the weight for higher level capsule 2 (right side).

In the figure, imagine that there are two higher level capsules, their output is represented by purple vectors  $v_1$  and  $v_2$ . The orange vector represents input from one of the lower level capsules and the black vectors represent all the remaining inputs from other lower level capsules.

We see that in the left part the purple output  $v_1$  and the orange input point in the opposite directions. In other words, they are not similar. This means their dot product will be a negative number and as result routing coefficient  $c_{11}$  will decrease. In the right part, the purple output  $v^2$  and the orange input point in the same direction. They are similar. Therefore, the routing coefficient  $c_{12}$  will increase. This procedure is repeated for all higher level capsules and for all inputs of each capsule. The result of this is a set of routing coefficients that best matches outputs from lower level capsules with outputs of higher level capsules.

# Capsule network

Remember that the output vector length can be interpreted as probability of a given feature being detected by the capsule.

We want the length of the output vector of a capsule to represent the probability that the entity represented by the capsule is present in the current input. We therefore use a non-linear "squashing" function to ensure that short vectors get shrunk to almost zero length and long vectors get shrunk to a length slightly below 1. We leave it to discriminative learning to make good use of this non-linearity.

Short just means small (in terms of some metric, usually Euclidean norm)

$$\mathbf{v}_j = \frac{\|\mathbf{s}_j\|^2}{1 + \|\mathbf{s}_j\|^2} \frac{\mathbf{s}_j}{\|\mathbf{s}_j\|}$$

$v_j \approx \|\mathbf{s}_j\| s_j$  for  $s_j$  is short  
 $v_j \approx \frac{s_j}{\|\mathbf{s}_j\|}$  for  $s_j$  is long

where  $\mathbf{v}_j$  is the vector output of capsule  $j$  and  $\mathbf{s}_j$  is its total input.

For all but the first layer of capsules, the total input to a capsule  $\mathbf{s}_j$  is a weighted sum over all "prediction vectors"  $\hat{\mathbf{u}}_{j|i}$  from the capsules in the layer below and is produced by multiplying the output  $\mathbf{u}_i$  of a capsule in the layer below by a weight matrix  $\mathbf{W}_{ij}$

$$\mathbf{s}_j = \sum_i c_{ij} \hat{\mathbf{u}}_{j|i}, \quad \hat{\mathbf{u}}_{j|i} = \mathbf{W}_{ij} \mathbf{u}_i$$

Where  $s_j$  is the output obtained from the previous step, and  $v_j$  is the output obtained after applying the non-linearity. The left side of the equation performs additional squashing, while the right side of the equation performs unit scaling of the output vector.

where the  $c_{ij}$  are coupling coefficients that are determined by the iterative dynamic routing process.

The coupling coefficients between capsule  $i$  and all the capsules in the layer above sum to 1 and are determined by a "routing softmax" whose initial logits  $b_{ij}$  are the log prior probabilities that capsule  $i$  should be coupled to capsule  $j$ .

$$c_{ij} = \frac{\exp(b_{ij})}{\sum_k \exp(b_{ik})}$$

A norm on a vector space  $V$  is a function

$$\|\cdot\| : V \rightarrow \mathbb{R}, \quad x \mapsto \|x\|, \quad \|x\|_2 = \sqrt{\sum_{i=1}^n x_i^2}, \quad x \in \mathbb{R}^n$$

which assigns each vector  $x$  its length  $\|x\| \in \mathbb{R}$



# Capsule network - dynamic routing

The agreement is simply the scalar product  $a_{ij} = \mathbf{v}_j \cdot \hat{\mathbf{u}}_{j|i}$ . This agreement is treated as if it was a log likelihood and is added to the initial logit,  $b_{ij}$  before computing the new values for all the coupling coefficients linking capsule  $i$  to higher level capsules.

In convolutional capsule layers, each capsule outputs a local grid of vectors to each type of capsule in the layer above using different transformation matrices for each member of the grid as well as for each type of capsule.

## Procedure 1 Routing algorithm.

```

1: procedure ROUTING( $\hat{\mathbf{u}}_{j|i}, r, l$ )
2:   for all capsule  $i$  in layer  $l$  and capsule  $j$  in layer  $(l + 1)$ :  $b_{ij} \leftarrow 0$ .
3:   for  $r$  iterations do                                remember that  $\mathbf{b}_i$  and  $\mathbf{c}_i$  are vectors
4:     for all capsule  $i$  in layer  $l$ :  $\mathbf{c}_i \leftarrow \text{softmax}(\mathbf{b}_i)$ 
5:     for all capsule  $j$  in layer  $(l + 1)$ :  $\mathbf{s}_j \leftarrow \sum_i c_{ij} \hat{\mathbf{u}}_{j|i}$ 
6:     for all capsule  $j$  in layer  $(l + 1)$ :  $\mathbf{v}_j \leftarrow \text{squash}(\mathbf{s}_j)$ 
7:     for all capsule  $i$  in layer  $l$  and capsule  $j$  in layer  $(l + 1)$ :  $b_{ij} \leftarrow b_{ij} + \hat{\mathbf{u}}_{j|i} \cdot \mathbf{v}_j$ 
return  $\mathbf{v}_j$ 
```

The dynamic routing algorithm allows us to better pass data between layers in the network. In practice, it is better to use a small number of routing iterations, because over iterating will lead to overfitting and poor performance.

In a nutshell, the essence of the dynamic routing algorithm could be seen as this: the lower level capsule will send its output to the higher level capsule that “agrees” with it.

Line 1: This line defines the procedure of ROUTING, which takes affine transformed input ( $\mathbf{u}$ ), the number of routing iterations ( $r$ ), and the layer number ( $l$ ) as inputs.

Line 2:  $b_{ij}$  is a temporary value that is used to initialize  $c_i$  in the end.

Line 3: The for loop iterates ‘ $r$ ’ times.

Line 4: The softmax function applied to  $\mathbf{b}_i$  makes sure to output a non-negative  $\mathbf{c}_i$ , where all the outputs sum to 1.

Line 5: For every capsule in the succeeding layer, the weighted sum is computed.

Line 6: For every capsule in the succeeding layer, the weighted sum is squashed.

Line 7: The weights  $b_{ij}$  are updated here.  $\mathbf{u}_{j|i}$  denotes the input to the capsule from low-level capsule  $i$ , and  $\mathbf{v}_j$  denotes the output of high-level capsule  $j$ .

In the following the loss function is detailed, as case study a digit classification is used (as in the original paper).

We are using the length of the instantiation vector to represent the probability that a capsule’s entity exists. We would like the top-level capsule for digit class  $k$  to have a long instantiation vector if and only if that digit is present in the image. To allow for multiple digits, we use a separate margin loss.  $L_k$  for each digit capsule,  $k$ :

$$L_k = T_k \max(0, m^+ - \|\mathbf{v}_k\|)^2 + \lambda (1 - T_k) \max(0, \|\mathbf{v}_k\| - m^-)^2$$

where  $T_k = 1$  iff a digit of class  $k$  is present and  $m^+ = 0.9$  and  $m^- = 0.1$ . The  $\lambda$  down-weighting of the loss for absent digit classes stops the initial learning from shrinking the lengths of the activity vectors of all the digit capsules. We use  $\lambda = 0.5$ . The total loss is simply the sum of the losses of all digit capsules.

more details on the loss are given on page 362

Step on line 7 captures the essence of the routing algorithm. This step looks at each higher level capsule  $j$  and then examines each input and updates the corresponding weight  $b_{ij}$  according to the formula. The formula says that the new weight value equals to the old value plus the dot product of current output of capsule  $j$  and the input to this capsule from a lower level capsule  $i$ . The dot product looks at similarity between input to the capsule and output from the capsule. Also, remember, the lower level capsule will send its output to the higher level capsule whose output is similar. This similarity is captured by the dot product.

# Capsule network

A simple CapsNet architecture is shown in Fig. ● The architecture is shallow with only two convolutional layers and one fully connected layer. Conv1 has 256,  $9 \times 9$  convolution kernels with a stride of 1 and ReLU activation. This layer converts pixel intensities to the activities of local feature detectors that are then used as inputs to the *primary capsules*.

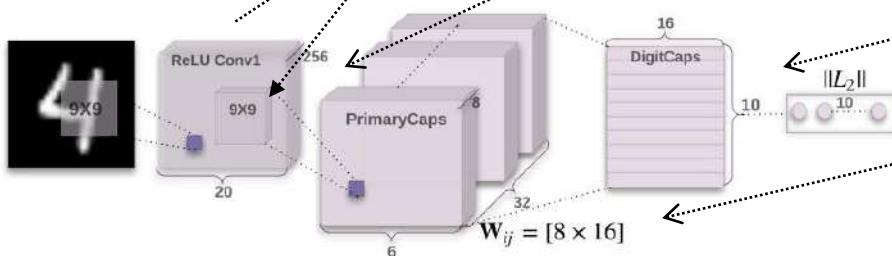
The primary capsules are the lowest level of multi-dimensional entities and, from an inverse graphics perspective, activating the primary capsules corresponds to inverting the rendering process. This is a very different type of computation than piecing instantiated parts together to make familiar wholes, which is what capsules are designed to be good at.

The second layer (PrimaryCapsules) is a convolutional capsule layer with 32 channels of convolutional 8D capsules (*i.e.* each primary capsule contains 8 convolutional units with a  $9 \times 9$  kernel and a stride of 2). Each primary capsule output sees the outputs of all  $256 \times 81$  Conv1 units whose receptive fields overlap with the location of the center of the capsule. In total PrimaryCapsules has  $[32 \times 6 \times 6]$  capsule outputs (each output is an 8D vector) and each capsule in the  $[6 \times 6]$  grid is sharing their weights with each other.

The final Layer (DigitCaps) has one 16D capsule per digit class and each of these capsules receives input from all the capsules in the layer below.

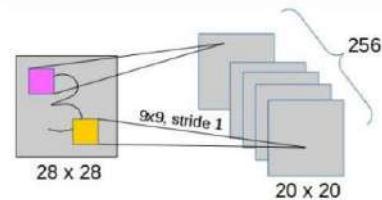
We have routing only between two consecutive capsule layers (*e.g.* PrimaryCapsules and DigitCaps). Since Conv1 output is 1D, there is no orientation in its space to agree on. Therefore, no routing is used between Conv1 and PrimaryCapsules. All the routing logits ( $b_{ij}$ ) are initialized to zero. Therefore, initially a capsule output ( $\mathbf{u}_i$ ) is sent to all parent capsules ( $\mathbf{v}_0 \dots \mathbf{v}_9$ ) with equal probability ( $c_{ij}$ ).

The length of the activity vector of each capsule in DigitCaps layer indicates presence of an instance of each class and is used to calculate the classification loss.  $\mathbf{W}_{ij}$  is a weight matrix between each  $\mathbf{u}_i, i \in [1, 32 \times 6 \times 6]$  in PrimaryCapsules and  $\mathbf{v}_j, j \in [1, 10]$ .



ten rows since the studied problem has 10 classes

We have  $6 \times 6 \times 32$   $\mathbf{W}$  matrices, each of that size  $8 \times 16$ ; if the the weights are shared between the capsules of each  $6 \times 6$  grid, means these  $6 \times 6$  capsules have the same  $\mathbf{W}$



In this CapsNet input is 28x28

The first layer is responsible for extracting the basic features of the image

The second layer (PrimaryCaps) is responsible for taking these basic features and finding more detailed patterns between them.

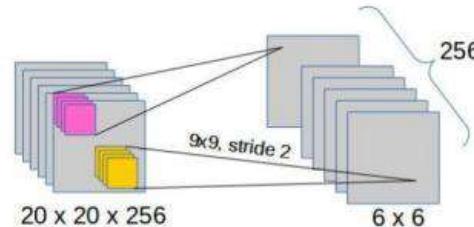
Matrix multiplication:

We perform a weight matrix multiplication between the info passed from the first to the second layer to encode the information of understanding spatial relationships.

# Capsule

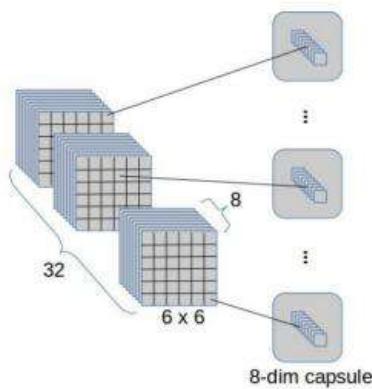
## Second convolutional layer or the PrimaryCaps layer:

1. another convolutional layer which produces 256 activation maps of  $6 \times 6$



2. Output of the second convolutional layer ( $6 \times 6 \times 256$ ) interpreted as a set of 32 "capsule activation maps" with capsule dimension 8.

A total of  $6 \times 6 \times 32 = 1152$  capsules (each of dimension 8)

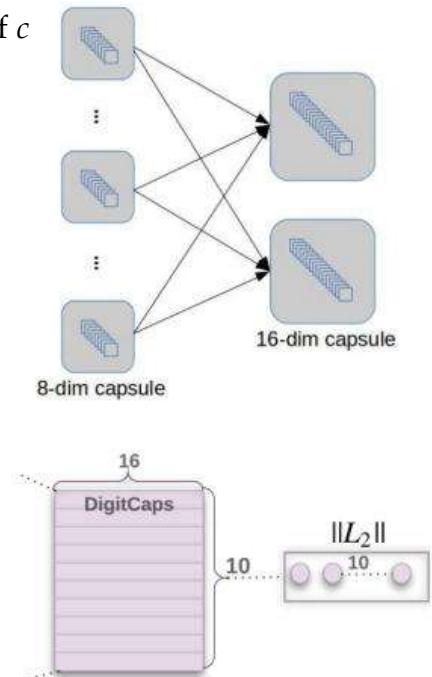


# Capsule

So, there are 1152 matrices, and also 1152  $c$  coefficients and 1152  $b$  coefficients used in the dynamic routing (for the definition of  $c$  and  $b$ , see pg. 359).

## Capsule-to-capsule layer or DigitCaps layers:

- The 1152 (lower level) capsules are connected to 10 (higher levels)
- The  $8 \times 16 W_{ij}$  is the weight matrix used for affine transformation against each 8-D capsule. The weights are shared. So, there are 1152 matrices  $8 \times 16$
- The 10 higher level capsules (of dimension 16) represent the 10 final "digit/class entities"
- This layer also has the "dynamic routing" in it.



## The loss function

Capsules use a separate margin loss  $L_c$  for each category  $c$  of digit capsules:

$$L_c = T_c \max(0, m^+ - \|v_c\|)^2 + \lambda(1 - T_c) \max(0, \|v_c\| - m^-)^2$$

$T_c=1$  if an object of class  $c$  is present.  $m_+=0.9$  and  $m_-=0.1$ ,  $\lambda=0.5$  down-weighting for absent digit classes For DigitCaps who do not match with the correct label,  $T_c$  will be zero and therefore the second term will be evaluated (corresponding to  $(1 - T_c)$  part).

Translated to english : if an object of class  $c$  is present, then  $\|v_c\|$  should be no less than 0,9. If not then  $\|v_c\|$  should be no more than 0,1

**Total loss is the sum of losses of all digit capsules**

$L_c$ loss term for one DigitCap $T_c$ calculated for correct DigitCap $m^+$ L2 norm $m^-$ L2 norm	$\max(0, m^+ - \ v_c\ )^2$ $+ \lambda(1 - T_c) \max(0, \ v_c\  - m^-)^2$ $1$ when correct $0$ when incorrect $0.5$ constant used for numerical stability $1$ when incorrect DigitCap, $0$ when correct	$\max(0, \ v_c\  - m^-)^2$ $0$ when correct $1$ when incorrect $0.5$ constant used for numerical stability $0$ when correct $1$ when incorrect $0$ when correct
---	--	---

Note: correct DigitCap is one that matches training label, for each training example there will be 1 correct and 9 incorrect DigitCaps

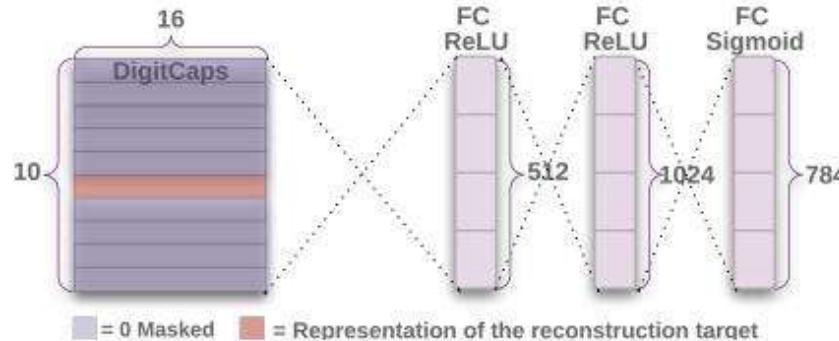


# Capsule network

Decoder structure to reconstruct a digit from the DigitCaps layer representation. The euclidean distance between the image and the output of the Sigmoid layer is minimized during training. We use the true label as reconstruction target during training.

Decoder takes a 16-dimensional vector from the correct DigitCap and learns to decode it into an image of a digit (note that it only uses the correct DigitCap vector during training and neglects the incorrect ones). Decoder is used as a regularizer, it takes the output of the correct DigitCap as input and learns to recreate an 28 by 28 pixels image, with the loss function being Euclidean distance between the reconstructed image and the input image. Decoder forces capsules to learn features that are useful for reconstructing the original image. The closer the reconstructed image is to the input image, the better.

The output of the last layer is 784 dimension (which after reshaping gives back a 28x28 decoded image, since in this example the images have that size).



Output: 784 (which after reshaping gives back a 28x28 decoded image).

We use an additional reconstruction loss to encourage the digit capsules to encode the instantiation parameters of the input digit. During training, we mask out all but the activity vector of the correct digit capsule. Then we use this activity vector to reconstruct the input image. The output of the digit capsule is fed into a decoder consisting of 3 fully connected layers that model the pixel intensities as described in Fig. ● We minimize the sum of squared differences between the outputs of the logistic units and the pixel intensities. We scale down this reconstruction loss by 0.0005 so that it does not dominate the margin loss during training.

survey on capsule networks 2022:

<https://www.sciencedirect.com/science/article/pii/S1319157819309322?via%3Dihub>

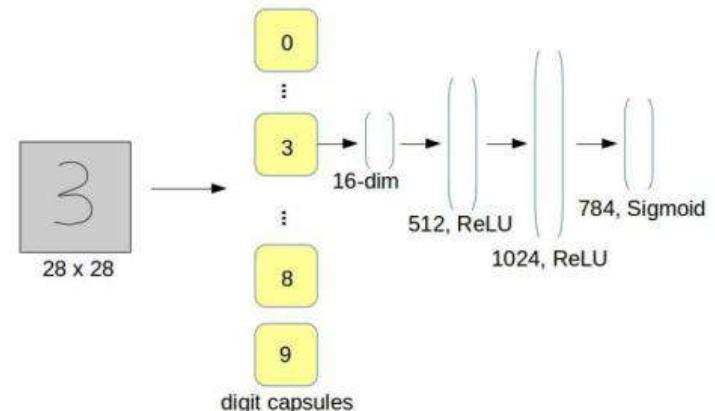
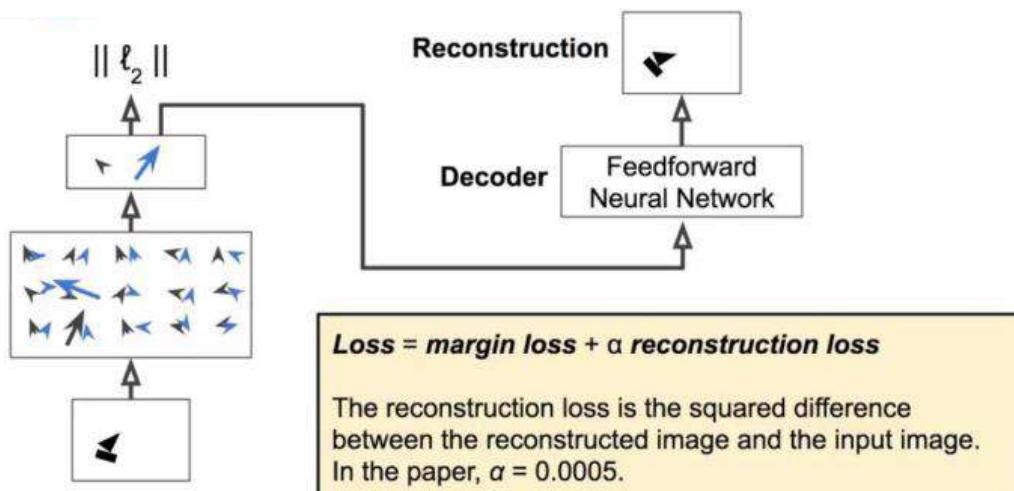
<https://arxiv.org/pdf/2206.02664.pdf>



# Reconstruction as regularization method

- Capsule Networks use a reconstruction loss as a regularization method to **encourage the digit capsules to encode the instantiation parameters** of the input digit
- In order to reconstruct the input from a lower dimensional space, the Encoder and Decoder needs to **learn a good matrix representation to relate the relationship between the latent space and the input**

During training, we mask out all but the activity vector of the correct digit capsule. Then we use this activity vector to reconstruct the input image.



# Capsule networks - conclusions

## Reconstruction as a regularization method

To summarize:

- using the reconstruction loss as a regularizer, the Capsule Network is able to learn a global linear manifold between a whole object and the pose of the object and its parts as a matrix of weights via **unsupervised learning**.
- the *translation invariance* is encapsulated in the matrix of weights, and not during neural activity, making the neural network *translation equivariance*.

A Capsule Network could be considered a "real-imitation" of the human brain. Unlike convolutional neural networks, which do not evaluate the spatial relationships in the given data, capsule networks consider the orientation of parts in an image as a key part of data analysis. They examine these hierarchical relationships to better identify images. The inverse-graphics mechanism which our brains make use of is imitated here to build a hierarchical representation of an image, and match it with what the network has learned. Though it isn't yet computationally efficient, there does seem to be an accuracy boost beneficial in tackling real-world scenarios. The Dynamic Routing of Capsules is what makes all of this possible. It employs an unusual strategy of updating the weights in a network, thus avoiding the pooling operation.



# other capsule based networks

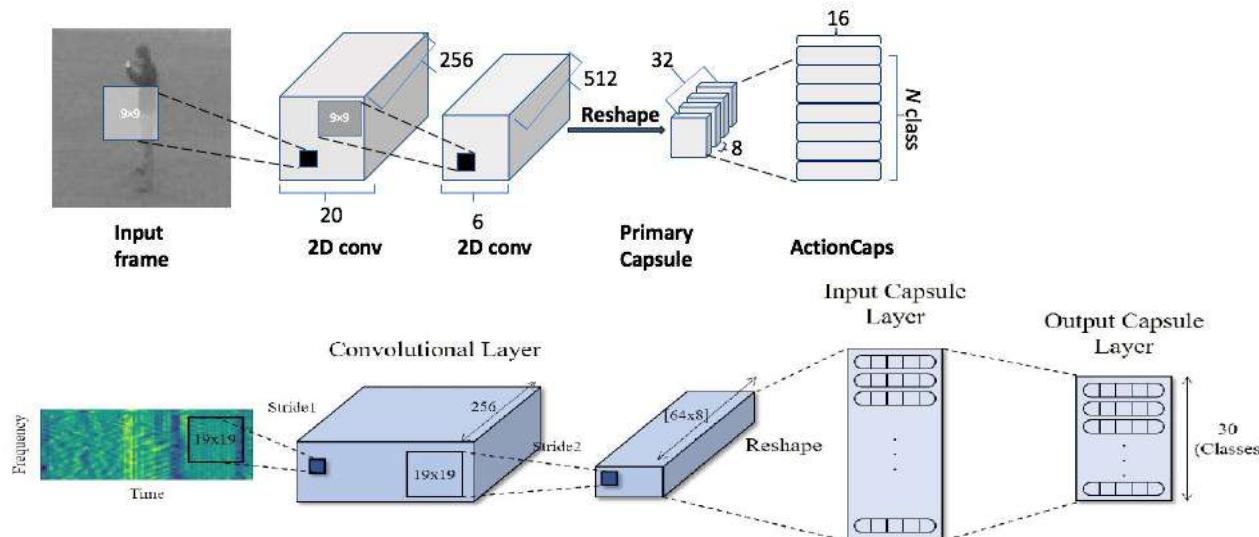
Video on capsule neural networks, with a focus on biomedical imaging applications.  
[https://www.youtube.com/watch?v=yY6sBVesVDw&ab\\_channel=RodneyLaLonde](https://www.youtube.com/watch?v=yY6sBVesVDw&ab_channel=RodneyLaLonde)

- Finally, several other capsule based networks have been proposed in the literature,

e.g. CNN-CapsNet, it can be divided into two parts:

CNN and CapsNet. First, an image is fed into a CNN model, and the initial feature

maps are extracted from its layer. Then, the initial feature maps are fed into CapsNet (the topology described in the previous pages).



- Transformer-based capsule routing algorithm have been already proposed <https://www.mdpi.com/1099-4300/24/5/678> see page 402 for details on transformer
- Ensemble CNN and capsule <https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0246988>
- Capsule for 3d point clouds <https://www.mdpi.com/2076-3417/11/4/1833>
- Siamese capsule networks [https://www.researchgate.net/publication/325262901\\_Siamese\\_Capsule\\_Networks](https://www.researchgate.net/publication/325262901_Siamese_Capsule_Networks)
- NLP <https://www.sentic.net/nlp-capsule.pdf>



# *Ensemble of networks*

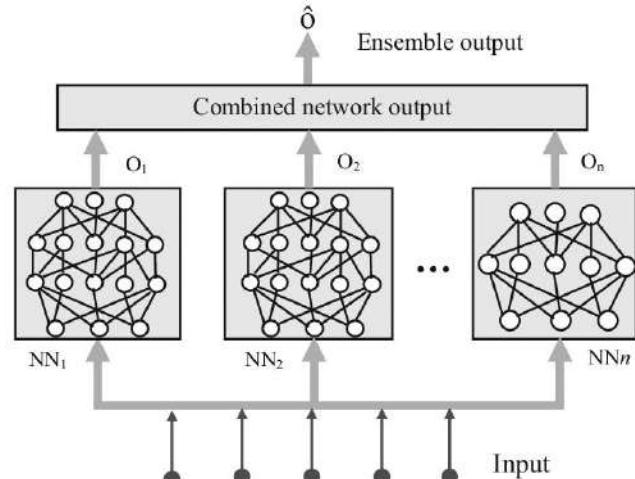
[http://www.scholarpedia.org/article/Ensemble\\_learning](http://www.scholarpedia.org/article/Ensemble_learning)

## Idea:

- Train several models separately for the same task
- At inference time: average results
- Thus, often also called “model averaging”

## Intuition:

- Different models make different errors on the test set
- By averaging we obtain a more robust estimate without a better model!
- Works best if models are maximally uncorrelated
- Winning entries of challenges are often ensembles (e.g., Netflix challenge), as empirically speaking it is very likely that using ensemble methods gives a 1-2% performance improvement in most tasks
- Drawback: requires evaluation of multiple models at inference time



A ensemble is an approach where several classifiers are used (usually in parallel, but sometimes also in cascade or in a hierarchical way) to perform the classification of patterns; the decisions of the individual classifiers are merged at a certain level of the classification process.

It has been demonstrated (theoretically but above all in practice) that the use of combinations of classifiers (combination of classifiers, classifier fusion, ensemble learning) can improve performance, even a lot.

Be pragmatic! In practice, investing time in the "push" optimization of a single classifier is generally less convenient than alongside (the initial classifier) other classifiers.



# Ensemble

Notice that the combination is effective only when the individual classifiers are (at least partially) independent from each other, i.e. they do not make the same mistakes.

Independence (or diversity) is normally achieved:

Using different features for training the classifiers;

Using different image preprocessing approaches;

Using different classification algorithms;

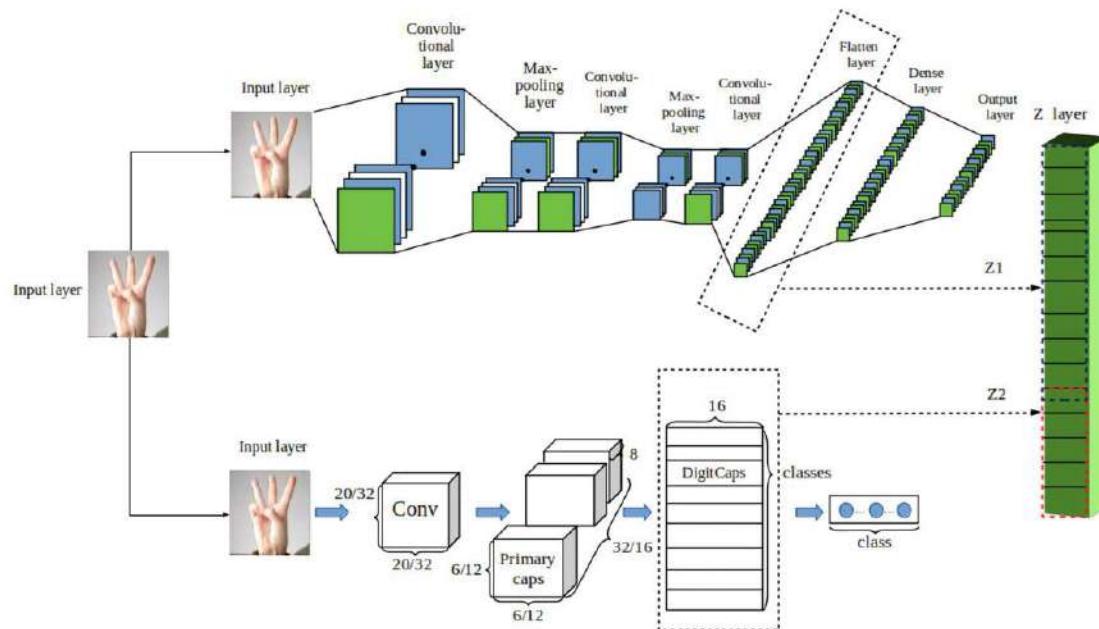
Using different loss functions;

Training the same classification algorithm on different portions of the training set (bagging);

Insisting on training on incorrectly classified patterns (boosting).

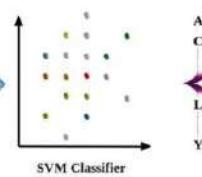
The combination can be done at the decision level or at the confidence level.

Further details: <https://towardsdatascience.com/ensemble-methods-bagging-boosting-and-stacking-c9214a10a205>

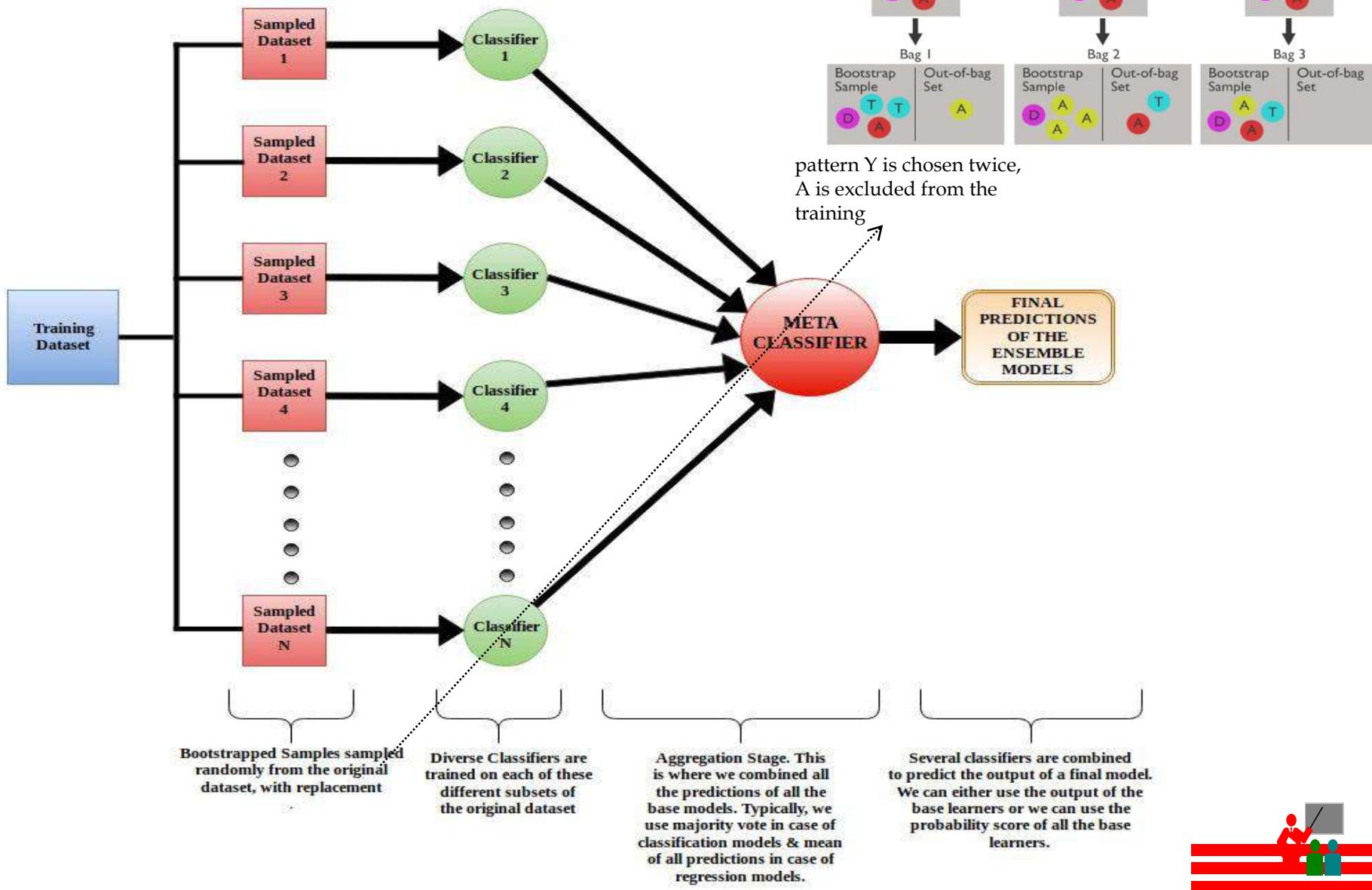


Ensemble of convolutional neural network and capsule network

[https://www.researchgate.net/publication/358252620\\_Improving\\_hand\\_gestures\\_recognition\\_capabilities\\_by\\_ensembling\\_convolutional\\_networks](https://www.researchgate.net/publication/358252620_Improving_hand_gestures_recognition_capabilities_by_ensembling_convolutional_networks)



# Bagging

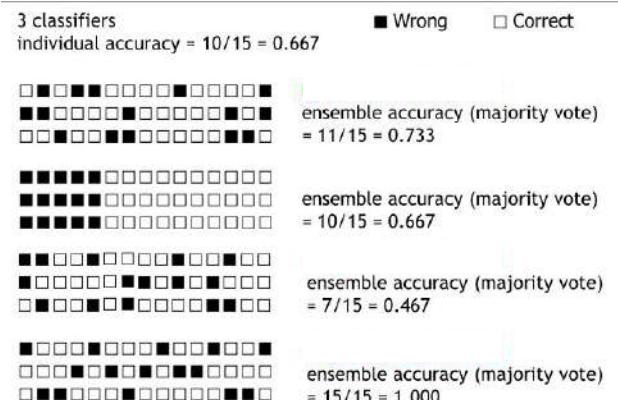


# Majority Voting

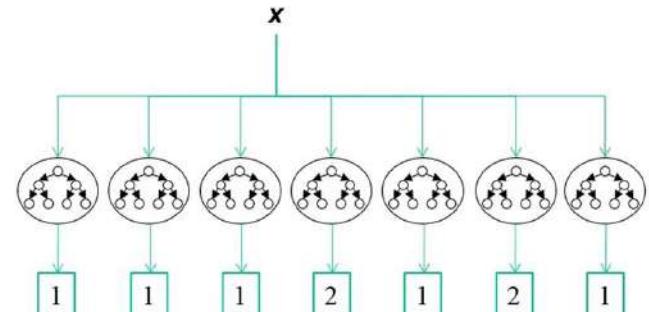
In the following discussion, we assume that only the class labels are available from the classifier outputs. Let us define the decision of the  $t^{\text{th}}$  classifier as  $d_{t,j} \in \{0, 1\}$ ,  $t = 1, \dots, T$  and  $j = 1, \dots, C$ , where  $T$  is the number of classifiers and  $C$  is the number of classes. If  $t^{\text{th}}$  classifier chooses class  $\omega_j$ , then  $d_{t,j} = 1$ , and 0, otherwise.

There are three versions of majority voting, where the ensemble choose the class (i) on which all classifiers agree (*unanimous voting*); (ii) predicted by at least one more than half the number of classifiers (*simple majority*); or (iii) that receives the highest number of votes, whether or not the sum of those votes exceeds 50% (*plurality voting* or just *majority voting*). The ensemble decision for the plurality voting can be described as follows: choose class  $\omega_J$ , if

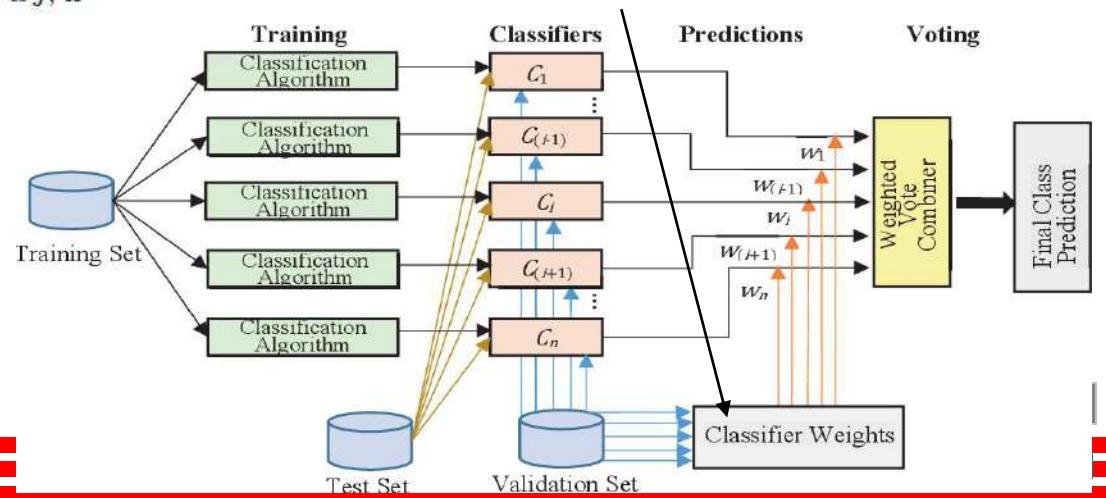
$$\sum_{t=1}^T d_{t,J} = \max_{j=1}^C \sum_{t=1}^T d_{t,j}.$$



An ensemble of seven networks. Every network provides its own classification for the instance  $x$ . Using majority voting instance  $x$  should be classified as "1" since five network out of seven vote for "1".



