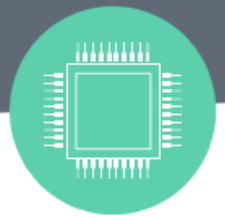
NEURAL NETWORKS EXPLAINED

How AI works behind the scenes

June 2020


Thomas de Mareuil

AGENDA

I. INTRODUCTION
II. DEEP LEARNING & NEURAL NETWORKS
III. CONVOLUTIONAL NEURAL NETWORKS (CNN)
IV. RECURRENT NEURAL NETWORKS (RNN)
V. OTHER TYPES OF DEEP LEARNING MODELS
VI. APPENDIX

1

INTRODUCTION

Machine Learning overview
Why do we need neural networks?

MACHINE LEARNING OVERVIEW – THE SUPERVISED LEARNING FRAMEWORK

Most of what we call today “machine learning”, “deep learning” or “AI” is based on the simple principle of **supervised learning:**

① TRAINING

Training data

Data points (X_i)	Labels (Y_i)
	0
	1
	2
	3
	4

Learning algorithm

Goal is to **train a prediction function** (a classifier) f.

- Loss function (predictions vs. truth):
  - Quadratic loss / MSE: $L(Y, f(X)) = |Y - f(X)|2$
  - 0-1 loss: $L(Y, f(X)) = 1_{Y \neq f(X)}$
- Minimization of loss: $f^*(X) = argmin_f \, \mathbb{E}[L(Y, f(X))]$
  - Exact computation
  - Gradient Descent: $w_{k+1} = w_k - \eta \, \nabla f(w_k)$

② PREDICTION

New input

$X_{new} =$ 

Classifier

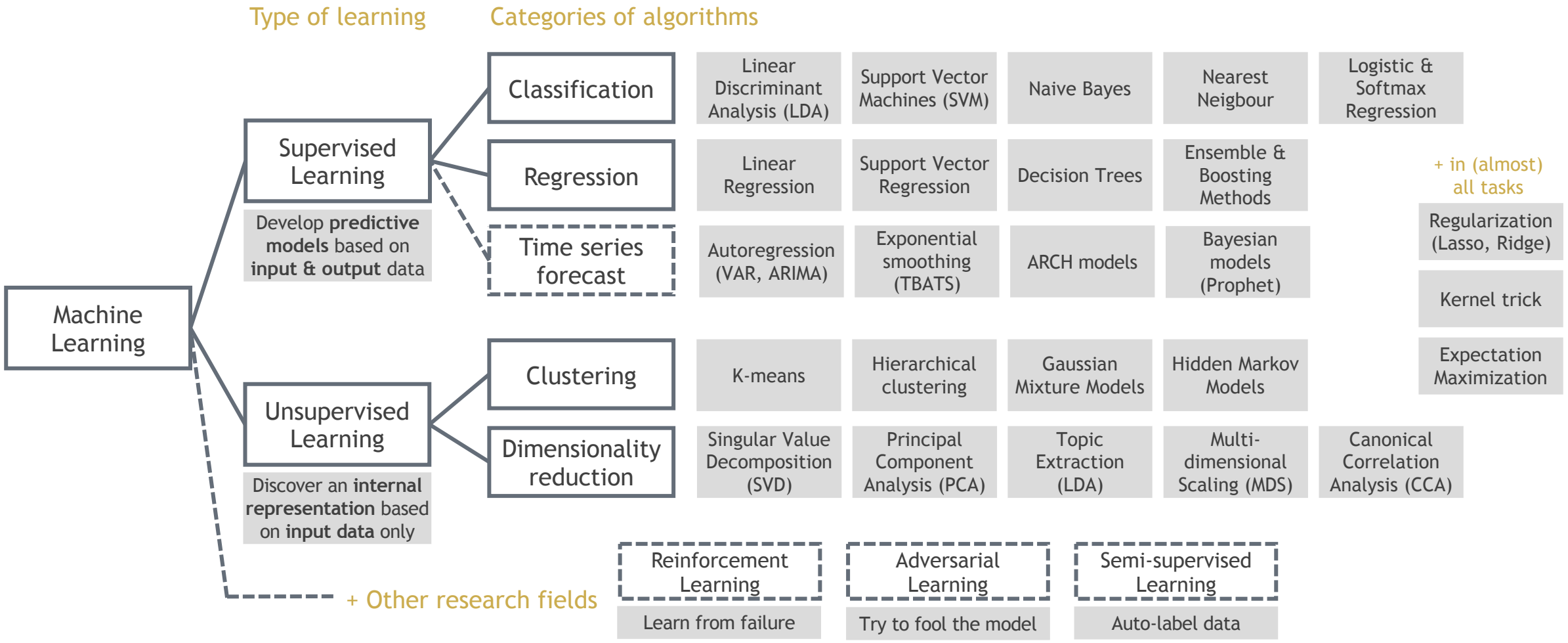
$f^*(X_{new}) = Y_{new}$

Label

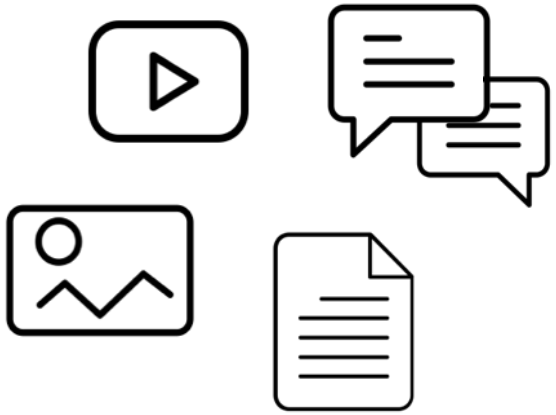
$Y_{new} = 2$

A COMPREHENSIVE VIEW OF MACHINE LEARNING

Here is just an overview of most machine learning fields. We'll see that deep learning / neural nets can improve on almost all of them!



WHY DO WE NEED DEEP LEARNING & NEURAL NETWORKS?



More complex tasks: big data (more inputs), distributed computing

Higher performance: more parameters, non-linear mappings

Special types of data: images, videos, text, speech…

Deep learning and its tools (neural networks, gradient learning) can be used in almost all the fields of machine learning presented in the previous slide (supervised, unsupervised, reinforcement, etc.)!

2

DEEP LEARNING & NEURAL NETWORKS

History
Learning methods
Implementation with Python

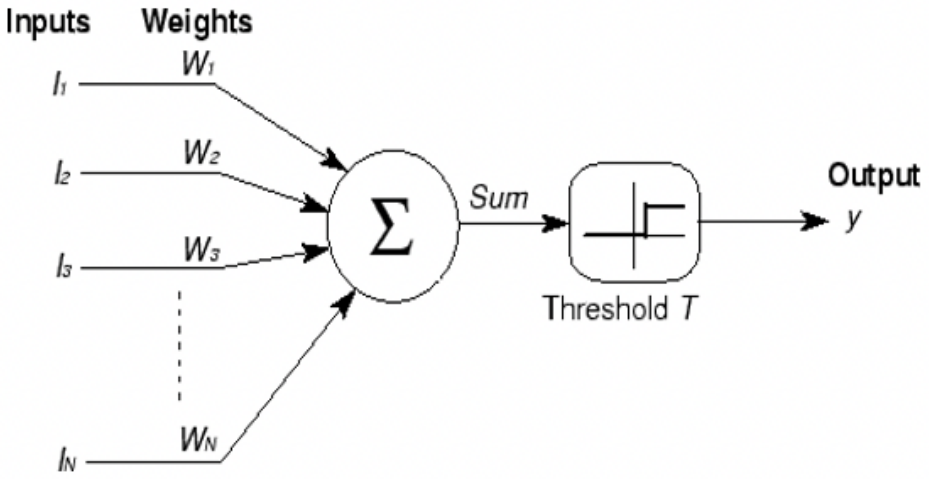
THE HISTORY OF AI AND NEURAL NETWORKS: NOT A RECENT IDEA

- Unsurprisingly, the initial idea of neural networks was to simulate how neurons in the brain function.
- Basically, **a neuron is a cell that receives an input signal, transforms it, and outputs another signal**.
- Same with an artificial neuron: it receives data as an input, performs computations on it, and outputs another value. It also includes an **activation threshold**, to determine if the neuron "fires" – i.e. it outputs a value only if the result of its computations corresponds to a certain threshold. Basically, a neuron is just a node of functions.

The first modeling of this process was the **McCulloch-Pitts neuron**, in **1943**. It takes values as inputs, computes a **weighted sum**, and outputs 0 or 1 if the result is below or above threshold (**linear activation**).

The weights are the heart of the system, and by changing them to specific numerical values, we can **obtain the desired output** (e.g. a prediction, based the input and learned weights).

The neuron computation is of the form: $y = T(\sum_{k=1}^{N} W_k I_k + b)$.



A basic McCulloch-Pitts neuron

THE FIRST NEURAL NETWORK: THE PERCEPTRON

The first practical use of these artificial neurons is the ***Perceptron***, developed by F. Rosenblatt in **1958** at Cornell University (link).

His goal was to understand the decision systems in the eye of a fly, which determine its flee response: he installed cameras (the inputs) on a patchboard (to try different combinations) and connected them to processors (for computations) through potentiometers (to adapt weights, using a basic iterative algorithm).

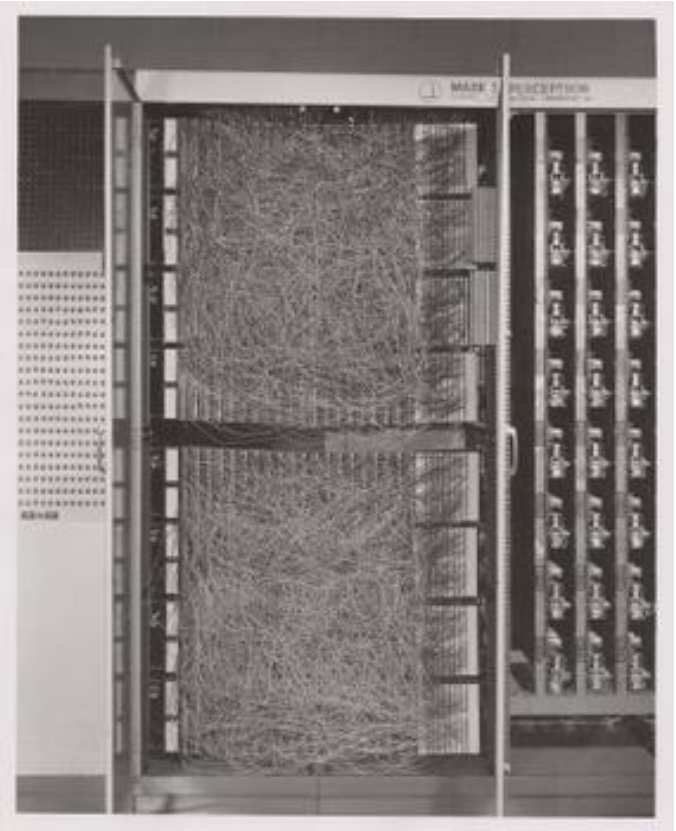
The Perceptron project was funded by the US Navy, who stated at the time:

> *"The Navy revealed the embryo of an electronic computer today that it expects **will be able to walk, talk, see, write, reproduce itself and be conscious of its existence**. Later perceptrons will be able to recognize people and call out their names, and to instantly translate speech in one language to speech and writing in another language."*
>
> – Press conference, 7 July 1958, The New York Times.

Limits:

- By design, the Perceptron performs only linear computations.
- It can only process data with one level of complexity (1 single neuron layer).

The *Mark I Perceptron*, F. Rosenblatt, 1958



GROWING COMPLEXITY: THE MULTI-LAYER PERCEPTRON (1/2)

The complexity problem was quickly solved by stacking several neurons together to form different **layers** of computations. You then include non-linearity by passing the data through **non-linear activation functions**.

You get a **multi-layer perceptron**, which is the basic architecture for all modern neural network. It includes are **3 types of layers**:

Input layer

- The layer receiving input data and performing the first computations
- Input data has to be numerical, i.e. text or images have to be transformed into numerical vectors beforehand
- Data pre-processing is usually the most time consuming part in machine learning!

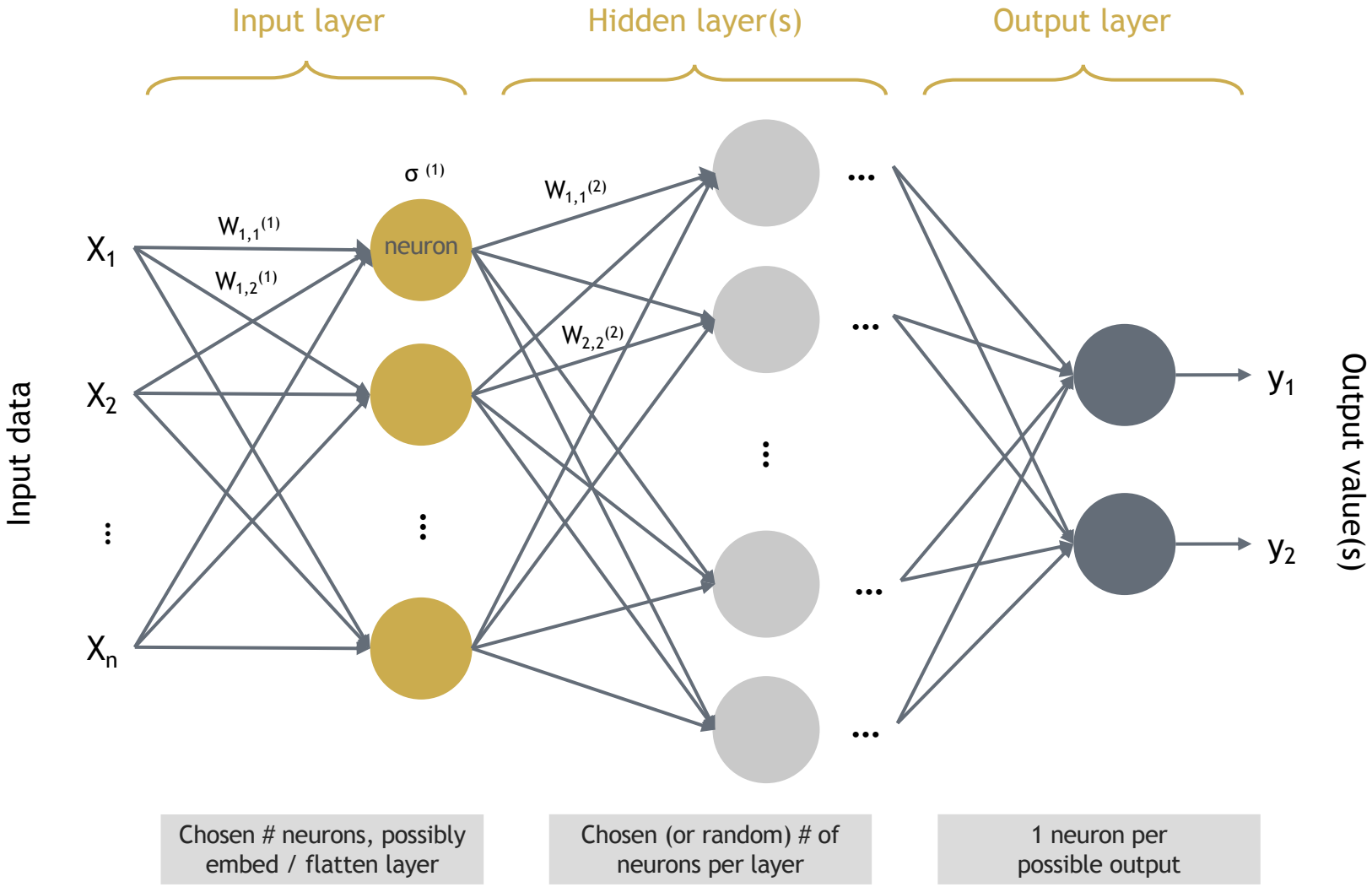
Hidden layer(s)

- They are each made of several neurons, which perform 1 linear and 1 non-linear (activation) computation (ex: ReLu function)
- Each neuron transmits its output to the next hidden layer
- The output of a given layer is called a “hidden representation” of our data.

Output layer

- It outputs the desired result, which can be of different sorts
- For example, each neuron in the output layer can represent a class and output the probability that the original input belongs to this class.

GROWING COMPLEXITY: THE MULTI-LAYER PERCEPTRON (2/2)



- The multi-layer perceptron is a **fully-connected** or **'dense' network**, i.e. all inputs and outputs are connected together throughout the layers. This will not be the case with other architectures.
- In the 60's, several critics (notably Minsky & Papert, 1969, link), lack of funding and too young technological hardware led to a period where AI and neural networks were put aside: the **"AI Winter"**.

Notations:

$W_{1,2}^{(2)}$: weight applied to the 1st input of the 2nd neuron in the 2nd layer (including bias $b_2^{(2)}$)
$\sigma^{(2)}$: activation function of the 2nd layer (similar for all neurons in the layer)
$Z_2^{(2)}$: input of the 2nd neuron in the 2nd layer
$A_2^{(2)}$: output of the 2nd neuron in the 2nd layer, such that $A_2^{(2)} = \sigma^{(2)} (\Sigma_k W_{k,2}^{(2)} \cdot Z_{k,2}^{(2)})$

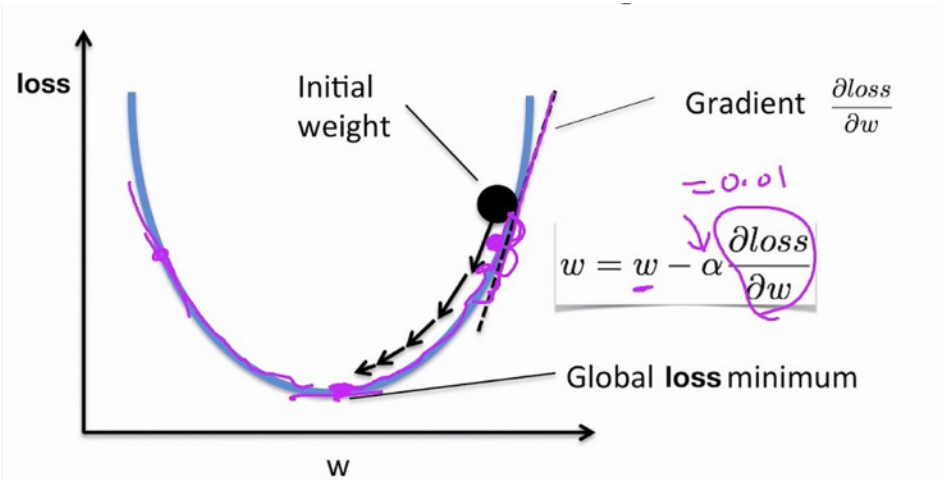
HOW NEURAL NETWORKS LEARN: GRADIENT DESCENT & BACKPROPAGATION

Turning point: the (re)invention and improvement of backpropagation by G. Hinton (1986, link), Y. Le Cun (1989, link) and Y. Bengio (1998, link), the 'godfathers' of deep learning.

- Backpropagation is the process in which **weights are updated** throughout all the network in order to **minimize the error at output level** (e.g. minimize $global\ loss\ function = |\ output\ value - desired\ value\ or\ label\ |2$).
- We want to learn the **weights that set this loss function to its minimum**, i.e. that **set the derivative (gradient) to 0**. But these are multidimensional and composed functions, too complex to solve directly – so we use gradient descent:

1) Starting from the last layer, we **compute the gradient** through all layers (i.e. **backwards**) using the **chain rule** (derivative of composed functions)
2) We then **move weights slightly** in the **direction given by the gradient**
3) We **repeat this process** multiple times **until the loss converges** (i.e. stabilizes at a minimum value).

We can tune multiple parameters to make the update less computationally expensive and to converge faster: change learning rate (step size) for each update, compute gradients w.r.t. only 1 weight or 1 coordinate at each step, add a momentum term, etc.



Gradient descent

NEURAL NETWORKS IMPLEMENTATION WITH PYTHON

Why Python?

- **Free and exhaustive libraries**, with a very active **community**
- Language **flexibility** and **accessibility** (reducing time to code and debug)
- **Seamless integration with C and C++** code modules

Main frameworks:

- **TensorFlow**: developed by Google, a flexible deep learning library, with implementations of most DL methods and algorithms
- **Keras**: a simplified API on top of TensorFlow
- **PyTorch**: the DL library developed by Facebook
- **Scikit-learn**: the most famous machine learning library, but still useful in DL pipelines for some data pre-processing tasks
- **Spark MLlib**: the Spark framework to scale machine learning / deep learning algorithms on distributed systems (computer clusters)

Specialized libraries:

- **Natural Language Processing (NLP)**: NLTK (text processing), SpaCy
- **Computer Vision**: OpenCV, Raster Vision (satellite imagery)
- **FastAI**: simplified API on top of PyTorch

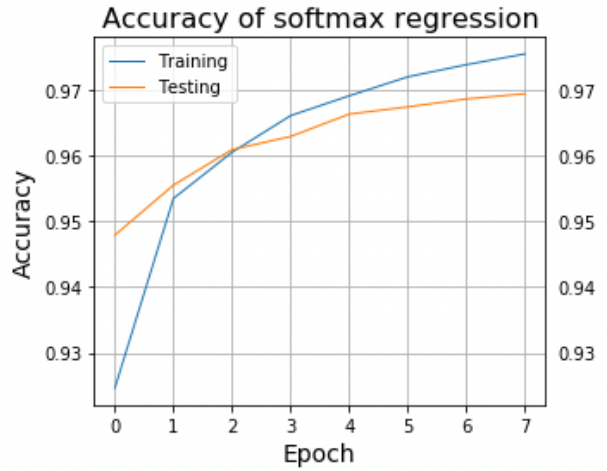
PyTorch
TensorFlow
Keras
scikit learn
APACHE Spark MLlib
spaCy
OpenCV
NLTK Natural Language Toolkit
fast.ai
rastervision

EXAMPLE: HANDWRITTEN DIGITS RECOGNITION (MNIST)

- The MNIST dataset (link) is a famous basic example of neural networks application
- It contains images of **handwritten digits** (60k for training, 10k for testing), labelled from 0 to 9
- A simple model we could use is: a flattening layer (to convert each image to a vector of numbers), 1 dense hidden layer with ReLu activation, and 1 dense output layer with 10 neurons and softmax activation, which **outputs the probability for each of the 10 classes**:

Image sample -
28x28 pixels each, 1 channel (gray)