We can assign a weight to each classifier, these weights must be chosen using only training/validation data (e.g. a grid search)



# Q-statistic

Let  $\mathbf{Z} = \{\mathbf{z}_1, \dots, \mathbf{z}_N\}$  be a labelled data set,  $\mathbf{z}_j \in \mathcal{R}^n$  coming from the classification problem in question. For each classifier  $D_i$  we design an  $N$ -dimensional output vector  $\mathbf{y}_i = [y_{1,i}, \dots, y_{N,i}]^T$  of *correct classification*, such that  $y_{j,i} = 1$ , if  $D_i$  recognises correctly  $\mathbf{z}_j$ , and 0, otherwise. There are various statistics to assess the similarity of  $D_i$  and  $D_k$

Yule suggested that the  $Q$  statistic be used as a measure of association. The  $Q$  statistic for two classifiers is

$$Q_{i,k} = \frac{N^{11}N^{00} - N^{01}N^{10}}{N^{11}N^{00} + N^{01}N^{10}}$$

where  $N^{ab}$  is the number of elements  $\mathbf{z}_j$  of  $\mathbf{Z}$  for which  $y_{j,i} = a$  and  $y_{j,k} = b$  see Table

For statistically *independent* classifiers,  $Q_{i,k} = 0$ .  $Q$  varies between  $-1$  and  $1$ . The correlation between two binary classifier outputs (correct/wrong)  $\mathbf{y}_i$  and  $\mathbf{y}_k$  is

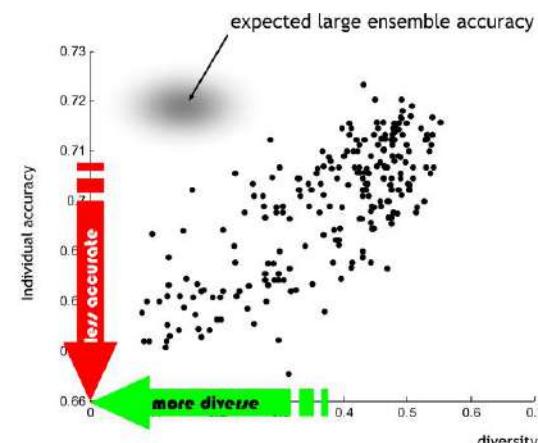
$$\rho_{i,k} = \frac{N^{11}N^{00} - N^{01}N^{10}}{\sqrt{(N^{11} + N^{10})(N^{01} + N^{00})(N^{11} + N^{01})(N^{10} + N^{00})}}$$

For any two classifiers,  $Q$  and  $\rho$  have the same sign, and it can be proved that  $|\rho| \leq |Q|$ . We chose  $Q$  to measure the dependency because it has been designed for  $2 \times 2$  contingency tables. It is also simpler to calculate from the table entries.

Table A  $2 \times 2$  table of the relationship between a pair of classifiers

	$D_k$ correct (1)	$D_k$ wrong (0)
$D_i$ correct (1)	$N^{11}$	$N^{10}$
$D_i$ wrong (0)	$N^{01}$	$N^{00}$

Total,  $N = N^{00} + N^{01} + N^{10} + N^{11}$ .



$N^{11}$  indicates the number of patterns that both classifiers correctly classify,  $N^{00}$  indicates the number of patterns that both classifiers wrongly classify. A value of 0 means that the classifiers are uncorrelated. I mean the two classifiers are independent of each other

# Combining continuous outputs

$d_{tj}(x)$  = support given by the  $t$ th classifier to the  $j$ th class for the instance  $x$

$w_j$  = weight of the  $j$ th classifier

$T$  = total number of classifiers

$\mu_j(x)$  = total support for the  $j$ th class for instance  $x$

- (A) *Sum rule* The total support for a class is calculated as the sum of the supports given to that class by all the classifiers. After calculating the total supports for all the classes, the class with the highest support is selected as the final output.

Let's see an example, we have 2 classifiers and two classes, the support of the first classifier is 0.2 (first class) 0.6 (second class), the second classifier has a support of 0.9 0.1, sum rule  $0.2 + 0.9 \cdot 0.1 + 0.6$ , the pattern is classified as class '1'.

- (B) *Mean rule* This rule is similar to the sum rule but the total support is normalized by  $1/T$  ( $T$  = number of classifiers).

$$\mu_j(x) = \frac{1}{T} \sum_{t=1}^T d_{tj}(x)$$

- (C) *Weighted sum rule* By this rule, the support for a class is calculated by the sum of the product of the classifiers' weight and their respective supports.

$$\mu_j(x) = \sum_{t=1}^T w_t d_{tj}(x)$$

classifier outputs are often normalized (e.g. to the  $[0, 1]$  interval), and these values are interpreted as the support given by the classifier to each class, or as class-conditional posterior probabilities. E.g. if we combine two classifiers where the first has output  $[0,1]$  and the second  $[-1000,1000]$  obviously we have to normalize them before the fusion, one of the most used method is to normalize the outputs  $X$  to mean 0 and variance 1

Standard score	$\frac{X - \mu}{\sigma}$
----------------	--------------------------

- (D) *Product rule* Here, supports provided by the classifiers to a particular class are multiplied to obtain the final support for that class. This rule is very sensitive to the pessimistic classifiers as a low support can remove any chance of getting selected by that class.

$$\mu_j(x) = \prod_{t=1}^T d_{tj}(x)$$

The product is the most "probabilistically" correct method (the joint probability is calculated as a probability product) but only in the case of statistical independence. the sum is often preferable to the product as it is more robust. In the product, only one classifier indicating zero confidence for a class is sufficient to bring the confidence to zero for that class

- (E) *Maximum rule* As the name suggests, this rule selects the maximum of all the supports of the different classifiers for a particular class.

$$\mu_j(x) = \max_{t=1}^T \{d_{tj}(x)\}$$

- (F) *Minimum rule* This rule selects the minimum of all the supports of the different classifiers for a particular class.

$$\mu_j(x) = \min_{t=1}^T \{d_{tj}(x)\}$$

we choose the maximum of the minimums, therefore a class that has not received too low confidence from any classifier

- (G) *Median rule* This rule selects the median of the supports of the different classifiers for a particular class.

$$\mu_j(x) = \text{median}_{t=1}^T \{d_{tj}(x)\}$$

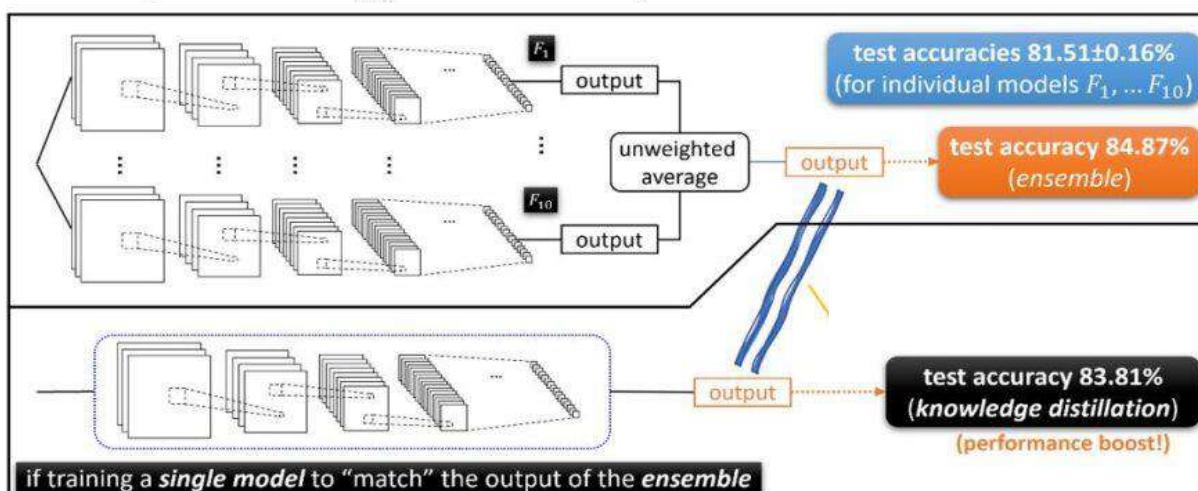
# Knowledge Distillation

In deep learning competitions like Kaggle, ensembles are super famous. Basically, an ensemble (aka teacher) is when we average multiple trained model outputs for prediction. This simple technique is great for improving test-time performance. However, it becomes  $N$  times slower during inference, where  $N$  indicates the number of trained models. This is an issue when we deploy such neural networks in embedded devices. To address it, an established technique is [knowledge distillation](#).

*Knowledge distillation* simply trains a new randomly initialized model to match the output of the ensemble (an  $N$  times bigger set of models).

for further details:  
<https://neptune.ai/blog/knowledge-distillation>

The main drawback of an ensemble is that it increases the computational power needed to classify a pattern as we are using multiple approaches. Let's see an example, while the ensemble is great for improving performance during testing, it gets 10 times slower during the inference time (that is, the test time): we need to calculate the outputs of 10 neural networks instead of one. This is a problem when we implement such models in a low energy mobile environment. To solve it, a fundamental technique called distillation of knowledge has been proposed. For example, in a straightforward distillation approach, the distillation of knowledge simply trains another individual model to match the output of the whole.





# Data Augmentation

2023 survey: <https://arxiv.org/abs/2301.02830>

Now let's see some data augmentation methods.

Data augmentation is a technique used to increase the amount of data by adding slightly modified copies of already existing training pattern or newly created synthetic data from existing patterns. It helps to reduce overfitting when training a machine learning model.

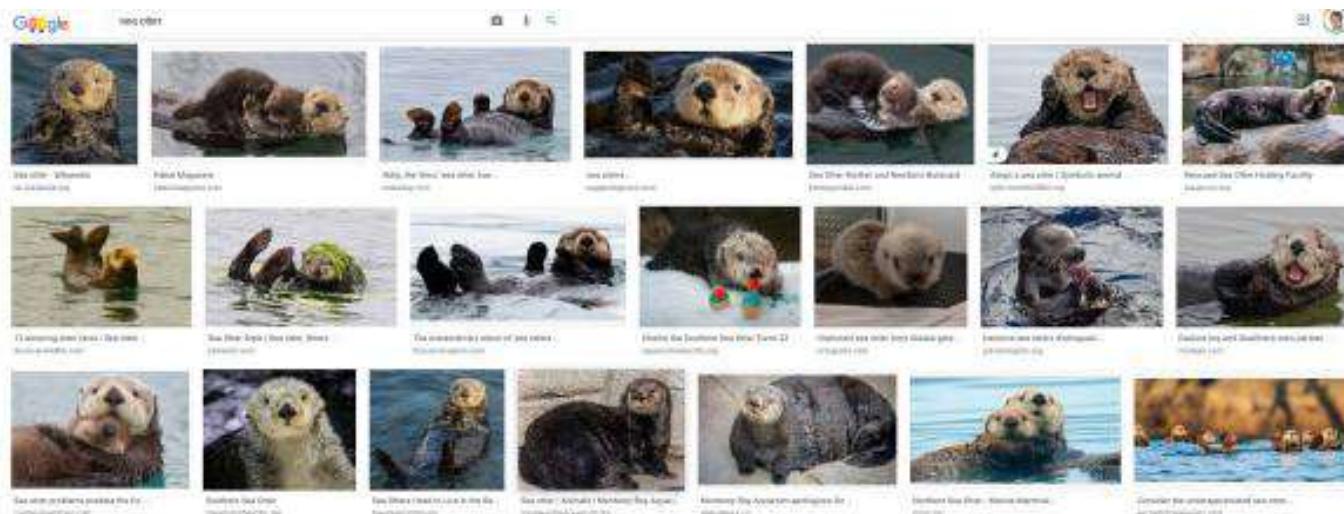


# Data Augmentation

in the real world there is a very wide variation between images of the same class, as you can see from these otter images in the wild. To improve the classification performance we should train the network using images representing different variants of the given class, these training patterns are not always available, for this reason it is important to increase the size of the training set with artificially created images.

## Motivation:

- Deep neural networks must be **invariant** to a wide variety of input variations. We want our model to be invariant (not affected by) to input variations, as we want it to classify all inputs irrespective of their variations correctly. For e.g. we would want our model to correctly classify both an otter lying on its back and an otter sitting on a rock as an otter, despite the variations in these 2 images.



Variations of images from a single class

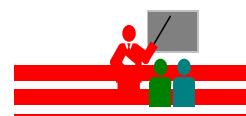
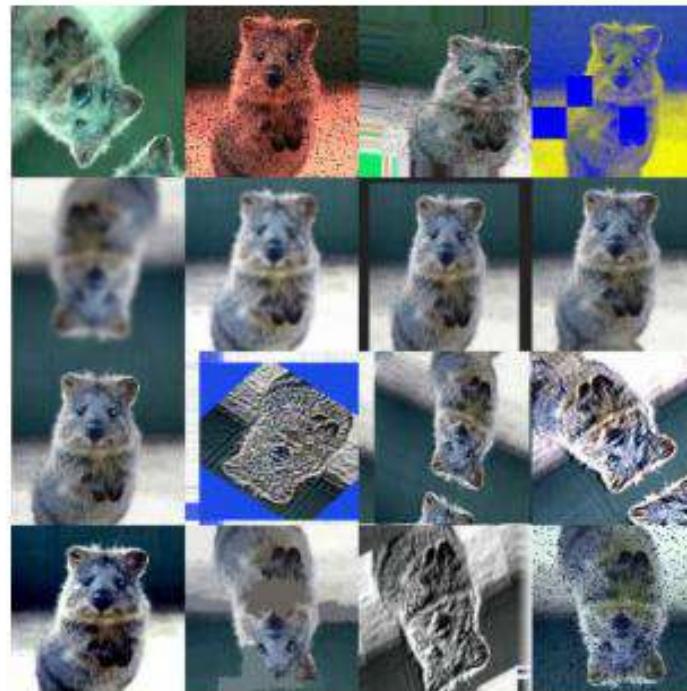
- Often in real-life data there exist **large intra-class variations** in terms of pose, appearance, lighting etc. This makes image classification in particular an extremely hard task.

# Data augmentation

How do we tackle intra-class variations and improve generalization ?

- Best way towards better generalization is to **train on more data**. However, data in practice often limited.
- Goal of data augmentation: create “**fake**” **data** from the existing data (on the fly) and add it to the training set. This process is “on the fly” as we do not store the augmented images. The augmented images are generated randomly (by applying random transformations) from each image in our batch at the beginning of every training iteration.
- New data must **preserve semantics** i.e. the augmented data of a class should not change the semantic meaning of the class category. For e.g. augmenting the images of a class of dogs to look like cats (with label as dogs) is not a good idea.
- Even **simple operations** like translation or adding per-pixel noise often already greatly improve generalization
- <https://github.com/aleju/imgaug> is a popular library on GitHub where implementations of different image augmentation techniques can be found.

The best way to address intraclass variation and reduce the risk of overfitting is to increase the size of the training set by creating fake images, even with simple geometric transformations such as rotation and translation.



# Data augmentation - geometric transformation

## Image Cropping:

- Randomly crop and re-scale images to original image size
- Do not crop regions from the image which are too small, as then semantics of the image category could be lost

## Affine Transformation:

- Apply a single affine/ linear transformation to the 2D image space i.e. scale, rotate, shear an image etc.
- Blank spaces left after the transformation can be filled by using different strategies like using constant color, expand image colors, replicate colors, etc.



## Perspective Transformation:

- Similar to Affine transformation albeit a perspective transform has more degrees of freedom
- The perspective effect causes certain regions of the image to be squeezed and other regions to be enlarged



## Piecewise Affine Transformation:

- Similar to Affine transformation albeit now a single transformation is not applied to the whole image
- The image is perceived as an underlying grid. A piece-wise affine transformation is applied independently to each grid cell. This causes different regions in the image to be distorted differently



# Data augmentation - local filters

## Gaussian Blur:

- Applies the gaussian blur filter onto the given image with the  $\sigma$  of the gaussian distribution being chosen randomly from an interval specified by the user
- Helps the model to recognize objects at different resolutions
- Also the process of capturing data using a camera induces a small amount of blur. Thus applying gaussian blur also helps the model to be robust to such camera-induced variations in captured images



## Image Sharpening:

- Does the opposite of introducing blur i.e. introduces sharpness into input images

## Edge Detection:

- Uses the edge detected version of a given image
- Is important to verify whether the edge detected version are still reasonable and do not change / make it impossible to decipher semantics of the class



# Data augmentation - adding noise

A popular data augmentation technique which typically involves introducing a more *structured* random per-pixel noise.

## Why is studying noise important ?

- Deep Networks are highly sensitive to noise in images. Thus, it is very likely that adding even a small amount of noise (just to the test data) would lead to drastic reduction in performance, even though to the human eye the "noisy" test dataset would look almost similar to the original test dataset

Following are some of the most popular additive-noise augmentation techniques to improve generalization performance:

### Gaussian Noise:

- Add a fixed Gaussian noise to each pixel in an image. The additive noise is chosen randomly for different images



### Salt and Pepper Noise:

- Each pixel in an image is turned into either black or white depending on a specified probability



## Why are Color Transformations important ?

- Color transformations have played a key-role in the success of most neural networks that work well on the Imagenet dataset
- Cameras produce different color spectrum's based on the type of sensors used and the kind of white-balancing performed. A more important reason behind this is the change in lighting conditions while capturing data, for e.g. the colors captured by a camera would change dramatically for the same scene depending on the time of the day i.e. sunset would have warmer colors, mid-day would have brighter colors, etc. We thus want our models to be invariant to such variations so that our generalization performance increases. This is exactly why color transformations are an essential form of data augmentation

Following are some of the most popular color transformations techniques to improve generalization performance:

### Contrast:

- Change the contrast of images to either become faint or have stronger contrast.



### Brightness:

- Change the brightness of the whole images to make them either darker or brighter



### Brightness per Channel:

- Instead of changing brightness of the whole image, change brightness for each channel instead



# Data augmentation - color transformation

It is almost impossible for the training set to have all the possible variations of light that can occur in the images. Digital image data is usually encoded as a tensor of the dimension (height×width×color channels). Performing augmentations in the color channels space is another strategy that is very practical to implement. The RGB values can be easily manipulated with simple matrix operations to increase or decrease the brightness of the image. More advanced color augmentations come from deriving a color histogram describing the image. Changing the intensity values in these histograms results in lighting alterations such as what is used in photo editing applications.



## Weathers

Even though weather effects might sound complex they are still very easy to compute and can be done on-the-fly without requiring any sophisticated graphics engine.

### Snow:

- Introduces a snow-like effect on the images



# *Data augmentation - weathers*

### Clouds:

- Introduces an effect on the images which look like clouds



### Fogs:

- Introduces a fog effect on the images



# Data augmentation

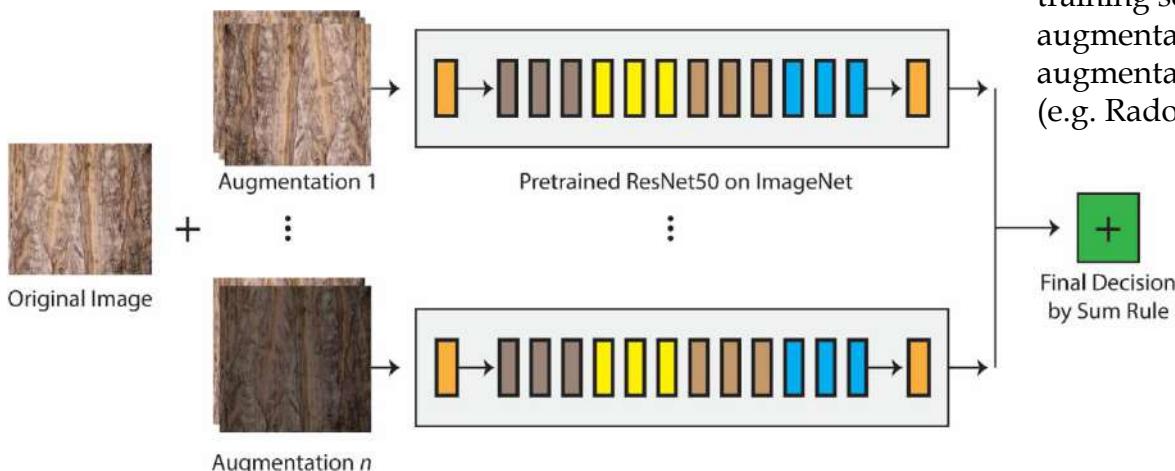
## Random Combinations

In practice, all of the aforementioned augmentation techniques are combined and then applied randomly on-the-fly to different images drawn randomly from within a mini-batch.

for instance, we can rotate an image and then apply the fog filter.

## Important Remarks about Data Augmentation

- When comparing two networks, make sure you **use the same augmentation**. This is because data augmentation is an extremely powerful strategy and could significantly improve performance. If the same augmentation strategy is not used in both networks, one could be misled into thinking that their idea was responsible for better performance, whereas in reality it was just a better data augmentation strategy
- Consider data augmentation as a **part of your network design**
- It is important to specify the right distributions (often done empirically). Not all transformations would be applicable or beneficial to improving performance in a given task. Thus the kind of transformations that would help in improving performance and their corresponding strength (hyperparameters) need to be chosen empirically
- Can also be **combined with ensemble idea**:



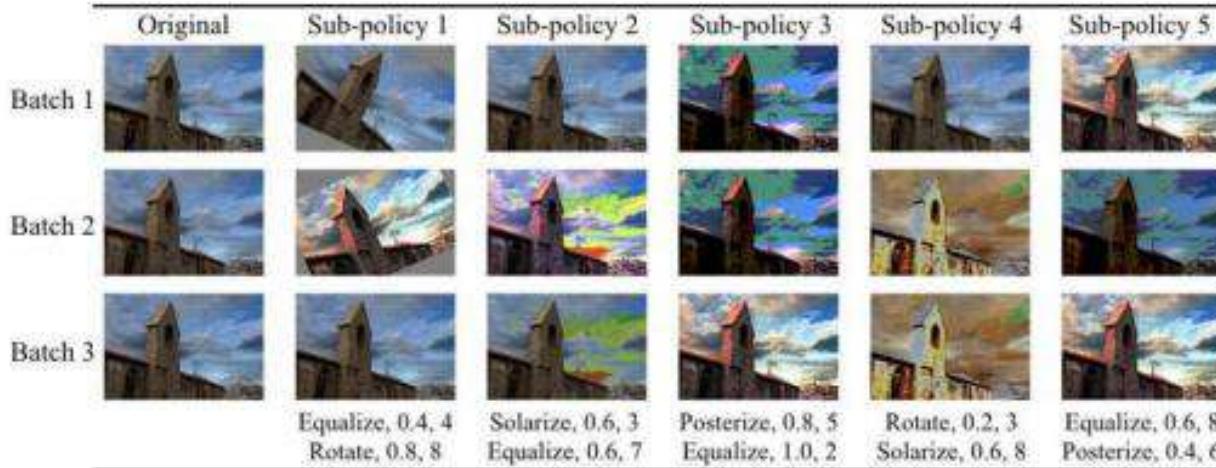
<https://ieeexplore.ieee.org/document/10025727> -> each network is trained using a different training set, each training set is obtained by applying a different set of data augmentation methods. In that work some data augmentation approaches based on features transform (e.g. Radon Transform, Fourier Transform...) are proposed.



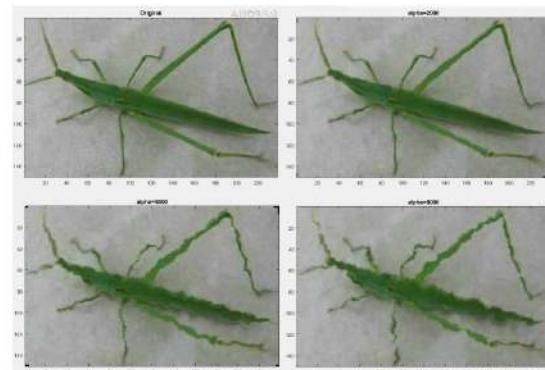
# AutoAugment / Elastic deformation

- AutoAugment uses reinforcement learning to find strategies automatically:

<https://arxiv.org/abs/1805.09501>



Elastic deformation transforms a given image by applying a randomly generated displacement field to its pixels by a value extracted from the standard uniform distribution in the range [-1,1]. The resulting horizontal and vertical displacement fields are passed through a low-pass filter to smooth the image.



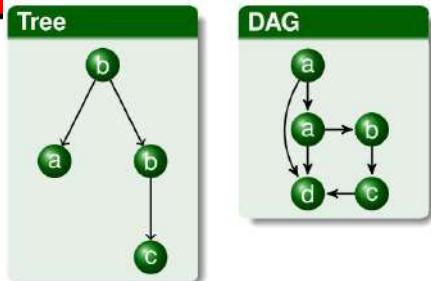


# *Graph Convolutional Networks (GCN)*

<https://blogs.nvidia.com/blog/2022/10/24/what-are-graph-neural-networks/>



# Graph Convolutional Networks



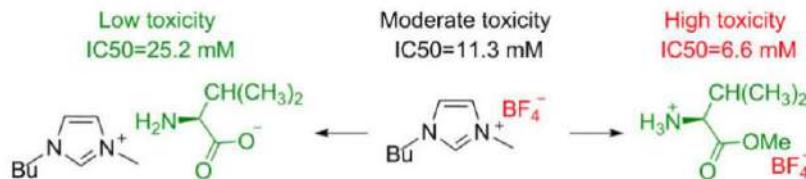
GCNs were first introduced in "Spectral Networks and Deep Locally Connected Networks on Graphs" (Bruna et al, 2014), as a method for applying neural networks to graph-structured data.

Using a process called message passing, GNNs organize graphs so machine learning algorithms can use them.

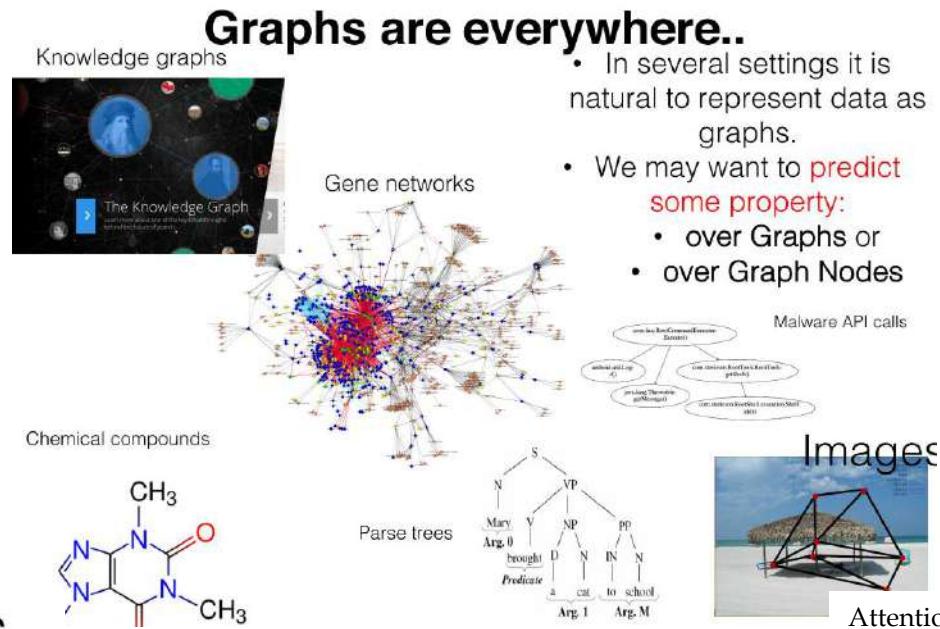
The graphs can take any shape or size and contain any type of data, including images and text.

Message passing embeds into each node information about its neighbors. AI models employ the embedded information to find patterns and make predictions.

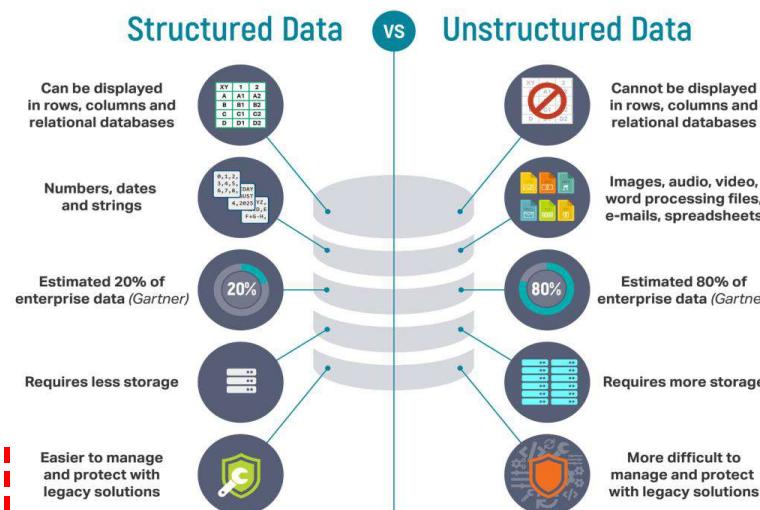
## Classification/Regression on Graphs



- Dataset composed of N pairs  $\{(G_i, y_i), 1 \leq i \leq N\}$ 
  - Each graph:
    - $n_i$  vertices
    - (possibly) discrete label associated to each node:  $l(v)$
    - $d$  vectorial **attributes** associated to each node:  $a(v)$  or  $X \in \mathbb{R}^{n_i \times d}$
- Given an unseen graph  $G$ , the task is to predict the correct target

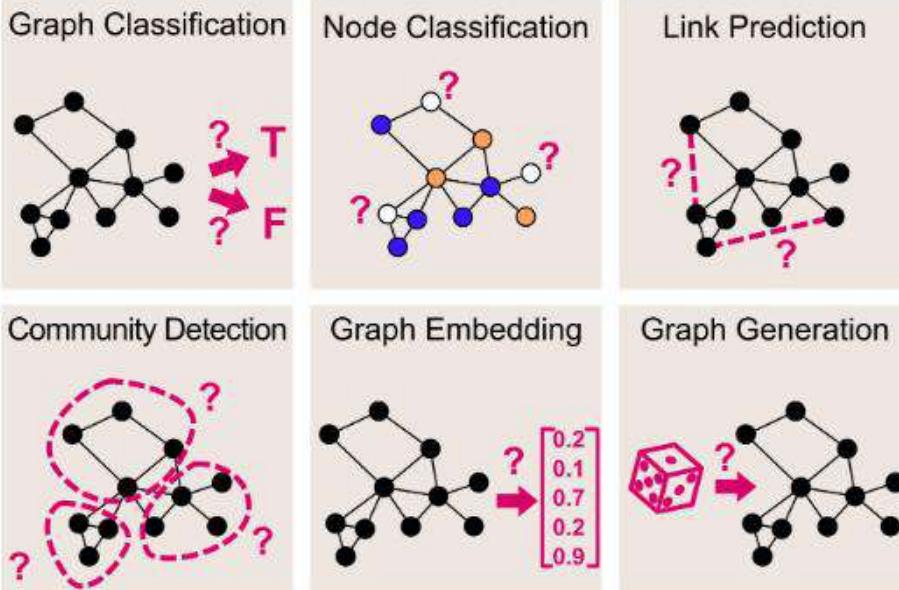


Attention, we say that images or words are structured data only when such objects are described by means of a vector, matrix or tensor; they are generally not considered structured data.



# Graph Convolutional Networks

- Graph Classification — given a graph, predict to which of a set of classes it belongs
- Node Classification — given a graph with incomplete node labelling, predict the class of the remaining nodes
- Link Prediction — given a graph with incomplete adjacency matrix, predict for each pair of nodes whether they are connected
- Community Detection (a.k.a. Graph Clustering) — given a graph, partition its nodes into clusters based on its edge structure
- Graph Embedding — given a graph, map it into a vector while preserving relevant information
- Graph Generation — learn a distribution a set of given graphs, and sample from this distribution to generate new similar graphs



GNNs in computer vision

Using regular CNNs, machines can distinguish and identify objects in images and videos. Although there is still much development needed for machines to have the visual intuition of a human. Yet, GNN architectures can be applied to image classification problems.

One of these problems is scene graph generation, in which the model aims to parse an image into a semantic graph that consists of objects and their semantic relationships. Given an image, scene graph generation models detect and recognize objects and predict semantic relationships between pairs of objects.

However, the number of applications of GNNs in computer vision is still growing. It includes human-object interaction, few-shot image classification, and more.

Graphs are used also for data art, e.g.

<https://www.instagram.com/p/Bj9igibAReb>

<https://medium.com/cuevagallery/data-have-a-soul-in-the-art-of-kirell-benzi-fa809433dcea>

GNNs in Natural Language Processing

In NLP, we know that the text is a type of sequential data which can be described by an RNN or an LSTM. However, graphs are heavily used in various NLP tasks, due to their naturalness and ease of representation.

Recently, there has been a surge of interest in applying GNNs for a large number of NLP problems like text classification, exploiting semantics in machine translation, user geolocation, relation extraction, or question answering.

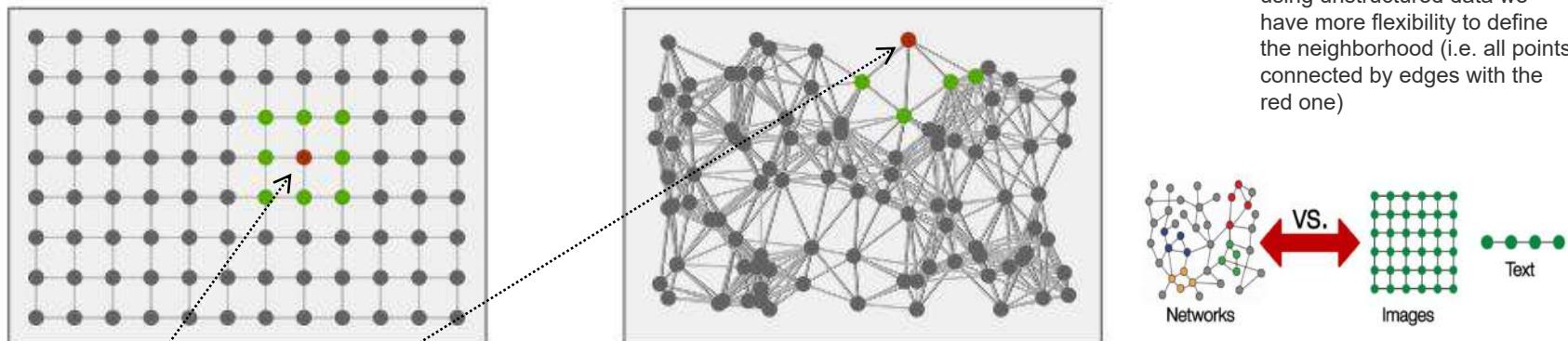
We know that every node is an entity and edges describe relations between them. In NLP research, the problem of question answering is not recent. But it was limited by the existing database. Although, with techniques like GraphSage (<https://arxiv.org/abs/1706.02216>), the methods can be generalized to previously unseen nodes.



# Graph Convolutional Networks

Multi layer perceptrons (MLPs) are very flexible function approximators. Theoretically a MLP with only one layer can already act as a universal function approximator, given that this layer can grow infinitely wide. However, MLPs do not scale well. If the input to an MLP is large the amount of model parameters grows large as well. Furthermore, the more parameters a neural network has, the more it tends to overfit: Its generalization capability decreases. For structured signal grids like 2D images or 1D time series a CNN addresses this issue and allows good generalization due to its convolution operation over the regular grid of signals, which reduces the number of signals. Unfortunately a lot of signals cannot be described in such a structured way such as molecules or natural language. These signals may be better described with the help of graphs. In order to exploit graph structured data, a model class is necessary that scales better than an MLP when receiving large input data without loosing its predictive power. Further on this model has to be more flexible than a CNN in order to exploit the local connectivity structure of any graph - not only structured ones - as prior information.

CNN has mainly focused on structured data; graphs, on the other hand, can handle unstructured data. Unstructured data can take any shape or size and contain any type of data, including images and text. See the following figures, structured data on the left, unstructured data on the right. In both cases a convolution is applied to calculate the red point, using structured data we have more constraints to define the neighborhood of a point, using unstructured data we have more flexibility to define the neighborhood (i.e. all points connected by edges with the red one)



**Comparison of graph structures.** An illustration of a regular structured graph (left) and an unstructured graph (right). The red dots marks the point of interest that is calculated by exploiting the graph structure and the neighboring nodes (green).

Graphs are descriptors of a signal structure, where the signals are described as nodes (points) and the similarity between signals with edges (lines between signals). Fig. shows a comparison between a structured graph (left) and an unstructured graph (right). On both graphs a convolution operation is applied in order to calculate the red dot by exploiting the graph structure and the neighboring nodes (green). On the structured graph on the left a convolution filter is applied, that computes a value for the red dot by calculating the dot product between the elements in green and red. The equivalent operation is done on the right by using a convolution that exploits the locality in the graph.



# Graph Convolutional Networks

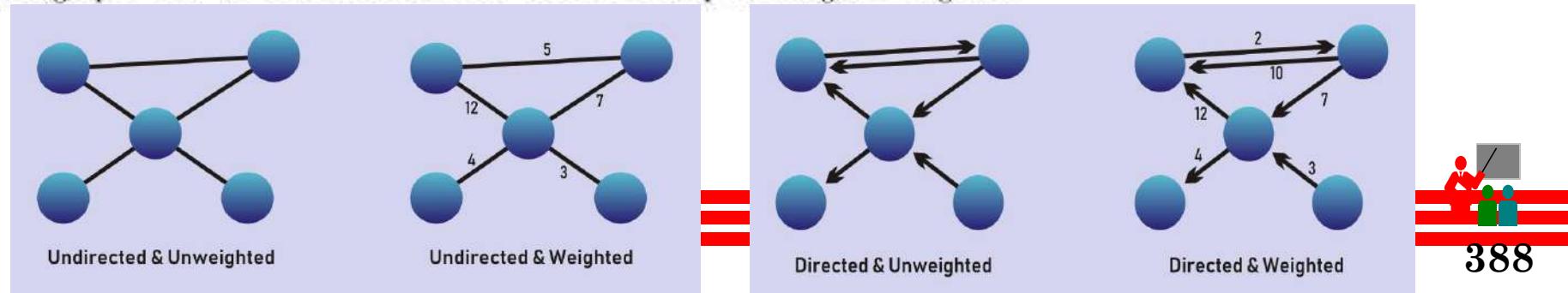
GCNs operate on graphs. Therefore some basic knowledge of graphs and operations on graphs is presented in this section.

A graph can be represented as a triplet  $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{W})$  with vertices  $\mathcal{V}$ , edges  $\mathcal{E}$  and weights  $\mathcal{W}$ . Vertices or nodes are defined as a set of  $N$  numerical labels  $\mathcal{V} = \{1, \dots, N\}$ . Edges are ordered pairs of these numerical labels  $(i, j)$ , where  $(i, j) \in \mathcal{E}$  is interpreted as "node  $i$  is influenced by node  $j$ ". Weights  $w_{ij} \in \mathbb{R}$  are numbers associated to edges  $(i, j)$  that determine the strength of the influence that node  $j$  has on node  $i$ . Depending on these basic properties, a graph can be classified as a directed graph or as a symmetric graph.

In the case of a **directed graph** the edge  $(i, j)$  differs from edge  $(j, i)$ . Thus, a connection between two nodes can be one-way only, meaning that  $(i, j) \in \mathcal{E}$  and  $(j, i) \notin \mathcal{E}$ . Furthermore, if a connection between two nodes is bidirectional  $\{(i, j), (j, i)\} \subseteq \mathcal{E}$ , their weights can be different  $w_{ij} \neq w_{ji}$ .

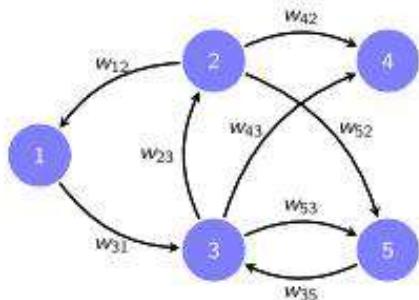
The **undirected or symmetric graph** is a directed graph, with the special property that its edge set and its weights are symmetric. Therefore, if the edge set  $\mathcal{E}$  contains  $(i, j)$  it implies that  $(j, i) \in \mathcal{E}$  as well.

A special case in directed and symmetric graphs is the **unweighted graph**, where all existing connections  $(i, j) \in \mathcal{E}$  have weights  $w_{ij} = 1$ . Since the weights of a graph convey valuable information as a prior for GCNs, most graphs that are encountered in the context of deep learning are weighted.