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               100480
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1290
=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
```



- This model relies on more than **100k parameters** (weights) which are updated using gradient descent, to minimize the loss function (negative loglikelihood : $\theta \mapsto -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{10}\mathbb{1}_{Y_i=k}\log\mathbb{P}_\theta(Y_i=k|X_i)$)
- For training, the gradient is computed using batches of 32 samples (up to the 60k samples), and model performance is assessed using the 10k test samples. The whole data set is used 8 times – in the graph you can see **how the model learns and improves accuracy**.

3

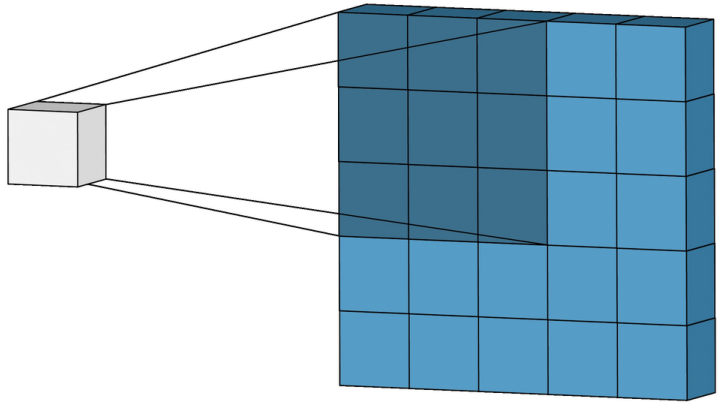
CONVOLUTIONAL NEURAL NETWORKS (CNN)

Architecture
Famous CNNs
Examples

CONVOLUTIONAL NEURAL NETWORKS (CNN): A SPARSE ARCHITECTURE FOR SPATIAL STRUCTURE RECOGNITION

A problem with the MLP (or “feed-forward”, “dense” network) is that it doesn’t take into account the spatial organization of data – e.g. if pixels values are permuted, the output remains the same, while the image is actually very different.

- CNNs tackle this problem by applying **convolutions** to the data grid. The idea is that a **“filter”**, or “kernel” (a small square matrix of numbers, for ex. of size 3x3) is **moved all along the image**, and applies each time some computations to the area it covers.
- Depending on the filter, the output represents a special feature of the image - for example, a filter can apply weights such that it keeps only the pixel values corresponding to horizontal lines.
- With different filters you can detect different “low-level” features (points, corners…), and when you put **several convolutional layers successively**, in a **deep network**, you end up detecting more complex (“high-level”), combined features.



*Thanks to **backpropagation**, your network can learn the filter weights that reveal the most relevant features for your problem. For example, if your goal is to detect panels for an autonomous vehicle, you feed your model with labelled images, and **by minimizing the prediction error it learns how to detect circles and rectangles** !*

-1	0	1
-2	0	2
-1	0	1

Horizontal

-1	-2	-1
0	0	0
-1	-2	-1

Vertical

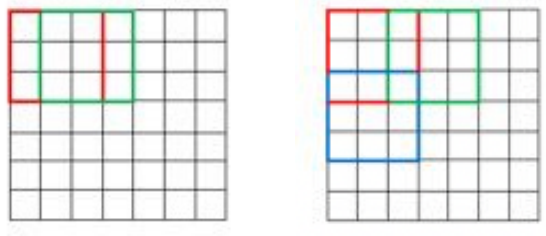
Kernels used in the Sobel edge detection

CONVOLUTIONAL NEURAL NETWORKS (CNN): A SPARSE ARCHITECTURE FOR SPATIAL STRUCTURE RECOGNITION

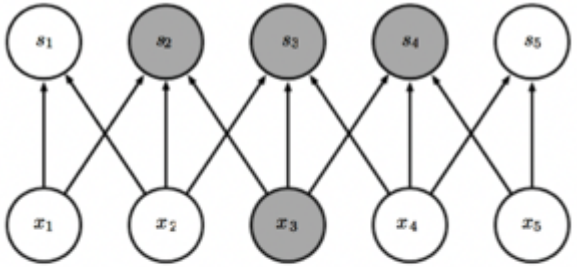
You can adapt many parameters to make sure the network performs well:

- ***Kernel size and stride***: you can adjust the size and 'speed' at which the filter moves.
- ***Pooling***: after scanning an image, you can average results 2 by 2 (2x2 AvgPooling), or you can keep only the maximum values (MaxPooling), in order to reveal only the most important features (and resize the output by the way).
- ***Dropout***: random killing of some neurons throughout the network, in order to add randomness and **avoid "overfitting"** the training data (more robust **generalization**)

> *Each **convolutional layer** (including filter **Convolutions, + Pooling, + non-linear Activation** as usual) outputs a new representation of the image, called a feature mapping, and can transmit it to the next layer for further analysis.*

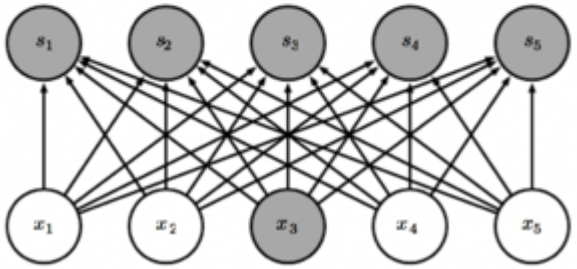


Stride = 1 vs. stride = 2



Another great advantage of CNNs is their sparse architecture:

- In a convolution with a kernel of width 3, **only 3 outputs are affected by the input x_1**, vs. when we use complete matrix multiplication all inputs are connected to all outputs. We say that the connectivity is sparse vs. dense.
- It results in **a lot less parameters**, improving memory and computation speed.



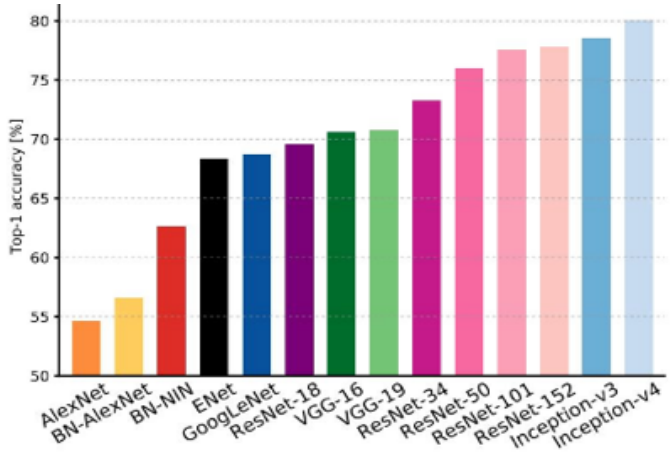
Sparse vs. dense architecture

FAMOUS CNNs – THE IMAGENET CHALLENGE

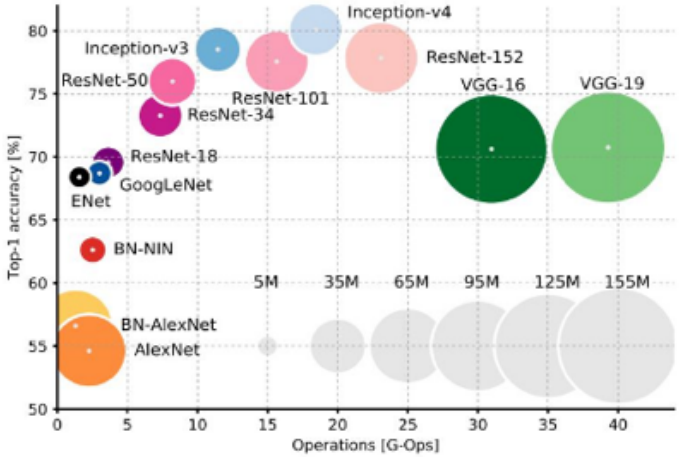
The first famous CNN is LeNet (link), by Y. LeCun in his landmark 1998 article about character recognition (MNIST).

The next famous CNNs are all outcomes of the "ImageNet" challenge, a **global open competition** (started in 2010, annually) that aims at **advancing research in AI**. It contains +14M images with +20k different classes to predict. Major milestones in the competition are:

- AlexNet (Univ. of Toronto, link): winner in 2012 with a test accuracy of 84.6%, +10.8 ppts better than the runner-up. This was made feasible due to the use of Graphics processing units (GPUs, an essential ingredient of the deep learning revolution) which allow to train more complex models thanks to parallelization properties.
- VGG-16 (Oxford, link): winner in 2013 with 92.7% accuracy, another big leap forward, thanks to a deeper architecture.
- GoogleNet (Google, link) winner in 2014 with 93.3% accuracy, made of 22 layers and a new building block called "inception module".
- ResNet (Microsoft, link): winner in 2015 with 96.4% accuracy, very deep (152 layers) but still not-so-expensive to train thanks to "residual blocks", which allow to skip certain layers (see architecture in Appendix).
- DenseNet (Facebook, link, 2017): improvement on ResNet thanks to alternating conv and dense blocks, with overall less parameters (see architecture in Appendix).



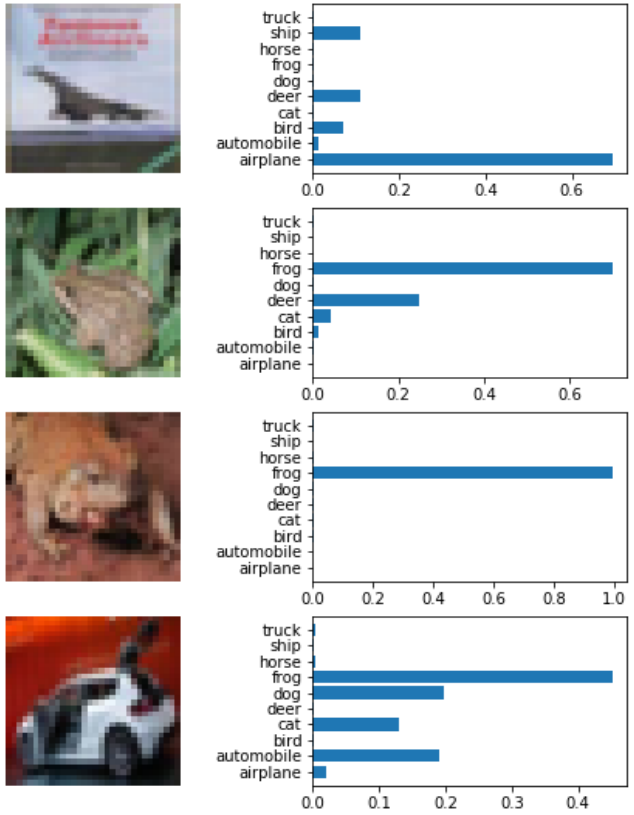
ImageNet models accuracy and computation speed



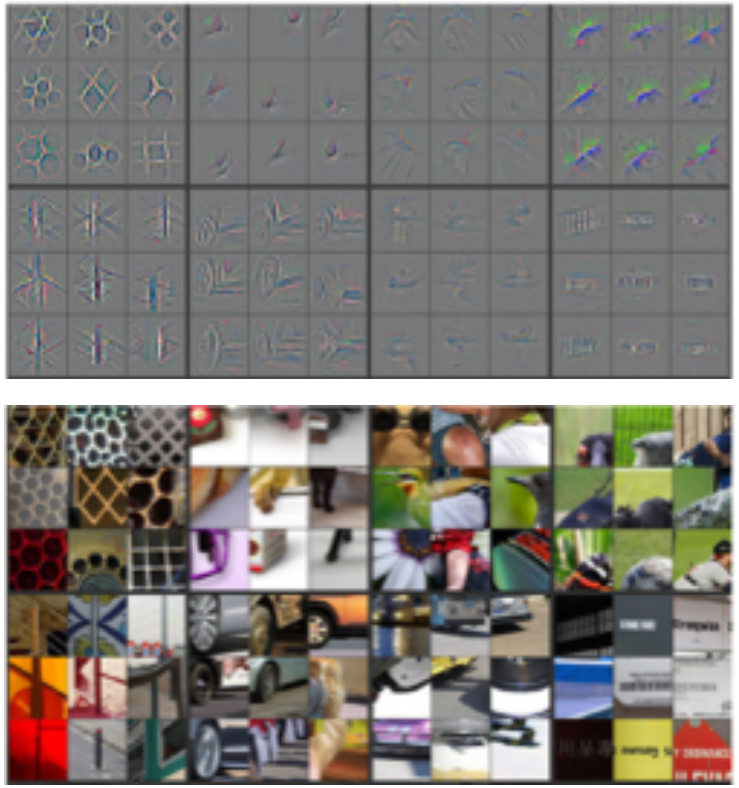
EXAMPLE (I): IMAGE CLASSIFICATION

The most straightforward application of CNNs might be image classification: you give a model thousands of images with their labels (classes), it learns how to recognize them and outputs the class probability.

Output probabilities on the "CIFAR-10" (10 classes) dataset:



Feature mapping: what a model "sees" (output of AlexNet's 3rd conv layer)



Frame annotation based on recognized content:

EXAMPLE (II): OBJECT DETECTION

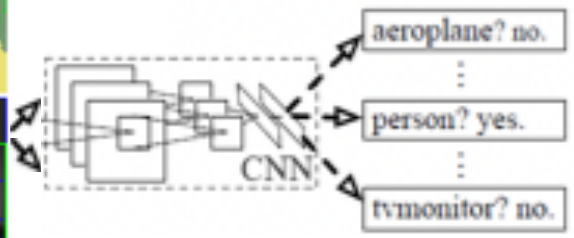
Object detection goes further than simple classification by defining a **bounding box** around the recognized object.

A popular approach is the Region-based CNN model (R-CNN, 2014, link), which is a 2-stage detection model:

1) First, the model determines where the objects could be located in the image, using a **"selective search"** algorithm which iteratively groups together similar pixels.

2) Then it puts a bounding box around each category-independent region proposal, and **uses a CNN** to classify it.

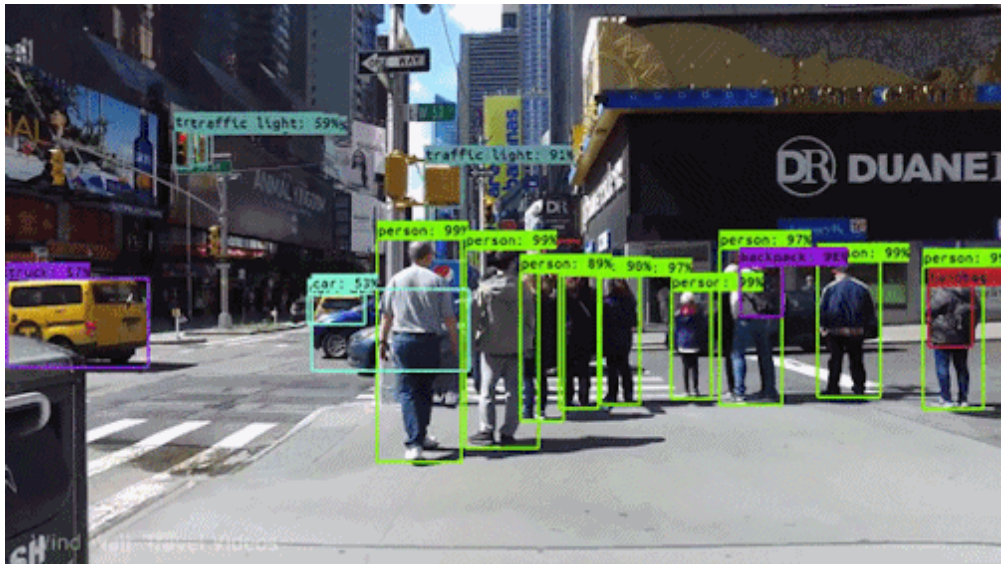
Selective search iterative process



CNN region classification

The "You Only Look Once" (from YOLO, 2015, link, to YOLOv4, 2020, link) and "Single Shot Detector" (SSD, 2016, link) models use the same principle, but they predict bounding boxes and class probabilities from the full image in a single, trainable pipeline.

Therefore, they are much faster and can easily **process video**:



EXAMPLE (III) – SEMANTIC SEGMENTATION

Semantic segmentation goes even further by **classifying each individual pixel** in an image, in order to **outline the shape** of the recognized objects.

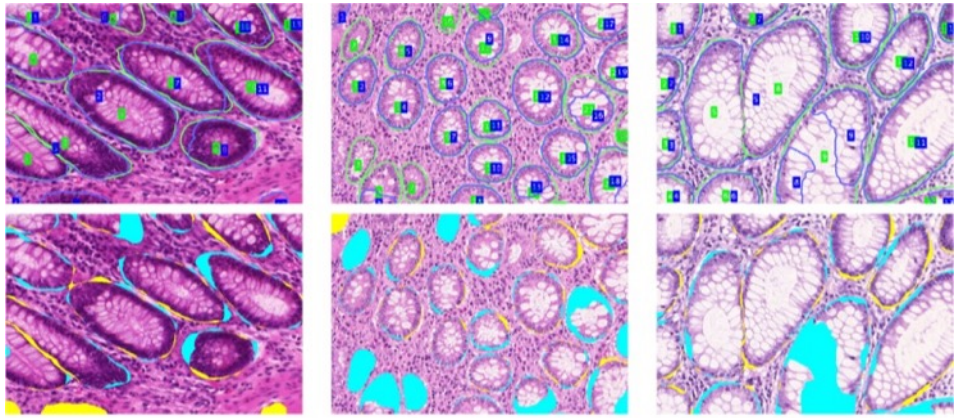
Semantic segmentation models were pioneered by **SegNet** (2015, link) and are continuous being improved, notably through the **COCO** (Common Objects in COntext) reference competition and dataset.

They are now used in multiple fields:

- Autonomous vehicles (ICNet, 2017, link and gif on the right)
- Medical image analysis (U-Net, 2015, link, and FCN, 2017, link)
- Satellite image analysis (e.g. Mask R-CNN by Facebook, state-of-the-art, 2017, link)

…And a lot more!

Real-time video segmentation

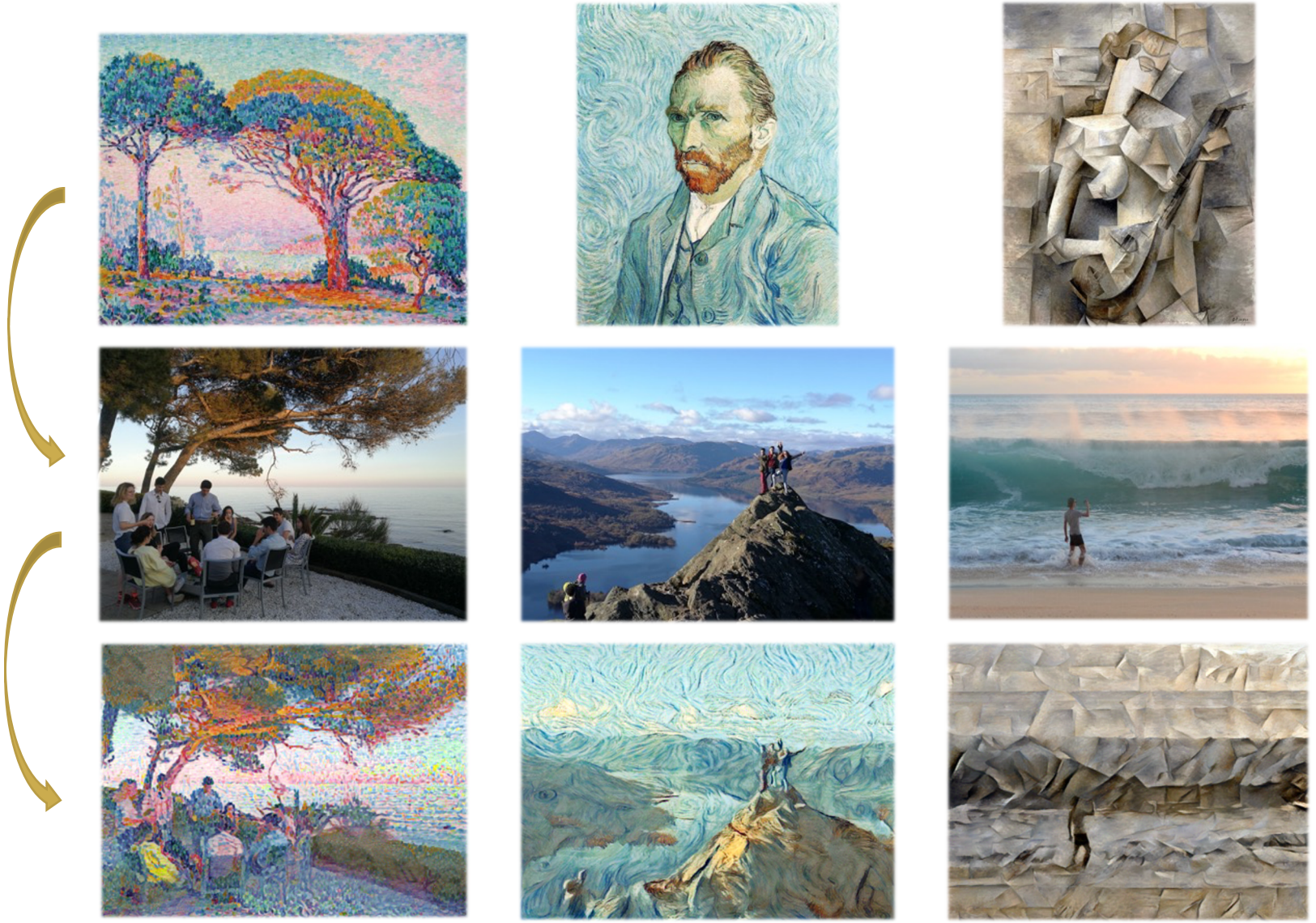


Histopathological tumor detection

Satellite image analysis

EXAMPLE (IV): NEURAL STYLE TRANFER

- We can use CNNs to **apply a new artistic style** to an image (2015, link) .
- The idea is that it is possible to use CNN feature maps to separate the style and the content representation from an image. Here, I ran a ‘content’ image and a ‘style’ image through the pre-trained VGG19 to obtain their content and style representations.
- I then defined a dual loss function, which corresponds (1) to the difference in style between the ‘input’ and the ‘style’ image, and (2) to the difference in content between the ‘input’ and the ‘output’ image.
- Minimizing this function with gradient descent transfers style from one image to the other, while making sure content isn't lost in the process.
- More recent approaches use GANs for faster and better results (link, 2019).



4

RECURRENT NEURAL NETWORKS (RNN)

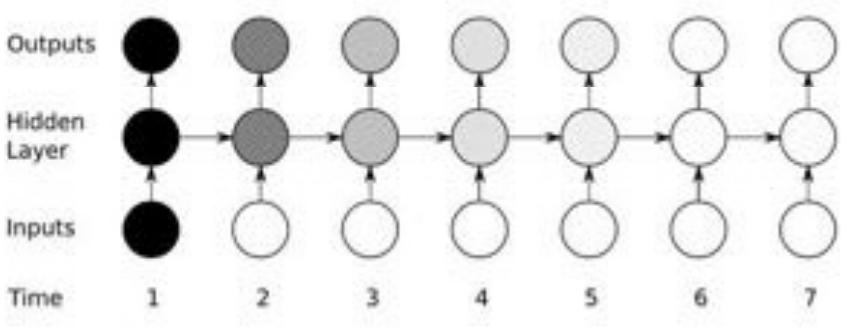
Architecture
Famous RNNs
Examples

RECURRENT NEURAL NETWORKS (RNN): A 'RECURRENT' ARCHITECTURE FOR SEQUENTIAL DATA ANALYSIS

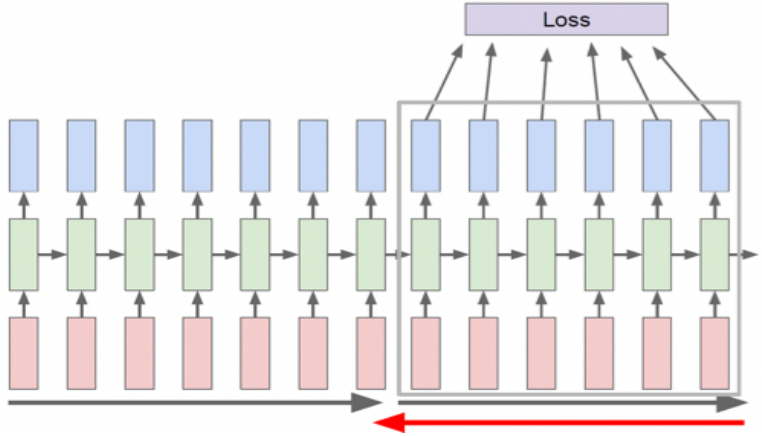
Previously seen networks are not so good at dealing with sequential or "temporal" data, because (1) they lack **"memory"** (i.e. they don't consider the inputs as connected together) and (2) they have a **fixed architecture** (fixed input and output sizes).

- RNNs solve this problem thanks to an architecture where **neurons in the hidden layers are connected together**. They not only transmit their output to the next layer, but also to the neuron next to them.
- As a consequence, RNNs **can take into account past instances of data** to predict future instances of a variable. Each output is computed based on inputs + on the "state matrix" corresponding to the outputs of the past steps. It works especially for stock market forecast, next-word prediction, etc.!
- This recurrent connectivity is even more **biologically realistic**. RNNs can also be mixed with CNN layers to get the best of both worlds.

*When training RNNs, 1 gradient step is very costly, as it requires to pass through the entire connected dataset. You are also at risk of the **"vanishing gradient"** problem: too many derivations bring gradients to 0 (i.e. no more weight update). So you usually work with batches of data steps. See appendix for maths!*



RNN structure (↑) and learning process (↓) for sequential data



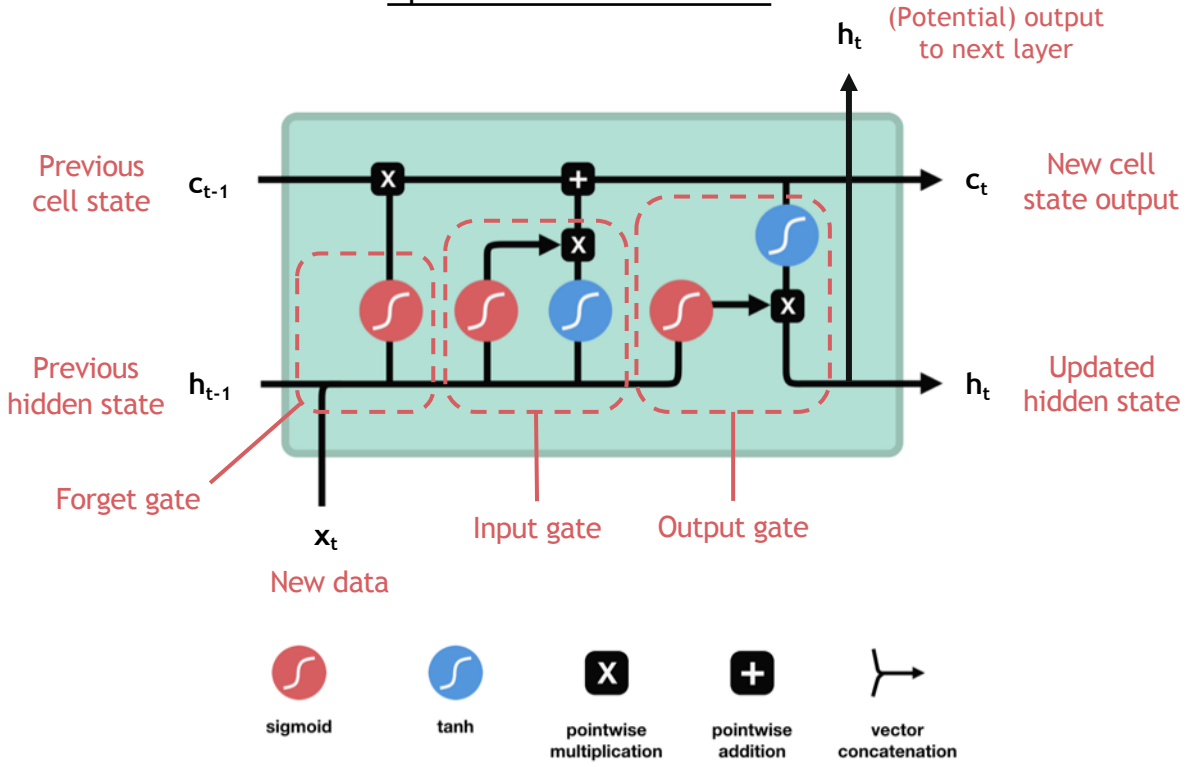
FAMOUS RNNs (1/2) – LONG SHORT-TERM MEMORY CELLS (LSTM)

The LTSM cells (invented in 1997 – link, improved in 2015 – link) and their close cousin the "Gated Recurrent Units" (GRU, 2015 - link) are special types of neurons that use the principle of **"gates"** to regulate the flow of information and **improve the way the network "remembers"**. This one is a little complex, stay focused!

Each LSTM neuron includes 2 data flows: the "cell state" (result of the computations of a single neuron, i.e. a sort of "short term memory") and the "hidden state" (outputs passed and updated from cell to cell, i.e. a sort of "long term memory").

1) At the entry of each cell, the "forget gate" receives the past hidden state and new data, and eliminates part of them.
2) The past hidden state and the new data also take a 2nd way to the "input gate", which runs several computations on them.
3) The results of the 2 firsts gates are combined with the previous cell state (line on top) to form the **new cell state**.
4) Finally, the "output gate" combines the new cell state and the inputs from the start (hidden state + new data) to produce the **new hidden state** (updated long-term memory).

Operations in an LSTM unit

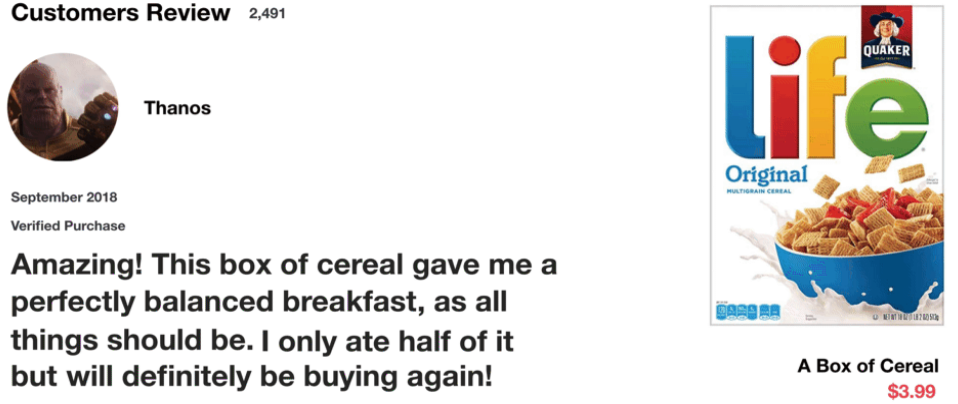


*See appendix for animated version

FAMOUS RNNs (2/2) – LONG SHORT-TERM MEMORY CELLS (LSTM)

- In short, the **forget gate** decides what is relevant to keep from prior steps. The **input gate** decides what information is relevant to add based on computations at the **current step**. The **output gate** uses current and past states to perform the update of the **hidden state**.