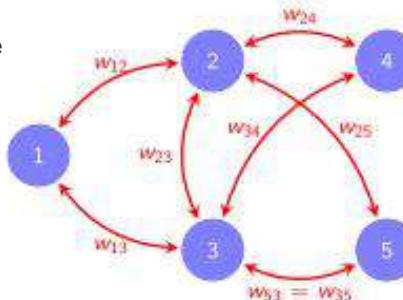


# Graph Convolutional Networks

In the directed graph the edges are represented as an arrow (from a starting node to an ending node), the weights ' $w_{ij}$ ' and ' $w_{ji}$ ' could be different. Instead, in the symmetric graph ' $w_{ij}$ ' and ' $w_{ji}$ ' are the same.



**Directed graph.** Nodes are illustrated by numerical labelled purple dots, edges are denoted by arrow lines and weights with a description in the form  $w_{ij}$ .



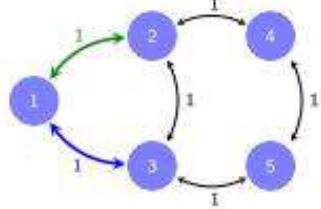
**Symmetric graph.** The illustration is similar to Due to the symmetry of the graph the edges are directed in both directions and there is exactly one weight per edge.

**Graph Matrix Representation.** The visual graph with its connections can be mathematically represented with the help of different matrices. A common representation of the edges of a graph  $\mathcal{G}$  is the **adjacency matrix  $\mathbf{A}$** . In this sparse  $N \times N$  matrix, each row and each column represent a vertex and each entry  $A_{ij}$  contains the weight  $w_{ij}$  of all connections  $i, j \in \mathcal{E}$ . Furthermore if the graph is symmetric its holds that  $\mathbf{A} = \mathbf{A}^T$

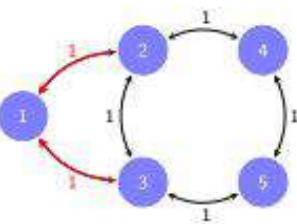
We don't allow multi-edges between two nodes

The elements of the adjacency matrix indicate whether the pairs of vertices are adjacent or not in the graph.

If the graph is undirected (i.e. all its edges are bidirectional), the adjacency matrix is symmetric.



$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$



degree matrix, see next page

$$\mathbf{D} = \begin{pmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 2 \end{pmatrix}$$

**Unweighted graph and corresponding matrices.** The unweighted graph is transformed into respective matrices  $\mathbf{A}$  and  $\mathbf{D}$ . Colors mark the corresponding location of the edges in the matrices

the degree matrix of an undirected graph is a diagonal matrix that contains information about the degree of each vertex, that is, the number of edges attached to each vertex. In a directed graph, the term degree may refer to indegree (the number of incoming edges at each vertex) or outdegree (the number of outgoing edges at each vertex)

# Laplacian Matrix

Another representation is the **degree matrix**  $\mathbf{D}$  that contains the degree of a vertex on its diagonal axis

The **degree**  $d_i$  of node  $i$  is the sum of weights of its incident edges:

in this formula the notation  $w_{ij}$   
used in the previous page is  
used, see top-left graph

$$d_i = \sum_{j \in \mathcal{N}(i)} w_{ij}, \quad \text{it is the formula for directed weighted graph}$$

where  $\mathcal{N}(i)$  - the **neighborhood** - is the set of nodes that influence node  $i$ :

$$\mathcal{N}(i) = \{j | (i, j) \in \mathcal{E}\}.$$

The diagonal  $D_{ii}$  therefore contains the degree  $d$  of vertex  $i$ :  $D_{ii} = d_i$ .

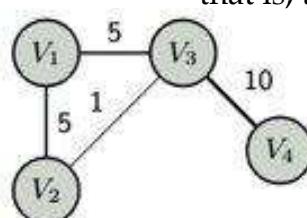
The **Laplacian matrix**  $\mathbf{L}$  combines matrices  $\mathbf{D}$  and  $\mathbf{A}$  as follows:

$$\mathbf{L} = \mathbf{D} - \mathbf{A}.$$

It therefore contains  $\mathbf{D}$  on its diagonal and  $\mathbf{A}$  negated on its off-diagonals. Written explicitly in terms of graph weights the entries consist of  $L_{ij} = -A_{ij} = -w_{ij}$  and  $L_{ii} = d_i = \sum_{j \in \mathcal{N}(i)} w_{ij}$ .

In an adjacency matrix, the interpretation of the entry  $w_{ij}$  can vary depending on the convention used in different sources or contexts. Both interpretations are valid, but they represent different conventions and can lead to different interpretations of the matrix. In one convention, the entry  $w_{ij}$  represents the weight of the edge connecting vertex  $i$  to vertex  $j$ .

A different notation, where the entry  $w_{ij}$  represents the weight of the edge connecting vertex  $j$  to vertex  $i$ , is sometimes referred to as the "transpose adjacency matrix" notation. See the difference definition of  $w_{ij}$  of the above figure and the top-left figure of the previous page.

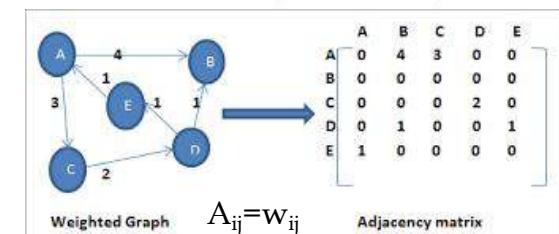


$$D = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$A = \begin{pmatrix} V_1 & V_2 & V_3 & V_4 \\ V_1 & 0 & 5 & 0 \\ V_2 & 5 & 0 & 1 & 0 \\ V_3 & 5 & 1 & 0 & 10 \\ V_4 & 0 & 0 & 10 & 0 \end{pmatrix}$$

$$L = \begin{pmatrix} 2 & -5 & -5 & 0 \\ -5 & 2 & -1 & 0 \\ -5 & -1 & 3 & -10 \\ 0 & 0 & -10 & 1 \end{pmatrix}$$

undirected,  
weighted graph



# Laplacian Matrix

► The Laplacian matrix of a graph with adjacency matrix  $\mathbf{A}$  is  $\Rightarrow \mathbf{L} = \mathbf{D} - \mathbf{A}$

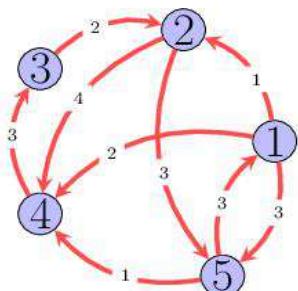
► Can also be written explicitly in terms of graph weights  $A_{ij} = w_{ij}$

$\Rightarrow$  Off diagonal entries  $\Rightarrow L_{ij} = -A_{ij} = -w_{ij}$

$\Rightarrow$  Diagonal entries  $\Rightarrow L_{ii} = d_i = \sum_{j \in n(i)} w_{ij}$

This hold if there are no loops in the graph

$$\mathbf{L} = \begin{bmatrix} 2 & -1 & -1 & 0 & 0 \\ -1 & 3 & -1 & -1 & 0 \\ -1 & -1 & 3 & 0 & -1 \\ 0 & -1 & 0 & 2 & -1 \\ 0 & 0 & -1 & -1 & 2 \end{bmatrix}$$

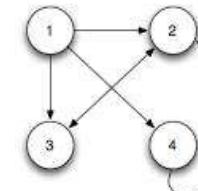


the sum of the weights of the incident edges to '4' is  $4+2+1=7$

► An example directed graph.

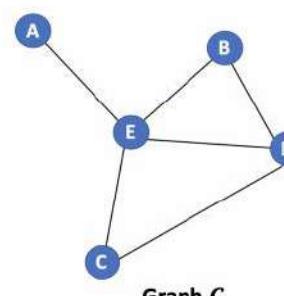
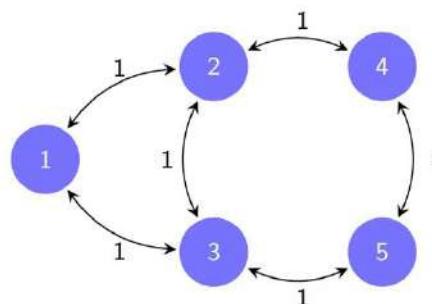
$$D_{\text{in}} = \begin{bmatrix} 3 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 7 & 0 \\ 0 & 0 & 0 & 0 & 6 \end{bmatrix}$$

In-degree matrix.



1	2	3	4	5
0	1	1	1	0
0	0	1	0	0
0	1	0	0	0
0	0	0	1	0
0	1	0	0	0

there is no a circular edge (loop) on '1',  $w_{11}=0$



A	B	C	D	E
0	0	0	0	1
0	0	0	1	1
0	0	0	1	1
0	1	1	0	1
1	1	1	1	0

Adjacency matrix  $A$

A	B	C	D	E
1	0	0	0	0
0	2	0	0	0
0	0	2	0	0
0	0	0	3	0
0	0	0	0	4

Degree matrix  $D$

# Laplacian Matrix

Both the Laplacian and the adjacency matrix can be normalized to achieve a more homogeneous representation of a graph. That is especially helpful for asymmetric graphs, where some nodes have a lot of neighbors and/or a high degree and some nodes have only a few neighbors and/or a low degree. The **normalized adjacency matrix** expresses weights relative to node degrees:

$$\bar{A}_{ij} = \frac{w_{ij}}{\sqrt{d_i d_j}}.$$

The **normalized Laplacian matrix** is similarly defined as:

$$\bar{\mathbf{L}} = \mathbf{I} - \bar{\mathbf{A}}$$

The Laplacian matrix, the adjacency matrix and the normalized forms of both are in the following represented by the **Graph Shift Operators  $\mathbf{S}$** . That is, because for the theoretical analysis of GCNs the specific graph matrix representation is irrelevant. Nevertheless during deployment the specific representation matters and leads to different results. It holds that if  $\mathcal{G}$  is symmetric  $\mathbf{S} = \mathbf{S}^\top$ .

**Graph Signal Diffusion.** Given the mathematical representation of a graph  $\mathcal{G}$  as  $\mathbf{S}$  that captures the structure of this graph, a **graph signal** can be defined as a vector  $\mathbf{x} \in \mathbb{R}^N$  that assigns a value  $x_i \in \mathbb{R}$  to every node  $i$ . In that way  $\mathbf{S}$  encodes the expected proximity or similarity between components of  $\mathbf{x}$ . A multiplication  $\mathbf{Sx}$  yields to a diffused signal  $\mathbf{y}$  over  $\mathcal{G}$ :

$$\mathbf{y} = \mathbf{Sx}$$

If  $\mathbf{S}$  is the adjacency matrix that would yield to  $y_i = \sum_j w_{ij} x_j$ . The operation mixes the neighboring values of  $x_i$  and diffuses its signal along the edges over  $\mathcal{G}$ . Since one application of  $\mathbf{S}$  moves the signal of  $\mathbf{x}$  only one edge at a time, a **diffusion sequence** over  $k$  steps can be defined as

$$\mathbf{x}_{k+1} = \mathbf{S} \mathbf{x}_k \quad \text{with} \quad \mathbf{x}_0 = \mathbf{x}$$

or equivalently in form of a power sequence

$$\mathbf{x}_k = \mathbf{S}^k \mathbf{x}.$$

Adjacency and Laplacian matrices are graph shift operators.

Many other graph shift operators are already proposed, e.g. read

<https://openreview.net/pdf?id=0OlrlVrsHwQ> for other operators.

Graph signal Diffusion means applying a graph shift operator to the graph, a diffusion sequence applies the graph shift operator multiple times. The idea is that we want to model a diffusion process by a shift operator. This process propagates the components of the diffused signal  $\mathbf{y} = \mathbf{x}\mathbf{G}$  from node to node and consequently indicates how much signal is sent to neighboring nodes. The graph signal is the set of values assigned to the nodes of the graph.

$$\begin{aligned} \mathbf{A}^0 &= \mathbf{I}, \\ \mathbf{A}^1 &= \mathbf{A}, \\ \mathbf{A}^k &= \underbrace{\mathbf{AA} \cdots \mathbf{A}}_{k \text{ times}}. \end{aligned}$$

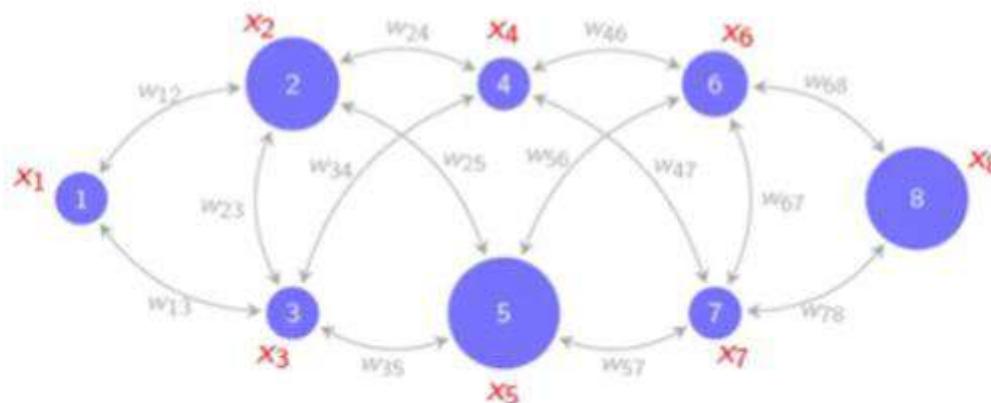


# Graph Signal

A graph signal  $f$  is a function  $f : V \rightarrow R$ . An equivalent interpretation is to view  $f$  as an  $n$ -dimensional vector, where the component corresponds to  $v \in V$  is denoted by  $f(v)$ .

- ▶ Consider a given graph  $\mathcal{G}$  with  $n$  nodes and shift operator  $S$
- ▶ A graph signal is a vector  $x \in \mathbb{R}^n$  in which component  $x_i$  is associated with node  $i$
- ▶ To emphasize that the graph is intrinsic to the signal we may write the signal as a pair  $\Rightarrow (S, x)$

Here you can see a graph with 8 nodes, a graph signal  $x$  is the vector whose elements are stored in the nodes. Therefore, the graph is an expectation of similarity between the elements of  $x$ , that is, if there is an edge between two elements of  $x$  we can assume that these two elements are more similar than two nodes that are not connected by an edge.



- ▶ The graph is an expectation of proximity or similarity between components of the signal  $x$

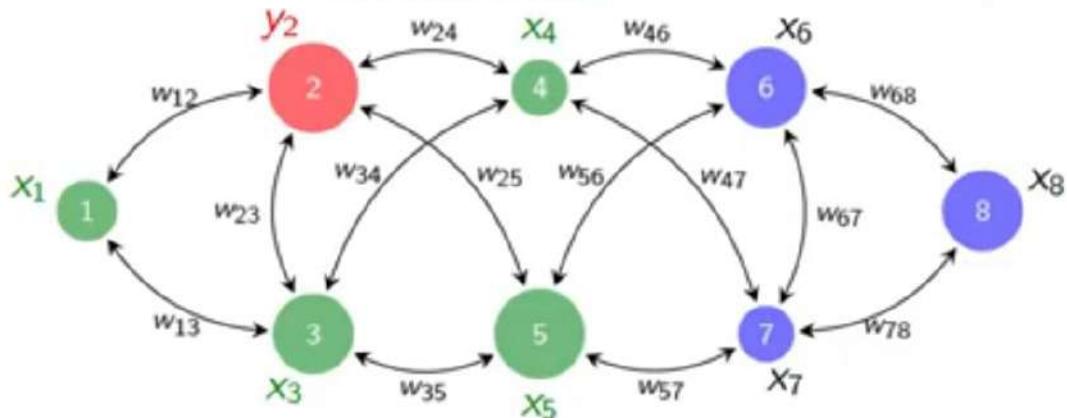


# Graph Signal

Define **diffused signal**  $\mathbf{y} = \mathbf{S}\mathbf{x}$   $\Rightarrow$  Components are  $y_i = \sum_{j \in n(i)} w_{ij} x_j$

The diffused signal is the application of the shift operator to the signal. In this example, the shift operator is the degree matrix. It is a local operation, only the nodes of the neighborhood of the given node are considered for updating its value. the green nodes are the neighboring nodes of the red node.

Codifies a **local operation** where components are mixed with components of **neighboring nodes**.



the green nodes are the neighboring nodes of the red node

- Compose the diffusion operator to produce **diffusion sequence**  $\Rightarrow$  defined recursively as

$$\mathbf{x}^{(k+1)} = \mathbf{S}\mathbf{x}^{(k)}, \quad \text{with } \mathbf{x}^{(0)} = \mathbf{x}$$

- Can **unroll** the recursion and write the diffusion sequence as the **power sequence**  $\Rightarrow \mathbf{x}^{(k)} = \mathbf{S}^k \mathbf{x}$

Diffusion sequence: The shift operator can be applied multiple times by changing the vector  $x$  values.



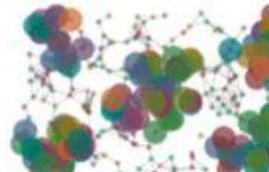
$$\mathbf{x}^{(0)} = \mathbf{x} = \mathbf{S}^0 \mathbf{x}$$



$$\mathbf{x}^{(1)} = \mathbf{S}\mathbf{x}^{(0)} = \mathbf{S}^1 \mathbf{x}$$



$$\mathbf{x}^{(2)} = \mathbf{S}\mathbf{x}^{(1)} = \mathbf{S}^2 \mathbf{x}$$



$$\mathbf{x}^{(3)} = \mathbf{S}\mathbf{x}^{(2)} = \mathbf{S}^3 \mathbf{x}$$



# Graph convolution

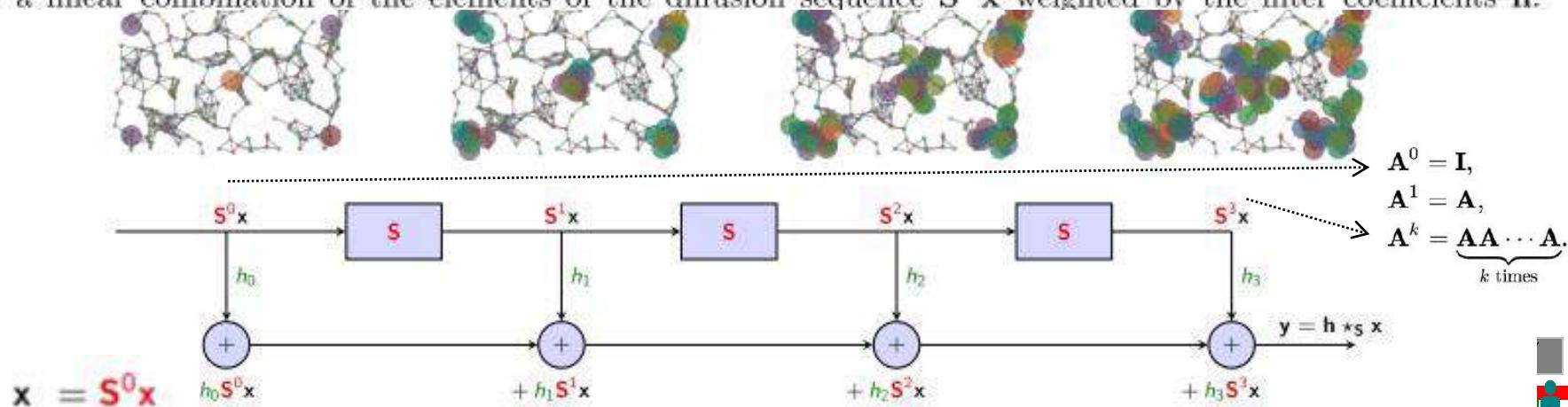
Graph convolution filters are the basic building block of a GCN. Given  $\mathbf{S}$  and filter coefficients  $h_k$ , a graph convolution is a polynomial on  $\mathbf{S}$

$$\mathbf{H}(\mathbf{S}) = \sum_{k=0}^{\infty} h_k \mathbf{S}^k$$

The result of applying the filter  $\mathbf{H}(\mathbf{S})$  to the signal  $\mathbf{x}$  is the signal  $\mathbf{y} = \mathbf{H}(\mathbf{S})\mathbf{x} = \sum_{k=0}^{\infty} h_k \mathbf{S}^k \mathbf{x}$ . As a short hand notation this is written as  $\mathbf{y} = \mathbf{h} *_{\mathbf{S}} \mathbf{x}$ , where  $*_{\mathbf{S}}$  denotes the **graph convolution** operation of filter  $\mathbf{h} = \{h_k\}_{k=0}^{\infty}$  on signal  $\mathbf{x}$  and graph shift operator  $\mathbf{S}$ . The filter coefficient  $h_k$  determines the importance of a particular diffusion state  $k = 0, \infty$  that is multiplied by  $\mathbf{x}$ . Since in practice only a finite amount of diffusion states  $K - 1$  is observed, it yields that

$$\mathbf{y} = \mathbf{h} *_{\mathbf{S}} \mathbf{x} = h_0 \mathbf{S}^0 \mathbf{x} + h_1 \mathbf{S}^1 \mathbf{x} + \cdots + h_{K-1} \mathbf{S}^{K-1} \mathbf{x} = \sum_{k=0}^{K-1} h_k \mathbf{S}^k \mathbf{x}.$$

The convolution therefore successively aggregates information from local to global neighborhoods. This is done by a linear combination of the elements of the diffusion sequence  $\mathbf{S}^k \mathbf{x}$  weighted by the filter coefficients  $\mathbf{h}$ .



**Graph convolutions as diffusion operators.** A graph convolution can be visualized in a block diagram. Iteratively a shift of  $\mathbf{S}$ , a weighting of  $\mathbf{S}^k \mathbf{x}$  with  $h_k$  and a summation of the weighted terms is applied.

# Graph convolution

**Learning Graph Filters.** After knowing what graph convolutions are, an algorithm for learning the filter coefficients  $\mathbf{h}$  of this graph convolution operation can be defined. This is equivalent to training a GCN with a single layer only. Let  $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i, \mathbf{S}_i)\}$  denote a dataset of input signals  $\mathbf{x}_i$ , output signals  $\mathbf{y}_i$  and graph shift operators  $\mathbf{S}_i$  of a graph. It is important to note that a dataset can contain *various* graph shift operators  $\mathbf{S}_i$ . Thus, a single graph filter can be trained on multiple graphs in order to become more robust during deployment even for unseen graph representations. The graph convolution  is denoted by  $f_{\mathbf{h}}(\mathbf{x}, \mathbf{S})$ . A prediction of an output signal  $\hat{\mathbf{y}}$  can be calculated using the following equation:

$$\hat{\mathbf{y}} = f_{\mathbf{h}}(\mathbf{x}, \mathbf{S}) = \sum_{k=0}^{K-1} h_k \mathbf{S}^k \mathbf{x}.$$

During the training a loss  $\mathcal{L}$  is minimized between the predicted output signal  $\hat{\mathbf{y}} = f_{\mathbf{h}}(\mathbf{x}, \mathbf{S})$  and the original output signal  $\mathbf{y}$  in order to find the optimal values for the filter coefficients  $\mathbf{h}$ :

$$\mathbf{h}^* = \operatorname{argmin}_{\mathbf{h}} \sum_{(\mathbf{x}, \mathbf{y}, \mathbf{S}) \in \mathcal{D}} \mathcal{L}(f_{\mathbf{h}}(\mathbf{x}, \mathbf{S}), \mathbf{y})$$

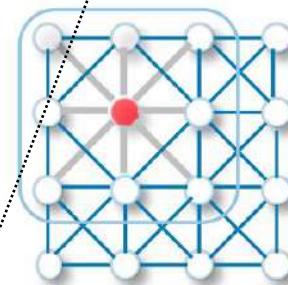
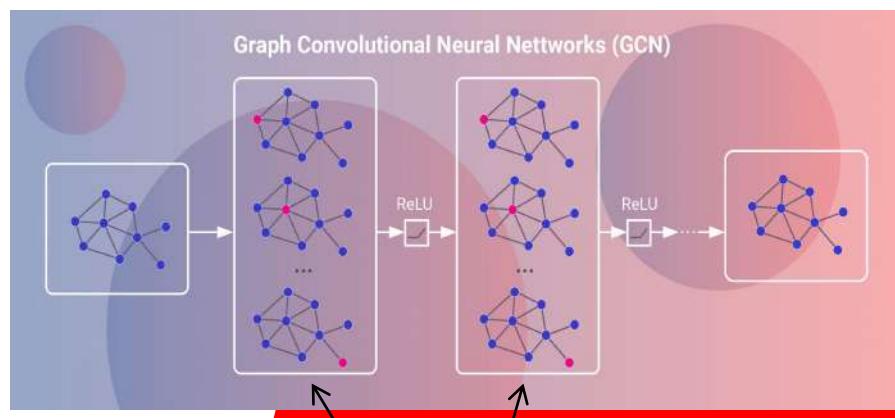


Illustration of 2D Convolutional Neural Networks (left) and Graph Convolutional Networks (right)

It has two convolution layers, we can use different shift operator in these two layers

# Graph perceptron

**Graph Perceptron.** The graph filters as defined above have limited expressive power as they can only learn linear mappings. In order to achieve a higher expressive power, the graph filters are combined with point-wise non-linearities  $g(\cdot)$ , such as sigmoid, tanh or ReLU activation functions. This function is named **graph perceptron**, since it introduces the same features for GCN as the perceptron for general neural network graph perceptron can be expressed as:

$$f_{\mathbf{h}}(\mathbf{x}, \mathbf{S}) = g \left( \sum_{k=0}^{K-1} h_k \mathbf{S}^k \mathbf{x} \right).$$

Because of the introduced non-linearity, the graph perceptron is able to express a larger function class.

**Graph Convolution Networks.** A GCN can now be defined by stacking multiple layers of graph perceptrons on top of each other. Similar to a MLP, the GCN is recursively composed

$$\mathbf{x}_\ell = g \left( \sum_{k=0}^{K-1} h_{\ell k} \mathbf{S}^k \mathbf{x}_{\ell-1} \right),$$

where it is assumed that the input to the first layer is set to the input signal  $\mathbf{x}_0 = \mathbf{x}$ . Note that compared to  $\mathbf{x}$  and  $h_k$  have index  $\ell$  now to denote the layer. A short hand notation for the recursive application of the graph perceptron over  $L$  layers is:

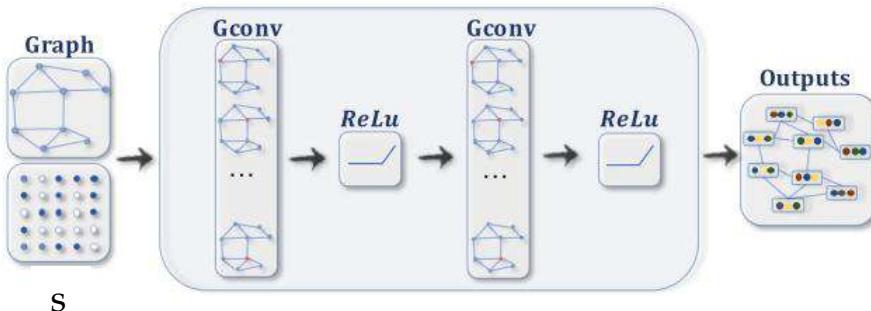
$$\hat{\mathbf{y}} = f_{\mathcal{H}}(\mathbf{x}, \mathbf{S}) = \mathbf{x}_L,$$

where  $\mathcal{H}$  denotes a set of  $L$  vectors of trainable filter coefficients  $\mathcal{H} = \{\mathbf{h}_1, \dots, \mathbf{h}_L\}$ .

# Graph convolution networks

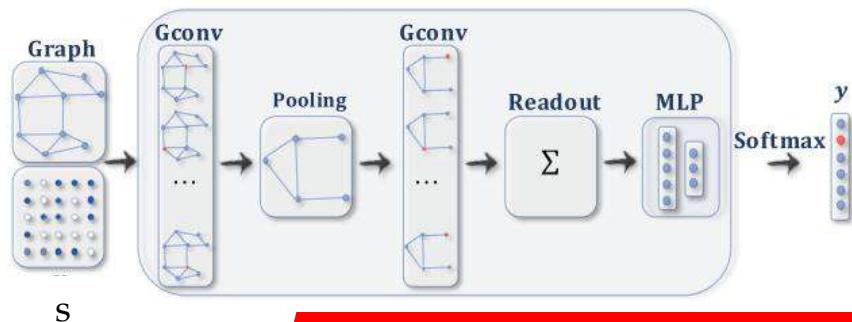
**Learning Graph Convolution Networks.** The filter parameters of a GCN are learned in same way as in the single-layer-case but with the filter coefficients of all layers  $\mathcal{H}$  as the optimization objective:

$$\mathcal{H}^* = \operatorname{argmin}_{\mathcal{H}} \sum_{(\mathbf{x}, \mathbf{y}, \mathbf{S}) \in \mathcal{D}} \mathcal{L}(f_{\mathcal{H}}(\mathbf{x}, \mathbf{S}), \mathbf{y}).$$



GCN for node classification.

A graph convolutional layer encapsulates each node's hidden representation by aggregating feature information from its neighbors. After feature aggregation, a non-linear transformation is applied to the resulted outputs. By stacking multiple layers, the final hidden representation of each node receives messages from a further neighborhood. Node-level outputs relate to node regression and node classification tasks. GCN can extract high-level node representations by information propagation/ graph convolution. In the output level we have a feature vector that describe each node, now we can use these feature vectors for training a classifieres for node classification or use them for clustering the node.

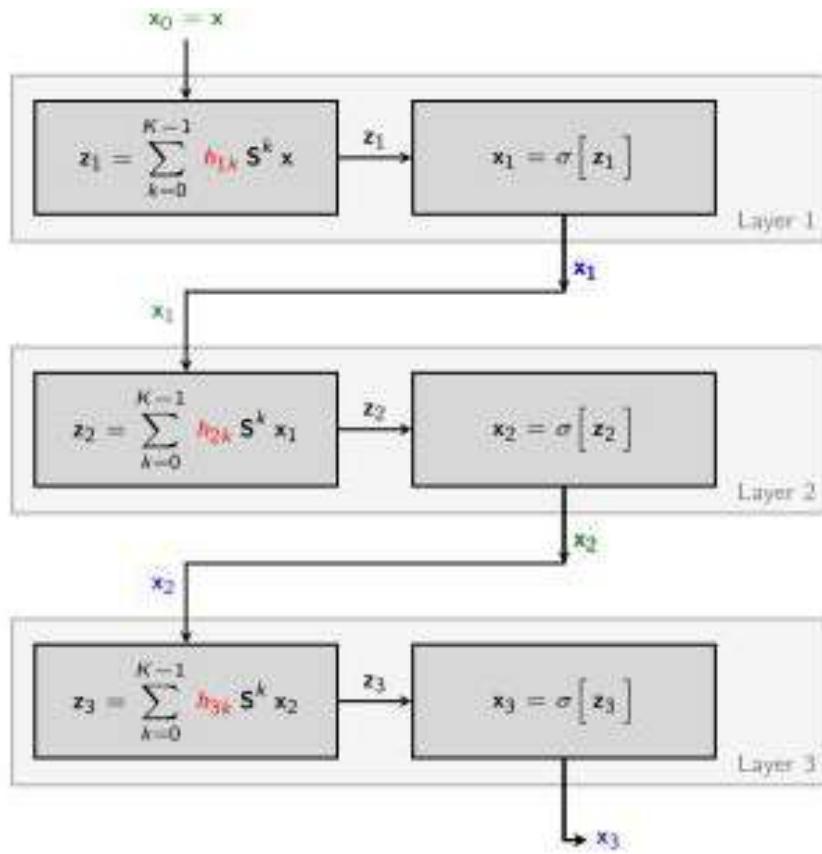


GCN for graph classification.

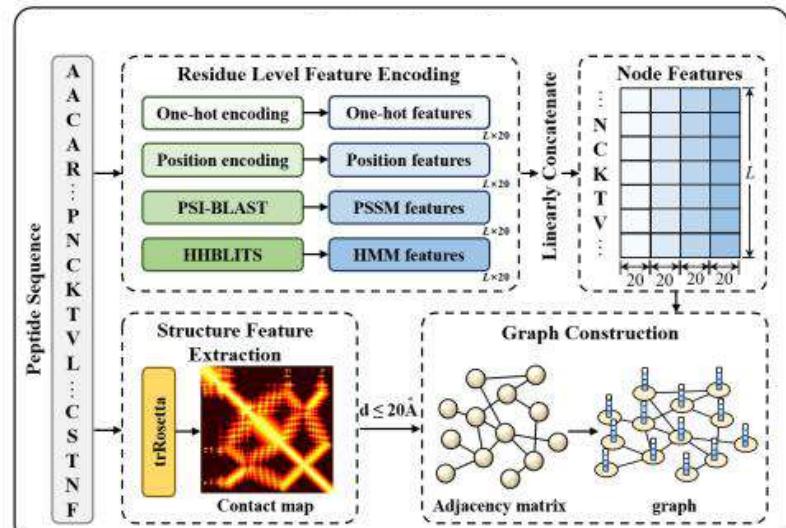
A GCN with pooling and readout layers for graph classification. A graph convolutional layer is followed by a pooling layer to coarsen a graph into sub-graphs so that node representations on coarsened graphs represent higher graph-level representations. A readout layer summarizes the final graph representation by taking the sum/mean of hidden representations of sub-graphs (e.g. you can cluster the graph for obtaining the sub-graphs).

A pooling layer can be any graph clustering algorithm that group sets of nodes together, a max pooling takes the max values from the cluster.

# Graph convolution networks



**Block diagram of a GCN.** The input vector  $x_0$  is passed into the first graph perceptron in the first layer  $\ell = 1$ . The output  $x_1$  is equally passed into layer  $\ell = \ell + 1$  until the final output  $x_3$  is produced.

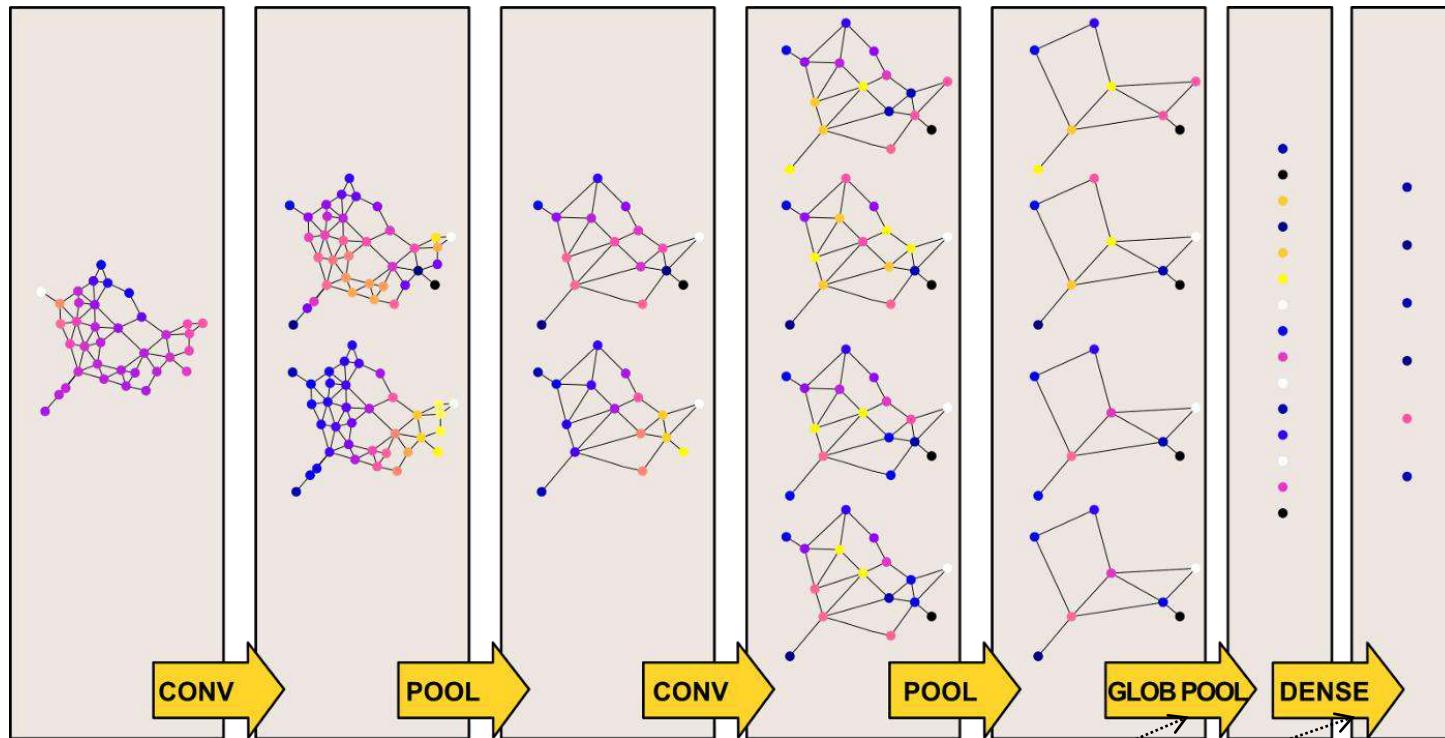


The flowchart of feature extraction and graph construction process. The original pattern is a peptide, an amino-acid sequence, the goal is to represent the peptide as a graph. Antimicrobial peptides (AMPs) as biomolecules have therapeutic functions. AMPs sequences have broad antibiotic-resistant activity against Gram-negative bacteria, cancer cells, fungi, etc.

For each amino-acid a feature vector is extracted, we extract 4 different features, including one-hot features, position encoding features, PSSM features, and HMM features (for details: <https://doi.org/10.1093/bioinformatics/btac715>). Each node is related to a given amino acid, the value of each node is the feature vector that describes the amino-acid.

Now, we have to choose the edges that connect the nodes. We can build the adjacency matrix considering the distances among the amino-acid, considering the peptide 3d structure, this information is obtainable from the contact maps, e.g. you can obtain it by using trRosetta (Yang, J., et al. Improved protein structure prediction using predicted interresidue orientations. Proc Natl Acad Sci U S A 2020;117(3):1496-1503.). If the distance is lower than 20 angstrom ( $1 = 0.1$  nm) then there is an edge that connects the two amino acids

# GCN for classification



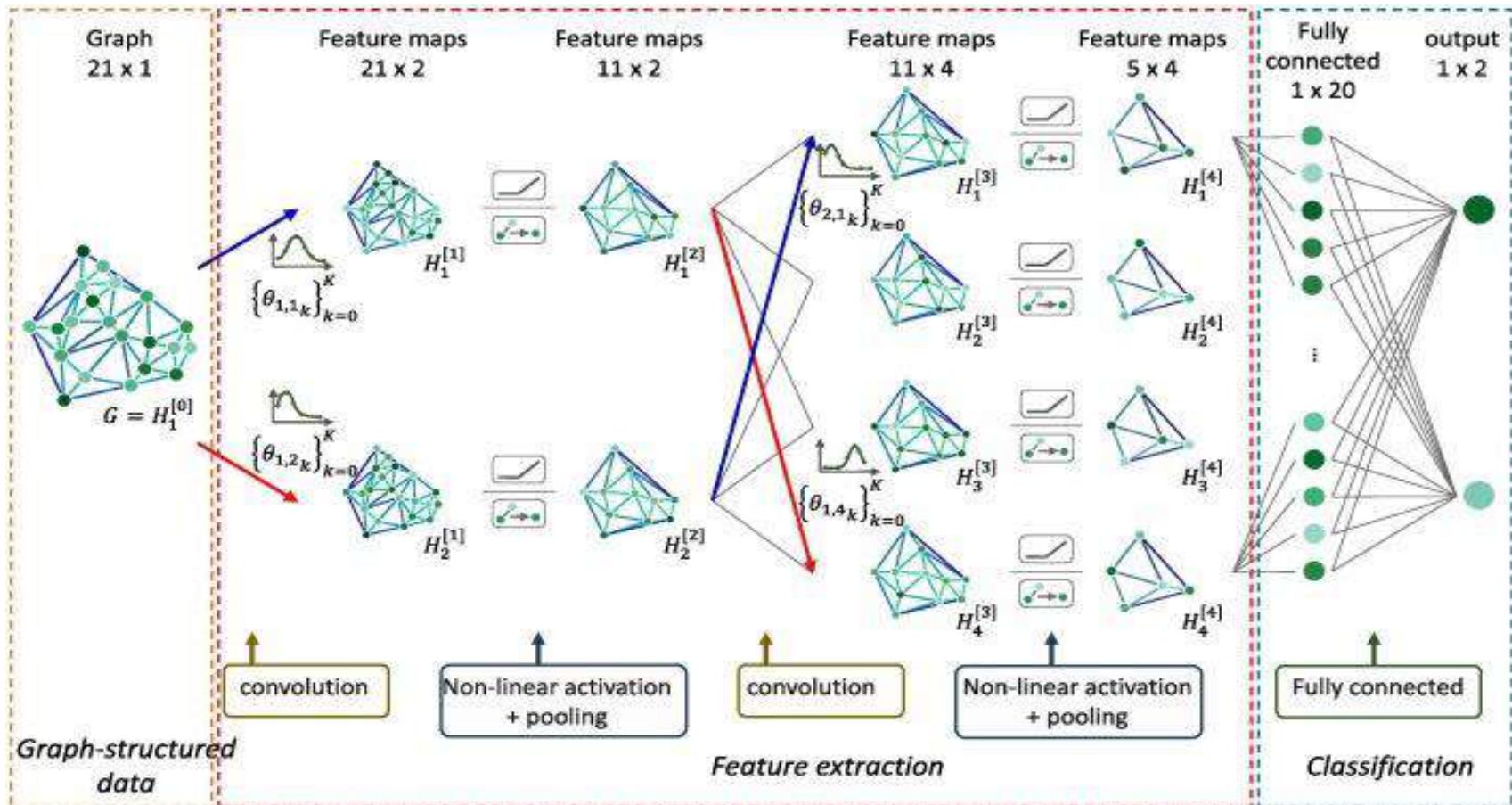
Concerning pooling layers, we can choose any graph clustering algorithm that merges sets of nodes together while preserving local geometric structures.

A distinction often found in the literature is that between “regular” and global pooling, which is extremely evident, to the point where global pooling is usually referred to as a separate operation called “readout”.

Specifically, global pooling indicates those methods that reduce a graph to a single node, discarding all topological information.

As for traditional CNNs, a GCN for classification consists of several layers, e.g. convolutional and pooling layers for feature extraction, followed by the final fully-connected layers.

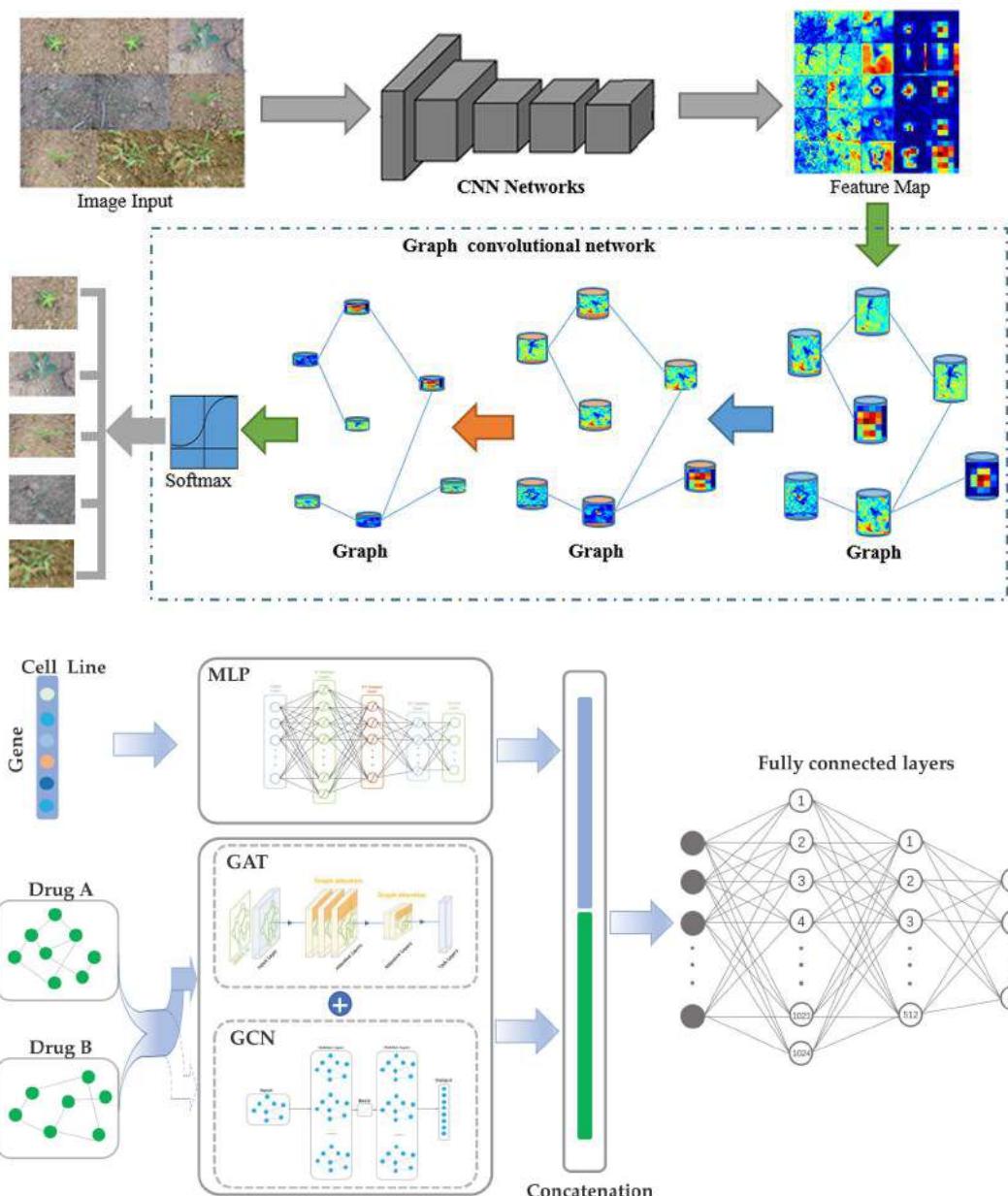
# GCN for classification



GCNs are a hot topic in current scientific research and many new methods are published every month, e.g. attention-based graph convolutional layer have been recently proposed in:  
<https://link.springer.com/content/pdf/10.1007/s00138-021-01251-0.pdf>



# GCNN examples



CNN feature extraction: each image feature is extracted using a CNN model, and we can finally obtain a CNN feature set; a GCN graph is constructed based on extracted CNN feature  $X$  and vertex set  $V$ . Here  $V$  consists of  $V_{train}$  and  $V_{test}$ , where  $V_{train}$  and  $V_{test}$  are labeled and unlabeled vertices separately.

The adjacency matrix  $A$  is the Euclidean distance of each two weed image features.

GCN is to obtain the label information of unlabeled vertices. The GCN model naturally combines graph structures and vertex features in the convolution, and propagated over the graph through multiple layers. Through the graph convolution layer by layer, unlabeled vertices update their features. Then the final obtained features of unlabeled vertices are fed into the softmax classifier, and the output is label information of unlabeled vertices. Thus all unlabeled images are recognized by a softmax classifier trained based on features of labeled vertices  $V_{train}$ .

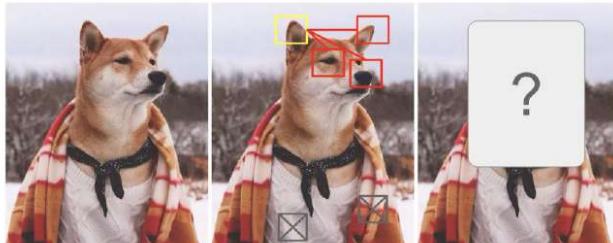
<https://www.sciencedirect.com/science/article/pii/S0168169919321349>

For each pairwise drug combination, the input layer firstly receives the molecular graphs of two drugs and gene expression profiles of one cancer cell line that was treated by these two drugs. We tested two type of Graph Neural Networks (GNN), graph attention network (GAT) and graph convolution network (GCN), to extract features of drugs. The genomic feature representation of cancer cells is encoded by a multi-layer perception (MLP). The embedding vectors are subsequently concatenated as the final feature representation of each drug-pair-cell-line combination, which is propagated through the fully-connected layers for the binary classification of drug combinations (synergistic or antagonistic). For example, triple-negative breast cancer is a malignant tumor with strong invasiveness, high metastasis rate and poor prognosis. Lapatinib or Rapamycin alone has little therapeutic effect, but their combined treatment has been reported to significantly increase the apoptosis rate of triple-negative breast cancer cells

<https://www.biorxiv.org/content/10.1101/2021.04.06.438723v2.full.pdf>

# Transformer

<https://blogs.nvidia.com/blog/2022/03/25/what-is-a-transformer-model/>



The name of the dog is Shiba Inu.

If we focus at the features of the dog in the red boxes, like his nose, right pointy ear and mystery eyes, we'll be able to guess what should come in the yellow box.

However, by just looking at the pixels in the gray boxes, you won't be able to predict what should come in the yellow box. The attention mechanism weighs the pixel in the correct boxes more w.r.t the pixel in the yellow box. While the pixel in the gray boxes would be weighed less.

A **language model** models the probability distribution over a sequence of **discrete tokens**  $\mathbf{x} = (x_1, \dots, x_T)$ . Each of these tokens can take a value from a **vocabulary**  $\mathcal{V}$  ( $x_t \in \mathcal{V}$ ) and it can be for example a word, a character or a byte, depending on the model. The last token of such a sequence is a special `<EOS>` token to indicate the end of a sentence. Therefor:  $x_T = <\text{EOS}>$ . This means that whenever we predict the next word or token, we can either predict a word form the vocabulary or we can predict the end of sentence token.

$$p(\mathbf{x}) = p(x_1, \dots, x_T) = \prod^T p(x_t|x_1, \dots, x_{t-1})$$

## Word Language Model Example:

$$\begin{aligned} p(\text{The dog ran away } <\text{EOS}>) &= p(\text{The}) p(\text{dog}|\text{The}) p(\text{ran}|\text{The dog}) \\ &\quad p(\text{away}|\text{The dog ran}) p(<\text{EOS}>|\text{The dog ran away}) \end{aligned}$$

As we can see in the example above, the sentence 'The dog ran away `<EOS>`' decomposes based on the product rule into the conditional distributions  $p$  of 'The', times  $p$  of 'dog' given 'The', times  $p$  of 'ran' given 'The' and 'dog' and so forth. We can see that language models in general but also in particular the given word language model are **autoregressive** models that predict the next token given all the previous tokens in this sentence or sequence. If a model is good then it has a high probability of predicting likely next words.



# Transformer/ Tokenization

Ashish Vaswani; Noam Shazeer; Niki Parmar;  
Jakob Uszkoreit et al. "Attention is all you need",  
2017

In 2017 there was another big step and it was the transformer paper

- The fundamental innovation of the Transformer is the **self-attention layer**
- For each position  $t$  in the sequence we compute an attention over the other positions in the sequence
- The transformer uses **multiple heads** (because multiple heads are empirically better), i.e., it computes the attention operation multiple times ( $K = 8$  in the original implementation)
- **Self-attention** then constructs a tensor  $A[k, t_1, t_2]$  – the strength of the attention weight from  $t_1$  to  $t_2$  for head  $k$ . So the attention that  $t_1$  pays to  $t_2$  in a particular layer of the transformer for a particular head  $k$ .

Attention is so key to transformers the Google researchers almost used the term as the name for their 2017 model.

"Attention Net didn't sound very exciting," said Vaswani.

Jakob Uszkoreit, a senior software engineer on the team, came up with the name Transformer.

"I argued we were transforming representations, but that was just playing semantics," Vaswani said.

Tokenization is the process of encoding a string of text into transformer-readable token ID integers.

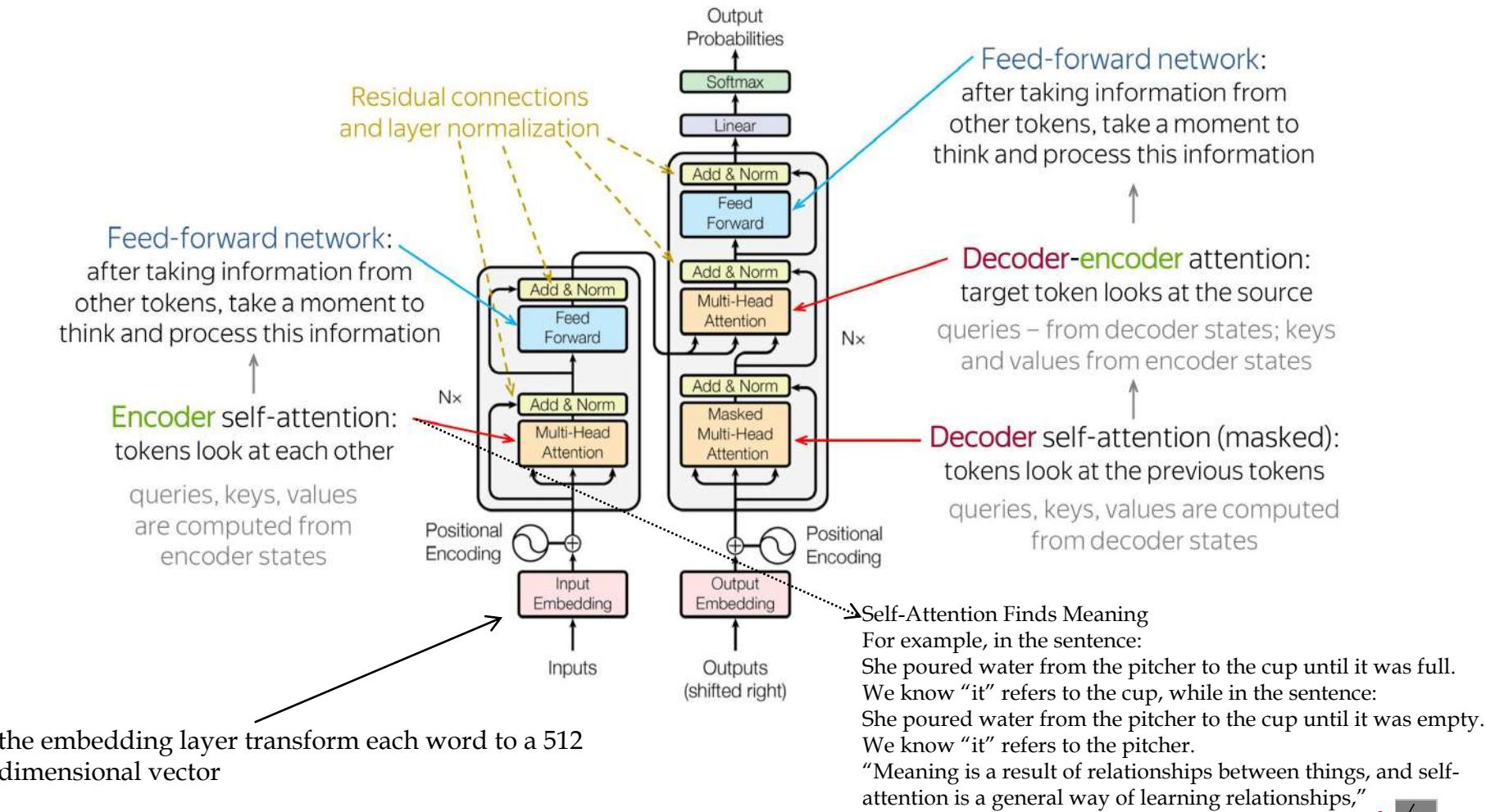


During inference/training, these token IDs are read by an embedding layer in our transformer model – which maps the token ID to a dense vector representation of that token (imagine a numerical representation of the meaning of our word).

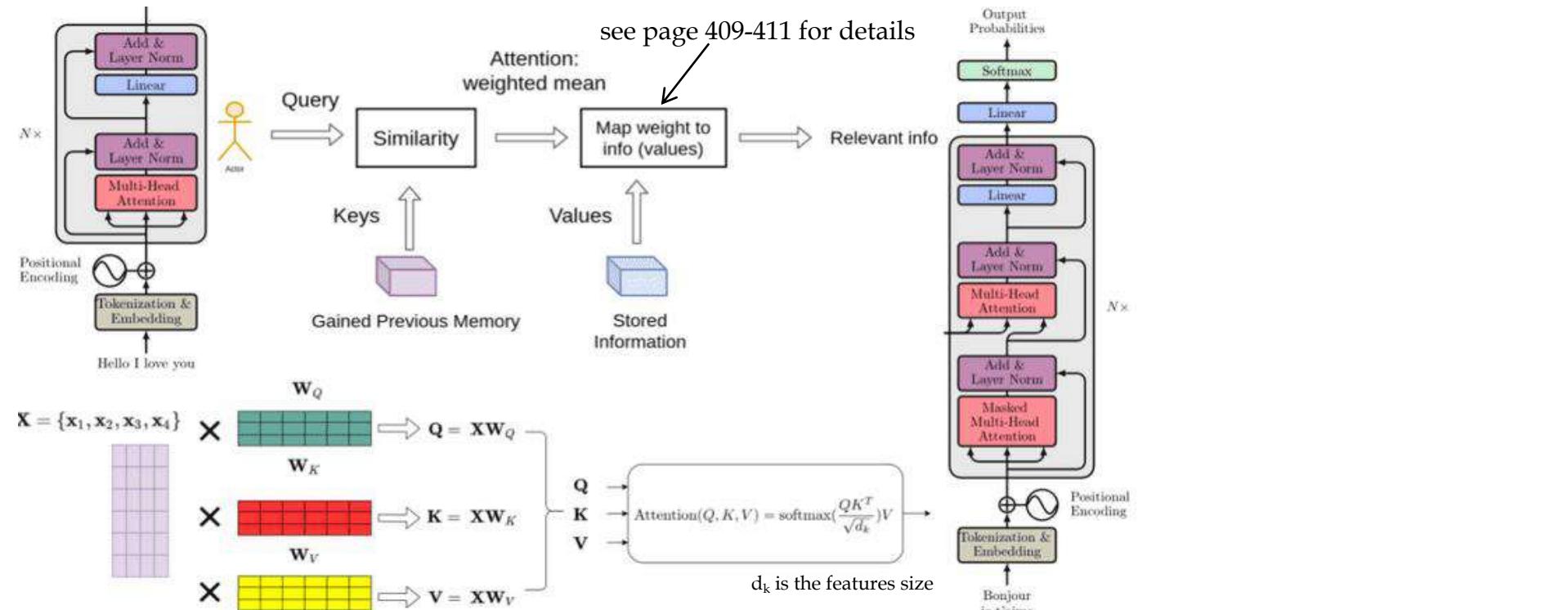
i.e. each word is represented with a feature vector, there some method for representing words as vectors, for details you could read <https://en.wikipedia.org/wiki/Word2vec>

# Transformer - general picture

<https://data-science-blog.com/blog/2021/04/07/multi-head-attention-mechanism/>



# Transformer - general picture



the input token  $X$  is projected by the linear projection layer  $W_q$ ,  $W_k$ , and  $W_v$  to embed  $Q$ ,  $K$ , and  $V$  vectors, respectively. Then, the dot product  $V$  with the attention matrix, which is obtained by the scaled dot product of  $Q$  and  $K$ .

The general idea is that we submit a query for each token in the input sequence. We then take those queries and match them against a series of keys that describe values that we want to know something about. The similarity of a given query to the keys determines how much information from each value to retrieve for that particular query. The output of the first matrix multiplication ( $QK^T$ ), where we take the similarity of each query to each of the keys, is known as the attention matrix. The attention matrix depicts how much each token in the sequence is paying attention to each of the keys (hence the  $n \times n$  shape).

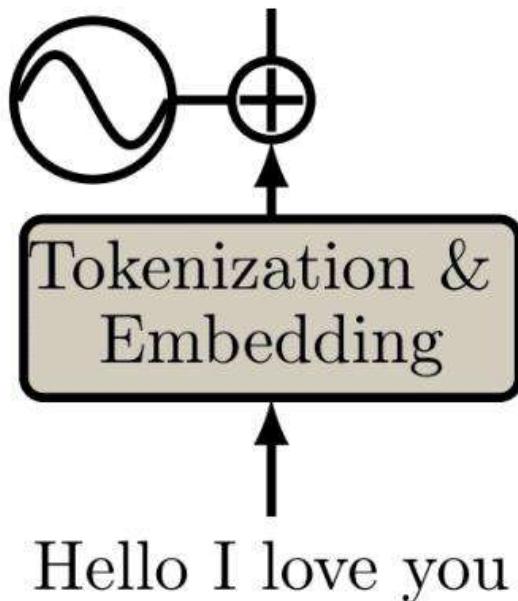
Attention is a rather general mechanism that can be used in a wide variety of problem domains. Consider the task of machine translation using an RNN model. Also, consider the problem of image classification using a basic CNN model. While an RNN produces a sequence of hidden state vectors, a CNN creates feature maps, where each region in the image is represented by a feature vector. The RNN hidden states are organized sequentially, while the CNN feature maps are organized spatially. Yet, attention can still be applied in both situations, since the attention mechanism does not inherently depend on the organization of the feature vectors. This characteristic makes attention easy to implement in a wide variety of models in different domains.

# Positional Encoding

Let's help them have a sense of order by slightly altering the embeddings based on the position. Officially, **positional encoding** is a set of small constants, which are added to the word embedding vector before the first self-attention layer.

So if the same word appears in a different position, the actual representation will be slightly different, depending on where it appears in the input sentence.

Positional  
Encoding



In the transformer paper, the authors came up with the sinusoidal function for the positional encoding. The sine function tells the model to pay attention to a particular wavelength  $\lambda$ . Given a signal  $y(x) = \sin(kx)$  the wavelength will be  $k = \frac{2\pi}{\lambda}$ . In our case the  $\lambda$  will be dependent on the position in the sentence.  $i$  is used to distinguish between odd and even positions.

Mathematically:

$$\vec{p}_t = \begin{bmatrix} \sin(\omega_1 \cdot t) \\ \cos(\omega_1 \cdot t) \\ \sin(\omega_2 \cdot t) \\ \cos(\omega_2 \cdot t) \\ \vdots \\ \sin(\omega_{d/2} \cdot t) \\ \cos(\omega_{d/2} \cdot t) \end{bmatrix}_{d \times 1} \quad (\omega_k = \frac{1}{1000^{\frac{2k}{d}}})$$

$d=512$ , the feature size of each word  
 $t$  is the order of the word in the sentence

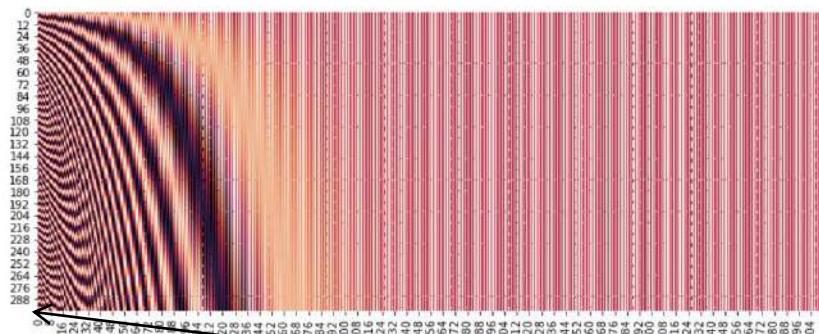
the  $p_t$  is added to the feature vector used to describe each word



# Position encoding

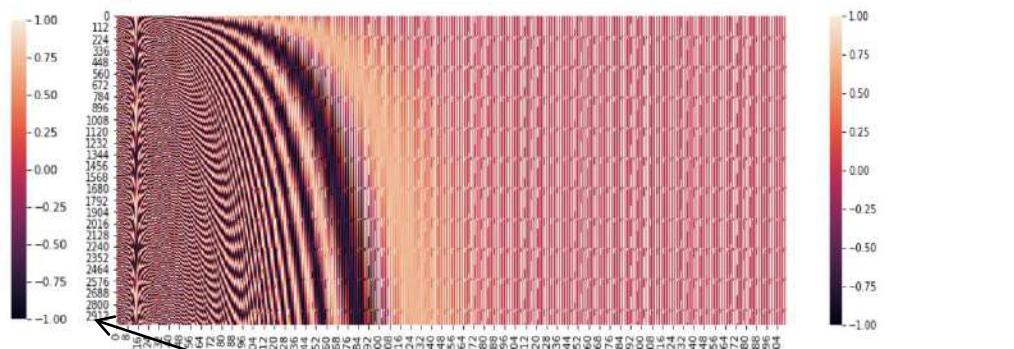
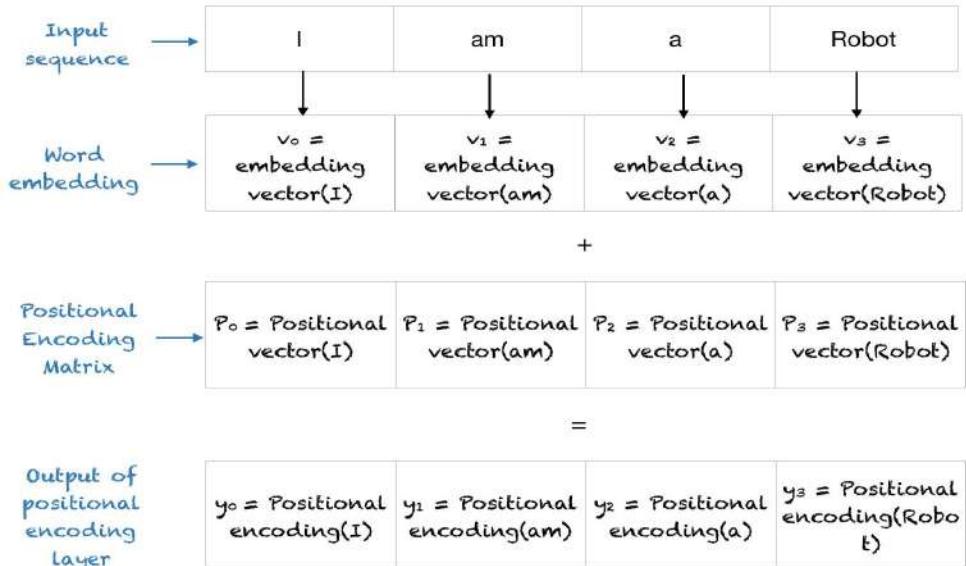
Sequence	Index of token, $k$	Positional Encoding			
		$i=0$	$i=0$	$i=1$	$i=1$
I	0	$P_{00}=\sin(0) = 0$	$P_{01}=\cos(0) = 1$	$P_{02}=\sin(0) = 0$	$P_{03}=\cos(0) = 1$
am	1	$P_{10}=\sin(1/1) = 0.84$	$P_{11}=\cos(1/1) = 0.54$	$P_{12}=\sin(1/10) = 0.10$	$P_{13}=\cos(1/10) = 1.0$
a	2	$P_{20}=\sin(2/1) = 0.91$	$P_{21}=\cos(2/1) = -0.42$	$P_{22}=\sin(2/10) = 0.20$	$P_{23}=\cos(2/10) = 0.98$
Robot	3	$P_{30}=\sin(3/1) = 0.14$	$P_{31}=\cos(3/1) = -0.99$	$P_{32}=\sin(3/10) = 0.30$	$P_{33}=\cos(3/10) = 0.96$

Positional Encoding Matrix for the sequence 'I am a robot'



Above is the heatmap of the position encoding matrix that we will add to the input that is to be given to the first encoder. I am showing the heatmap for the first 300 positions and the first 3000 positions. We can see that there is a distinct pattern that we provide to our Transformer to understand the position of each word. And since we are using a function comprised of sin and cos, we are able to embed positional embeddings for very high positions also pretty well as we can see in the second picture.

Interesting Fact: The authors also let the Transformer learn these encodings too and didn't see any difference in performance as such. So, they went with the above idea as it doesn't depend on sentence length and so even if the sentence is bigger than train samples, we would be fine.



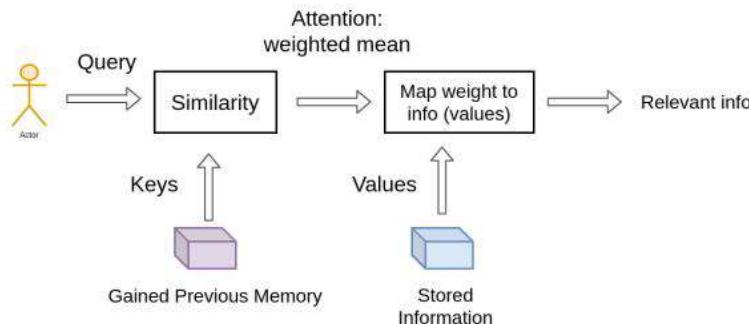
# Key-Value-Query concept

<https://towardsdatascience.com/how-gpt-works-a-metaphoric-explanation-of-key-value-query-in-attention-using-a-tale-of-potion-8c66ace1f470>

Let's start with an example of searching for a video on youtube.

When you search (**query**) for a particular video, the search engine will map your **query** against a set of **keys** (video title, description, etc.) associated with possible stored videos. Then the algorithm will present you the best-matched videos (**values**). This is the foundation of content/**feature-based lookup**.

Bringing this idea closer to the transformer's attention we have something like this:



In the single video retrieval, the attention is the choice of the video with a maximum relevance score.

But we can relax this idea. To this end, the main difference between attention and retrieval systems is that **we introduce a more abstract and smooth notion of 'retrieving' an object**. By defining a degree of similarity (weight) between our representations (videos for youtube) we can weight our query.

So, by moving one step forward, we further split the data into key-value pairs.

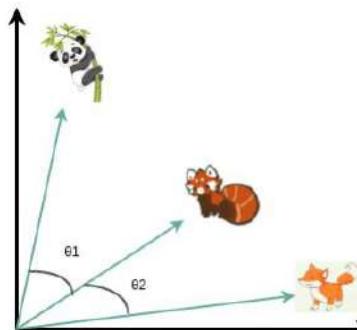
We use the **keys** to define the **attention weights** to look at the data and the **values** as the **information** that we will actually get.

For the so-called mapping, we need to **quantify similarity**, that we will be seeing next.

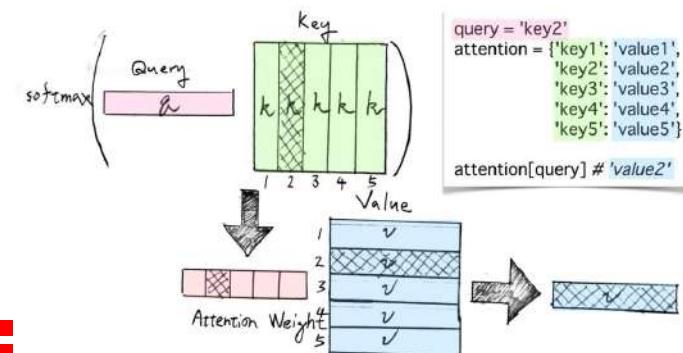
large dot product      small dot product

## Vector similarity in high dimensional spaces

In geometry, the **inner vector product** is interpreted as a vector projection. One way to define vector similarity is by computing the normalized inner product. In low dimensional space, like the 2D example below, this would correspond to the cosine value.

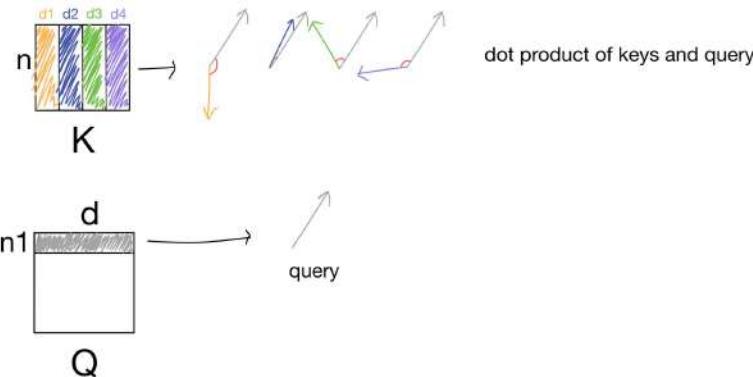


(Attention is a Dictionary object)



$$\text{sim}(a, b) = \cos(a, b) = \frac{a \cdot b}{\|a\| \|b\|} = \frac{1}{s} * a \cdot b$$

# Key-Value-Query concept

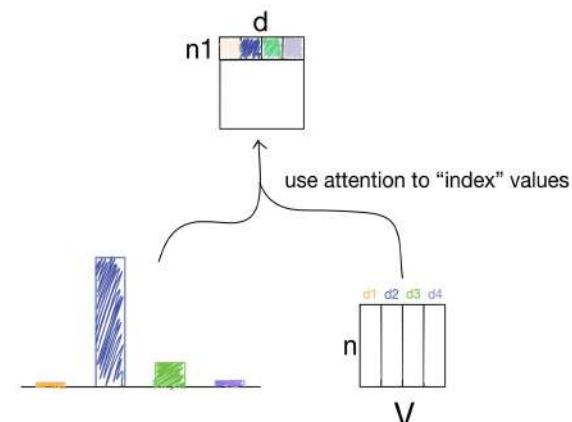


For keys close to the same direction as the query, the values will be large, while for keys differing in direction from the query vector the values would be low. These values are then passed through a softmax function to scale the values to a probability distribution that adds up to 1, and also sharpens the distribution so the higher values are even higher and lower values even lower.

These values are what make up the attention matrix, and are a representation of how much each query matches a given key.

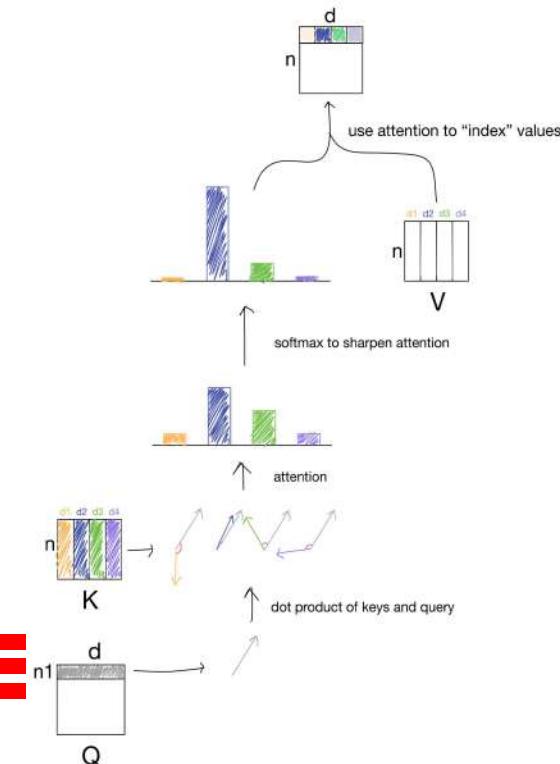
After we have our attention values, we need to index the information contained in the value matrix  $V$ , because, after all, now that we know how we want to route information, we need to actually route the information contained in the values.

To route the information, we simply take another dot product, but this time between the attention matrix and the value matrix.



In doing so we effectively "index" into the information for each value in the value matrix in proportion with the amount attention described in the attention matrix.

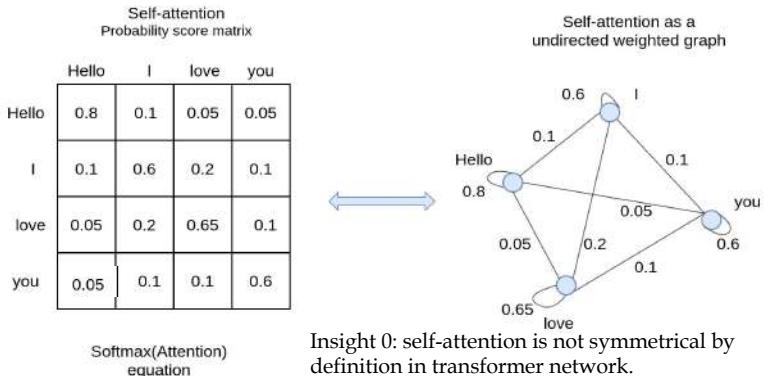
At the end of all this we in theory have a better, more information rich representation than prior to the attention operation.



# Self attention

"Self-attention, sometimes called intra-attention, is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence." ~  
Self-attention enables us to find correlations between different words of the input indicating the syntactic and contextual structure of the sentence.

Let's take the input sequence "Hello I love you" for example. A trained self-attention layer will associate the word "love" with the words 'I' and "you" with a higher weight than the word "Hello". From linguistics, we know that these words share a subject-verb-object relationship and that's an intuitive way to understand what self-attention will capture.

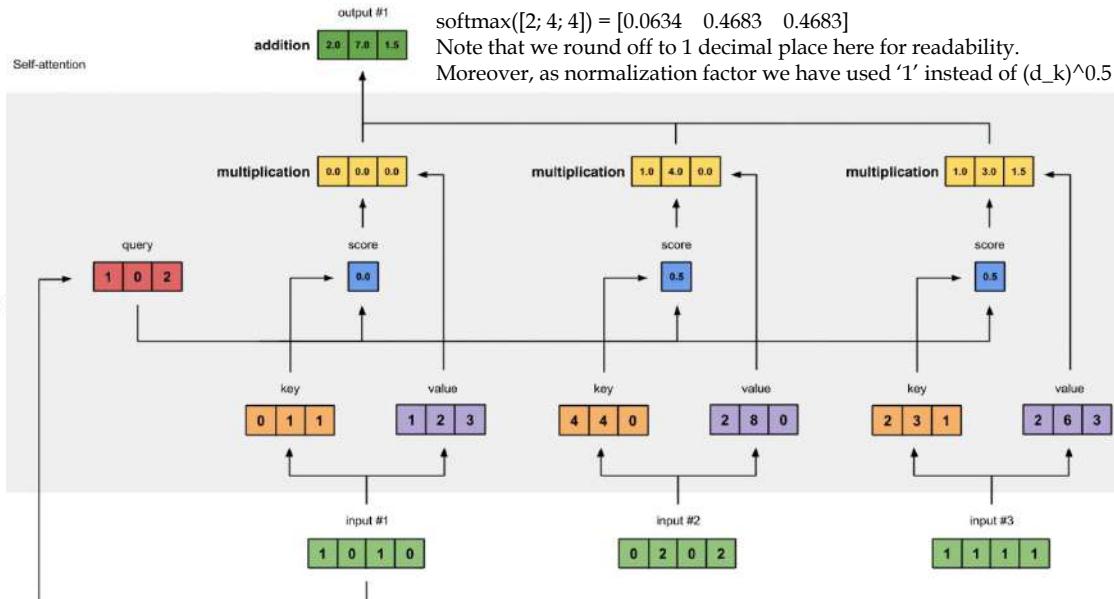


As you can see the diagonal entries are big. This is because the word contribution to itself is high. That is reasonable.

Having the Query, Value and Key matrices, we can now apply the self-attention layer as:

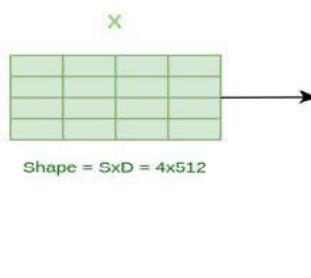
$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

<https://towardsdatascience.com/illustrated-self-attention-2d627e33b20a>  
the self-attention mechanism allows the inputs to interact with each other ("self") and find out who they should pay more attention to ("attention"). The outputs are aggregates of these interactions and attention scores. Every input must have three representations (see diagram below). These representations are called key (orange), query (red), and value (purple). For this example, let's take that we want these representations to have a dimension of 3 (output). Because every input has a dimension of 4, each set of the weights must have a shape of  $4 \times 3$ . To obtain attention scores, we start with taking a dot product between Input 1's query (red) with all keys (orange), including itself. Since there are 3 key representations (because we have 3 inputs), we obtain 3 attention scores (blue). Take the softmax across these attention scores (blue). The softmaxed attention scores for each input (blue) is multiplied by its corresponding value (purple). This results in 3 alignment vectors (yellow). We'll refer to them as weighted values. Take all the weighted values (yellow) and sum them element-wise. The resulting vector [2.0, 7.0, 1.5] (dark green) is Output 1, which is based on the query representation from Input 1 interacting with all other keys, including itself. Repeat for Input 2 & Input 3



# Self-attention

You can also use the rows of  $X$  to represent each word, obviously, the size of the  $W$ s will change with respect to the previous example.