- This system is very efficient: if new data was important, the gate weights (**which are learnt during training**) will make sure that it persists and becomes part of the long-term state matrix (the hidden state) and consequently weighs more in the next cell. Conversely, **non-essential information is eliminated** (see example on right).
- LSTM and GRU (a variant of LSTM) are therefore efficient at **finding patterns in sequential data**, by remembering context from left to right. You can also stack 2 RNNs on top of each other to read a sequence in both directions, and get a **bi-directional model**.

*We can also remark that with LSTM, the information passed through the network is a **weighted average** of the previous hidden states and cell states - therefore, it can be bounded and rescaled, which (partly) solves the **vanishing gradient problem** (i.e. gradient stays away from 0 when computing through time steps, which accelerates training and avoids to shrink long term information).*



$$\mathbf{f}_t = \sigma(W_{f,h}\mathbf{h}_{t-1} + W_{f,x}\mathbf{x}_t + b_f) \quad \rightarrow \text{Forget gate}$$

$$\mathbf{i}_t = \sigma(W_{i,h}\mathbf{h}_{t-1} + W_{i,x}\mathbf{x}_t + b_i) \quad \rightarrow \text{Input gate - part 1}$$

$$\mathbf{g}_t = \tanh(W_{g,h}\mathbf{h}_{t-1} + W_{g,x}\mathbf{x}_t + b_g) \quad \rightarrow \text{Input gate - part 2}$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t \quad \rightarrow \text{New cell state}$$

$$\mathbf{o}_t = \sigma(W_{o,h}\mathbf{h}_{t-1} + W_{o,x}\mathbf{x}_t + b_o) \quad \rightarrow \text{Output gate - part 1}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \quad \rightarrow \text{Output gate - part 2: new hidden state}$$

Each gate includes weighted average operations (weights learnt during training) and of a non-linear activation function.

LSTM equations

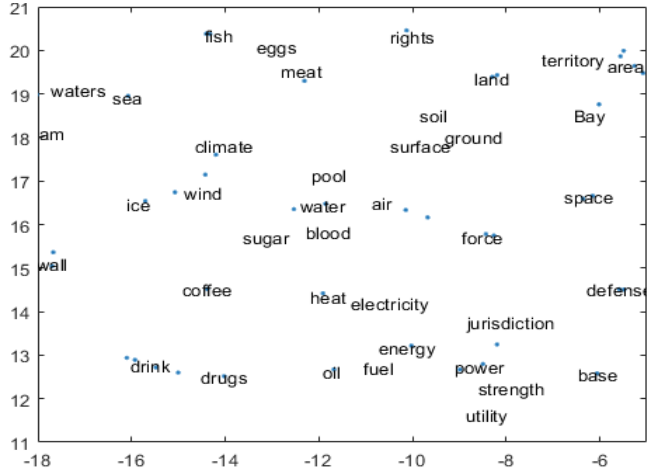
EXAMPLE (I): WORD EMBEDDINGS AND SENTIMENT ANALYSIS (1/2)

*The most straightforward use of RNNs might be **text analysis**, or NLP **(Natural Language Processing)**. But to analyse a sequence of words, you first need to **convert it into a sequence of numbers**, if possible carrying "meaning" - these word vectors are called word embeddings.*

There are several methods to create word embeddings:

- One-hot encoding: the most basic - take all the vocabulary in your corpus and assign each word a long vector of zeros with just a "1" at a specific position (e.g. word frequency).
- TF-IDF (Term Frequency - Inverse Document Frequency): statistical method to create vectors based on a more elaborate computation of word frequency, in order to penalize words that are very common (and not-so-important to capture meaning).
- Word2Vec (Google, 2013, link): creates **dense representations** (i.e. shorter vectors, with no zeros) that impressively capture meaning – see on right. It does so by using a neural network to either predict a word from context ("**Bag Of Words**" method) or predict the surrounding words from a given word ("**Skip-gram**" method). Model inputs are one-hot encoded words, and in the prediction process (hidden layers) it builds dense representations!
- More recent improvements include: **GloVe** (2014), **FastText** (2016), **ELMo** (2018), and most famously BERT (Google, 2018, link – followed by many variants such as "**CamemBERT**" for French words), current state-of-the-art, based on "**Transformers**" (see next section!).

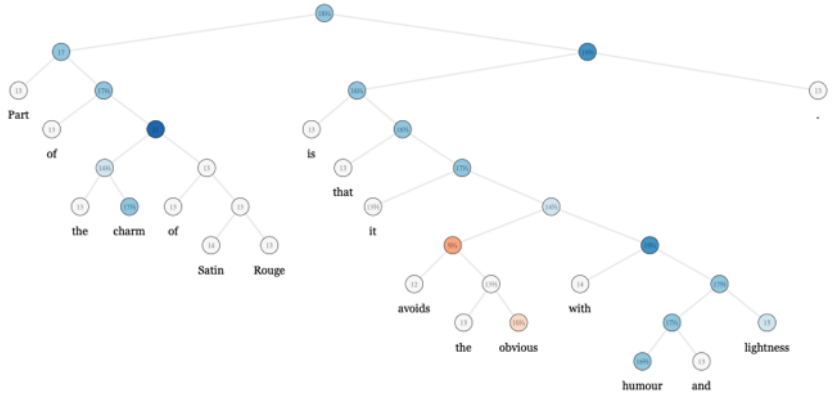
How vectors can represent meaning (Word2Vec)



Word embedding t-SNE plot (cosine similarity)

EXAMPLE (I): WORD EMBEDDINGS AND SENTIMENT ANALYSIS (2/2)

*Once you have word embeddings, you can "embed" your sentences and **feed them to a RNN**. The model can then output what you want - in the case of sentiment analysis, we want to **get a classification of sentences**, for example **"positive" vs. "negative" sentiment**.*



Sentence parsing

Word Cloud

Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	(None, None, 100)	1300000
lstm_3 (LSTM)	(None, None, 64)	42240
lstm_4 (LSTM)	(None, 32)	12416
dense_2 (Dense)	(None, 5)	165

Total params: 1,354,821
Trainable params: 1,354,821
Non-trainable params: 0

Simple model summary

- In this example, we try to determine the dominating sentiment in Rotten Tomatoes movie reviews (link).
- We can start by parsing the sentence (link) in several chunks, in order to (later) average predictions over several sub-phrases, for more accuracy.

- We can also visualize the most common words in negative reviews from our training set.
- Before going to the model, we might need to pad sentences with zeros so that they are all of the same length (same number of words).

- LTSM is a good choice for the "memory" property (ex: for "The movie was not so good", we need to remember the first negative words when we get to "good"!).
- The embedding output shape is of size 100, because each word is assigned a dense vector of size 100 (arbitrary size - could be more!).

EXAMPLE (II): ATTENTION MECHANISMS AND IMAGE CAPTIONING

*Beyond word embeddings, a great improvement to NLP was the introduction of "attention mechanisms" in the past few years, culminating with the famous **"Attention Is All You Need"** paper in 2017 (Google, link).*

- When computing predictions based on an input sequence, all parts of the sequence aren't of equal importance. Attention consists in providing the model with information about **which part of the sequence is more important to determine the model output**. In the model, each step of the input sequence corresponds to an internal "hidden state", so we compute an **"attention vector"** as a **weighted average of the model hidden states**, with **more weight on the important steps**. Weights can be obtained in different ways – see appendix for details.