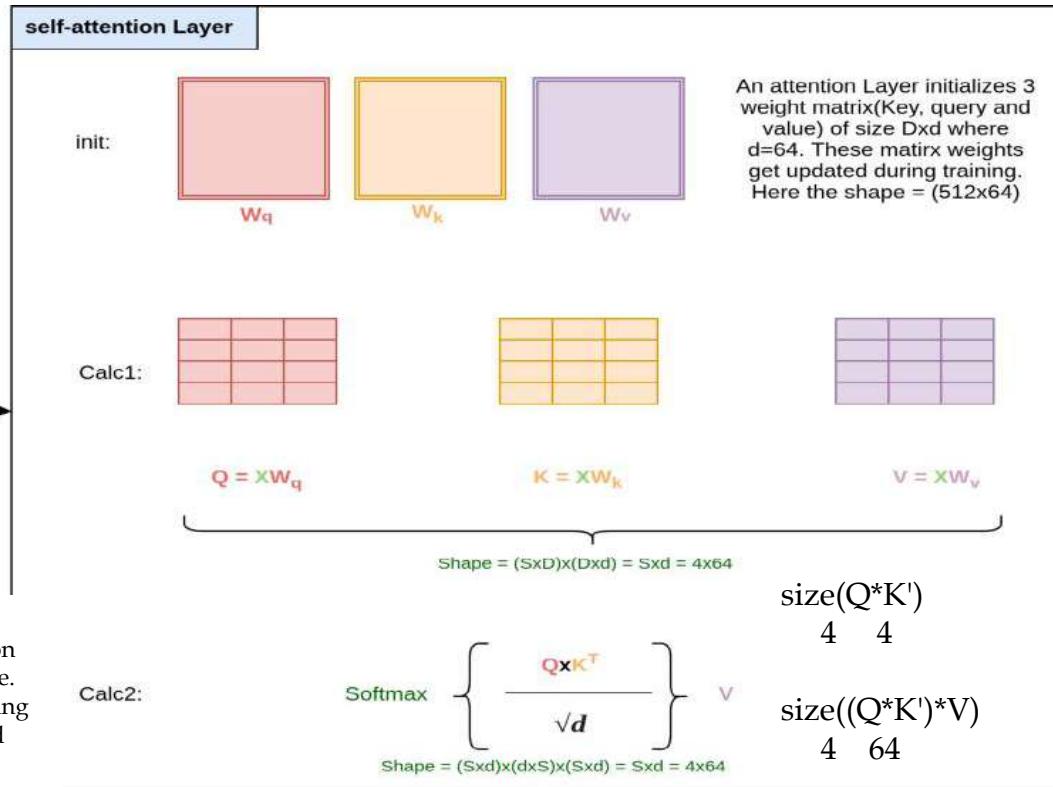


Content-based attention has distinct representations. The query matrix in the attention layer is conceptually the “search” in the database. The keys will account for where we will be looking while the values will actually give us the desired content. Consider the keys and values as components of our database.

Intuitively, the keys are the bridge between the queries (what we are looking for) and the values (what we will actually get).

Keep in mind that each vector to vector multiplication is a dot-product similarity. We can use the keys to guide our “search” and tell us where to look with respect to the input elements.

In other words, the keys will account for the computation of the attention on how to weigh the values based on our particular queries.



The feature model can be a recurrent neural network (RNN), a convolutional neural network (CNN), a simple embedding layer, a linear transformation of the original data, or no transformation at all. Essentially, the feature model consists of all the steps that transform the original input (e.g. a sentence) into the input  $X$  of the attention layer

The attention mechanism in Neural Networks tends to mimic the cognitive attention possessed by human beings. The main aim of this function is to emphasize the important parts of the information, and try to de-emphasize the non-relevant parts. Since working memory in both humans and machines is limited, this process is key to not overwhelming a system's memory. In deep learning, attention can be interpreted as a vector of importance weights. When we predict an element, which could be a pixel in an image or a word in a sentence, we use the attention vector to infer how much is it related to the other elements.

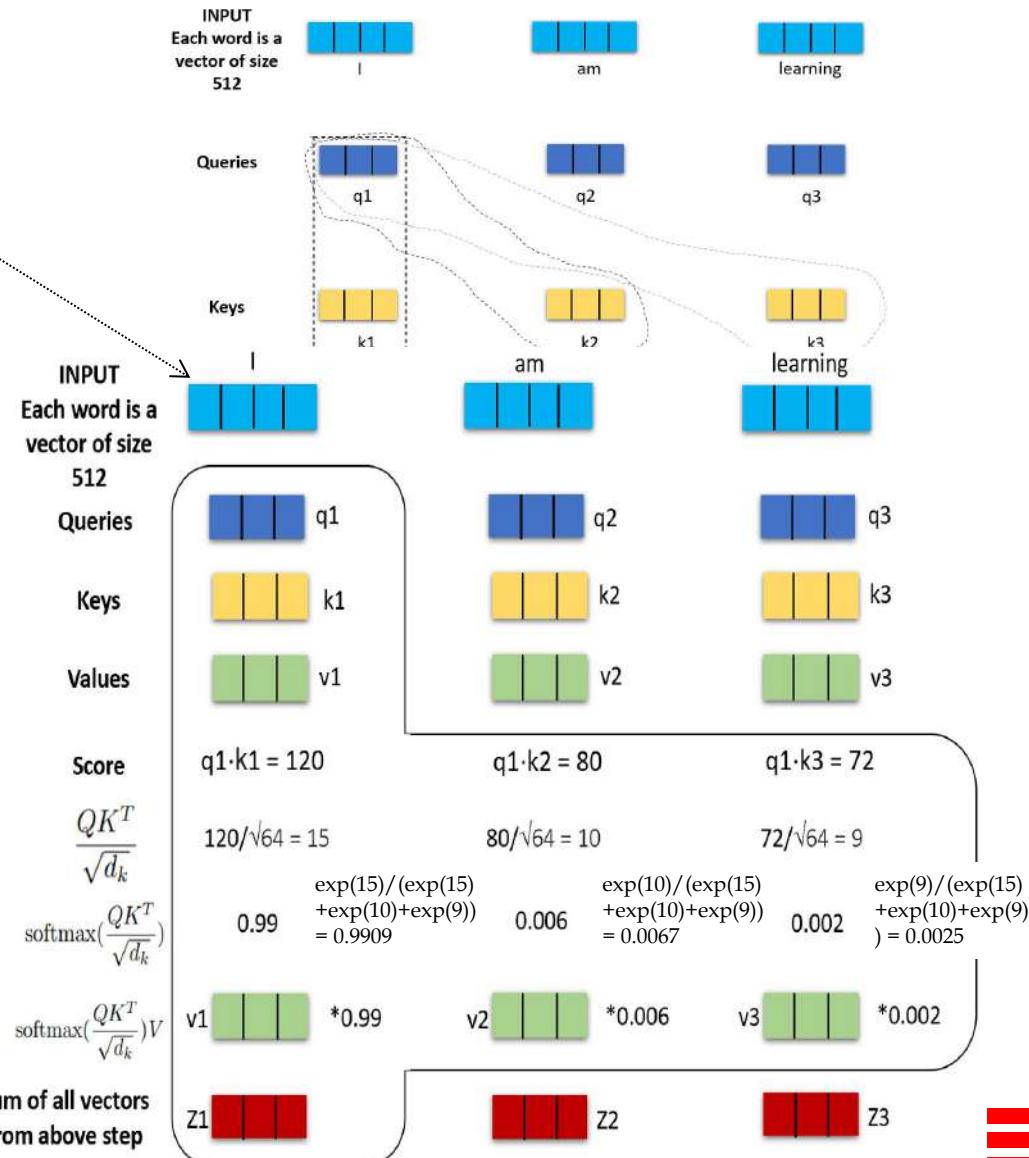
# Self-attention

the first word is "I" so we need to check the score of this word with all the words used in the sentence. The score is calculated by dot product of query and key vector of the respective words that we are scoring. For example, we are calculating score for "I" then score 1 ->  $q_1 \times k_1$ , score 2 ->  $q_1 \times k_2$  and score 3 ->  $q_1 \times k_3$ .

The  $\sqrt{d_k}$  is here simply as a scaling factor to make sure that the vectors won't explode.

Following the database-query paradigm we introduced before, this term simply finds the similarity of the searching query with an entry in a database. Finally, we apply a softmax function to get the final attention weights as a [probability distribution](#).

Remember that we have distinguished the Keys ( $K$ ) from the Values ( $V$ ) as distinct representations. Thus, the final representation is the self-attention matrix  $\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$  multiplied with the Value ( $V$ ) matrix.



# Multi-head attention

I like to think of it as multiple "linear views" of the same sequence. In the original paper, the authors expand on the idea of self-attention to multi-head attention. In essence, we run through the attention mechanism several times.

Each time, we map the independent set of Key, Query, Value matrices into different lower dimensional spaces and compute the attention there (the output is called a "head"). The mapping is achieved by multiplying each matrix with a separate weight matrix, denoted as  $\mathbf{W}_i^K, \mathbf{W}_i^Q \in R^{d_{model} \times d_k}$  and  $, \mathbf{W}_i^V \in R^{d_{model} \times d_k}$

To compensate for the extra complexity, the output vector size is divided by the number of heads. Specifically, in the vanilla transformer, they use  $d_{model} = 512$  and  $h = 8$  heads, which gives us vector representations of 64. Now, the model has multiple independent paths (ways) to understand the input.

The heads are then concatenated and transformed using a square weight matrix  $\mathbf{W}^O \in R^{d_{model} \times d_{model}}$ , since  $d_{model} = h d_k$        $h=8, d_k=d_{model}/h=512/8=64$

The heads are then concatenated and transformed using a square weight matrix  $\mathbf{W}^O \in R^{d_{model} \times d_{model}}$ , since  $d_{model} = h d_k$

Putting it all together we get:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \dots, \text{head}_h) \mathbf{W}^O$$

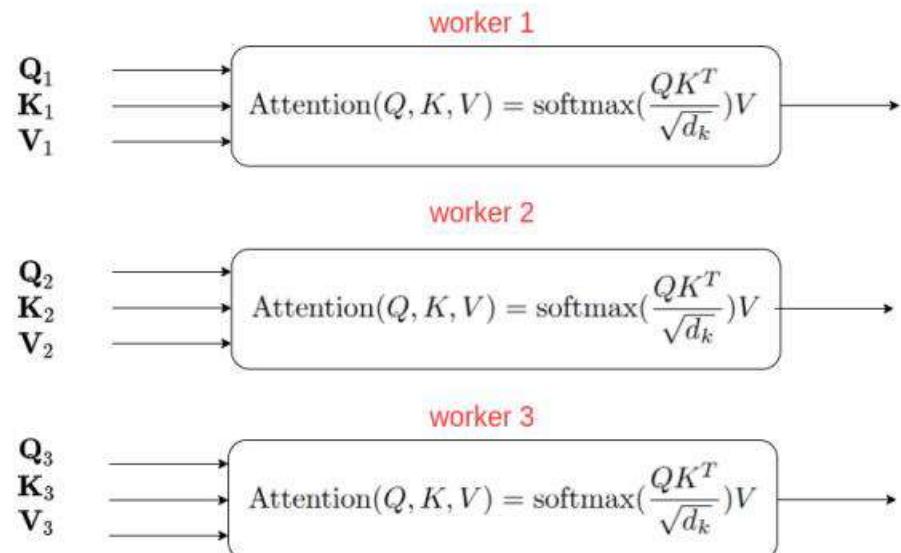
where head<sub>i</sub> = Attention  $(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V)$

where again:

$$\mathbf{W}_i^Q, \mathbf{W}_i^K, \mathbf{W}_i^V \in R^{d_{model} \times d_k}$$

Since heads are independent from each other, we can perform the self-attention computation in parallel on different workers:

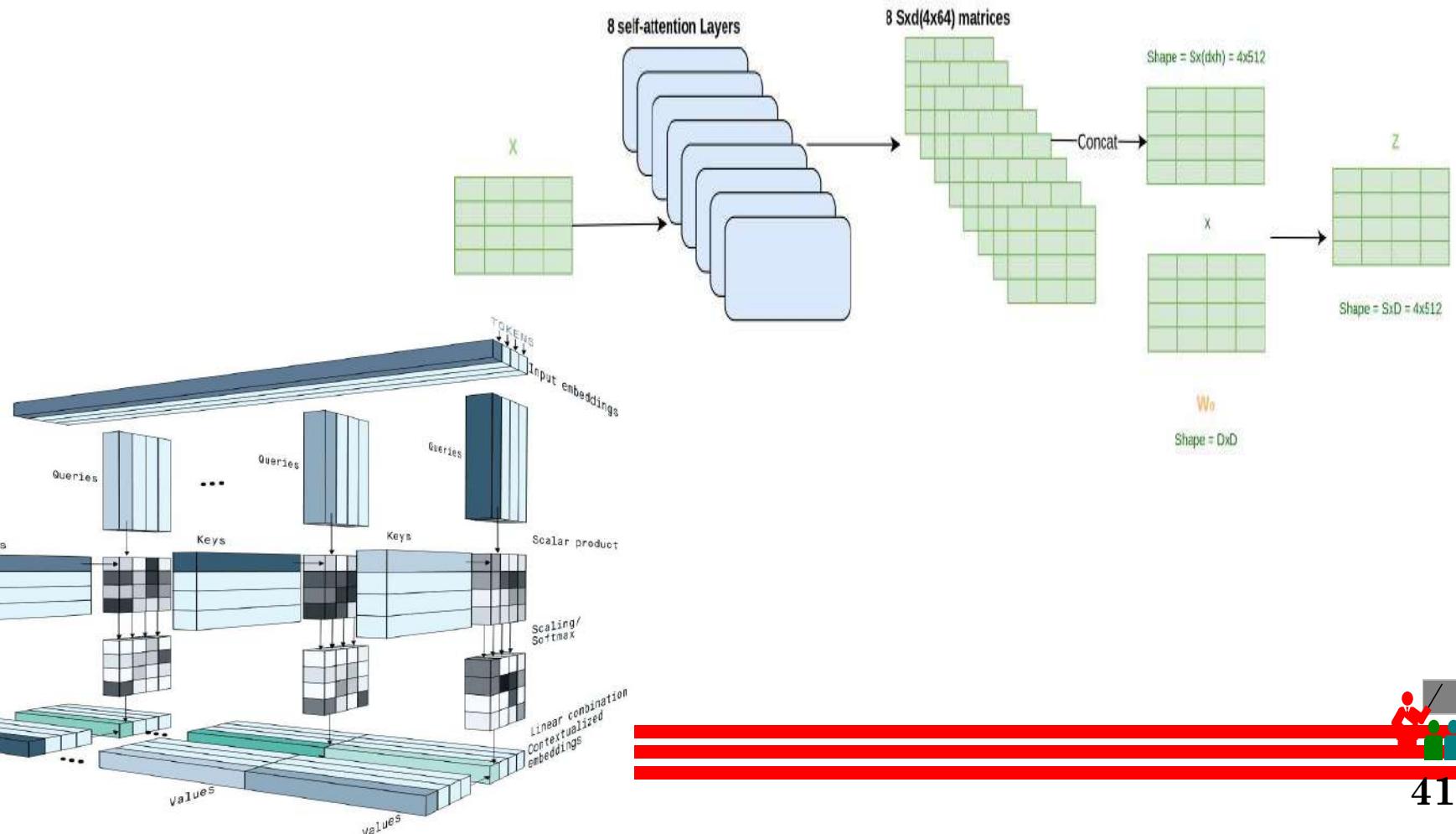
Each attention head can be implemented in parallel



# multi-head

The number of attention layers,  $h$ , is kept as 8 in the paper. So the input  $X$  goes through many self-attention layers parallelly, each of which gives a matrix  $z$  of shape. We concatenate these 8 ( $h$ ) matrices and again apply a final linear layer ( $W_0$ ).

What size do we get? For the concatenation operation we get a size of  $S \times D(4 \times 64 \times 8) = 4 \times 512$ . And multiplying this output by  $W_0$ , we get the final output  $Z$  with the shape of  $S \times D(4 \times 512)$  as desired.



# Normalization Layer

In Layer Normalization (LN), the mean and variance are computed across channels and spatial dims. In language, each word is a vector. Since we are dealing with vectors we only have one spatial dimension.

$$\mu_n = \frac{1}{K} \sum_{k=1}^K x_{nk}$$

$$\sigma_n^2 = \frac{1}{K} \sum_{k=1}^K (x_{nk} - \mu_n)^2$$

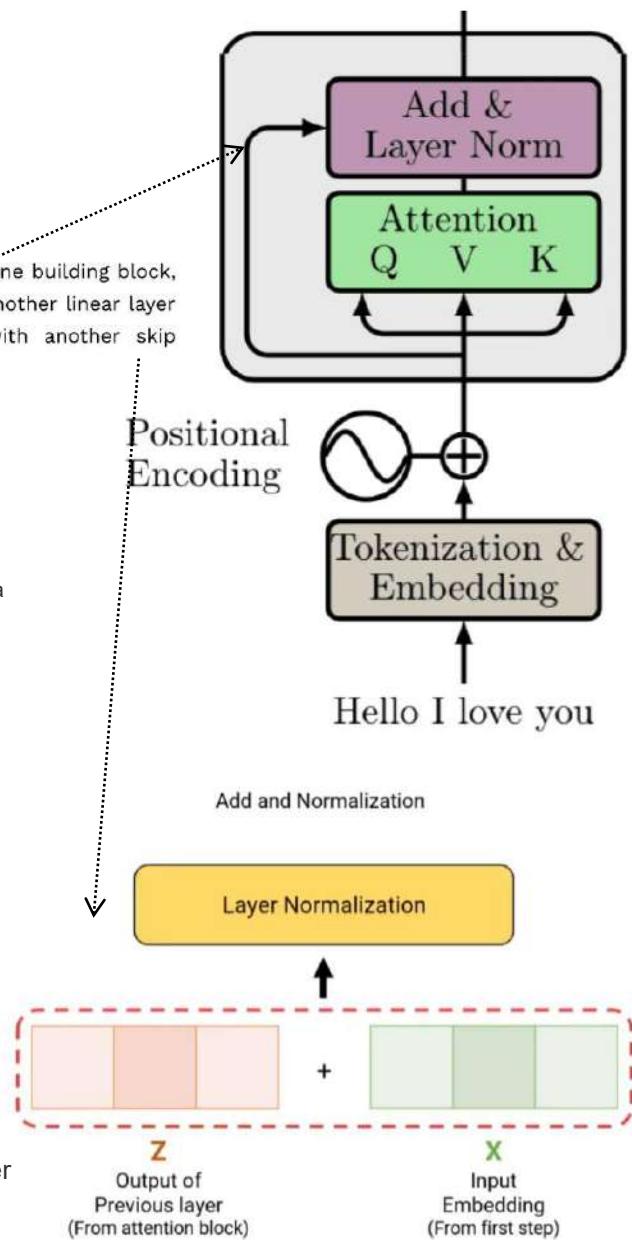
$$\hat{x}_{nk} = \frac{x_{nk} - \mu_n}{\sqrt{\sigma_n^2 + \epsilon}}, \hat{x}_{nk} \in R$$

It is important to note that here we are doing a layer normalization, not a batch normalization like in CNN. n indexes the pattern, in this case the words of the input sentence and k indexes the feature elements. These statistics are computed through the feature vector of a given pattern, so they are independent with respect to those calculated for other patterns.

The  $n$  dimension is the pattern (the words of the sentence) and  $k$  is the feature dimension. Note that in the above equations, for Batch Normalization (BN), K corresponds to the training samples, whereas for LN, this corresponds to the size of the feature vector. To summarize, with BN, the statistics are computed across the batch and are the same for each example in the batch. In contrast, in LN, the statistics are computed across feature vector of a given pattern (here each word), so they are independent of other patterns.

The attention layer output is one of two input of the normalization layer. We sum the output of the attention layer with the input of the attention layer, the result of the sum is normalized.

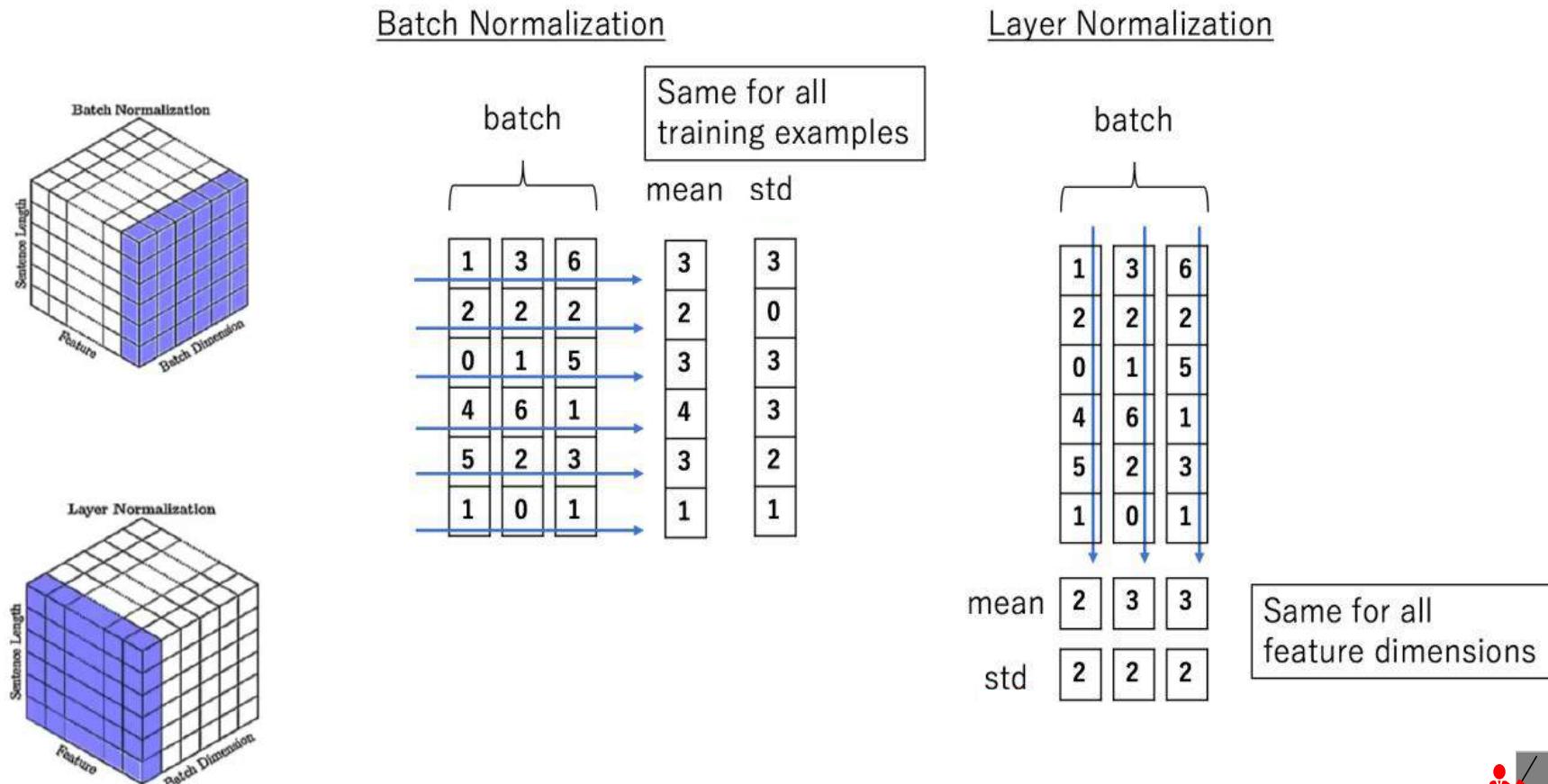
Even though this could be a stand-alone building block, the creators of the transformer add another linear layer on top and renormalize it along with another skip connection.



# Normalization Layer

each column is a pattern, e.g. a word, each row is a feature. The values of a column are the features that describe each pattern.

In batch normalization we normalize the features considering all the patterns of a given batch, in layer normalization we normalize each pattern considering its features.



# Linear layer

$$\mathbf{y} = \mathbf{x}\mathbf{W}^T + \mathbf{b}$$

Where  $\mathbf{W}$  is a matrix and  $\mathbf{y}, \mathbf{x}, \mathbf{b}$  are vectors.

In fact, they add two linear layers with dropout and non-linearities in between.

```
import torch.nn as nn
dim = 512
dim_linear_block = 512*4

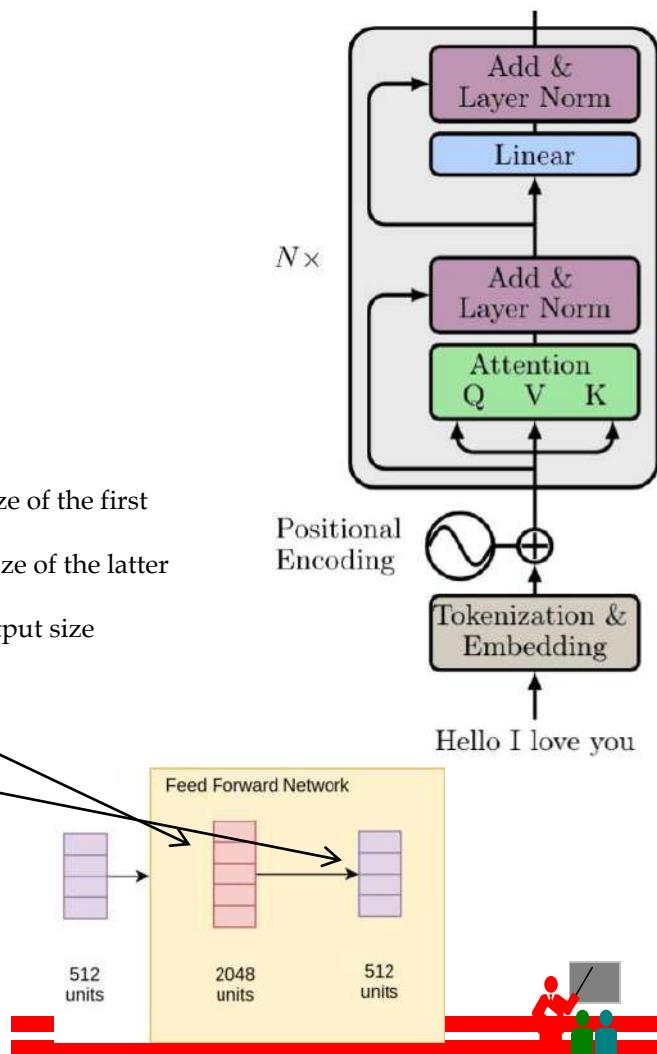
linear = nn.Sequential(
    nn.Linear(dim, dim_linear_block),
    nn.ReLU(),
    nn.Dropout(dropout),
    nn.Linear(dim_linear_block, dim),
    nn.Dropout(dropout))
```

This consists of two linear transforms with a ReLU activation in between.

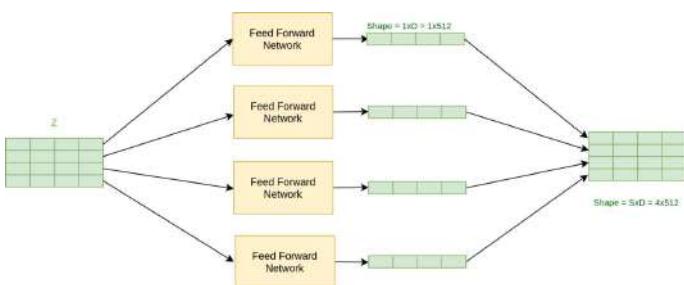
In this way the output has the same size of the input, e.g.:  
size( $x$ ) -> 1 512 input size  
size( $W$ ) -> 512 2048 matrix size of the first linear layer  
size( $W_1$ ) -> 2048 512 matrix size of the latter linear layer  
size( $X^TW^*W_1$ ) -> 1 512 output size

The main intuition is that they project the output of self-attention in a higher dimensional space ( $\times 4$  in the paper). This solves bad initializations and rank collapse. We will depict it in the diagrams simply as Linear.

Each word goes into the feed-forward network. The feed-forward network applies itself to each position parallelly (each position can be thought of as a word) and hence the name Position-wise feed-forward network. The feed-forward network also shares the weights.



# Linear layer

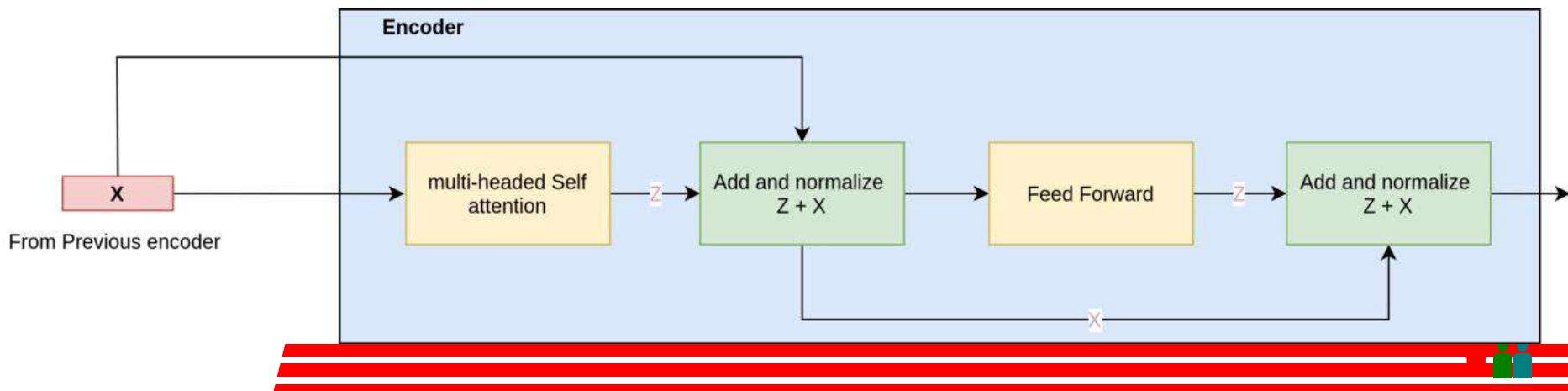
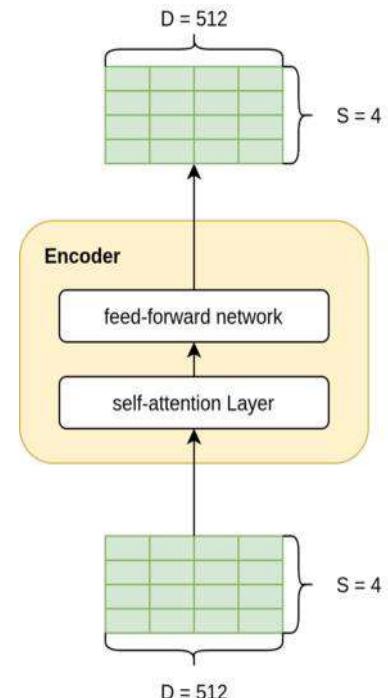


In this figure it is clear that each word (each row of this matrix) goes into a different feed-forward network, these networks can operate in parallel, moreover, all these networks share internal weights. The linear level output is the same size as the linear level input.

Each word goes into the feed-forward network.

The feed-forward network is applied to each position in a parallel manner (each position can be thought of as a word), hence the name feed-forward position-wise network. The feed-forward networks share the weights.

Notice that each input sentence could have a different number of words, so a different number of rows. Each matrix is a sentence (each row a word)



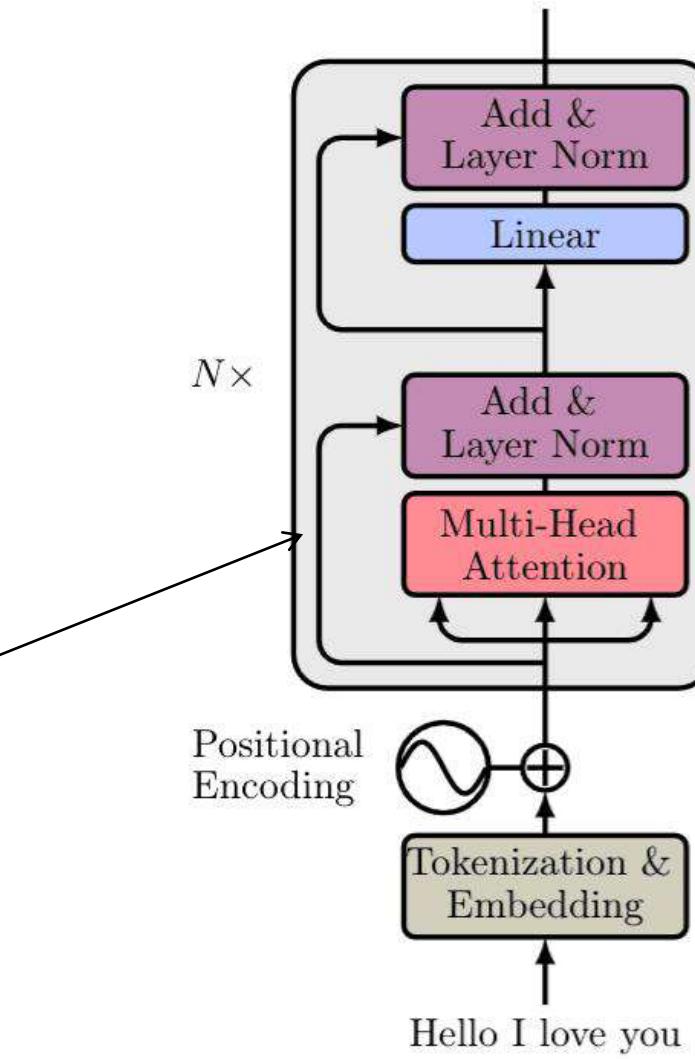
# Sum up: the Transformer encoder

To process a sentence we need these 3 steps:

1. Word embeddings of the input sentence are computed simultaneously.
2. Positional encodings are then applied to each embedding resulting in word vectors that also include positional information.
3. The word vectors are passed to the first encoder block.

Each block consists of the following layers in the same order:

1. A multi-head self-attention layer to find correlations between each word
2. A **normalization** layer
3. A residual connection around the previous two sublayers
4. A linear layer
5. A second normalization layer
6. A second residual connection



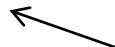
Note that the above block can be replicated several times to form the Encoder. In the original paper, the encoder composed of 6 identical blocks.

# Transformer decoder

<https://kikaben.com/transformers-encoder-decoder/>

The decoder consists of all the aforementioned components plus two novel ones. As before:

1. The output sequence is fed in its entirety and word embeddings are computed



be careful, the ouput is the input of the decoder (?!), yes, it is explained in the next pages.

2. Positional encoding are again applied

3. And the vectors are passed to the first Decoder block

Each decoder block includes:

1. A **Masked** multi-head self-attention layer

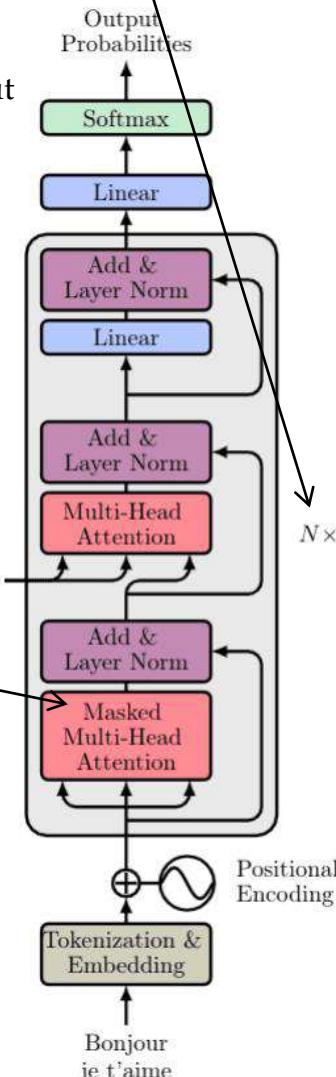
2. A normalization layer followed by a residual connection

3. A new multi-head attention layer (known as **Encoder-Decoder attention**) this input is the ouput of the encoder

4. A second normalization layer and a residual connection

5. A linear layer and a third residual connection

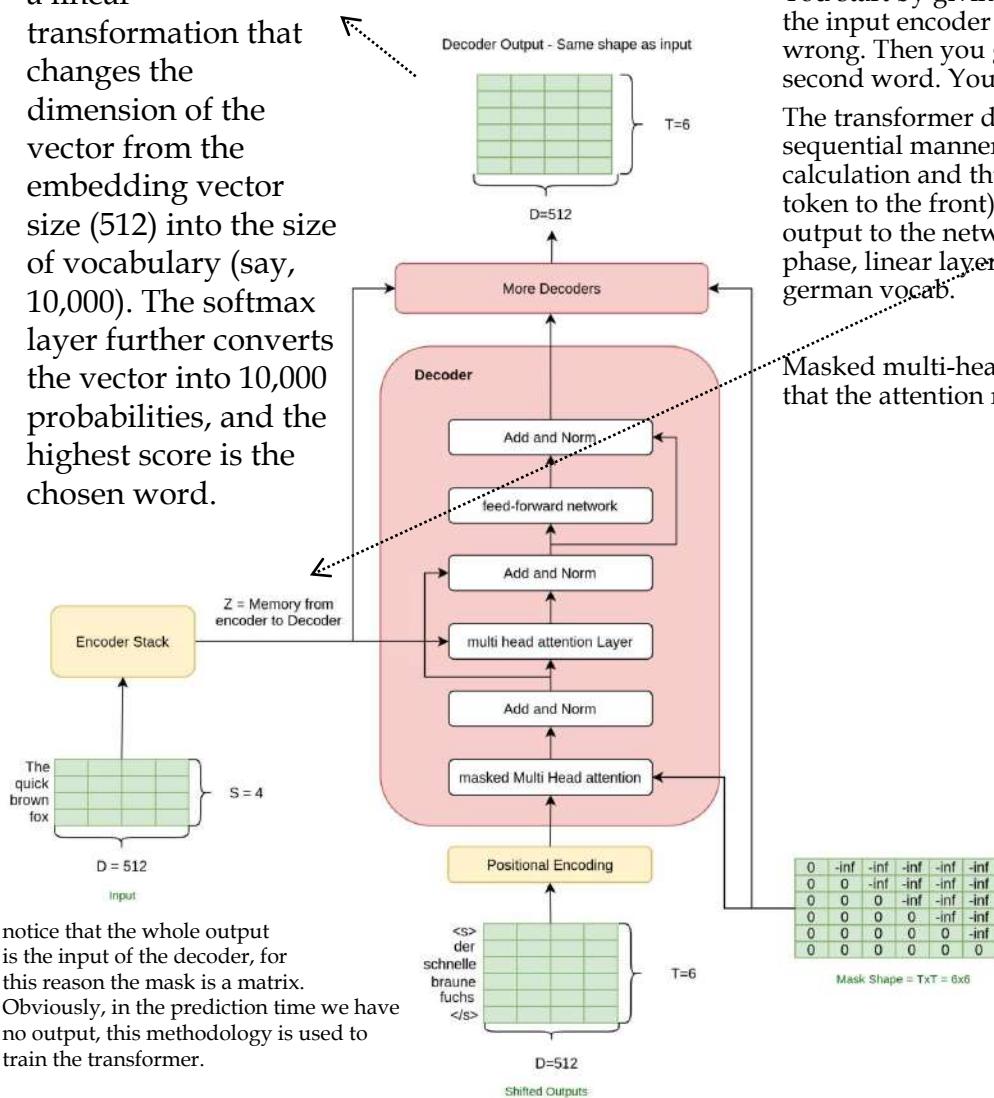
The decoder block appears again 6 times. The final output is transformed through a final linear layer and the output probabilities are calculated with the standard softmax function.



note that only for the training set you know the output

# The input is the output (?!?)

Then, the output vector from the decoder goes through a linear transformation that changes the dimension of the vector from the embedding vector size (512) into the size of vocabulary (say, 10,000). The softmax layer further converts the vector into 10,000 probabilities, and the highest score is the chosen word.



notice that the whole output is the input of the decoder, for this reason the mask is a matrix. Obviously, in the prediction time we have no output, this methodology is used to train the transformer.

but do I see the output we need flowing into the decoder as input?