- The principle of attention can also be applied to images, as shown in the the example below (Univ. of Montréal, 2015, link). In this model, **a CNN first "encodes"** the image features into hidden states (or put differently, into a "latent space"), then **a RNN "decodes"** the features to output the most probable image caption. For each output word, the decoder uses **2 sources of information**: the image features (captured through the encoder hidden states) and the **attention scores** (weighted average of hidden states/encoded features).

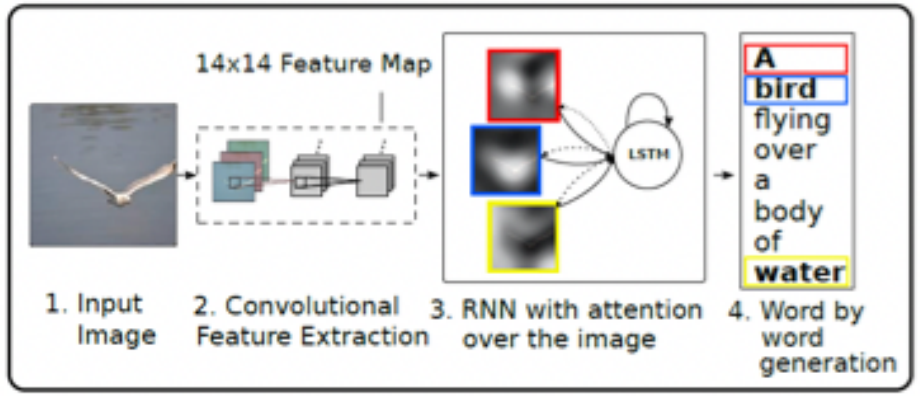
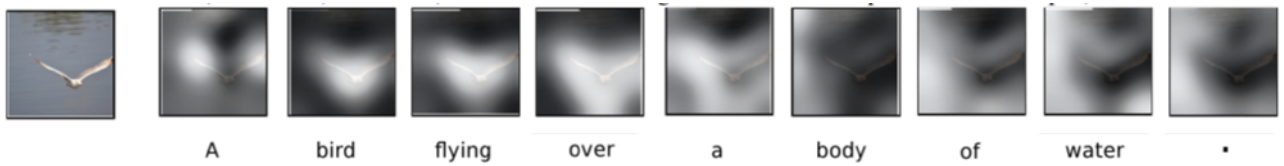


Image captioning model structure (link to paper)

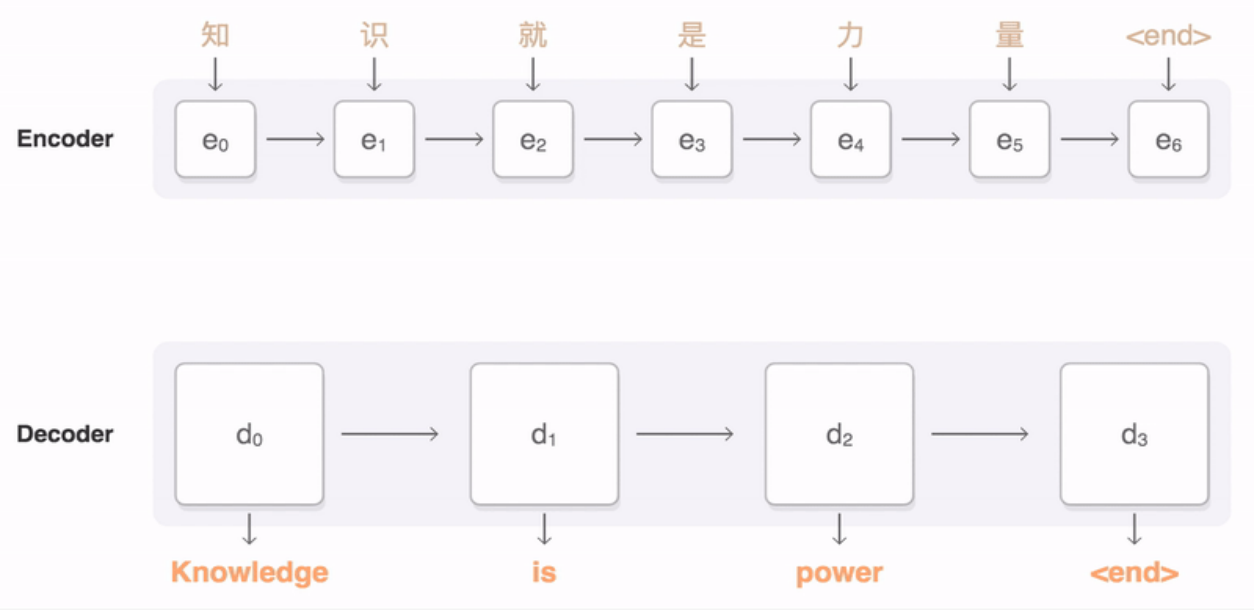


Where attention focuses (light areas) at each decoder LSTM step

EXAMPLE (III): SEQ2SEQ AND MACHINE TRANSLATION

*As you saw previously, it is possible to implement successively an **encoder** network and a **decoder** network, in order to first extract information from data into a latent feature space and then to compute a desired result - e.g. a caption, or text translation.*

- In the case of sequential data, this encoder-decoder architecture is known as **"sequence-to-sequence"** (**Seq2Seq**). Pioneering papers were published in 2014 (link, link) and have been extremely popular recently. Encoder and decoders can be **different types of neural networks** and can include **attention** mechanisms.
- **Google Translate** started using such a model in late 2016. Before **Neural Machine Translation** (NMT), translation was based on decades of research in **Statistical Machine Translation** (SMT), which used complex sets of rules to capture grammar structures.
- NMT provides faster, more accurate and scalable performance (you just need training data to learn a new language), but is less interpretable – **interpretability** is a classic problem in DL, where complex neural networks make it difficult to trace back errors!



Other applications of seq2seq architectures include, question answering, text summarization, conversational and generative models… you name it !

5

OTHER TYPES OF DEEP LEARNING MODELS

Generative Adversarial Networks (GANs)
Transformers / Autoencoders
Recommender systems
Self-Organizing Maps (SOMs)
Bayesian networks / Graph methods

GENERATIVE ADVERSARIAL NETWORKS FOR IMAGE CREATION (1/3)

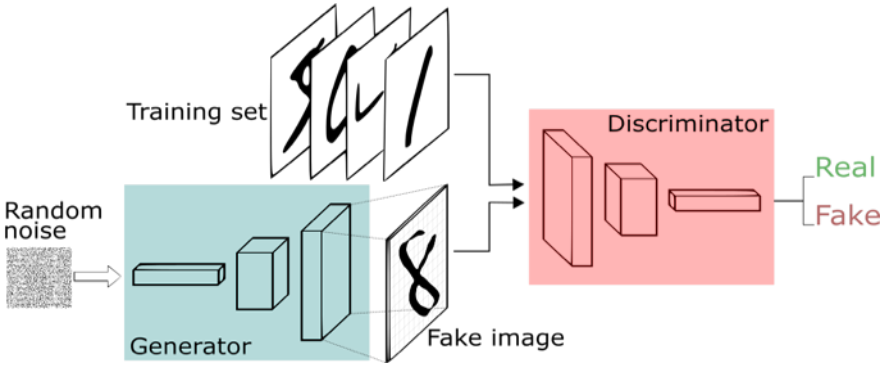
*Generative Adversarial Networks (GANs), originally proposed by **Ian Goodfellow** in 2014 (link), have been dubbed by Yann Le Cun as **"the most interesting idea in the last 10 years in ML"** (2016). They open a whole new field of creativity for artificial intelligence.*

The core principle of GANs is **"adversarial training"**, based on **2 modules**:

1) **Generator (G)**: the generator creates an image based on a certain input, for example a vector of **random noise** x, which goes through upsampling and conv layers (or through an autoencoder) to create a new image $G(x)$.
2) **Discriminator (D)**: the discriminator tries to classify the generated image as either fake ($D(G(x) = 0$) or real ($D(G(x) = 1$).

Training procedure

- We input to the discriminator both **real and fake images**, and our objective is to maximize the chance to recognize training set images as real and generated images as fake, i.e. the **maximum likelihood of the observed data** - eq. (1).
- On the other hand, we want the generator to output images with the highest possible value of $D(x)$, to **fool the discriminator** – see equation (2).
- Therefore we are in a "minimax" optimization game, modelled by equation (3). We train each module **alternatively** (i.e. we fix G's parameters and perform a single gradient descent iteration on D, then we switch sides), until the model **converges** - i.e. until D cannot tell the difference (output ½) !



Deep Convolutional GAN architecture (2016, link)

(1) $$\max_D V(D) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

recognize real images better — recognize generated images better

(2) $$\min_G V(G) = \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

Optimize G that can fool the discriminator the most.

(3) $$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

Original GAN objective functions

GENERATIVE ADVERSARIAL NETWORKS FOR IMAGE CREATION (2/3)

Image generation

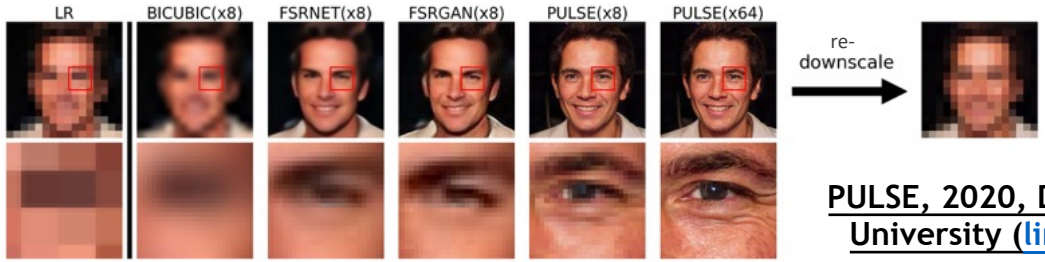
GANs have dramatically improved in the past few years, allowing us to generate entirely fake images in high quality. Go check the famous website *thispersondoesnotexist.com* (examples below)!

GANs can actually create any type of image that you teach them – paintings (see the start-up *noartist*), cars, houses, etc.

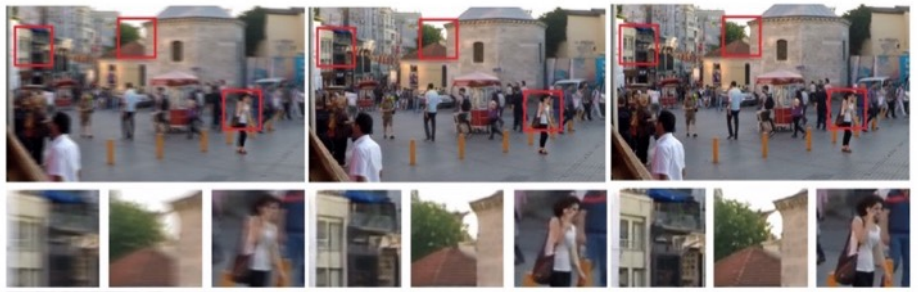
Face morphing (interpolation between GAN-generated images)

Super-resolution

Super-resolution is the task of re-building a high quality image from a pixelized image. The most performant solutions are based on GANs, and you can try a demonstration of the PULSE model (2020) through this link.



PULSE, 2020, Duke University (link)



DeblurGAN, 2018 (link)

Left: blurred, center: model, right: truth.

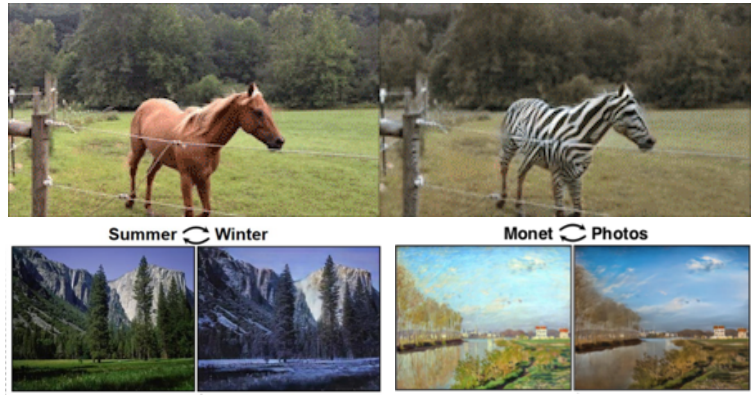
GENERATIVE ADVERSARIAL NETWORKS FOR IMAGE CREATION (3/3)

Finally, a very interesting thing about GANs is that the generator learns, in an unsupervised way, **how to map a vector to points on an image**. It can therefore be seen as **parameterizing the complicated surface of real data**. In particular, if the generator has an **encoder-decoder structure**, we can adjust several parameters and condition the way generators sample images from their latent space. See some examples of such techniques below.

Image-to-image translation

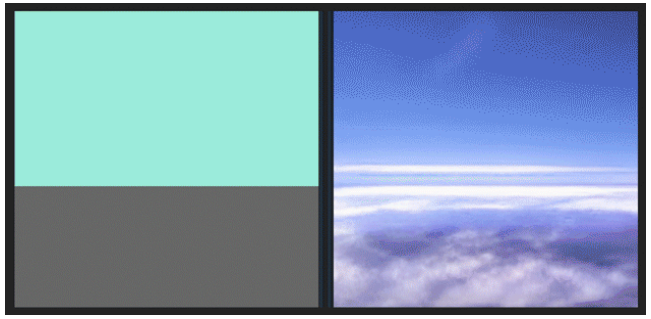
Pix2Pix model, Berkeley, 2016 (link)

Image generation from automatically-generated contours and from hand-drawn sketches



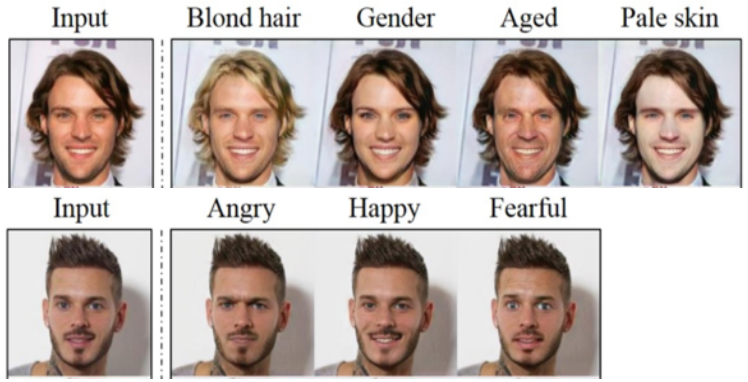
CycleGAN, Berkeley, 2017 (link)