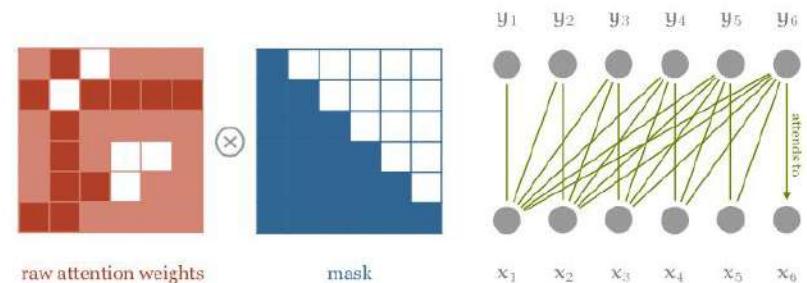
we can think of a transformer as a conditional language model in this case. A model that predicts the next word given an input word and an input encoder sentence (i.e. the ouput of the encoder) on which to condition upon or base its prediction on.

You start by giving the start token(<s>) and the model predicts the first word conditioned on the input encoder sentence. You change the weights based on if the prediction is right or wrong. Then you give the start token and the first word (<s> der) and the model predicts the second word. You change weights again. And so on.

The transformer decoder learns just like that but the beauty is that it doesn't do that in a sequential manner. It uses masking (see masked multihead in the next pages)) to do this calculation and thus takes the whole output sentence (although shifted right by adding a <s> token to the front) while training. Also, please note that at prediction time we won't give the output to the network, i.e. "prediction is right or wrong" can be made only in the training phase, linear layers and softmax on top to get the probability across all the words in the, e.g., german vocab.

Masked multi-head attention means the multi-head attention receives inputs with masks so that the attention mechanism does not use information from the hidden (masked) positions.

To use self-attention as an autoregressive model, we'll need to ensure that it cannot look forward into the sequence. We do this by applying a mask to the matrix of dot products, before the softmax is applied. This mask disables all elements above the diagonal of the matrix.



Masking the self attention, to ensure that elements can only attend to input elements that precede them in the sequence. Note that the multiplication symbol is slightly misleading: we actually set the masked out elements (the white squares) to  $-\infty$ .

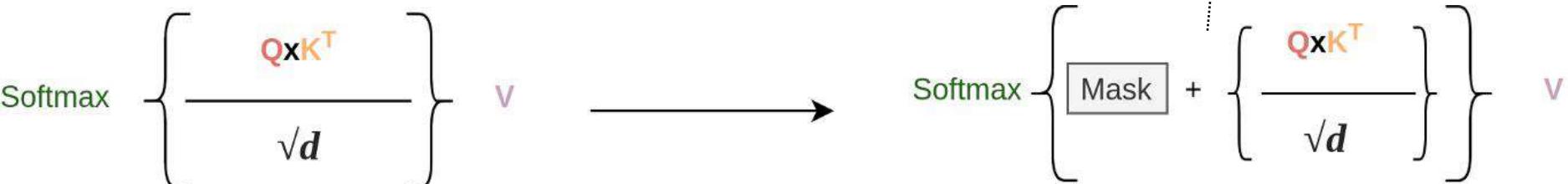
# Output

The output probabilities predict the next token in the output sentence. How? In essence, we assign a probability to each word simply keep the one with the highest score.

While most concepts of the decoder are already familiar, there are two more that we need to discuss. Let's start with the **Masked** multi-head self-attention layer.

## Masked Multi-head attention

In case you haven't realized, in the decoding stage, we predict one word (token) after another. In such NLP problems like machine translation, sequential token prediction is unavoidable. As a result, the self-attention layer needs to be modified in order to consider only the output sentence that has been generated so far.



we have a Masked Multi-Head attention Layer in our decoder, the aim of the matrix M is to mask our output (that is the input to the decoder) in a way that the network is never able to see the subsequent words since, otherwise, it can easily copy that word while training.

Mathematically we have:

$$\text{MaskedAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T + \mathbf{M}}{\sqrt{dk}}\right)\mathbf{V}$$

where the matrix M (mask) consists of zeros and -inf.

Zeros will become ones with the exponential while infinities become zeros.

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

$\exp(0)=1$   
 $\exp(-\infty)=0$

This effectively has the same effect as removing the corresponding connection. The remaining principles are exactly the same as the encoder's attention. And once again, we can implement them in parallel to speed up the computations.

Obviously, the mask will change for every new token we compute.

these two representations are equivalent since the elements of M are zero or infinite.

V

$$\text{Shape} = (Txn) \times (nxn) \times (Txn) = Txd = 6 \times 64$$

# Masked multi-head, some examples

**Masked multi-head attention** means the multi-head attention receives inputs with masks so that the attention mechanism does not use information from the hidden (masked) positions. The paper mentions that they used the mask inside the attention calculation by setting attention scores to negative infinity (or a very large negative number). The softmax within the attention mechanisms effectively assigns zero probability to masked positions.

Intuitively, it is as if we were gradually increasing the visibility of input sentences by the masks:

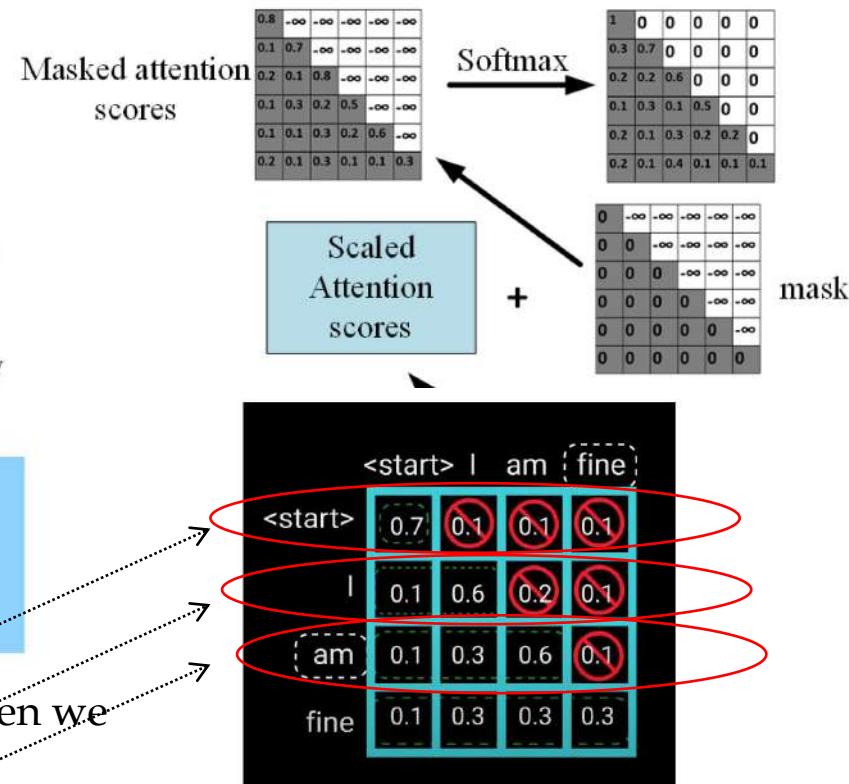
```
(1, 0, 0, 0, 0, ..., 0) => (<SOS>)
(1, 1, 0, 0, 0, ..., 0) => (<SOS>, 'Bonjour')
(1, 1, 1, 0, 0, ..., 0) => (<SOS>, 'Bonjour', 'le')
(1, 1, 1, 1, 0, ..., 0) => (<SOS>, 'Bonjour', 'le', 'monde')
(1, 1, 1, 1, 1, ..., 0) => (<SOS>, 'Bonjour', 'le', 'monde', '|')
```

- Notice that the mask will change for every new token we compute.
- e.g.

mask for the first token

mask for the second token

mask for the third token

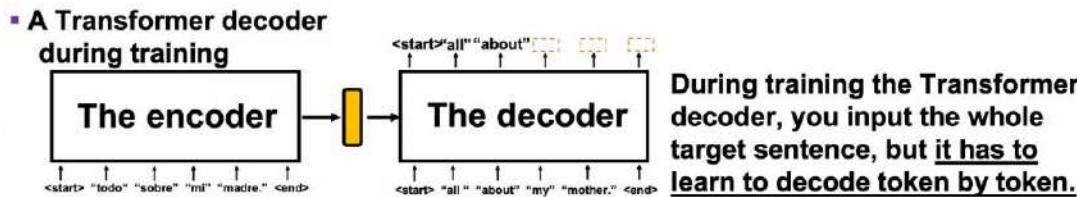


A depiction of Decoder's first Multi-headed Attention scaled attention scores. The word "am", should not any values for the word "fine". This is true for all other words.



# Masked multi-head

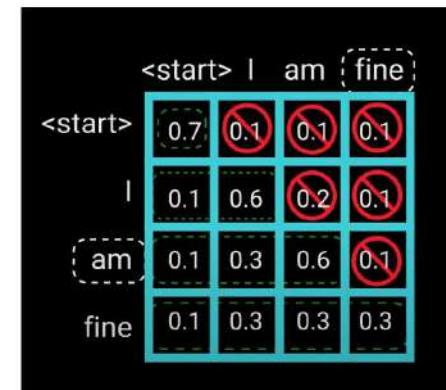
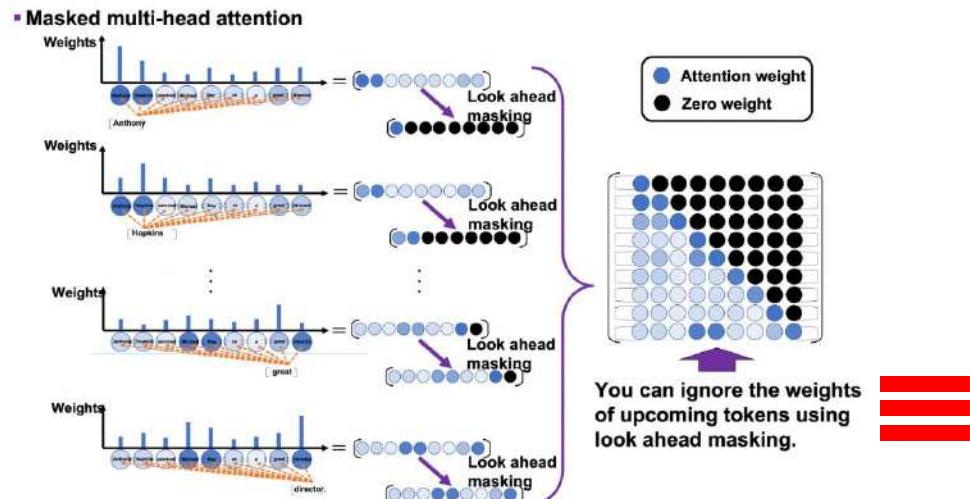
During training Transformer decoders, you input the whole sentence at once. That means Transformer decoders can see the whole sentence during training. That is as if a student preparing for a translation test could look at the whole answer sentences. It is easy to imagine that you cannot prepare for the test effectively if you study this way. Transformer decoders also have to learn to decode only based on the tokens they have generated so far.



During training you have to hide upcoming tokens in target phrases. In this example the next token to predict is "my", if you don't mask "my" in the decoder input it will be used in the prediction clearly distorting/biasing the results

In order to properly train a Transformer-based translator to learn such decoding, you have to hide the upcoming tokens in target sentences during training. During calculating multi-head attentions in each Transformer layer, if you keep ignoring the weights from up coming tokens like in the figure below, it is likely that Transformer models learn to decode only based on the tokens generated so far.

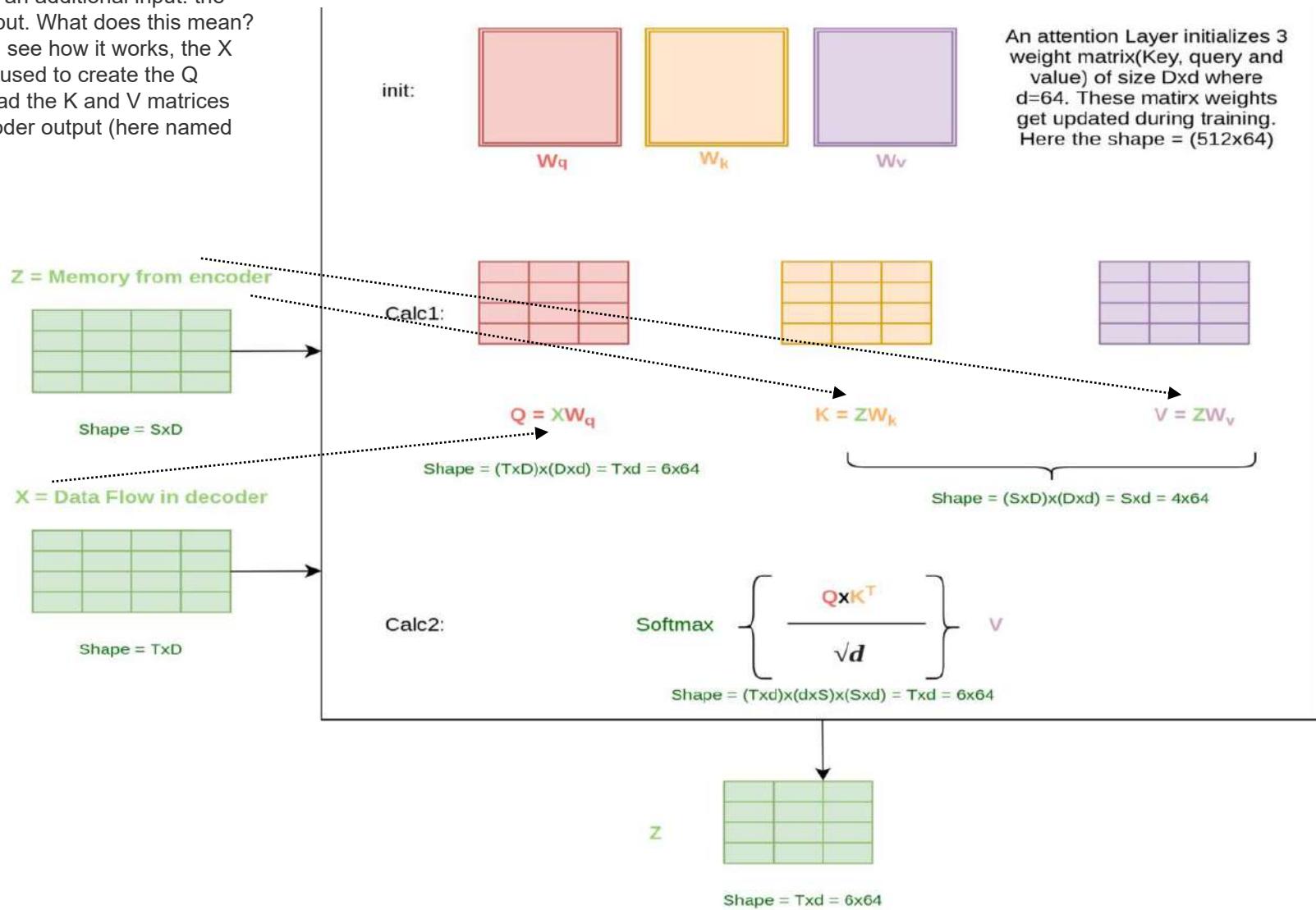
Let's take an example of calculating self attentions of an input "Anthony Hopkins admired Michael Bay as a great director." Also in this case you just calculate multi-head attention as usual, but when you get the histograms below, you apply look ahead masking to each histogram and delete the weights from the future tokens. In the figure below the black dots denote zero, and the sum of each row of the resulting attention map is also one.



A depiction of Decoder's first Multi-headed Attention scaled attention scores. The word "am", should not have any values for the word "fine". This is true for all other words.

# Decoder multi-head attention

The other attention layer of the decoder has an additional input: the encoder output. What does this mean? Here we can see how it works, the X input is only used to create the Q matrix, instead the K and V matrices use the encoder output (here named Z).



# Encoder-Decoder Attention

Encoder-Decoder attention: where the magic happens

This is actually where the decoder processes the encoded representation. The attention matrix generated by the encoder is passed to another attention layer alongside the result of the previous Masked Multi-head attention block.

The intuition behind the encoder-decoder attention layer is to combine the input and output sentence. The encoder's output encapsulates the final embedding of the input sentence. It is like our database. So **we will use the encoder output to produce the Key and Value matrices**. On the other hand, the output of the Masked

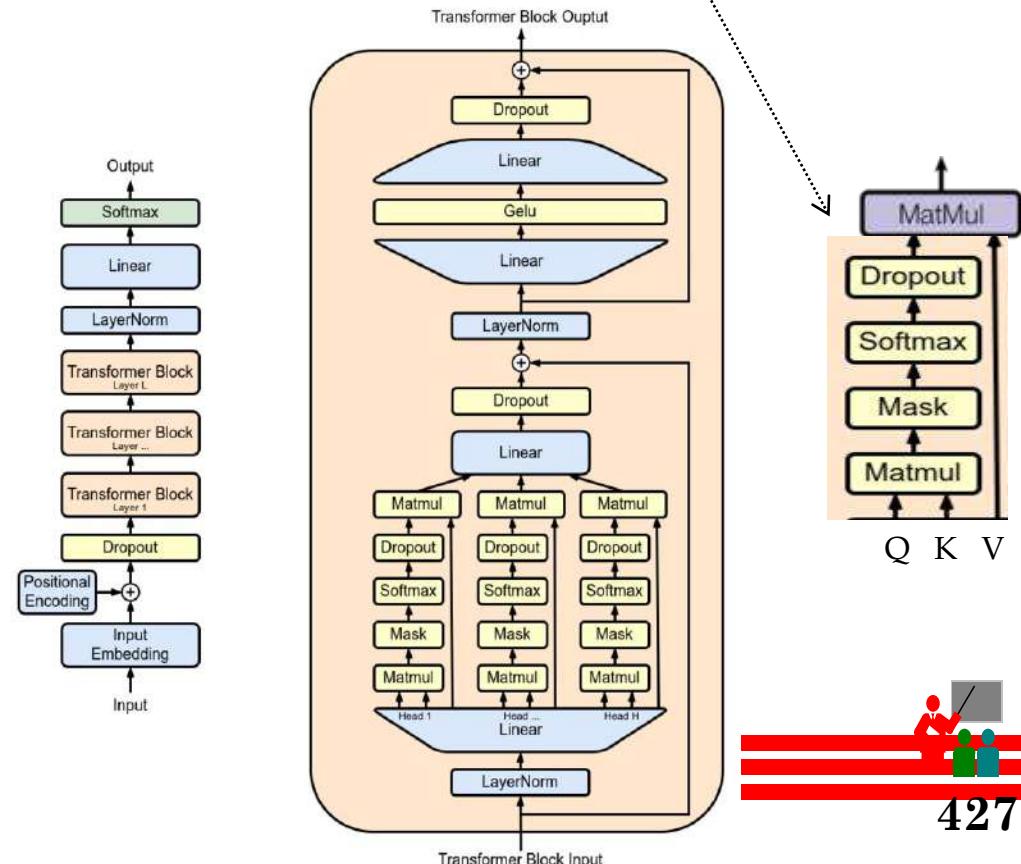
Multi-head attention block contains the so far generated new sentence and is represented as the Query matrix in the attention layer. Again, it is the “search” in the database.

The encoder-decoder attention is trained to associate the input sentence with corresponding output word.

It will eventually determine how related each English word is with respect to the French words. This is essentially where the mapping between English and French is happening.

Notice that the output of the last block of the encoder will be used in each decoder block.

Modern Language Models as GPT-3 use decoder-only transformer architectures, which apply a sequence of layers consisting of masked self-attention and feed forward transformations to the model's input. These architectures are explained in the natural language processing course.  
<https://towardsdatascience.com/language-model-scaling-laws-and-gpt-3-5cdc034e67bb>  
<https://towardsdatascience.com/large-language-models-gpt-1-generative-pre-trained-transformer-7b895f296d3b>





# Vision Transformer

Submitted on 22 Oct 2020  
<https://arxiv.org/abs/2010.11929>

First introduced in [An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale](#)

Vision Transformers (ViTs) have taken computer vision by storm, leading to hundreds of citations in the span of a few months. The paper's main goal was to show that a vanilla Transformer, once adapted to deal with data from the visual domain, could compete with some of the most performant convolutional neural networks (CNNs) developed up to that point.

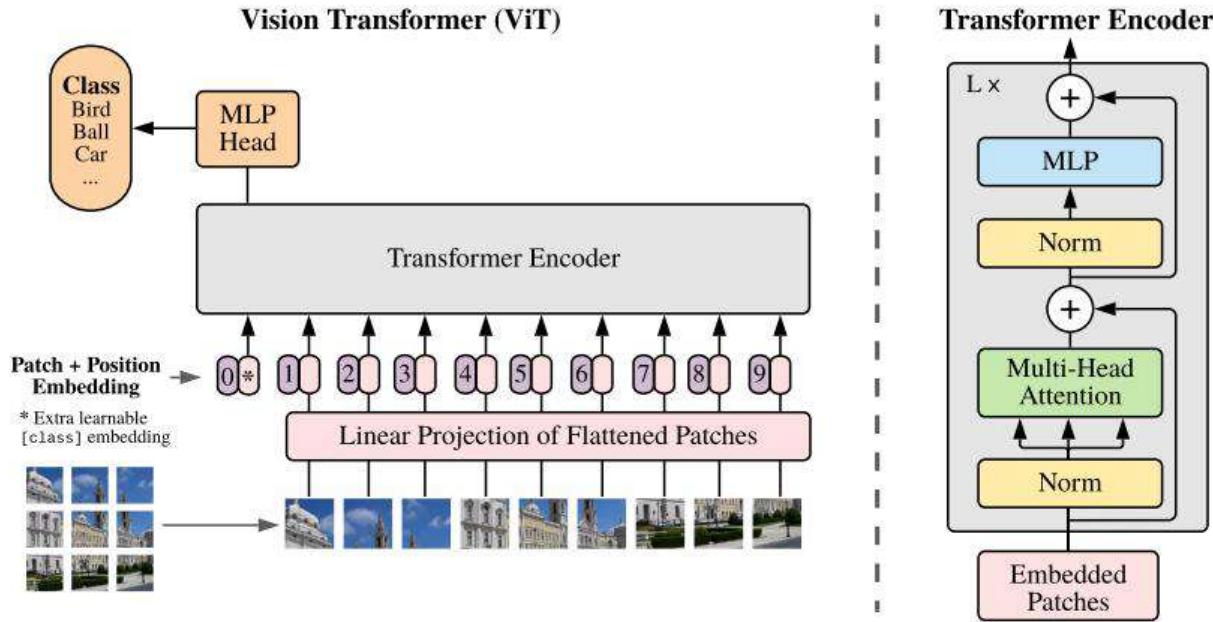
The Vision Transformer architecture is conceptually simple: divide the image into patches, flatten and project them into a  $D$ -dimensional embedding space obtaining the so-called patch embeddings, add positional embeddings (a set of learnable vectors allowing the model to retain positional information) and concatenate a (learnable) class token, then let the Transformer encoder do its magic. Finally, a classification head is applied to the class token to obtain the model's logits.

The model's performance was acceptable when trained on ImageNet (1M images), great when pre-trained on ImageNet-21k (14M images), and state-of-the-art when pre-trained on Google's internal JFT-300M dataset (300M images).

The striking performance improvement was due to the reduced inductive bias that characterizes Vision Transformers. By making fewer assumptions about the data, Vision Transformers could better adapt themselves to the given task. However, this ability came at a cost – when the sample size was too small (such as in the ImageNet case), the models overfit, resulting in degraded performance.

The goal for many follow-up papers would be that of matching (and surpassing) the performance of the best convolutional models in the “small” data regime – ImageNet (which is after all over a million images) and below.

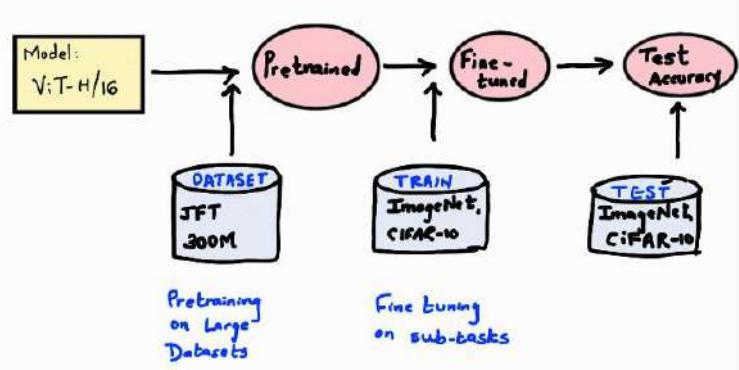
# Vision Transformer



Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable “classification token” to the sequence.

The total architecture is called Vision Transformer (ViT in short). Let's examine it step by step:

- Split an image into patches
- Flatten the patches
- Produce lower-dimensional linear embeddings from the flattened patches
- Add positional embeddings
- Feed the sequence as an input to a standard transformer encoder
- Pretrain the model with image labels (fully supervised on a huge dataset)



# ViT

In fact, the encoder block is similar to the original transformer. The only thing that changes is the number of those blocks. To this end, and to further prove that with more data they can train larger ViT variants, 3 models were proposed:

Model	Layers	Hidden size $D$	MLP size	Heads	Params
ViT-Base	12	768	3072	12	86M
ViT-Large	24	1024	4096	16	307M
ViT-Huge	32	1280	5120	16	632M

Hidden size  $D$  is the embedding size, which is kept fixed throughout the layers. Why keep it fixed? So that we can use short residual skip connections.

In case you missed it, there is no decoder in the game. Just an extra linear layer for the final classification called MLP head.

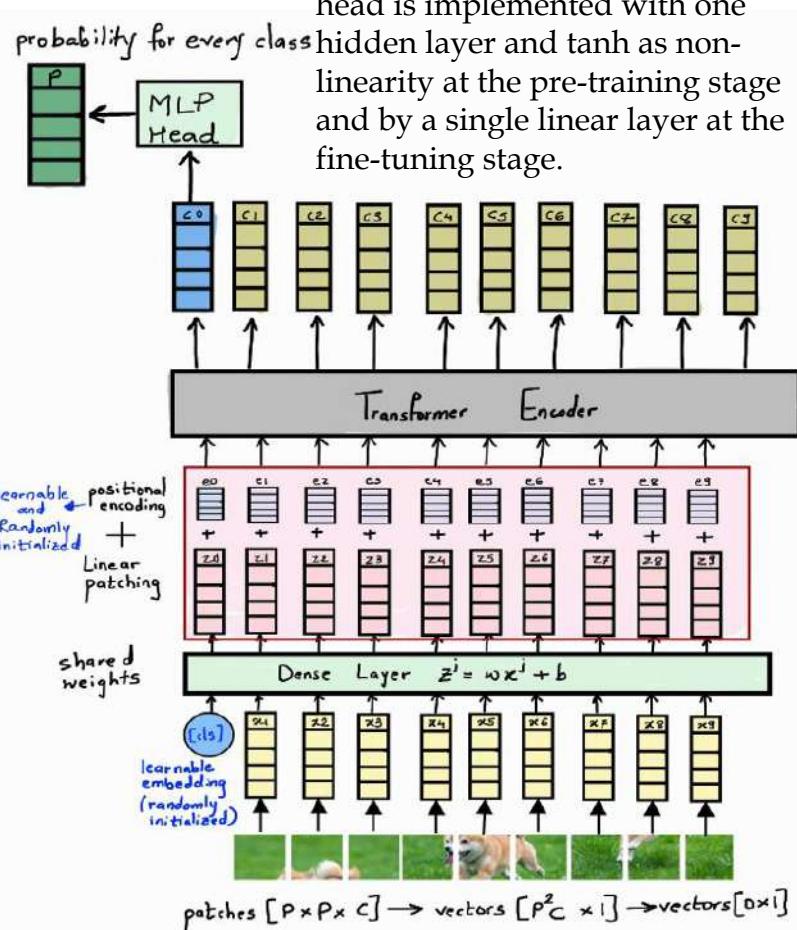
But is this enough?

Yes and no. Actually, we need a massive amount of data and as a result computational resources.

ViT is pretrained on the large dataset and then fine-tuned to small ones. The only modification is to discard the prediction head (MLP head) and attach a new linear layer to classify the pattern of the small dataset.

It is interesting that the authors claim that it is better to fine-tune at higher resolutions than pre-training. To fine-tune in higher resolutions, 2D interpolation of the pre-trained position embeddings is performed. The reason is that they model positional embeddings with trainable linear layers. Having that said, the key engineering part of this paper is all about feeding an image in the transformer.

Once we have our context vector  $C$ , we are only interested in the context token  $c_0$  for classification purposes. This context token  $c_0$  is passed through an MLP head to give us the final probability vector to help predict the class. The MLP head is implemented with one hidden layer and tanh as non-linearity at the pre-training stage and by a single linear layer at the fine-tuning stage.



## Training data-efficient image transformers & distillation through attention

was the first paper to show that models based on ViTs could be competitive on ImageNet without access to additional data.

# ViT - feeding image in the transformer

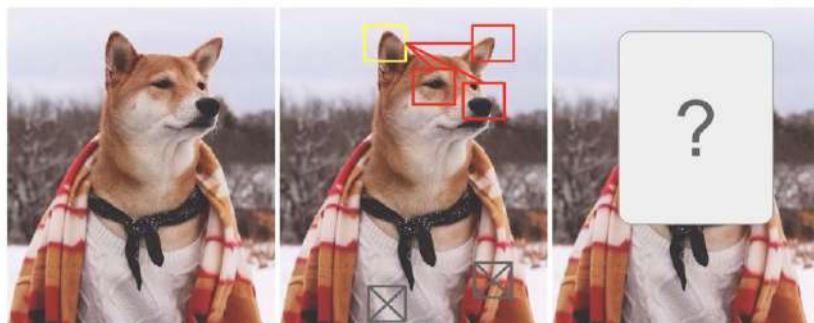
the image patch, i.e. [16,16,3] is flattened to 16x16x3. And what about going from patch to embeddings? the image patches are linearly projected into a vector using a learned embedding. The flattened patches is multiplied with embedding tensor of shape ( $P^2 * C \times D$ ), ( $P^2 * C$  is the length of the flattened image patch). The final embedded patches is now having shape of  $(1 \times D)$ . D is the model dimension.

E.g.  $P=16; C=3; D=15; f=\text{rand}(P^2 * P^2 * C, 1)'; W=\text{rand}(P^2 * P^2 * C, D); \text{size}(f^*W) = 15$

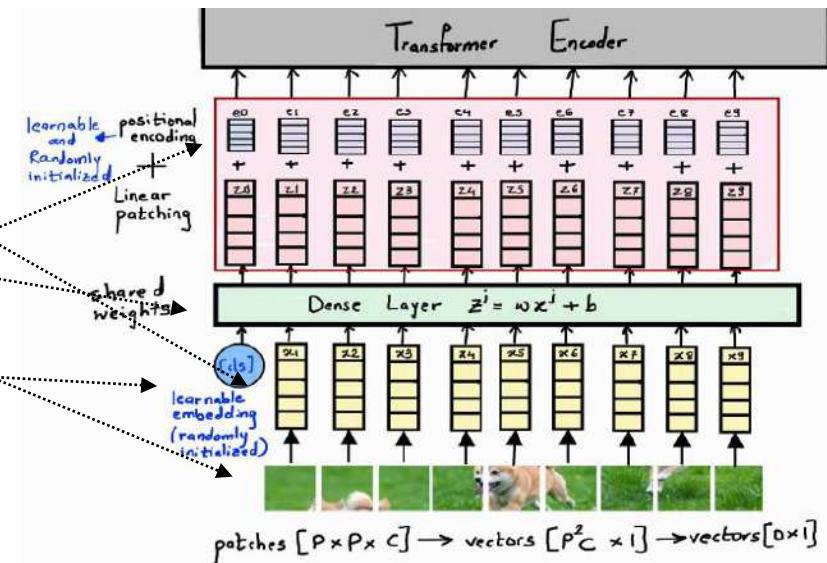
A cls token is prepended to the sequence of patch embeddings.

Now, we have to retain the positional information, before pass the data to the transformer encoder. To do that, the number of sequence of the image is passed to the encoder. They are just a learnable vector. Each of the positional embedding vector is parameterized and they will form a learnable positional embedding table. Positional embedding ( $E_{pos}$ ) is added to the sequence. It learns the positional information for each of the patches.

One of the interesting things about the Vision Transformer is that the architecture uses Class Tokens. These learnable classification Class Tokens (they are a set of parameters learned during the training step) are randomly initialized tokens that are prepended to the beginning of your input sequence. What is the reason for this Class Token and what does it do? Note that the Class Token is randomly initialized so it doesn't contain any useful information on its own. However, the Class Token is able to accumulate information from the other tokens in the sequence the deeper and more layers the Transformer is.



The name of the dog is Shiba Inu



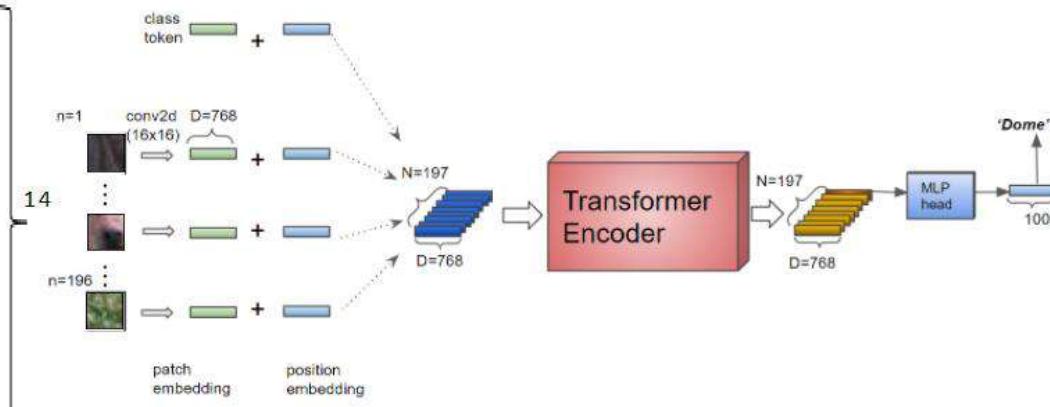
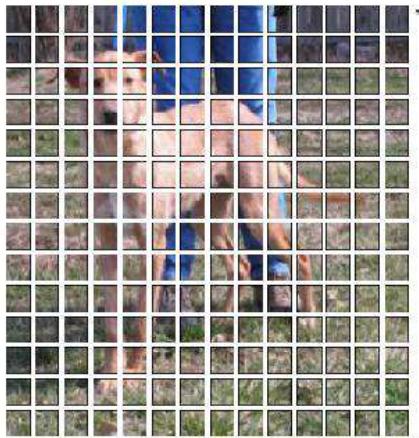
If we focus at the features of the dog in the red boxes, like his nose, right pointy ear and mystery eyes, we'll be able to guess what should come in the yellow box. However, by just looking at the pixels in the gray boxes, you won't be able to predict what should come in the yellow box. The attention mechanism weighs the pixel in the correct boxes more w.r.t the pixel in the yellow box. While the pixel in the gray boxes would be weighed less.

# Other interesting papers:

14

a recent survey is <https://arxiv.org/pdf/2203.01536.pdf> Augst 2022

survey on semantic segmentation using vision transformers <https://arxiv.org/pdf/2305.03273.pdf>  
ensemble of transformers for semantic segmentation <https://www.mdpi.com/2078-2489/14/12/657>  
ensemble of transformers and CNN for image classification <https://ieeexplore.ieee.org/document/10309107>



The input image is split into  $N$  patches ( $N = 14 \times 14$  vectors for ViT-Base) with dimension of 768 embedding vectors by learnable Conv2d ( $k=16 \times 16$ ) with stride=(16, 16).

<https://arxiv.org/pdf/2111.05464.pdf>

Transformer emerges as a powerful tool for visual recognition. In addition to demonstrating competitive performance on a broad range of visual benchmarks, recent works also argue that Transformers are much more robust than Convolutions Neural Networks (CNNs). Nonetheless, surprisingly, we find these conclusions are drawn from unfair experimental settings, where Transformers and CNNs are compared at different scales and are applied with distinct training frameworks. In this paper, we aim to provide the first **fair & in-depth** comparisons between Transformers and CNNs, focusing on robustness evaluations.

With our unified training setup, we first challenge the previous belief that Transformers outshine CNNs when measuring *adversarial robustness*. More surprisingly, we find CNNs can easily be as robust as Transformers on defending against adversarial attacks, if they properly adopt Transformers' training recipes. While regarding *generalization on out-of-distribution samples*, we show pre-training on (external) large-scale datasets is not a fundamental request for enabling Transformers to achieve better performance than CNNs. Moreover, our ablations suggest such stronger generalization is largely benefited by the Transformer's self-attention-like architectures per se, rather than by other training setups. We hope this work can help the community better understand and benchmark the robustness of Transformers and CNNs. The code and models are publicly available at <https://github.com/ytongbai/ViT-s-vs-CNNs>.

<https://arxiv.org/pdf/2112.04035.pdf>

Many deep neural network architectures loosely based on brain networks have recently been shown to replicate neural firing patterns observed in the brain. One of the most exciting and promising novel architectures, the Transformer neural network, was developed without the brain in mind. In this work, we show that transformers, when equipped with recurrent position encodings, replicate the precisely tuned spatial representations of the hippocampal formation; most notably place and grid cells. Furthermore, we show that this result is no surprise since it is closely related to current hippocampal models from neuroscience. We additionally show the transformer version offers dramatic performance gains over the neuroscience version. This work continues to bind computations of artificial and brain networks, offers a novel understanding of the hippocampal-cortical interaction, and suggests how wider cortical areas may perform complex tasks beyond current neuroscience models such as language comprehension.



# *Explainable AI*

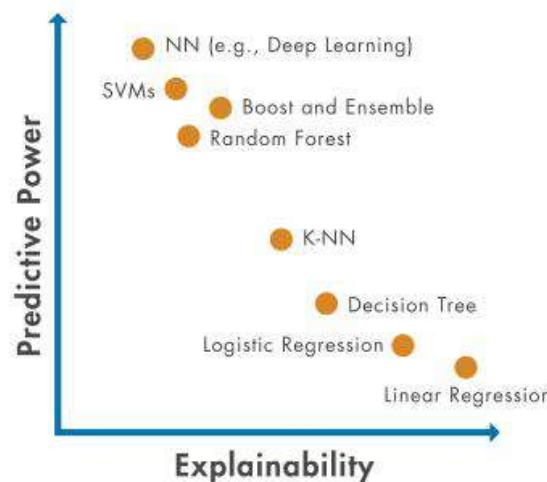
<https://www.vice.com/en/article/y3pezm/scientists-increasingly-can't-explain-how-ai-works>



# Explainable AI

Interpretability is the degree to which machine learning algorithms can be understood by humans. Machine learning models are often referred to as “black box” because their representations of knowledge are not intuitive, and as a result, it is often difficult to understand how they work. Interpretability techniques help to reveal how black-box machine learning models make predictions.

By revealing how various features contribute (or do not contribute) to predictions, interpretability techniques can help you validate that the model is using appropriate evidence for predictions, and find biases in your model that were not apparent during training. Some machine learning models, such as linear regression, decision trees, and generative additive models are inherently interpretable. However, interpretability often comes at the expense of power and accuracy.



*Trade-off between model performance and explainability.*

Interpretability and explainability are closely related. Interpretability is used more often in the context of (classic) machine learning, while in the context of deep neural networks many use “AI explainability.”

# *Explainable AI*

Practitioners seek model interpretability for three main reasons:

**Debugging:** Understanding where or why predictions go wrong and running “what-if” scenarios can improve model robustness and eliminate bias.

**Guidelines:** Black-box models may violate corporate technology best practices and personal preference

**Regulations:** Some government regulations require interpretability for sensitive applications such as in finance, public health, and transportation

Model interpretability addresses these concerns and increases trust in the models in situations where explanations for predictions are important or required by regulation.

Interpretability is typically applied at two levels:

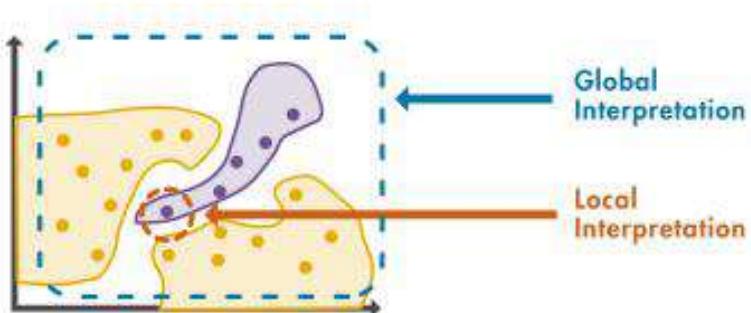
**Global Methods:** Provide an overview of the most influential variables in the model based on input data and predicted output

**Local Methods:** Provide an explanation of a single prediction result



# Explainable AI

illustrates the difference between the local and global scope of interpretability. You can also apply interpretability to groups within your data and arrive at conclusions at the group level, such as why a group of manufactured products were classified as faulty.



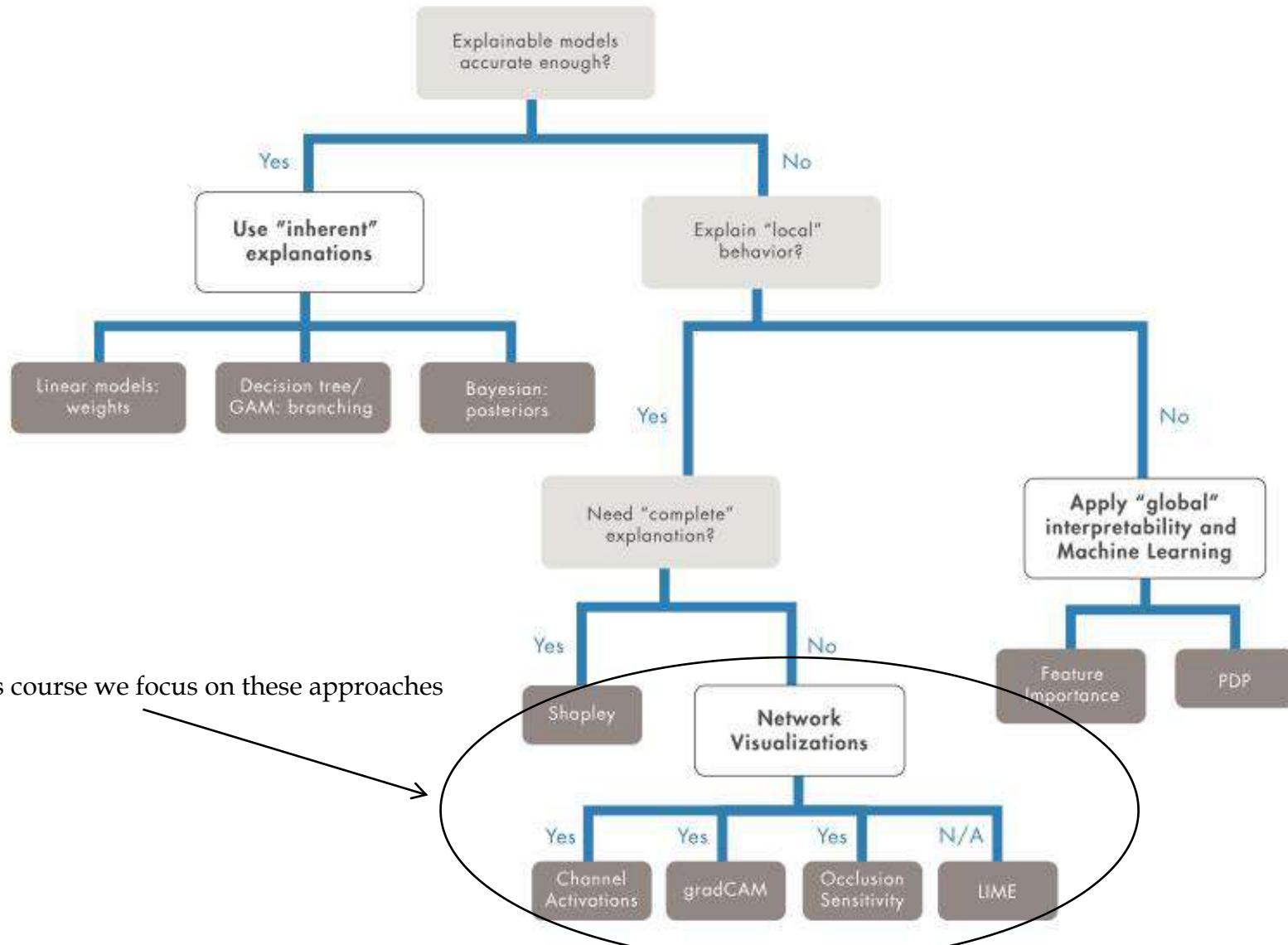
*Local versus global interpretability:*

*The two classes are represented by purple and orange dots.*

Popular techniques for local interpretability include Local Interpretable Model-Agnostic Explanations (LIME) and Shapley values. For global interpretability, many start with feature ranking (or importance)

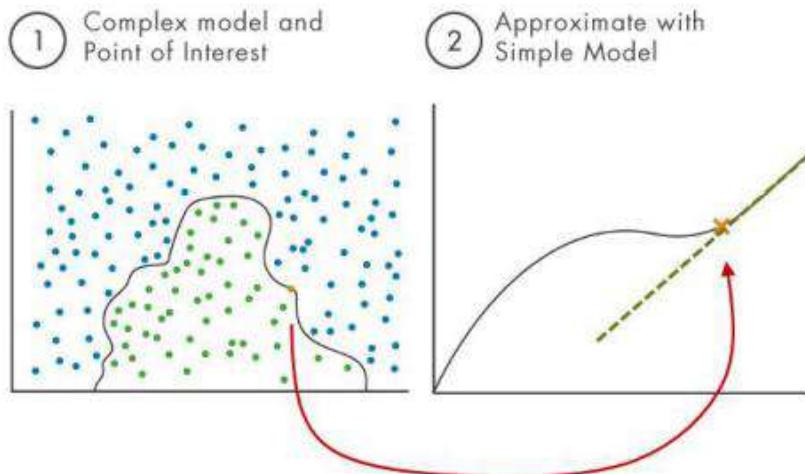


# Explainable AI



# LIME (*local approach*)

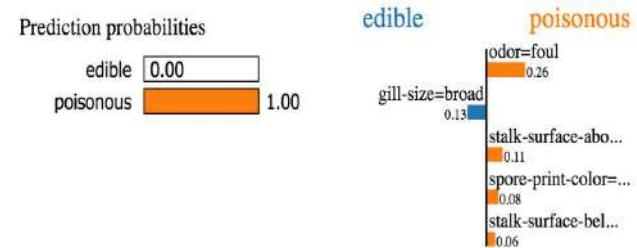
**Local Interpretable Model-Agnostic Explanations:** This approach involves approximating a complex model in the neighborhood of the prediction of interest with a simple interpretable model, such as a linear model or decision tree. You can then use the simpler model as a surrogate to explain how the original (complex) model works.



Strictly speaking, a partial dependence plot just shows that certain ranges in the value of a predictor are associated with specific likelihoods for prediction; that's not sufficient to establish a causal relationship between predictor values and prediction. However, if a local interpretability method like LIME indicates the predictor significantly influenced the prediction (in an area of interest), you can arrive at an explanation why a model behaved a certain way in that local area.

**Shapley Values:** This technique explains how much each predictor contributes to a prediction by calculating the deviation of a prediction of interest from the average.

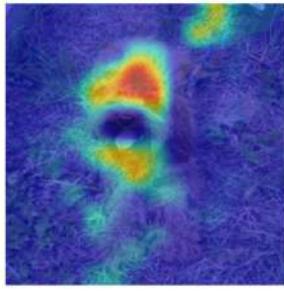
The output of LIME is a list of explanations, reflecting the contribution of each feature to the prediction of a data sample. This provides local interpretability, and it also allows to determine which feature changes will have most impact on the prediction. E.g. in the following example the pattern (e.g. food) is classified as "poisonous" since odor=foul



The interpretation of the Shapley value for feature value  $j$  is: The value of the  $j$ -th feature contributed  $\phi_j$  to the prediction of this particular instance compared to the average prediction for the dataset.



# Grad-CAM



Gradient-weighted class activation mapping (Grad-CAM) is an explainability technique that can be used to help understand the predictions made by a deep neural network. Grad-CAM, a generalization of the CAM technique, determines the importance of each neuron in a network prediction by considering the gradients of the target flowing through the deep network.

Grad-CAM computes the gradient of a differentiable output, for example class score, with respect to the convolutional features in the chosen layer. The gradients are spatially pooled to find the neuron importance weights. These weights are then used to linearly combine the activation maps and determine which features are most important to the prediction.

Suppose you have an image classification network with output  $y^c$ , representing the score for class  $c$ , and want to compute the Grad-CAM map for a convolutional layer with  $k$  feature maps (channels),  $A_{i,j}^k$ , where  $i,j$  indexes the pixels. The neuron importance weight is

$$\alpha_k^c = \underbrace{\frac{1}{N} \sum_i \sum_j}_{\substack{\text{Global average pooling} \\ \text{Gradients} \\ \text{via} \\ \text{backprop}}} \underbrace{\frac{\partial y^c}{\partial A_{i,j}^k}},$$

where  $N$  is the total number of pixels in the feature map. The Grad-CAM map is then a weighted combination of the feature maps with an applied ReLU:

$$M = \text{ReLU}\left(\sum_k \alpha_k^c A^k\right).$$

The ReLU activation ensures you get only the features that have a positive contribution to the class of interest. The output is therefore a heatmap for the specified class, which is the same size as the feature map. The Grad-CAM map is then upsampled to the size of the input data.

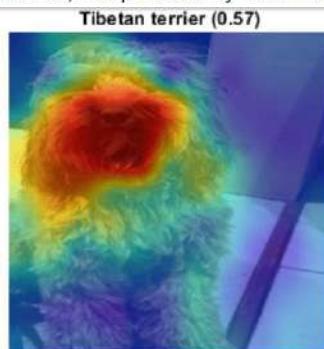
# Grad-CAM; Occlusion

Although Grad-CAM is commonly used for image classification tasks, you can compute a Grad-CAM map for any differentiable activation. For example, for semantic segmentation tasks, you can calculate the Grad-CAM map by replacing  $y^c$  with  $\sum_{(i,j) \in S} y_{ij}^c$ , where  $S$  is the set of pixels of interest and  $y_{i,j}^c$  is 1 if pixel  $(i,j)$  is predicted to be class  $c$ , and 0 otherwise

## Understand Network Predictions Using Occlusion

This example shows how to use occlusion sensitivity maps to understand why a deep neural network makes a classification decision. Occlusion sensitivity is a simple technique for understanding which parts of an image are most important for a deep network's classification. You can measure a network's sensitivity to occlusion in different regions of the data using small perturbations of the data. Use occlusion sensitivity to gain a high-level understanding of what image features a network uses to make a particular classification, and to provide insight into the reasons why a network can misclassify an image.

The `occlusionSensitivity` function perturbs small areas of the input by replacing it with an occluding mask, typically a gray square. The mask moves across the image, and the change in probability score for a given class is measured as a function of mask position. You can use this method to highlight which parts of the image are most important to the classification: when that part of the image is occluded, the probability score for the predicted class will fall sharply.

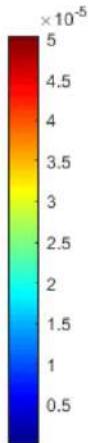


Again, the network strongly associates the dog's nose and mouth with the `Tibetan terrier` class. This highlights a possible failure mode of the network, since it suggests that images of Laika's face will consistently be misclassified as `Tibetan terrier`.

You can use the insights gained from the `occlusionSensitivity` function to make sure your network is focusing on the correct features of the input data. The cause of the classification problem in this example is that the available classes of GoogleNet do not include cross-breed dogs like Laika. The occlusion map demonstrates why the network is confused by these images of Laika. It is important to be sure that the network you are using is suitable for the task at hand.

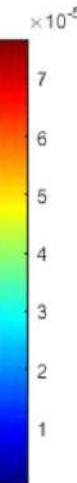


# LIME



Check the example code reported in the next page (link) to better understand how image LIME works.

The maps shows which areas of the image are important to the classification of `golden retriever`. Red areas of the map have a higher importance – when these areas are removed, the score for the `golden retriever` class goes down. The network focuses on the dog's face and ear to make its prediction of golden retriever. This is consistent with other explainability techniques like occlusion sensitivity or Grad-CAM.



For the `Labrador retriever` class, the network is more focused on the dog's nose and eyes, rather than the ear. While both maps highlight the dog's forehead, the network has decided that the dog's ear and neck indicate the `golden retriever` class, while the dog's eye and nose indicate the `Labrador retriever` class.





*Let's see an example (ImageLIME):*  
*Matlab\_Examples\ExplainableAI*





# *Quantum Deep Learning*

*interesting online course: <https://www.youtube.com/c/qiskit>*





# *Quantum Deep Learning*

- In the case of Quantum Computers, there is a particular behavior that governs the system; namely, quantum physics. Within quantum physics, we have a variety of tools that are used to describe the interaction between different atoms. In the case of Quantum Computers, these atoms are called "qubits" (we will discuss that in detail later). A qubit acts as both a particle and a wave. A wave distribution stores a lot of data, as compared to a particle (or bit).
- Loss functions are used to keep a check on how accurate a machine learning solution is. While training a machine learning model and getting its predictions, we often observe that predictions are not correct. The loss function is represented by some mathematical expression, the result of which shows by how much the algorithm has missed the target.
- A Quantum Computer also aims to reduce the loss function. It has a property called Quantum Tunneling which searches through the entire loss function space and finds the value where the loss is lowest, and hence, where the algorithm will perform the best and at a very fast rate.





# Bra-ket Notation

- In quantum mechanics and quantum physics, the “Bra-ket” notation or “Dirac” notation is used to write equations.
- The notation uses angle brackets,  $\langle \rangle$ , and a vertical bar,  $|$ , to construct “bras” and “kets”.
- A “ket” looks like this:  $|v\rangle$ . Mathematically it denotes a vector,  $v$ , in a complex vector space  $V$ . Physically, it represents the state of a quantum system.
- A “bra” looks like this:  $\langle f|$ . Mathematically, it denotes a linear function  $f: V \rightarrow C$ , i.e. a linear map that maps each vector in  $V$  to a number in the complex plane  $C$ .
- a linear function  $\langle f|$  act on a vector  $|v\rangle$  is written as:  
 $\langle f|v\rangle \in C$
- Wave functions and other quantum states can be represented as vectors in a complex state using the Bra-ket notation. Quantum Superposition can also be denoted by this notation.





# *qubits*

- Quantum Computing uses “qubits” instead of “bits”, which are used by classical computers. A bit refers to a binary digit, and it forms the basis of classical computing. The term “qbit” / “qubit” stands for Quantum Binary Digit. While bits have only two states – 0 and 1 – qubits can have multiple states at the same time. The value ranges between 0 and 1.
- To better understand this concept, take the analogy of a coin toss. A coin has two sides, Heads (1) or Tails (0). While the coin is being tossed, we don't know which side it has until we stop it or it falls on the ground. Look at the coin toss. Can you tell which side it has? It shows both 0 and 1 depending on your perspective; only if you stop it to look does it show just one side. The case with qubits is similar.
- This is called the superposition of two states. This means that the probabilities of measuring a 0 or 1 are generally neither 0.0 nor 1.0. In other words, there is a likelihood that the qubit is in various states at once. In the case of the coin toss, when we get our result (heads or tails) the superposition collapses.



# Quantum Deep Learning

The *qubit* is the basic unit of information in quantum computing. The power of quantum computing over classical computing derives from the phenomena of *superposition* and *entanglement* exhibited by qubits. Unlike a classical bit which has a value of either 0 or 1, superposition allows for a qubit to exist in a combination of the two states. In general, a qubit is represented as:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

$|0\rangle$  and  $|1\rangle$  represent the two computational *basis* states,  $\alpha$  and  $\beta$  are complex amplitudes corresponding to each, satisfying  $|\alpha|^2 + |\beta|^2 = 1$ . *Observing* a qubit causes a collapse into one of the basis states. The probability of each state being observed is proportional to the square of the amplitude of its coefficient, i.e. the probabilities of observing  $|0\rangle$  and  $|1\rangle$  are  $|\alpha|^2$  and  $|\beta|^2$  respectively. A qubit is physically realizable as a simple quantum system, for example the two basis states may correspond to the horizontal and vertical polarization of a photon. Superposition allows quantum computing systems to potentially achieve exponential speedups over their classical counterparts, due to the parallel computations on the probabilistic combinations of states.

- The qbit  $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$  has a 100% chance of collapsing to 0, and  $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$  has a 100% chance of collapsing to 1

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

observing the qbit causes a collapse of its state to standard bit values

$$|00\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad |01\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad |10\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad |11\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad \text{See pg 461 for details about } |00\rangle \dots$$

An individual photon can be described as having right or left circular polarization, or a superposition of the two. Equivalently, a photon can be described as having horizontal or vertical linear polarization, or a superposition of the two.  
[https://en.wikipedia.org/wiki/Photon\\_polarization](https://en.wikipedia.org/wiki/Photon_polarization)



# Entanglement

Entanglement refers to the phenomenon by which qubits exhibit correlation with one another. In general, a set of  $n$  entangled qubits exist as a superposition of  $2^n$  basis states. Observing one or more qubits among them causes a collapse of their states, and alters the original superposition to account for the observed values of the qubits. For example, consider the 2-qubit system in the following initial state:

$$|\psi\rangle = \frac{1}{\sqrt{3}}|00\rangle + \frac{1}{\sqrt{3}}|01\rangle + \frac{1}{\sqrt{6}}|10\rangle + \frac{1}{\sqrt{6}}|11\rangle$$

Suppose a measurement of the first qubit yields a value of 0 (which can occur with probability  $\frac{2}{3}$ ). Then,  $\psi$  collapses into:

$$|\psi'\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|01\rangle$$

Note that the relative probabilities of the possible states are conserved, after accounting for the state collapse of the observed qubits.

$$(1/3^{0.5})^2 + (1/3^{0.5})^2 = 2/3$$

we consider the component where the first element (i.e. first qbit) is zero

notice that the probability “second bit = 0” is the same that “second bit = 1”

we know that 0 is the first qubit after the measurement, the  $2^0$  qubit has a probability of zero equal to  $\alpha^2 = (1/3^{0.5})^2 + (1/6^{0.5})^2 = 0.5 \rightarrow \alpha = (1/2)^{0.5}$  the same for the probability of one

# QBit

A QuBit is the basic unit of quantum information – the quantum version of the classical binary bit physically realised with a two-state device. A QuBit is a two-level quantum-mechanical system. The below figure shows the main differences between a Bit and a QuBit. If with normal bits it is simply the number of representations we can represent that increases exponentially, with qubits it is the size of the vector space that increases exponentially. This means that: if in an 8-bit system a value can be in ONLY ONE of the  $(2^n)$  states, in an 8-qubit system a value can be with  $(2^n)$  different probabilities in all  $(2^n)$  different states (one probability for each different state). This is why a quantum computer cannot be simulated on an ordinary computer: only systems of very few qubits could be simulated.

Bits : 0 or 1		Qubits : 0 and 1
<b>State</b>	two possible exclusive states	two simultaneous possible states
<b>Initialization</b>	0 or 1	 0 or 1
<b>Internal representation</b>	0 or 1	vector of 2 dimensions
<b>Internal dimensionality</b>	1 binary digit	two floating numbers
<b>Modifications</b>	logical gates	quantum gates
<b>Read</b>	0 or 1, deterministic	0 or 1, probabilistic

<https://quantumcomputing.stackexchange.com/questions/4242/how-do-we-physically-initialize-qubits-in-a-quantum-register>

Quantum gates are physical devices acting on the qubits of the quantum registers.

# QuBIts

Once we have got an idea about the difference between a Bit and a QuBit, we are going to explore the registers. Indeed, one Qubit can do nothing alone, we need more to be able to represent more data. Here is an example using a register of 4 QuBIts :

Register of 4 Bits

0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

This register could represent 16 states (numbers from 0 to 15)

N = 10 QuBIts

Classical computer

N = 40 QuBIts

The biggest computer that exist

N = 100 QuBIts

I let you guess it

Register of 4 QuBIts

$$|0000\rangle + |0001\rangle + |0010\rangle + |0011\rangle + \\|0100\rangle + |0101\rangle + |0110\rangle + |0111\rangle + \\|1000\rangle + |1001\rangle + |1010\rangle + |1011\rangle + \\|1100\rangle + |1101\rangle + |1110\rangle + |1111\rangle$$

- / This register could represent 16 states that are in a superposition → we have 16 states in the same time !
- / With this way we can develop 16 calculations in the same time
- / With 8 Qubits, we can develop 256 calculations in the same time ... **with N QuBIts we can have  $2^N$  states at the same time**

# Register of $n$ qubits

The three most important basic principles that are used in QuBits :

Superposition : allows to have QuBits that are both in a state 0 and 1.

Entanglement : The entanglement makes it possible to connect the QuBits to each other to synchronize them, which makes it possible in particular to make copies of them, but without being able to read their contents or modify them independently.

Wave-particle duality [https://en.wikipedia.org/wiki/Wave%20particle\\_duality](https://en.wikipedia.org/wiki/Wave%20particle_duality) : the wave-particle duality makes it possible to interact in some cases with the QuBits or to make the QuBits interact with each other by interference in the context of quantum algorithms.

Superposition → information in QuBits

Entanglement → connection between QuBits

Wave-particle duality → interference in QuBits

A "qubit" is an abstraction of a two-dimensional quantum system. When you talk about qubits, you are not making any reference to the actual physical substrate that you are modeling. Any two-dimensional quantum system is "a qubit". The "amplitudes" of a qubit are the numbers we use to describe its state. The "wave properties" in quantum mechanics are reflected in the phenomenon of interference. There is not, however, some sort of "quantifier of waveness" of a state. Interference is just a foundational property of quantum systems. In some contexts wave properties might be more visible than in others, but any quantum state will display interference/wave phenomena under suitable circumstances. The interferences cause the qubit to collapse.

observing the qbit causes a collapse of its state to standard bit values

e.g. it can store:

101

	register of $n$ bits	register of $n$ qubits	
101	→ 2 $^n$ possible states <b>One at a time</b>	2 $^n$ possible states <b>Simultaneously</b>	↑ 0 0 0 0 0 1 0 1 0 0 1 1 1 0 0 1 0 1 1 1 0 1 1 1
	evaluable	partially evaluable	
	independent copies	Incopiables independently	
	individually erasable	indelible individually	
	lecture non destructive	lecture modify the present value	
	Deterministic	Probabilistic	



# Quantum Decoherence

The superposition of qubits causes issues like Quantum Decoherence. These are unwanted collapses that happen randomly and naturally because of noise in the system. Ultimately, this leads to errors in computation. If you think that a qubit is in a superposition when it isn't, and we do an operation on it, it's going to give you a different answer than you might have expected. This is why we run the same program over and over again many times, similar to training a machine learning model.

## What Causes Quantum Decoherence?

Quantum systems need to be isolated from the environment, because contact with the environment is what causes quantum decoherence, even a single molecule can be an issue.

Qubits are chilled to near absolute zero. When qubits interact with the environment, information from the environment leaks into them, and information from within the qubits leaks out. The information that leaks out is most likely needed for a future or current computation, and the information that leaks in is random noise.

This concept is exactly like the Second Law of Thermodynamics, which states that:

"The total entropy of an isolated system can never decrease over time, and is constant if and only if all processes are reversible. Isolated systems spontaneously evolve towards thermodynamic equilibrium, the state with maximum entropy." Quantum gates have to be reversible (see pg 461) because quantum mechanics is reversible (and even more specifically it is unitary). It's just an observed fact about the universe. In quantum physics, unitarity is the condition that the time evolution of a quantum state according to the Schrödinger equation is mathematically represented by a unitary operator.

[https://en.wikipedia.org/wiki/Unitary\\_operator](https://en.wikipedia.org/wiki/Unitary_operator).

Thus, quantum systems need to be in a state of coherence. Quantum decoherence is more visible in minute particles as compared to bigger objects, like a book or a table. It is a fact that all materials have a particular wavelength associated with them (wave-particle duality of matter), but the bigger the item, the lesser its wavelength.



# Unitary operator

In linear algebra, a complex square matrix  $U$  is **unitary** if its conjugate transpose  $U^*$  is also its inverse, that is, if

$$U^*U = UU^* = UU^{-1} = I,$$

where  $I$  is the identity matrix.

In physics, especially in quantum mechanics, the conjugate transpose is referred to as the Hermitian adjoint of a matrix and is denoted by a dagger ( $\dagger$ ), so the equation above is written

$$U^\dagger U = UU^\dagger = I.$$

In mathematics, specifically in operator theory, each linear operator  $A$  on a Euclidean vector space defines a **Hermitian adjoint** (or **adjoint**) operator  $A^*$  on that space according to the rule

$$\langle Ax, y \rangle = \langle x, A^*y \rangle,$$

where  $\langle \cdot, \cdot \rangle$  is the inner product on the vector space.

The Euclidean space  $R^n$ , where the inner product is given by the dot product

$$\langle (x_1, x_2, \dots, x_n), (y_1, y_2, \dots, y_n) \rangle = x_1^*y_1 + x_2^*y_2 + \dots + x_n^*y_n$$

Remember that inner product is not always equal to the dot product

[https://www.youtube.com/watch?v=WC9YW1ya31o&ab\\_channel=ProfessorNanoScience](https://www.youtube.com/watch?v=WC9YW1ya31o&ab_channel=ProfessorNanoScience)



# Wave-particle duality of matter

Einstein initially assumed that photons had zero mass, which made them a peculiar sort of particle indeed. In 1905, however, he published his special theory of relativity, which related energy and mass according to the famous equation:

$$E = hu = h \frac{c}{\lambda} = mc^2$$

the frequency of a sinusoidal wave is equal to the phase velocity  $v$  of the wave divided by the wavelength  $\lambda$  of the wave, if  $v = c$ , where  $c$  is the speed of light in vacuum, we obtain  $u=c/\lambda$  <https://en.wikipedia.org/wiki/Frequency>

According to this theory, a photon of wavelength  $\lambda$  and frequency  $u$  has a nonzero mass, which is given as follows:

$$m = \frac{E}{c^2} = \frac{hu}{c^2} = \frac{h}{\lambda c}$$

That is, light, which had always been regarded as a wave, also has properties typical of particles, a condition known as wave-particle duality (a principle that matter and energy have properties typical of both waves and particles). Depending on conditions, light could be viewed as either a wave or a particle.

In 1922, the American physicist Arthur Compton (1892–1962) reported the results of experiments involving the collision of x-rays and electrons that supported the particle nature of light. At about the same time, a young French physics student, Louis de Broglie (1892–1972), began to wonder whether the converse was true: Could particles exhibit the properties of waves? In his PhD dissertation submitted to the Sorbonne in 1924, de Broglie proposed that a particle such as an electron could be described by a wave whose wavelength is given by

$$\lambda = \frac{h}{mv}$$

where

- $h$  is Planck's constant,
- $m$  is the mass of the particle, and
- $v$  is the velocity of the particle.

Calculate the wavelength of a baseball, which has a mass of 149 g and a speed of 100 mi/h.

Recall that the joule is a derived unit, whose units are  $(kg \cdot m^2)/s^2$ . Thus the wavelength of the baseball is

$$\lambda = \frac{6.626 \times 10^{-34} J \cdot s}{(0.149 \text{ kg})(44.69 \text{ m} \cdot \text{s})} = \frac{6.626 \times 10^{-34} \text{ kg} \cdot \text{m}^2 \cdot \text{s}^{-2} \cdot \text{s}}{(0.149 \text{ kg})(44.69 \text{ m} \cdot \text{s}^{-1})} = 9.95 \times 10^{-35} \text{ m}$$

Given that the diameter of the nucleus of an atom is approximately  $10^{-14}$  m, the wavelength of the baseball is almost unimaginably small.



# Quantum Decoherence

Qubits are created using specific atomic particles, or even photons, which are light waves

[https://inst.eecs.berkeley.edu/~cs191/fa09/lectures/lecture4\\_fa09.pdf](https://inst.eecs.berkeley.edu/~cs191/fa09/lectures/lecture4_fa09.pdf). An electron of phosphorus is an example. To begin the process of creating a qubit, the atoms are placed in a superconducting magnet. This forces the electron into what we call a spin-down position <http://hyperphysics.phy-astr.gsu.edu/hbase/spin.html> [https://en.wikipedia.org/wiki/Spin\\_\(physics\)](https://en.wikipedia.org/wiki/Spin_(physics)). In an environment with a normal temperature, the electron's spin direction is not stable, as heat produces energy, forcing the electron to move. The environment needs to be contained in a supercooled refrigerator to create stability. For now, the need for large cooling systems ensures that quantum computers will be quite big for some time. Now to get the electron to spin up, a pulse of microwaves must be fired at it. Stopping the microwaves at a point anywhere between spin-down and spin-up creates a superposition. It is now a stable, coherent qubit. Any changes to temperature, light, sound, vibration, and other external factors will impact the qubit state. The big technical challenge today is making those quantum states of matter in the laboratory, and ultimately in quantum hardware, so that we have intrinsically, inherently, much more coherent and more stable qubits. So, the art of having stable qubits needs to be mastered. Pairs of entangled photons in polarization can be created when a laser beam passes through a nonlinear crystal such as beta barium borate.

Another option: we can consider the system consisting of the hydrogen atom. In this system, the state  $|0\rangle$  of the qubit can be represented by the first energy level ( $n = 0$ ), corresponding to the base state of the electron, and the state  $|1\rangle$  by the second energy level ( $n = 1$ ), corresponding to the excited state of the electron. The transition of the electron from one state to the other can be accomplished by subjecting the electron to a laser pulse of appropriate intensity, duration and wavelength. In an atom, the energy levels of the various electrons are discrete. Two of them can be selected to represent logical values 0 and 1. These levels correspond to particular excited states of the electrons in the atom.

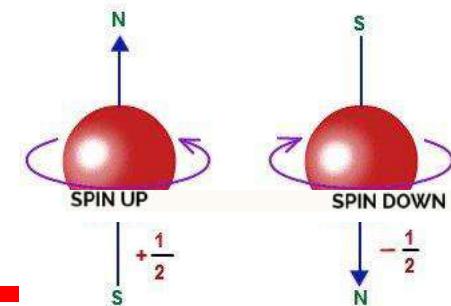
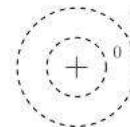
For details watch [https://www.youtube.com/watch?v=Re4l22ycc-k&ab\\_channel=QuTechAcademy](https://www.youtube.com/watch?v=Re4l22ycc-k&ab_channel=QuTechAcademy)

- The quantum computer has three main components:

Qubit area that houses the qubits;

A signal transfer method for transferring signals to the qubits;

Classical compute to execute a program that will send instructions.



# Quantum Entanglement

The idea of Quantum Entanglement refers to the idea that if we take two qubits, they are always in a superposition of two states. Here's an example. Suppose there is a box and inside of it, there is a pair of gloves. At random, one glove is taken out of the box. The box is then taken to a different room. The glove that was taken out was found to be right-handed, so we automatically know that the glove that is still inside the box is left-handed.

The same is the case with qubits. If one is in a spin-up position, then the other is automatically in the spin-down position. Like other, similar particles, electrons have a property called spin that can exist in one of two possible states (spin-up or spin-down). There does not exist a scenario where both the qubits are in the same state. In other words, they are always entangled. This is known as quantum entanglement. "spooky action at a distance": When measuring one of the entangled particles, instantaneously its state "collapses" and becomes definite. What is interesting is that, because of entanglement, the measurement instantaneously also affects the state of the other particle, regardless of the distance between them. While quantum entanglement can cause particles to collapse instantaneously over long distances, we can't use that to transport information faster than the speed of light. It turns out entanglement alone is not enough to send data. Quantum entanglement has no real equivalent in the classical world, so the best we can do is to use analogies. The best analogy I know of is this one: Imagine you have two ping-pong balls. You paint one red and one blue. You shuffle them behind your back and then seal them in two boxes. You put one box on a spaceship to Mars and the other on your desk.

When the spaceship arrives on Mars, open the box on your desk. You see a blue ping-pong ball. Immediately you know that the ping-pong ball on Mars is red. But here's something very, very important to understand this analogy: If you paint your blue ping-pong ball red, the one on Mars will NOT change from red to blue. The strange thing about quantum mechanics is the idea (and again this is a metaphor) that when you stick the two balls in a box, each is neither red nor blue. When you open the box on the desk, the ball inside turns blue and the one on Mars turns red. But again, painting your blue ball red will not turn the red ball on Mars blue.

Perhaps even stranger states are the three-qubit (or more) generalization <https://www.nature.com/articles/s41598-019-49805-7>

## Dual Principle

Qubits exhibit properties of both waves and particles. In fact, all objects do, but they can be observed more clearly in atomic-sized objects like the qubit. The wave-particle duality enables qubits to interact with each other by interference.

Quantum Coherence <https://www.qutube.nl/fundamentals-11/coherence-111> helps the quantum computer to process information in a way that classical computers cannot. A quantum algorithm performs a stepwise procedure to solve a problem, such as searching a database. It can outperform the best known classical algorithms. This phenomenon is called the Quantum Speedup.

If an object's wave-like nature is split in two, then the two waves may coherently interfere with each other in such a way as to form a single state that is a superposition of the two states. This concept of superposition is famously represented by Schrödinger's cat, which is both dead and alive at the same time when in its coherent state inside a closed box. Quantum coherence is based on the idea that all objects have wave-like properties. It's in many ways similar to the concept of quantum entanglement, which involves the shared states of two quantum particles instead of two quantum waves of a single particle. With that in mind, quantum coherence has any number of applications to the general concept of quantum mechanics. The superpositioning concept allows for the theoretical construction of a system built on qubits, which can have an array of values, either 1, or 0, or an indeterminate value. Quantum coherence is a desired property for a qubit. Its coherence time – the duration of the qubit coherence – is used to make quality comparisons between qubits. Coherence tells us something about how long a qubit retains its information, and thus dictates some sort of lifetime.



# Bloch's Sphere : a mathematical representation of the Qubit

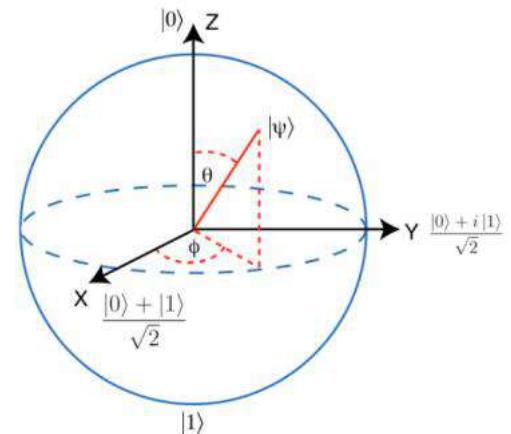
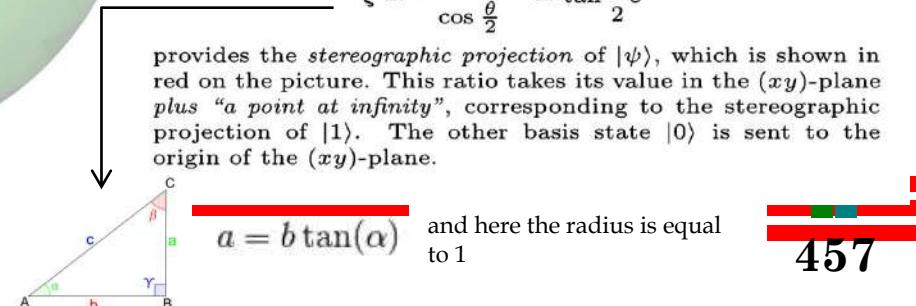
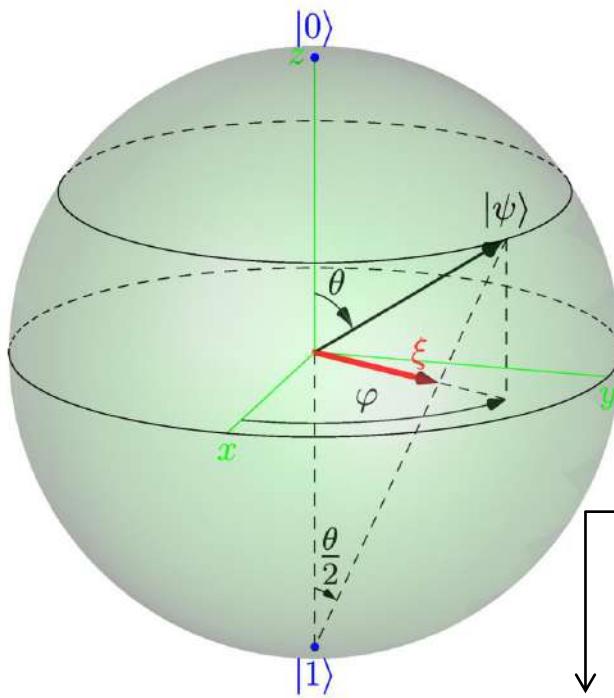
This model is linked to the representation of the state of a QuBit or of any quantum with two states by a two-dimensional vector whose so-called "norm" length is always 1. This vector has the particularity of having two elements: a real number  $\alpha$  and a complex number  $\beta$ .

Spherical Coordinates is a system defined by R (radius), theta and phi. In Bloch sphere, R is fixed to 1. Only Phi and Theta are variable. The qbit state function in the Bloch Sphere is defined as follows.

State of the qubit	Probability of the 0 state	Probability of the 1 state
$ \psi\rangle = \alpha 0\rangle + \beta 1\rangle$		
$ \alpha ^2 +  \beta ^2 = 1$		
The relationship between $\alpha$ and $\beta$ according to the <b>Max Born rule</b> , related to the <b>Schrodinger</b> wave function that defines the states $ 0\rangle$ and $ 1\rangle$		
$ \psi\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \xrightarrow{\text{Max Born rule}} \cos\left(\frac{\theta}{2}\right)$ $\xrightarrow{\text{Schrodinger wave function}} \exp(i\varphi)\sin\left(\frac{\theta}{2}\right)$		

$$0 \leq \theta \leq \pi$$

$$0 \leq \varphi \leq 2\pi$$



THE BLOCH SPHERE  
A stereographic representation of qubits

The simplest quantum state, namely the (pure) qubit, can be written

$$|\psi\rangle = \cos \frac{\theta}{2} |0\rangle + e^{i\varphi} \sin \frac{\theta}{2} |1\rangle$$

and shown on the *Bloch sphere* as the vector with spherical polar coordinates  $\theta$  and  $\varphi$ . Of course, this representation of  $|\psi\rangle$  on the sphere is *not* a linear combination of the representations of the basis states  $|0\rangle$  and  $|1\rangle$  at the poles of the sphere. However this graphical representation is not an artificial one. Indeed, taking the ratio of the two coordinates

$$\xi = \frac{e^{i\varphi} \sin \frac{\theta}{2}}{\cos \frac{\theta}{2}} = \tan \frac{\theta}{2} e^{i\varphi}$$

provides the *stereographic projection* of  $|\psi\rangle$ , which is shown in red on the picture. This ratio takes its value in the  $(xy)$ -plane plus "a point at infinity", corresponding to the stereographic projection of  $|1\rangle$ . The other basis state  $|0\rangle$  is sent to the origin of the  $(xy)$ -plane.