Cycle consistency (i.e. the output of a 1st generator could be re-processed and match the original image) allows to train the translation model without paired examples.



GauGAN, Nvidia, 2019 (link)

Photorealistic image synthesis from segmentation maps (labelled sketches that depict the layout of a scene), named after post-Impressionist painter Paul Gauguin.



StarGAN, 2018 (link)

StarGAN learns the mappings among multiple domains in a single generator and a single discriminator.

See other examples of models in Appendix.

TRANSFORMERS: PIXEL RNN AND CNN

WIP

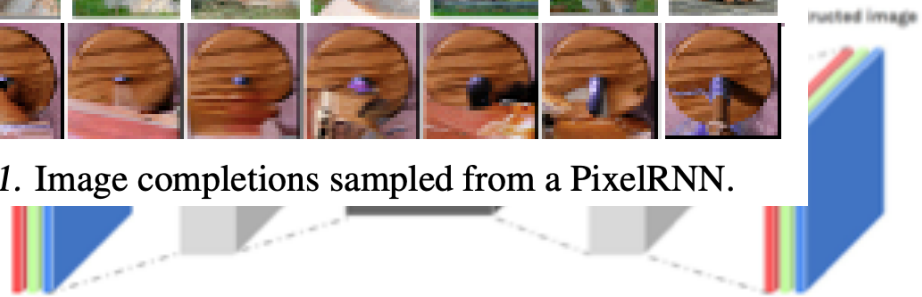
https://arxiv.org/pdf/1601.06759.pdf Pix2Pix

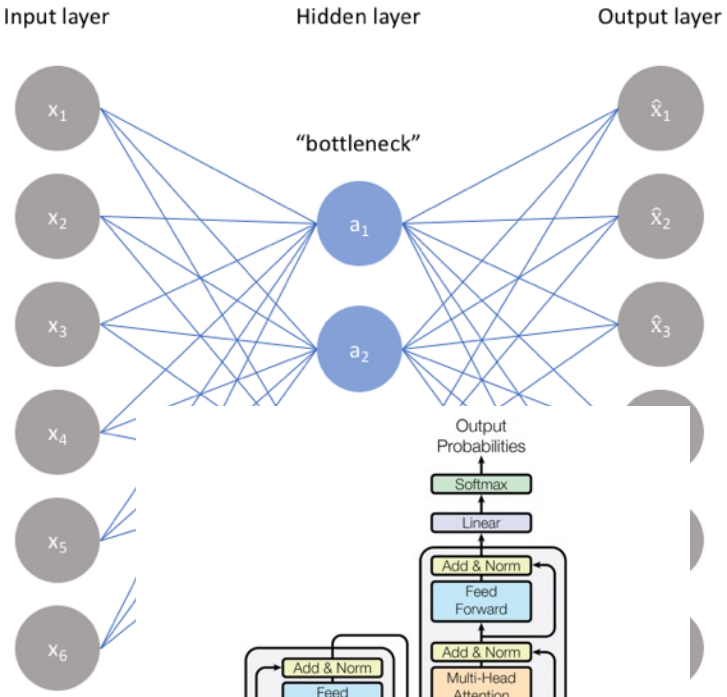
https://arxiv.org/pdf/1802.05751.pdf image transformer
https://openai.com/blog/image-gpt/#rf2 open AI image GPT

Autoencoders — Representing Data in a Compressed Way

Most neural networks take in data and make some types of decisions.
Autoencoders have a different task, and that is to figure out a way to compress data but maintain the same quality.
Traditionally in machi
and the goal of the ne
are the same as the i
So, in this architectur
hidden layer is smalle
is called the **"bottle**
a way to compress th
compression is often
maintain high quality.

Figure 1. Image completions sampled from a PixelRNN.





Multi-head attention layers

WIP

AUTOENCODERS FOR SPEECH AND MUSIC SYNTHE

This leaves room for impressive image generation and manipulation possibilities, such as for example doing arithmetic on faces in the latent vector space: [man with glasses] - [man without glasses] + [woman without glasses] = [woman with glasses]. See examples in the next slide.

Autoencoders *encode* input data as vectors. They create a hidden, or compressed, representation of the raw data. They are useful in dimensionality reduction; that is, the vector serving as a hidden representation compresses the raw data into a smaller number of salient dimensions. Autoencoders can be paired with a so-called decoder, which allows you to reconstruct input data based on its hidden representation, much as you would with a restricted Boltzmann machine.

Variational autoencoders are generative algorithm that add an additional constraint to encoding the input data, namely that the hidden representations are normalized. Variational autoencoders are capable of both compressing data like an autoencoder and synthesizing data like a GAN. However, while GANs generate data in fine, granular detail, images generated by VAEs tend to be more blurred.

WaveNet

Music transformer https://magenta.tensorflow.org/music-transformer, https://magenta.tensorflow.org/transformer-autoencoder

https://cloud.google.com/text-to-speech

Wavenet, 2016: https://arxiv.org/pdf/1609.03499.pdf, https://deepmind.com/blog/article/wavenet-generative-model-raw-aud
Parallel wavenet, 2017: https://arxiv.org/pdf/1711.10433.pdf, https://deepmind.com/blog/article/high-fidelity-speech-synthes
Blog post https://towardsdatascience.com/wavenet-google-assistants-voice-synthesizer-a168e9af13b1

Open AI music models: musenet https://openai.com/blog/musenet/, jukebox: https://openai.com/blog/jukebox/

WIP

NEURAL NETWORKS TO TRAIN RECOMMENDER SY

The classic recommender system architecture is the Collaborative filtering model. It is based on factorization of a matrix compiling all explicit feedbacks from users – for example, all movie ratings per user.

Netflix organized a famous competition in 2006-2010 to improve state-of-the-art in this domain.

Today, the most recent approaches use neural networks to include, next to explicit feedback, implicit feedback, content features and content/user/system metadata, which are all embedded and fed to neural networks, which provide even more accurate recommendations.

https://towardsdatascience.com/building-a-recommendation-system-using-neural-network-embeddings-1ef92e5c80c9

WIP

SELF-ORGANIZING MAPS FOR UNSUPERVISED CLU

WIP

GRAPH METHODS & RANDOM WALKS FOR NODE D

BAYESIAN NETWORKS

Variational autoencoders (GANs)
Attention mechanisms
Radial basis function network
Restricted Boltzmann machine
Deep belief networks (graph) with walk-sleep algo
Bayesian networks

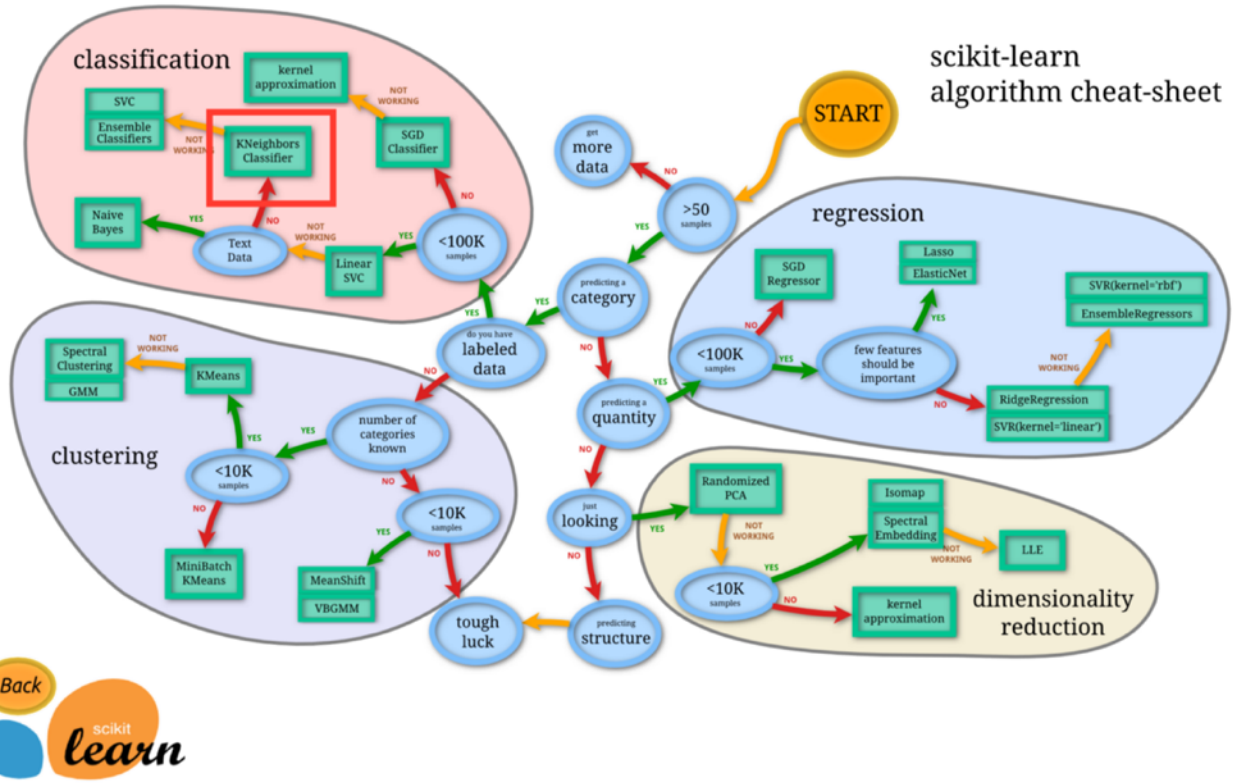
6

APPENDIX

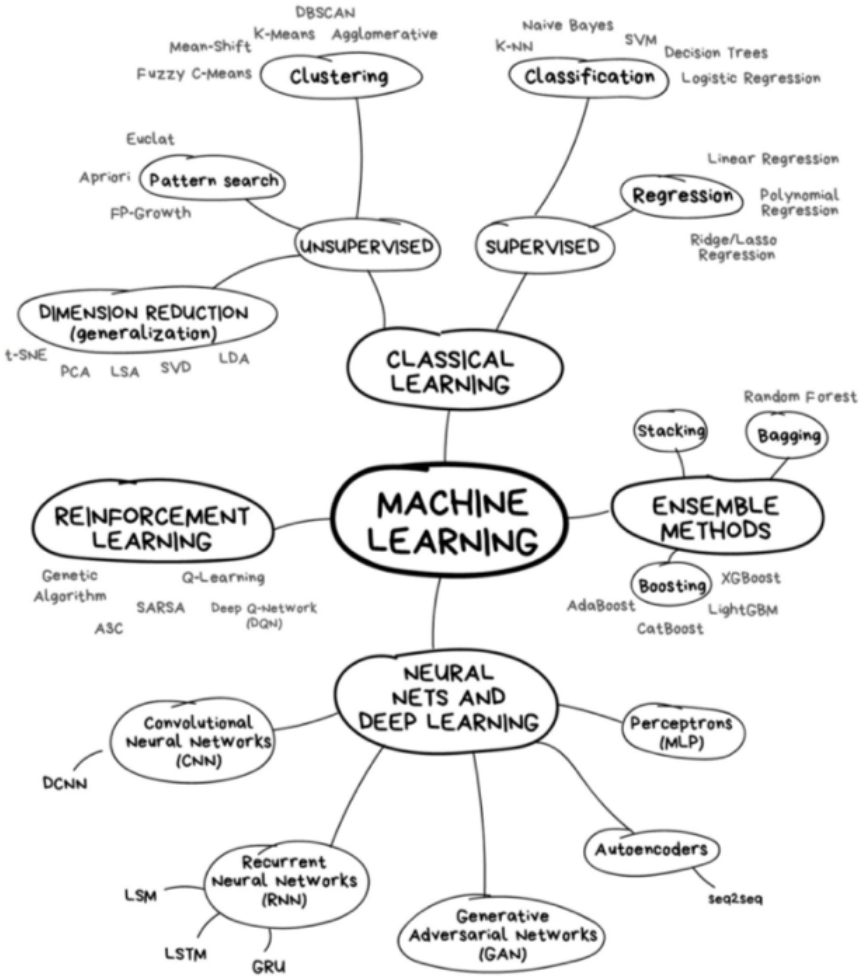
More information!

Supplementary material – Machine Learning

- **Other Machine Learning overviews**

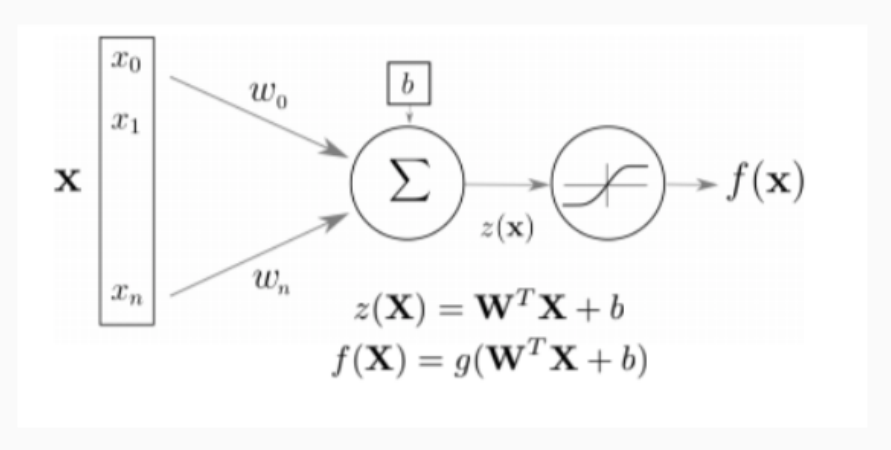


Back

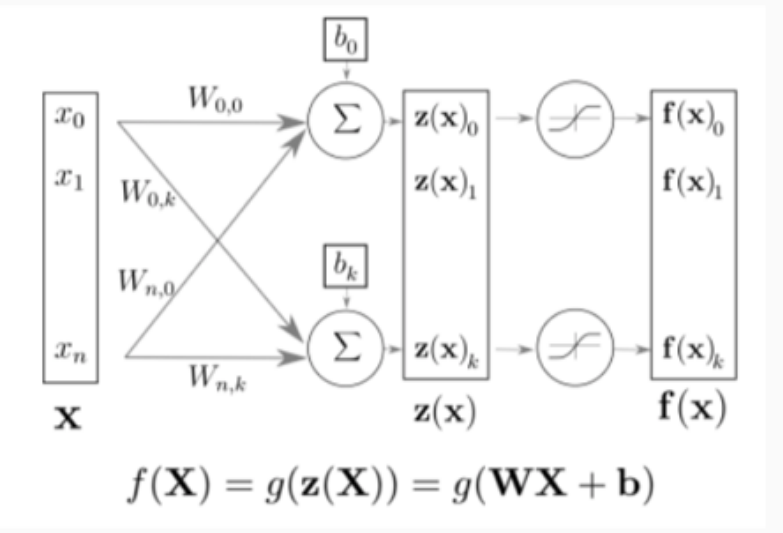


Supplementary material – MULTI-LAYER PERCEPTRON (ANNs)

- **Other neuron and FFNN diagrams**



A neuron

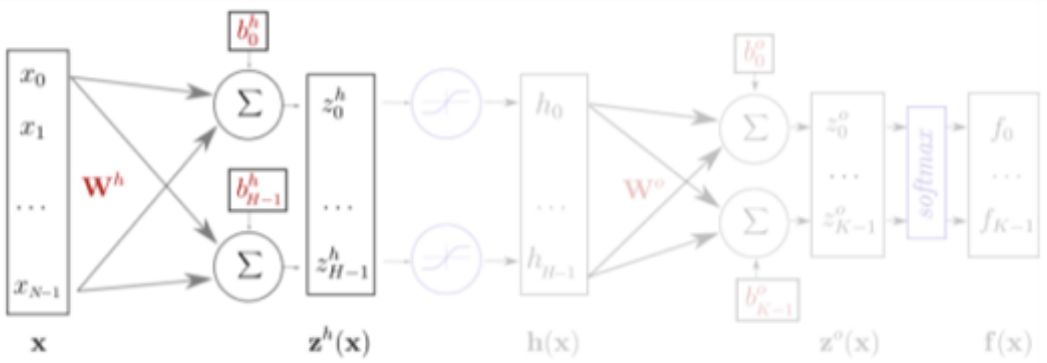


Layer of neurons and hidden states

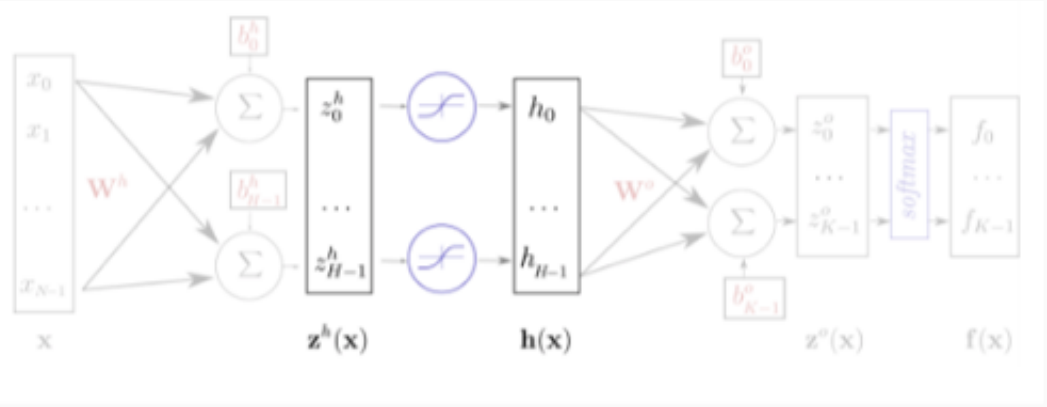
Supplementary material - ANNs

- **Feed-forward network, step-by-step**

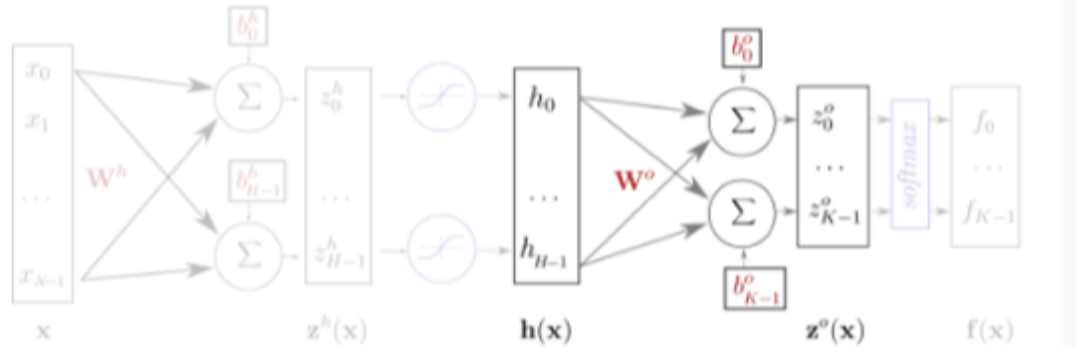
1



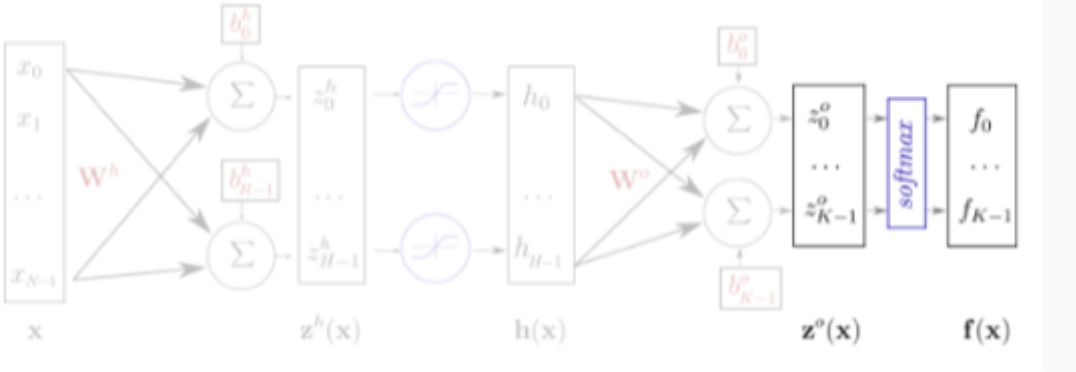
2



3



4



→ X **input in** $\mathbb{R}^d$.

→ $z^h(X)$ **pre-activation in** $\mathbb{R}^H$, with **weight** $W^h \in \mathbb{R}^{d \times H}$ and **bias** $b^h \in \mathbb{R}^H$.

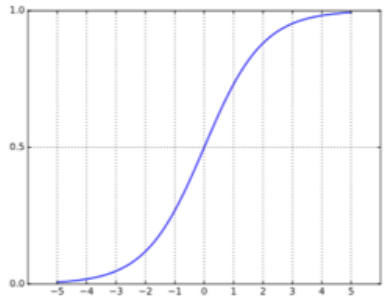
→ g **any activation function** to produce $h \in \mathbb{R}^H$.

→ $z^o(X)$ **pre-activation in** $\mathbb{R}^M$, with **weight** $W^o \in \mathbb{R}^{H \times M}$ and **bias** $b^o \in \mathbb{R}^M$.

→ Apply the **softmax function to produce the output**, i.e. $\mathbb{P}(Y = m|X)$ for $1 \leqslant m \leqslant M$.

Supplementary material - ANNs

- **Different activation functions**

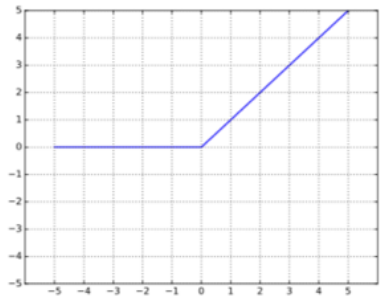


Sigmoid

- Sigmoid function

$$x \mapsto \frac{\exp(x)}{1+\exp(x)}$$

- Problems:
  1. Sigmoid is not a zero centered function -> need for rescaling data.
  2. Saturated function: gradient killer -> need for rescaling data
  3. Plus: exp is a bit computational expensive

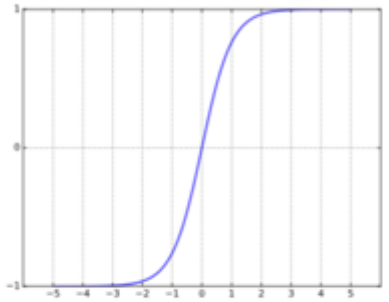


Rectified Linear Unit

- ReLU / positive part

$$x \mapsto \max(0, x)$$

- Problems:
  1. Not a saturated function.
  2. Computationally efficient