# Bloch's Sphere

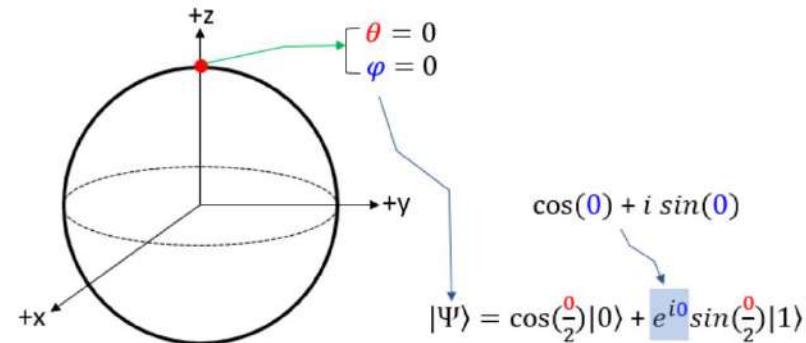
$$e^{ix} = \cos x + i \sin x,$$

In case not familiar with Euler form of the equation and for easy calculation, we can rewrite

$$|\Psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + (\sin(\varphi) + i \cos(\varphi)) \cdot \sin\left(\frac{\theta}{2}\right)|1\rangle$$

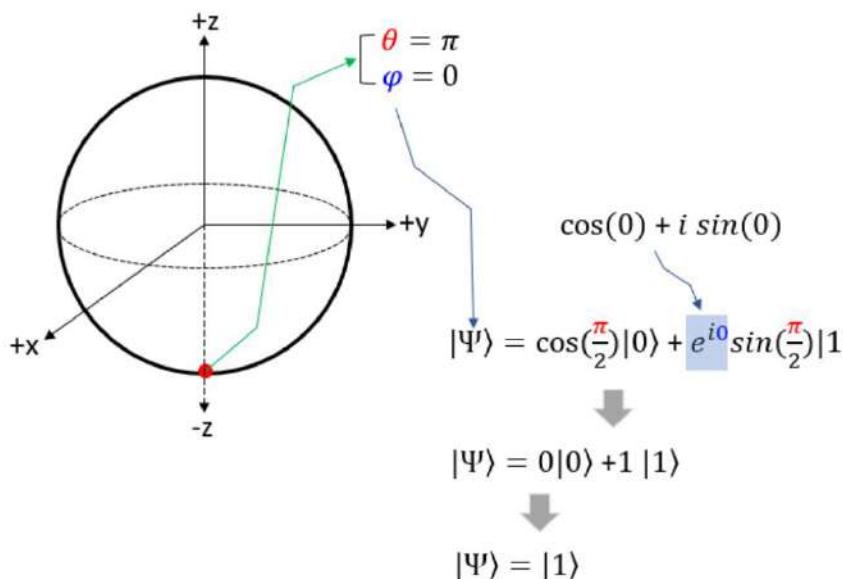
||

$$|\Psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\varphi} \sin\left(\frac{\theta}{2}\right)|1\rangle$$



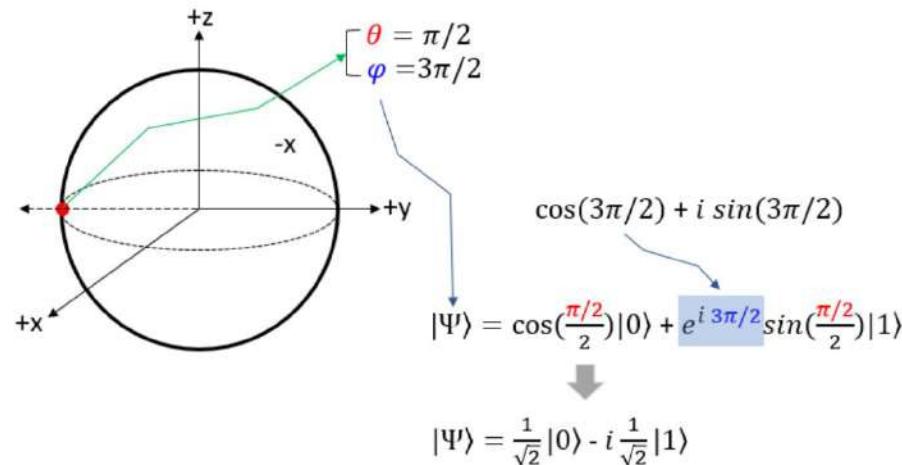
$$|\Psi\rangle = 1|0\rangle + 0|1\rangle$$

$$|\Psi\rangle = |0\rangle$$



$$|\Psi\rangle = 0|0\rangle + 1|1\rangle$$

$$|\Psi\rangle = |1\rangle$$



$$|\Psi\rangle = \frac{1}{\sqrt{2}}|0\rangle - i \frac{1}{\sqrt{2}}|1\rangle$$

# Other basis

- The basis of  $|1\rangle$  and  $|0\rangle$  is the most common in quantum computing. The  $|+\rangle$  state and the  $|-\rangle$  state are defined as following in terms of the  $|0\rangle$ ,  $|1\rangle$  basis. They are both in an equal superposition, with equal probabilities of  $\frac{1}{2}$  of measuring 0 and 1. The difference is solely a phase difference.

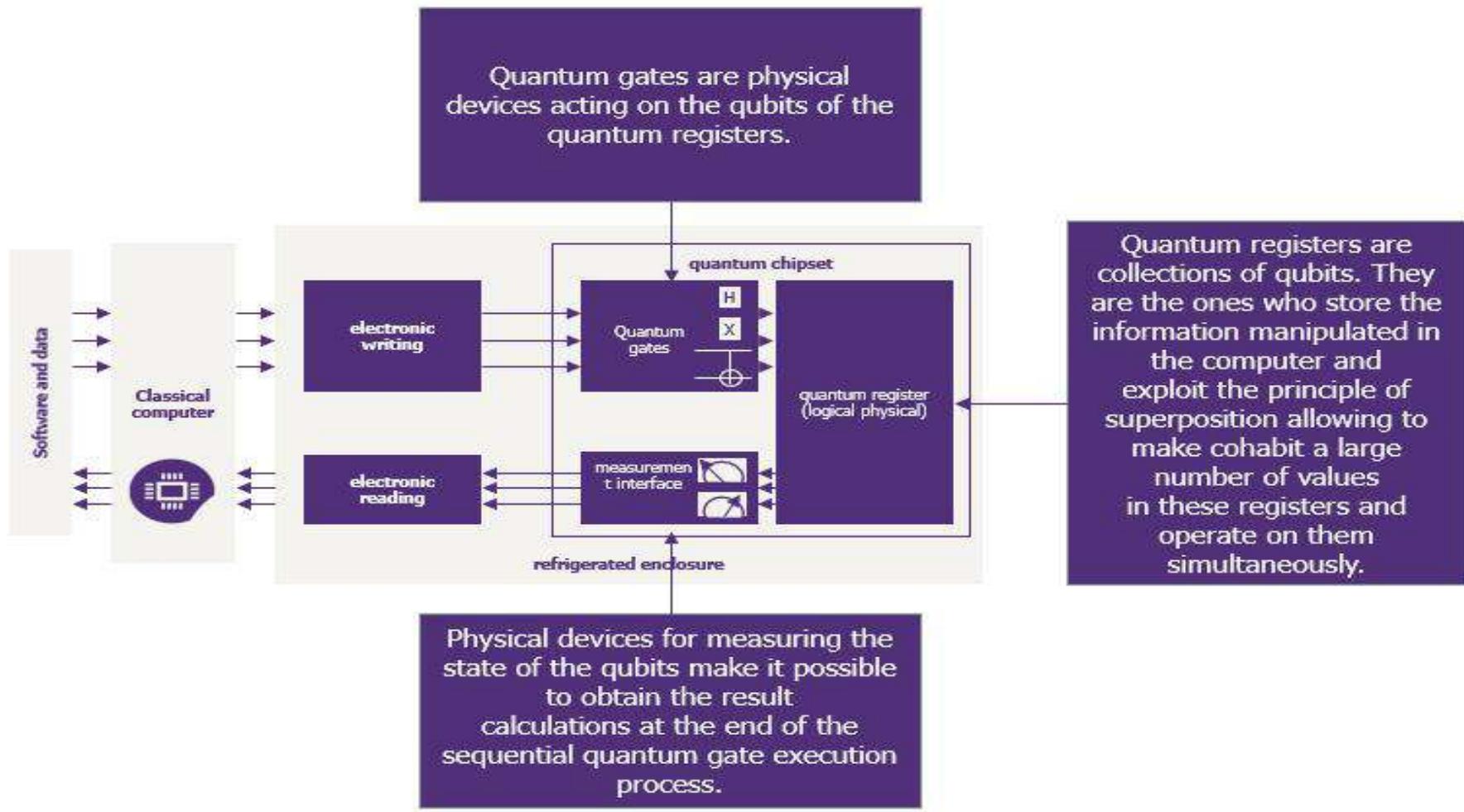
$$\begin{aligned}
 & |0\rangle, |1\rangle && \left. \begin{array}{l} \text{Z Basis} \\ \text{Z Measurement} \end{array} \right\} \\
 & |+\rangle := \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) && \left. \begin{array}{l} \text{X Basis} \\ \text{X Measurement} \end{array} \right\} \\
 & |-\rangle := \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) && |0\rangle = \frac{1}{\sqrt{2}}(|+\rangle + |-\rangle) \quad |1\rangle = \frac{1}{\sqrt{2}}(|+\rangle - |-\rangle).
 \end{aligned}$$

- In quantum mechanics, when we take a measurement the state collapses onto one of the states of the system. Therefore for these bases there are two different types of measurement. The first is called a Z measurement and in taking this measurement we collapse the state onto one of the states in the  $|0\rangle$ ,  $|1\rangle$  (Z basis). The second is called an X measurement and this collapses the state onto one of the  $|+\rangle$ ,  $|-\rangle$  states (X basis).

$$\begin{aligned}
 \alpha|0\rangle + \beta|1\rangle &= \frac{\alpha}{\sqrt{2}}(|+\rangle + |-\rangle) + \frac{\beta}{\sqrt{2}}(|+\rangle - |-\rangle) \\
 &= \frac{\alpha+\beta}{\sqrt{2}}|+\rangle + \frac{\alpha-\beta}{\sqrt{2}}|-\rangle.
 \end{aligned}$$



# *Global architecture of a quantum computer :*



# Basic operations

- We represent classical bits in vector form as  $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$  for 0 and  $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$  for 1

## Operations on one classical bit (cbit)

Identity	$f(x) = x$		$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$
Negation	$f(x) = \neg x$		$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$
Constant-0	$f(x) = 0$		$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$
Constant-1	$f(x) = 1$		$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$

↑  
Ax=b where A is the matrix related to the operation (Identity, ... , Constant-1)

## Review: tensor product of vectors

$$\begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \otimes \begin{pmatrix} y_0 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_0 \begin{pmatrix} y_0 \\ y_1 \end{pmatrix} \\ x_1 \begin{pmatrix} y_0 \\ y_1 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} x_0 y_0 \\ x_0 y_1 \\ x_1 y_0 \\ x_1 y_1 \end{pmatrix}$$

$$\begin{pmatrix} 1 \\ 2 \end{pmatrix} \otimes \begin{pmatrix} 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 3 \\ 4 \\ 6 \\ 8 \end{pmatrix}$$

$$\begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \otimes \begin{pmatrix} y_0 \\ y_1 \end{pmatrix} \otimes \begin{pmatrix} z_0 \\ z_1 \end{pmatrix} = \begin{pmatrix} x_0 y_0 z_0 \\ x_0 y_0 z_1 \\ x_0 y_1 z_0 \\ x_0 y_1 z_1 \\ x_1 y_0 z_0 \\ x_1 y_0 z_1 \\ x_1 y_1 z_0 \\ x_1 y_1 z_1 \end{pmatrix}$$

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

- Reversible means given the operation and output value, you can find the input value
  - For  $Ax = b$ , given  $b$  and  $A$ , you can uniquely find  $x$
- Operations which permute are reversible; operations which erase & overwrite are not
- Identity and Negation are reversible
- Constant-0 and Constant-1 are not reversible
- Quantum computers use only reversible operations, so we will only care about those
  - In fact, all quantum operators are their own inverses

The evolution of quantum states is restricted by the unitarity property of quantum mechanics; that is, every operation on a (normalized) quantum state must keep the sum of probabilities of all possible outcomes at exactly 1. Any quantum gate must thus be implemented as a unitary operator (see page 453), and it is therefore reversible.

## Representing multiple cbits

$$|00\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$|01\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

$$|1\rangle = |100\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$|10\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

$$|11\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

- We call this tensored representation the **product state**

- We can **factor** the product state back into the **individual state** representation

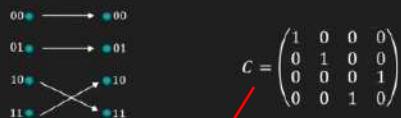
- The product state of  $n$  bits is a vector of size  $2^n$



# Basic operations

## Operations on multiple cbits: CNOT

- Operates on pairs of bits, one of which is the "control" bit and the other the "target" bit
- If the control bit is 1, then the target bit is flipped
- If the control bit is 0, then the target bit is unchanged
- The control bit is always unchanged
- With most-significant bit as control and least-significant bit as target, action is as follows:



$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \text{ on the basis } [ |00\rangle, |01\rangle, |10\rangle, |11\rangle ]$$

## Operations on multiple cbits: CNOT

$$C|10\rangle = C\left(\begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix}\right) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = |11\rangle$$

$$C|11\rangle = C\left(\begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix}\right) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = |10\rangle$$

$$C|00\rangle = C\left(\begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix}\right) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = |00\rangle$$

$$C|01\rangle = C\left(\begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix}\right) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |01\rangle$$

10, 1 is the control bit, the other bit is flipped; therefore 10 -> 11

- We represent classical bits in vector form as  $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$  for 0 and  $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$  for 1
- Operations on bits are represented by matrix multiplication on bit vectors
- Quantum computers only use reversible operations
- Multi-bit states are written as the tensor product of single-bit vectors
- The CNOT gate is a fundamental building block of reversible computing



# Basic operations

## Qbits and superposition

- Surprise! We've actually been using qbits all along!
- The cbit vectors we've been using are just special cases of qbit vectors
- A qbit is represented by  $\begin{pmatrix} a \\ b \end{pmatrix}$  where  $a$  and  $b$  are Complex numbers and  $\|a\|^2 + \|b\|^2 = 1$ 
  - The cbit vectors  $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$  and  $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$  fit within this definition

- Multiple qbits are similarly represented by the tensor product  $\begin{pmatrix} a \\ b \end{pmatrix} \otimes \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} ac \\ ad \\ bc \\ bd \end{pmatrix}$ 
  - Note that  $\|ac\|^2 + \|ad\|^2 + \|bc\|^2 + \|bd\|^2 = 1$
- For example, the system  $\begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} \otimes \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} = \begin{pmatrix} \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \end{pmatrix}$  (note that  $\left\|\frac{1}{2}\right\|^2 = \frac{1}{4}$ , and  $\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} = 1$ )
  - There's a  $\frac{1}{4}$  chance each of collapsing to  $|00\rangle$ ,  $|01\rangle$ ,  $|10\rangle$ , or  $|11\rangle$

- How can a qbit have a value which is not 0 or 1? This is called superposition.
- Superposition means the qbit is both 0 and 1 at the same time
- When we **measure** the qbit, it **collapses** to an actual value of 0 or 1
  - We usually do this at the end of a quantum computation to get the result
- If a qbit has value  $\begin{pmatrix} a \\ b \end{pmatrix}$  then it collapses to 0 with probability  $\|a\|^2$  and 1 with probability  $\|b\|^2$ 
  - For example, qbit  $\begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix}$  has a  $\left\|\frac{1}{\sqrt{2}}\right\|^2 = \frac{1}{2}$  chance of collapsing to 0 or 1 (coin flip)
  - The qbit  $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$  has a 100% chance of collapsing to 0, and  $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$  has a 100% chance of collapsing to 1
- How do we operate on qbits? The same way we operate on cbits: with matrices!
- All the matrix operators we've seen also work on qbits (bit flip, CNOT, etc.)
- Matrix operators model the effect of some device which manipulates qbit spin/polarization without measuring and collapsing it

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \frac{1}{2} \\ \frac{\sqrt{3}}{2} \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{3}}{2} \\ \frac{1}{2} \end{pmatrix}$$

- There are several important matrix operators which only make sense in a quantum context



# Basic operations

The Hadamard gate takes a 0- or 1-bit and puts it into exactly equal superposition

$$H|0\rangle = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix}$$

$$H|1\rangle = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{pmatrix}$$

'-' since the operation has to be reversible

- The Hadamard gate also takes a qbit in exactly-equal superposition, and transforms it into a 0- or 1-bit! (This should be unsurprising – remember operations are their own inverse!)

$$\begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

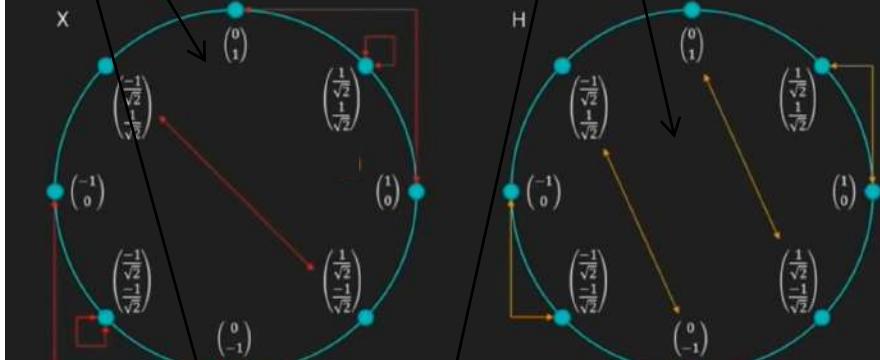
$$\begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

- We can transition out of superposition without measurement!
- We can thus structure quantum computation deterministically instead of probabilistically

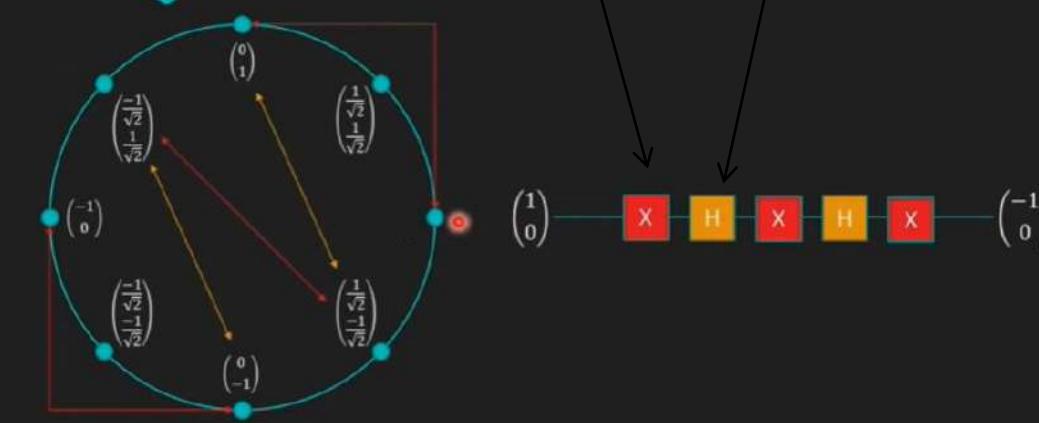
```
H =
0.7071  0.7071
0.7071 -0.7071
>> InpH=[1 0];
InpH =
1
0
>> H*InpH
ans =
1.0000
0
>> H*InpH
ans =
0.7071
0.7071
```

flip hadamard

The unit circle state machine



The unit circle state machine



# Quantum operators

**Quantum operators** In classical computing, two fundamental logic gates (AND and OR) perform irreversible computations, i.e. the original inputs cannot be recovered from the output. Quantum gates (which operate on qubits) are constrained to be *reversible*, and operate on the input state to yield an output of the same dimension. In general, quantum gates are represented by unitary matrices, which are square matrices whose inverse is their complex conjugate.

An  $n$ -qubit system exists as a superposition of  $2^n$  basis states. Its state can be described by a  $2^n$  dimensional vector containing the coefficients corresponding to each basis state. For example, the  $|\psi\rangle$  vector above may be described by the vector  $[\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{6}}, \frac{1}{\sqrt{6}}]^T$  using the basis vectors.

Thus, a  $n$ -qubit quantum gate  $H$  represents a  $2^n \times 2^n$  unitary matrix that acts on the state vector. Two common quantum gates are the Hadamard and CNOT gates. The Hadamard gate acts on 1-qubit and maps the basis states  $|0\rangle$  and  $|1\rangle$  to  $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$  and  $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$  respectively. The CNOT gate acts on 2-qubits and maps  $|a, b\rangle$  to  $|a, a \oplus b\rangle$ . In other words, the first bit is copied, and the second bit is flipped if the first bit is 1. The unitary matrices corresponding to the Hadamard and CNOT gates are:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \text{ on the basis } [ |0\rangle, |1\rangle ]$$

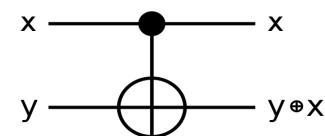
The Hadamard gate for multiple qubits transforms each of the qubits according to the Hadamard gate. A simple example is the Hadamard transformation for a register with only  $n = 2$  qubits for which the Hadamard gate is represented, as follows:  $H^{\otimes 2} = H \otimes H$ . When applied to a register

$$\begin{aligned} H^{\otimes 2}|00\rangle &= H|0\rangle \otimes H|0\rangle = |+\rangle \otimes |+\rangle, \\ &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle), \end{aligned}$$

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

[https://en.wikipedia.org/wiki/Unitary\\_matrix](https://en.wikipedia.org/wiki/Unitary_matrix)

CNOT: Control NOT gate

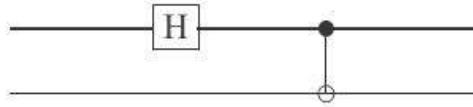


input		output	
x	y	x	y+x
0>	0>	0>	0>
0>	1>	0>	1>
1>	0>	1>	1>
1>	1>	1>	0>



# Quantum operators

We can generate the Bell states with a Hadamard gate and a CNOT gate. Consider the following diagram:



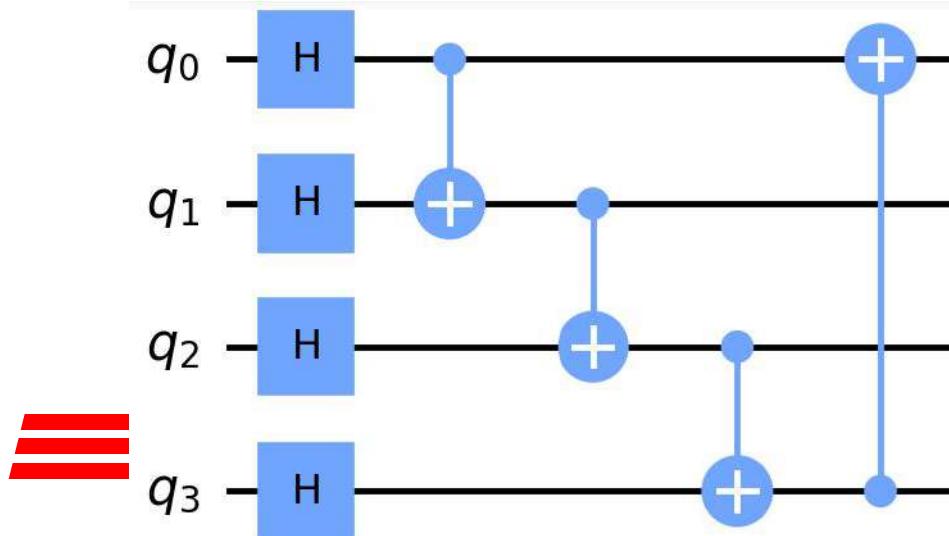
The first qubit is passed through a Hadamard gate and then both qubits are entangled by a CNOT gate.

If the input to the system is  $|0\rangle \otimes |0\rangle$ , then the Hadamard gate changes the state to

$$\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|10\rangle ,$$

and after the CNOT gate the state becomes  $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ , the Bell state  $|\Phi^+\rangle$ .

Applying Hadamard gates to apply superposition into qubits and CNOT gates to generate entanglement.



# Pauli Operators

The Pauli  $X$ ,  $Y$  and  $Z$  matrices are so-called because when they are exponentiated, they give rise to the *rotation operators*, which rotate the Bloch vector  $\vec{r}_\rho$  about the  $\hat{x}$ ,  $\hat{y}$  and  $\hat{z}$  axes, by a given angle  $\theta$ :

$$R_x(\theta) \equiv e^{-i\frac{\theta}{2}X}$$

$$R_y(\theta) \equiv e^{-i\frac{\theta}{2}Y}$$

$$R_z(\theta) \equiv e^{-i\frac{\theta}{2}Z}$$

Now, if operator  $A$  satisfies  $A^2 = I$ , it can be shown that

$$e^{i\theta A} = \cos(\theta)I + i \sin(\theta)A$$

reminder:

$$e^{i\theta} = \cos \theta + i \sin \theta$$

$$e^{-i\theta} = \cos \theta - i \sin \theta$$

## Unitary operators

Defined by the property  $U^{-1} = U^\dagger$ .  
All Pauli matrices are unitary.

E.g.

$$\sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \rightarrow \sigma_y^\dagger = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

$$\sigma_y \sigma_y^\dagger = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \mathbf{1} \Rightarrow \sigma_y^\dagger = \sigma_y^{-1}$$

therefore you can use  
 $\cos((\theta/2)*I-i*\sin(\theta/2)*A)$   
 for calculating the rotation  
 operators

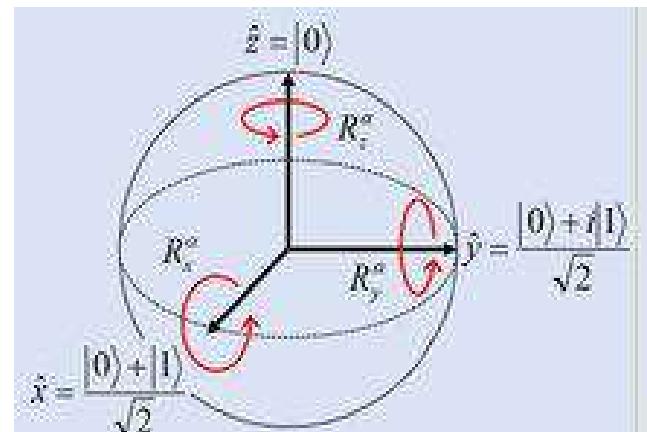
$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Rotation matrices:

$$\hat{R}_x(\theta) = \begin{pmatrix} \cos \frac{\theta}{2} & -i \sin \frac{\theta}{2} \\ -i \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{pmatrix}$$

$$\hat{R}_y(\theta) = \begin{pmatrix} \cos \frac{\theta}{2} & -\sin \frac{\theta}{2} \\ \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{pmatrix}$$

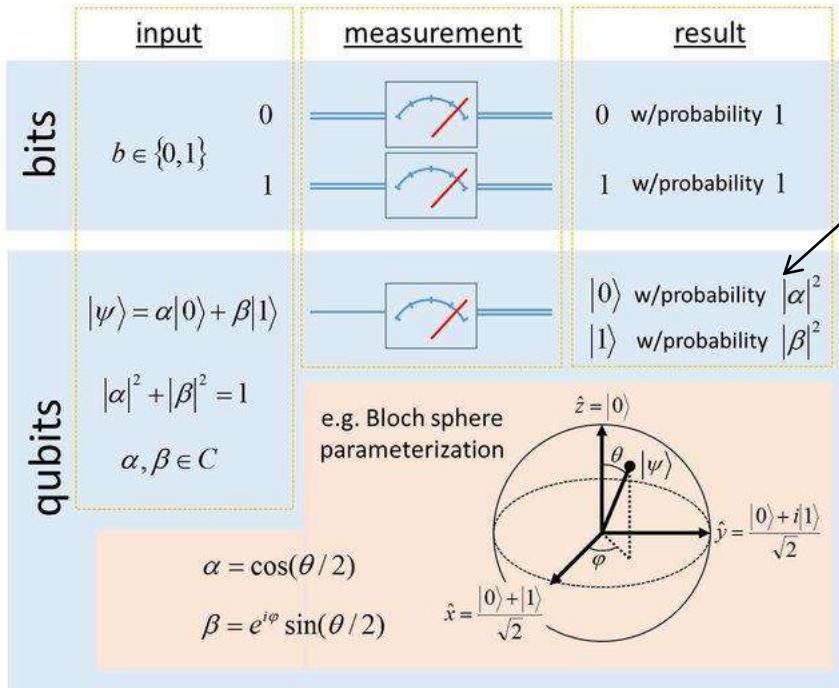
$$\hat{R}_z(\theta) = \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix}$$



[https://en.wikipedia.org/wiki/Conjugate\\_transpose](https://en.wikipedia.org/wiki/Conjugate_transpose)

# QM postulate

Circuit symbol for measurement:  $|\psi\rangle \xrightarrow{\text{M}}$



QM postulate: quantum measurement is described by a set of operators acting on the state space of the system. The aim of these operations is to obtain the probability  $p$  of a measurement result  $m$ : the result is a classical bit that will be 0 or 1 with probability  $|\alpha|^2$  and  $|\beta|^2$ , respectively.

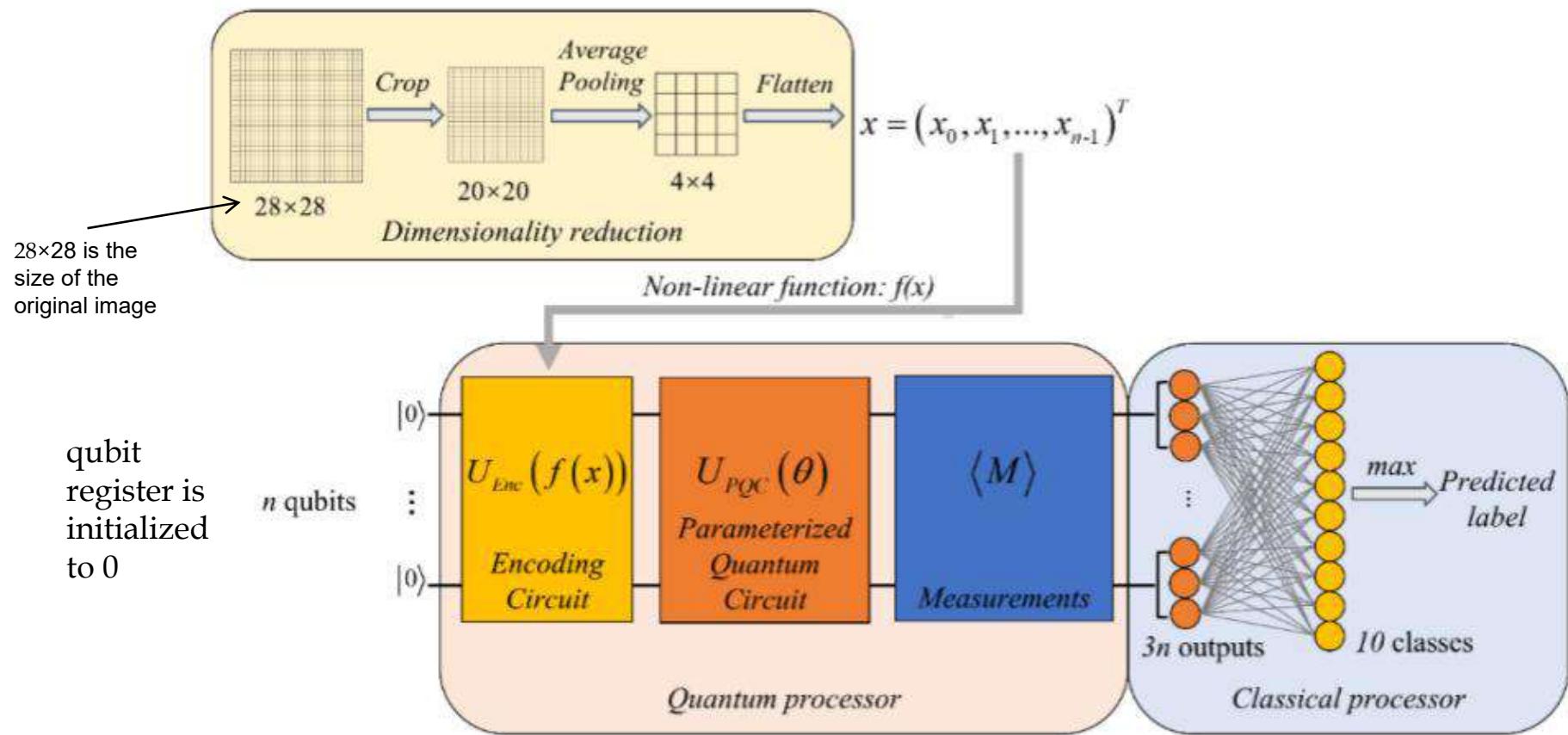
The details of the measurement techniques are beyond the scope of this course



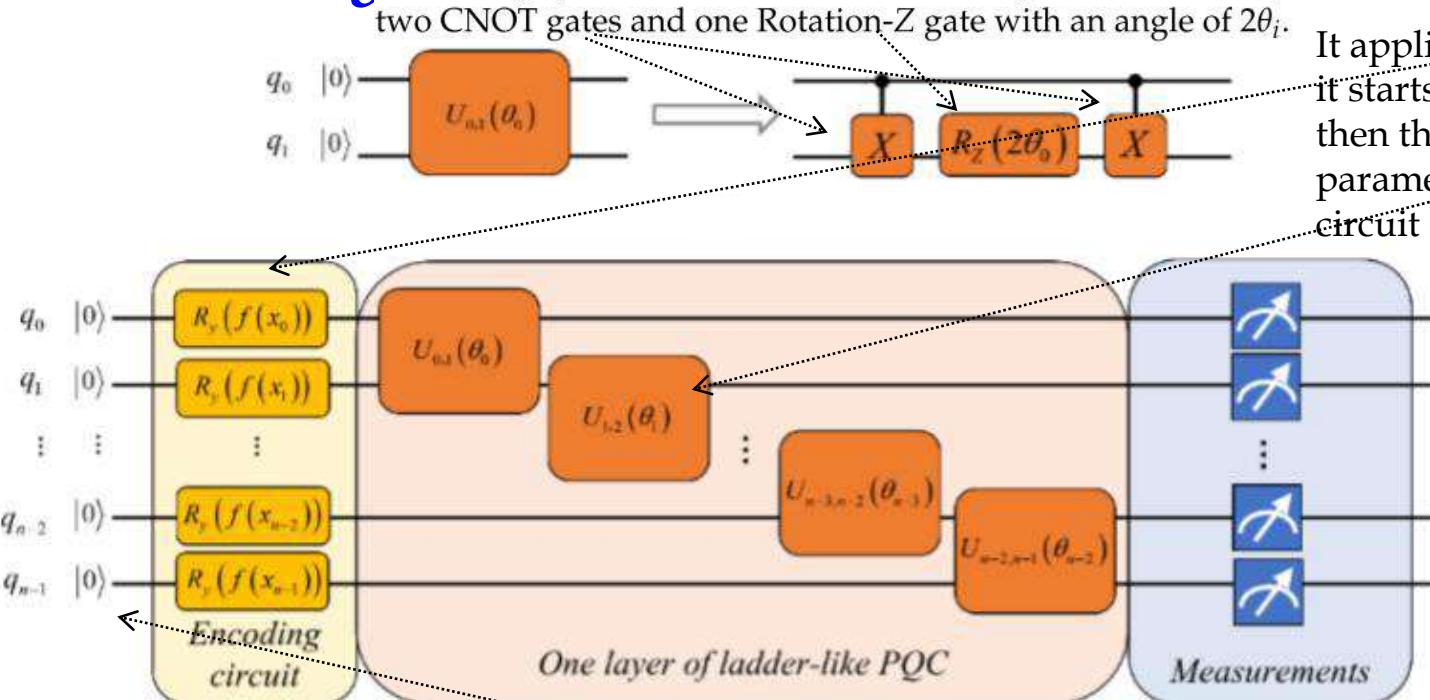
# Hybrid quantum network, an example

“Classical” layers can be used to represent the images as features, than the feature vector is used to feed a quantum processor, the quantum layers have set of weights that are tuned for minimizing a given loss; notice that the classification is performed using classical methods. The measurement outcomes are fed into a fully connected layer with the softmax activation function to generate the predicted label. The loss function between the predicted label and the true label is computed, and a classical optimizer is employed to update parameters.

Essentially, the quantum processor is a feature extractor (in this network) for the fully connected layers.



# Hybrid quantum network



The data vector is encoded in a quantum state  $|\varphi_{in}\rangle$  by applying an encoding circuit  $U_{Enc} = \otimes_{i=0}^{n-1} R_y(f(x_i))$  on an initial state  $|0\rangle^{\otimes n}$ , where  $f(x_i) = \arcsin(x_i)$  is a nonlinear function that maps the data to angle space, and  $R_y(\theta)|0\rangle = \cos(\theta/2)|0\rangle + \sin(\theta/2)|1\rangle$  is a rotation operation which guarantees that the amplitude of a single qubit is real.

$$|\varphi_{in}\rangle = \begin{bmatrix} \cos(f(x_0)/2) \\ \sin(f(x_0)/2) \end{bmatrix} \otimes \dots \otimes \begin{bmatrix} \cos(f(x_{n-1})/2) \\ \sin(f(x_{n-1})/2) \end{bmatrix} \quad (1)$$

This nonlinear quantum encoding circuit can map the input data into a higher-dimensional space, which will facilitate the subsequent classification process.

The parameterized quantum circuit  $U_{PQC}(\theta)$  performs a series of linear transformations on the input state.

$x$  is the vector that represents a given pattern, see previous page



# Hybrid quantum network

The ladder-like circuit is the main part of the PQC quantum circuit to perform the calculation. This architecture is capable of disentangling the input state as much as possible, even with a small number of quantum gates. In this ladder-like circuit, two-qubit quantum gates in steps of one are applied. In other words, a family of quantum gates with different parameters is performed on each pair of nearest-neighbor qubits, which allows to capture the quantum correlations of a specific scale on the same layer of the network. In the terms of the input state  $|\phi_{in}\rangle$ , the effective information (discrete labels in the classification task) is embedded in the quantum subsystem. Therefore, the PQC is used for performing quantum computations to extract the information hidden in this quantum subsystem. Specifically, the purpose of implementing unitary gates is to remove the superposition in quantum data, leaving the information containing the label. Then, with an appropriate measurement strategy, the classical information is extracted, but it may not be the direct representation of the label, so further classical classification is required.

$U_{PQC}(\vec{\theta})$  consists of unitary gates  $U_i(\theta_i)$  with parameters  $\vec{\theta} = \{\theta_0, \theta_1, \dots, \theta_{m-1}\}$ , and  $m$  is the number of quantum gates. In the learning stage, the parameters are optimized by the classical Adam optimizer, so that the initial state can evolve to the desired state through the operation of the PQC.