  3. Empirically, convergence is faster than sigmoid/tanh.
  4. Plus: biologically plausible



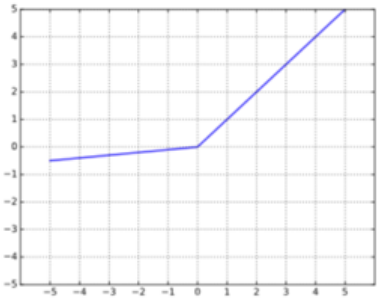
Tanh

- Hyperbolic tangente function

$$x \mapsto \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$$

- Problems:
  1. Tanh is a zero centered function -> no need for rescaling data
  2. Saturated function: gradient killer -> need for rescaling data
  3. Plus: exp is a bit computational expensive

Note: $\tanh(x) = 2\sigma(2x) - 1$.



Leaky ReLu / Parametric ReLu / Absolute Value Rectification

$$x \mapsto \max(\alpha x, x)$$

- Leaky ReLU: $\alpha = 0.1$

  ["Rectifier nonlinearities improve neural network acoustic models", Maas et al. 2013]

- Absolute Value Rectification:

  $\alpha = -1$

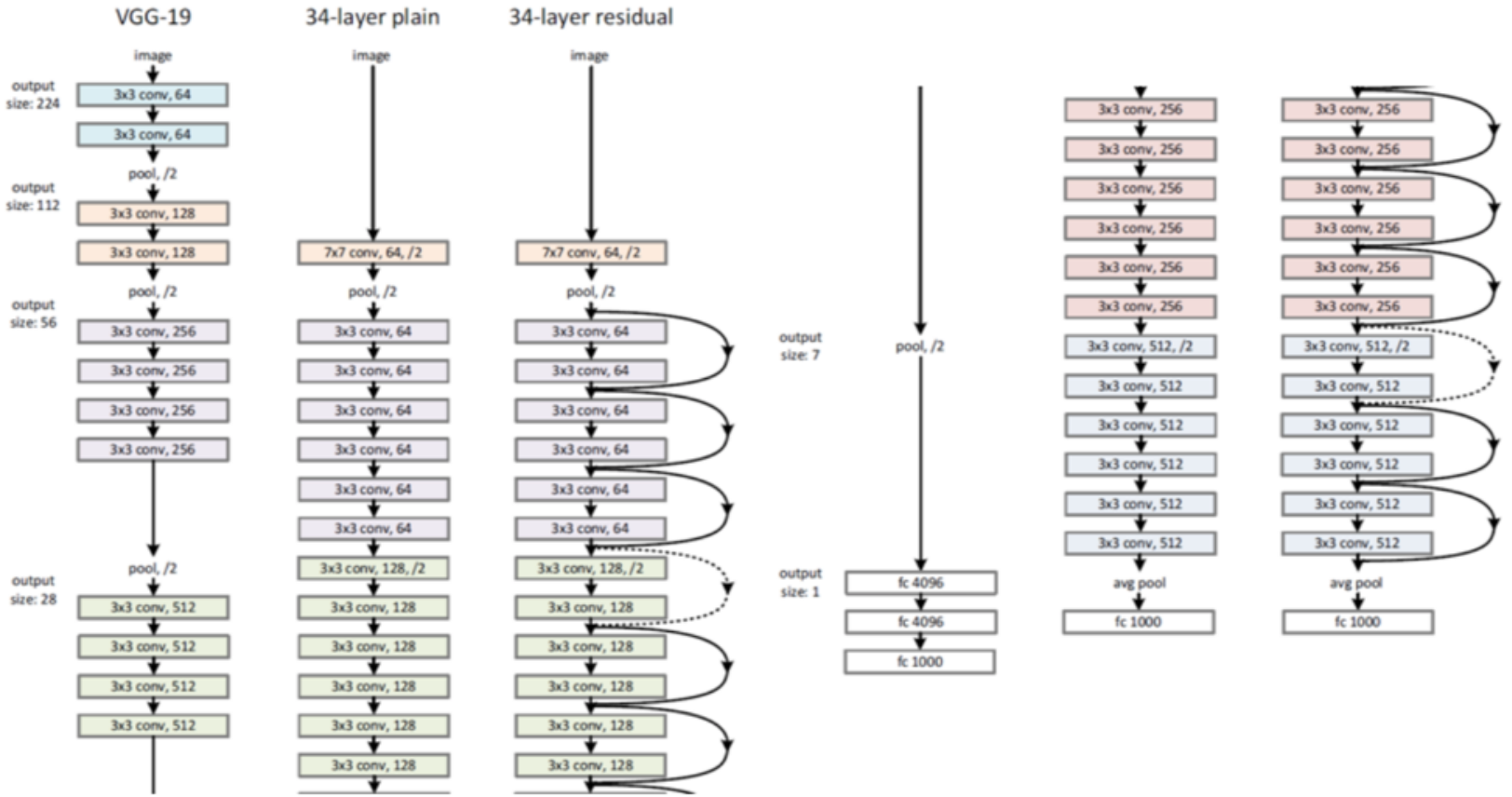
  ["What is the best multi-stage architecture for object recognition?", Jarrett et al. 2009]

- Parametric ReLU: α optimized during backpropagation. Activation function is learned.

  ["Empirical evaluation of rectified activations in convolutional network", Xu et al. 2015]

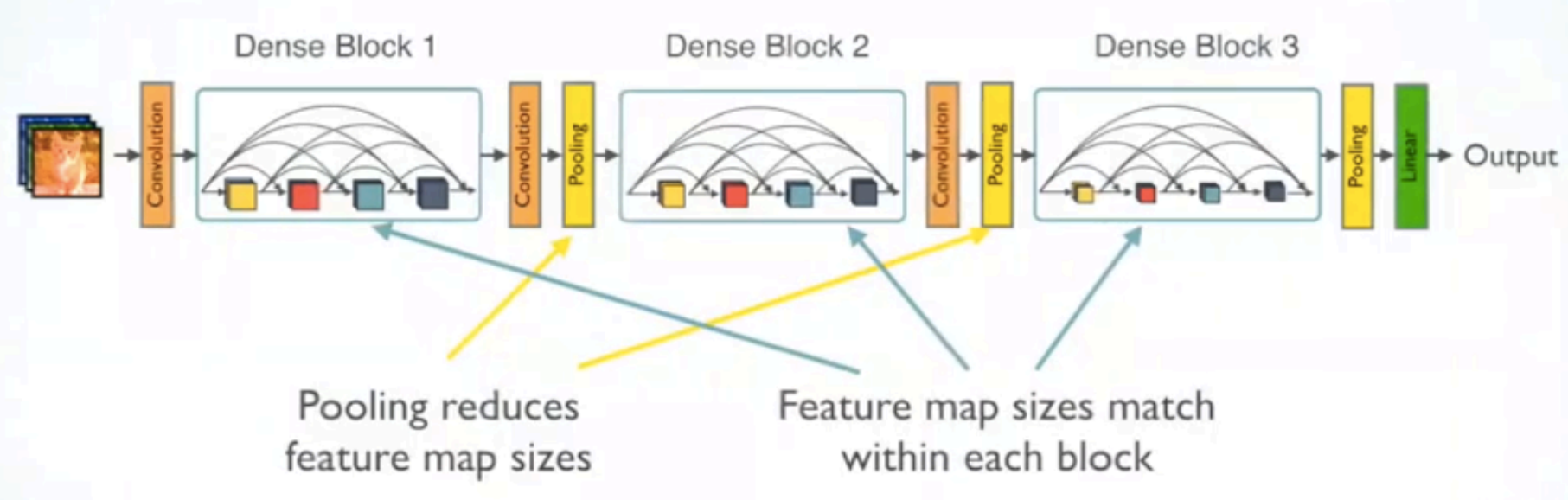
Supplementary material - CNNs

- **Structure of ResNet**



Supplementary material - CNNs

- **Structure of DenseNet**



Supplementary material - CNNs

- **“Classic” data augmentation techniques**

1.1 **Sampling** [“Imagenet large scale visual recognition challenge”, Russakovsky et al. 2015]

2.2 **Translation/shifting** [“Deep convolutional neural networks and data augmentation for environmental sound classification”, Salamon and Bello 2017]

3.3 **Horizontal reflection/mirroring** [“Mirror, mirror on the wall, tell me, is the error small?”, H. Yang and Patras 2015]

4.4 **Rotating** [“Holistically-nested edge detection”, Xie and Tu 2015]

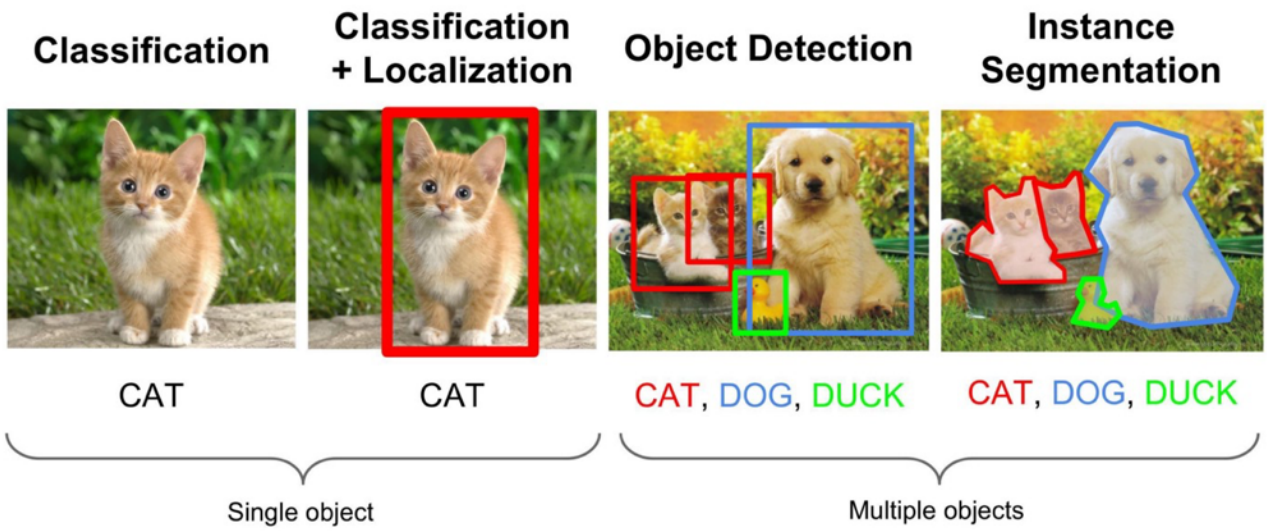
5.5 **Various photometric transformations**

6.6 Add noise to input

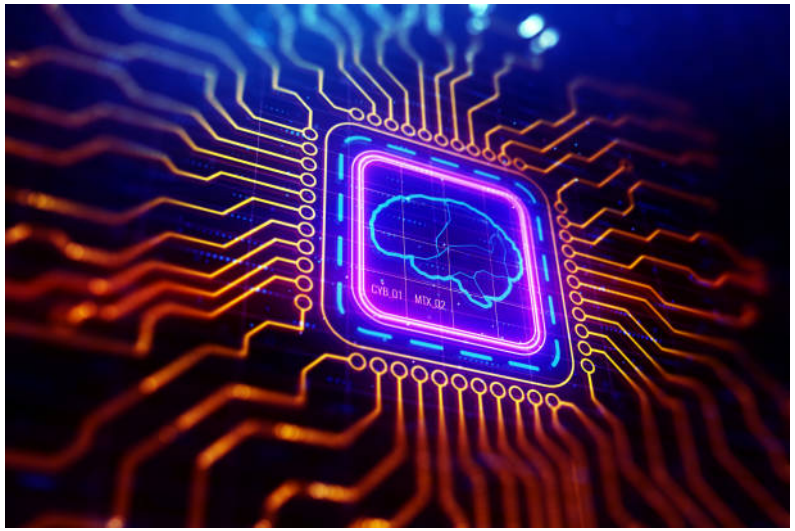
- **Another interesting side-application of image segmentation with CNNs: Pose Detection, for example the OpenPose model (2016, link)**

Supplementary material - CNNs

- **Difference between various object detection tasks**



- **Nice illustration**



Supplementary material - RNNs

- **Some RNN mathematics**

Hidden neuron:

$$\mathbf{h}_t = \tanh(W_{HH}\mathbf{h}_{t-1} + W_{IH}\mathbf{x}_t + \mathbf{b}_h)$$

Output neuron:

$$\hat{y}_t = \mathrm{softmax}(W_{HO}\mathbf{h}_t + \mathbf{b}_{out})$$

Backpropagation equation

$$\frac{\partial L_T}{\partial W_h} = \frac{\partial L_T}{\partial y_T}\sum_{k=1}^{T}\frac{\partial y_T}{\partial \mathbf{h}_T}\left(\prod_{m=k+1}^{T}\frac{\partial \mathbf{h}_m}{\partial \mathbf{h}_{m-1}}\right)\frac{\partial \mathbf{h}_k}{\partial W_h}$$

where

$$\frac{\partial \mathbf{h}_m}{\partial \mathbf{h}_{m-1}} = \prod_{m=k+1}^{j} W_h^T \mathrm{diag}(\tanh'(W_h\mathbf{h}_{m-1} + W_x\mathbf{x}_m)).$$

"Vanishing gradient"

Recall that the 2-norm of a matrix A is given by

$$\begin{aligned}\|A\|_2 &= \sup_{\mathbf{x}\neq 0}\frac{\|A\mathbf{x}\|_2}{\|\mathbf{x}\|_2}\\ &= \sup\{\|A\mathbf{x}\|_2, \|\mathbf{x}\|_2 = 1\}\\ &= \sqrt{\lambda_{max}(A^TA)}.\end{aligned}$$

Assuming that $\tanh'(u) \leq \gamma$, and that the largest eigenvalue of W_h^T is bounded above strictly by $1/\gamma$.

$$\left\|\frac{\partial \mathbf{h}_m}{\partial \mathbf{h}_{m-1}}\right\| \leq \left\|W_h^T\right\| \left\|\mathrm{diag}(\tanh'(W_h\mathbf{h}_{m-1} + W_x\mathbf{x}_m))\right\| < 1.$$

Thus, there exists $0 < \eta < 1$ such that

$$\left\|\prod_{m=k+1}^{T}\frac{\partial \mathbf{h}_m}{\partial \mathbf{h}_{m-1}}\right\| \leq \eta^{T-k}.$$

As $T - k$ gets larger, the contribution of the kth term to the gradient decreases exponentially fast.

Supplementary material - RNNs

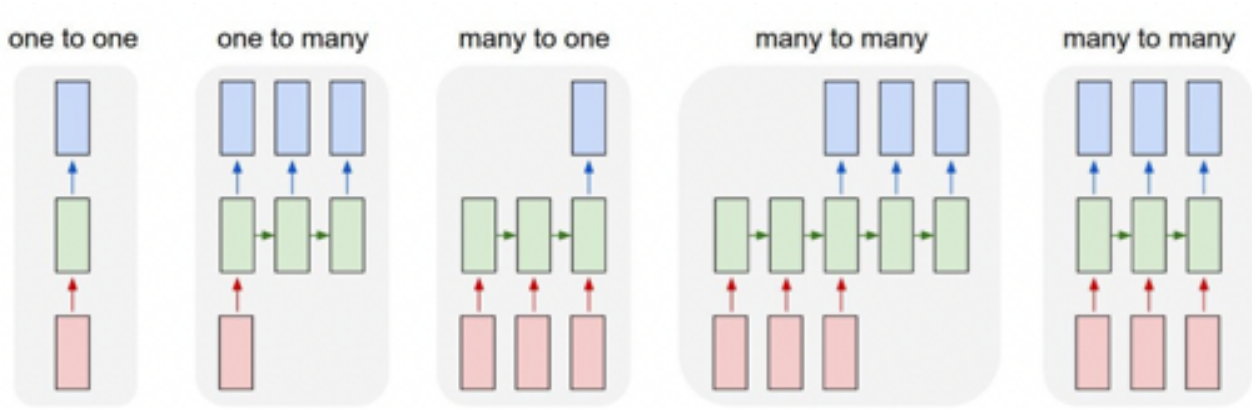
- **Clever weight initialization techniques**

✓ The weight matrix *Wh* is initialized as the identity, biases are set to zero, with the ReLU activation function. [“A simple way to initialize recurrent networks of rectified linear units”, Le et al. 2015]

✓ Learn a weight matrix that is a mixture of Identity matrix and another matrix (mix of long-term and small-term dependencies). [“Learning longer memory in recurrent neural networks”, Tomas Mikolov, Joulin, et al. 2014]

✓ Initialize *W* randomly among definite positive matrix (real and positive eigenvalues) with one eigenvalue of 1 and the other less (or equal) than 1

- **RNNs offer a lot of variability**

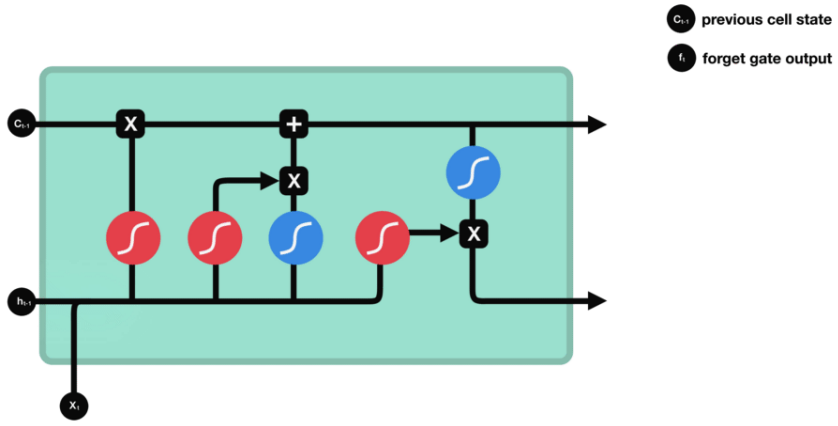


Supplementary material - RNNs

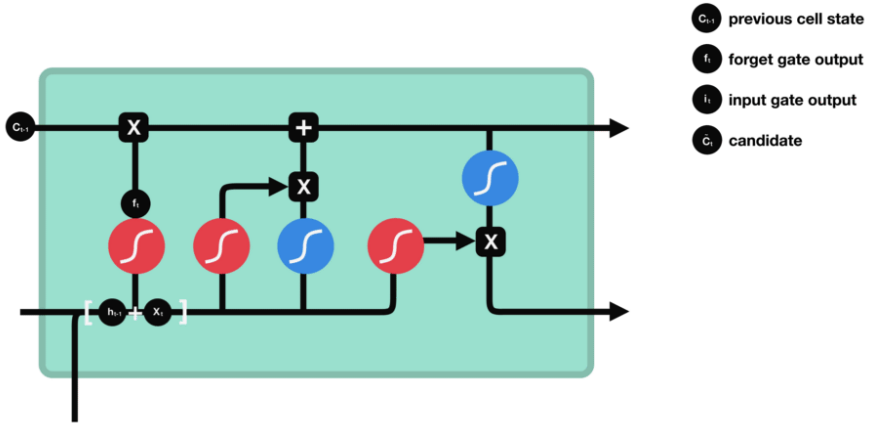
To sum up on LSTM, the forget gate decides what is relevant to keep from prior steps. The input gate decides what information is relevant to add from the current step. The output gate determines what the next hidden state should be.

- **LSTM gifs**

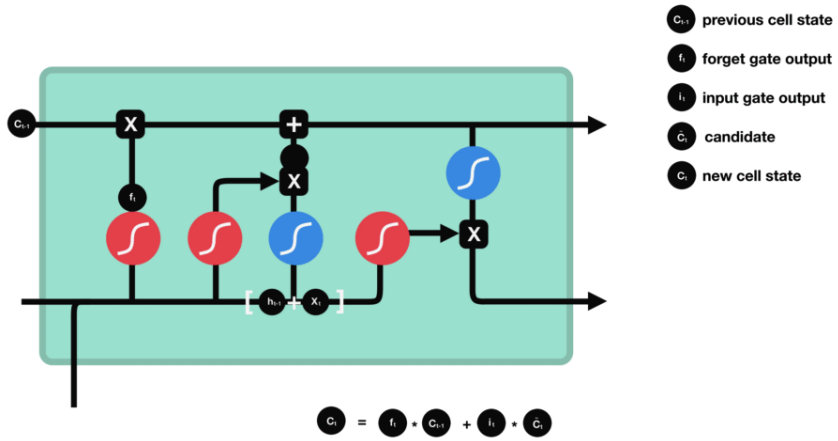
Forget gate



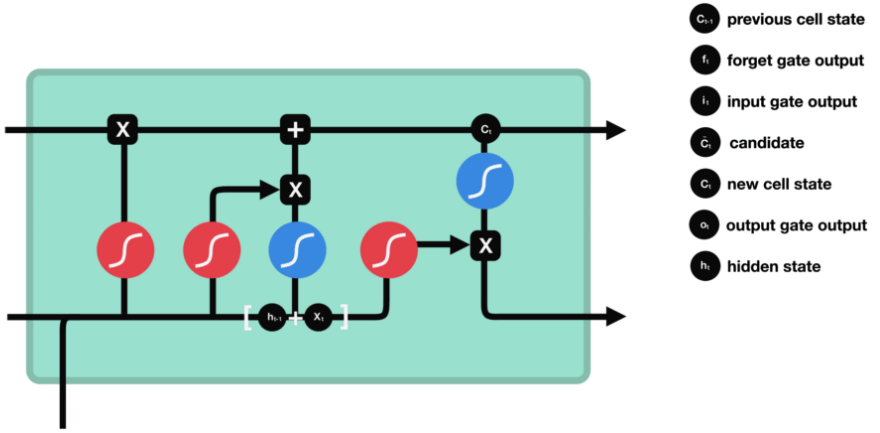
Input gate



Cell state



Output gate



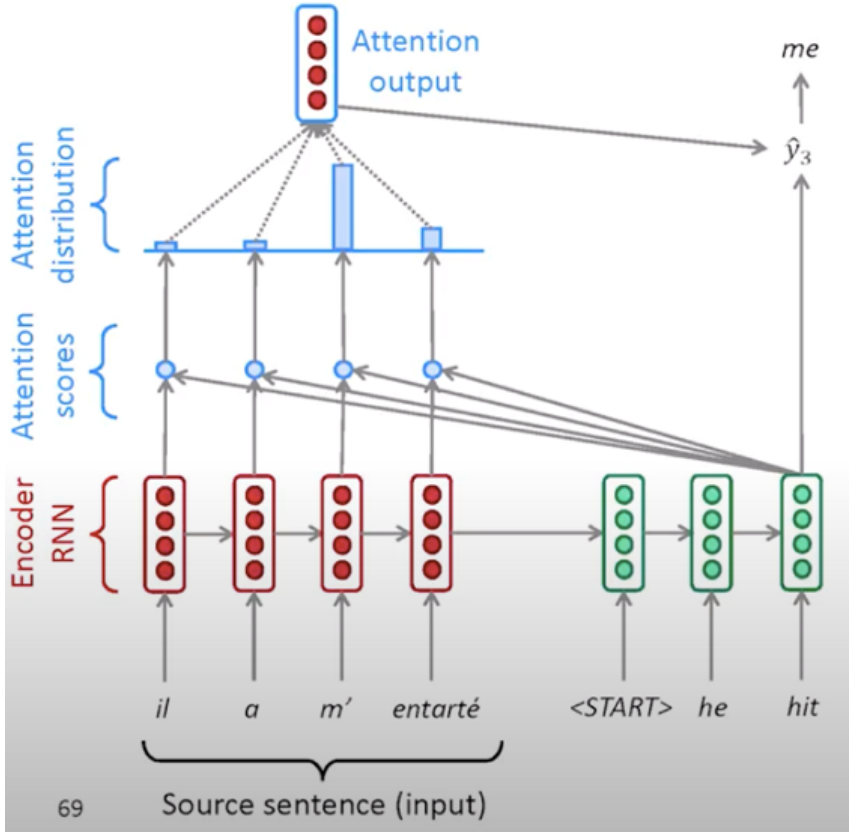
Supplementary material - RNNs

- **Details about attention for machine translation**

At each decoder step, attention scores are computed as the dot product between encoder and decoder hidden states. Scores go through softmax to create an attention distribution, which provides weights for the attention vector computation.

Excellent lecture at Stanford about NLP and Attention (end of the video): link.

A great advantage is it helps put more emphasis and therefore "remember" the start of very long sequences (longer than what LSTM can handle).
It therefore also helps with the vanishing gradient problem.



Supplementary material - GANs

- More *thispersondoesnotexist* faces

- GANs applied to other images than faces (StyleGAN)

Figure 10. Uncurated set of images produced by our style-based generator (config F) with the LSUN BEDROOM dataset at 256^2. FID computed for 50K images was 2.65.

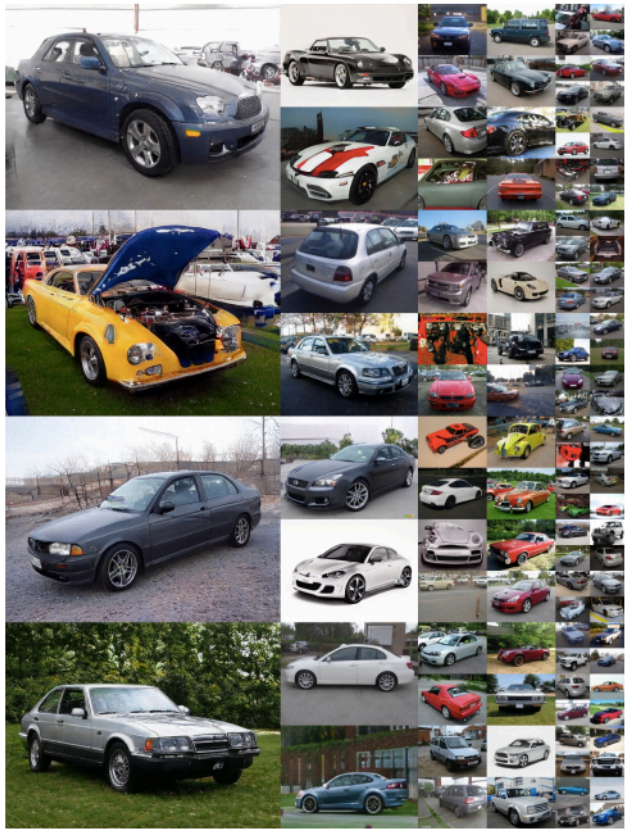
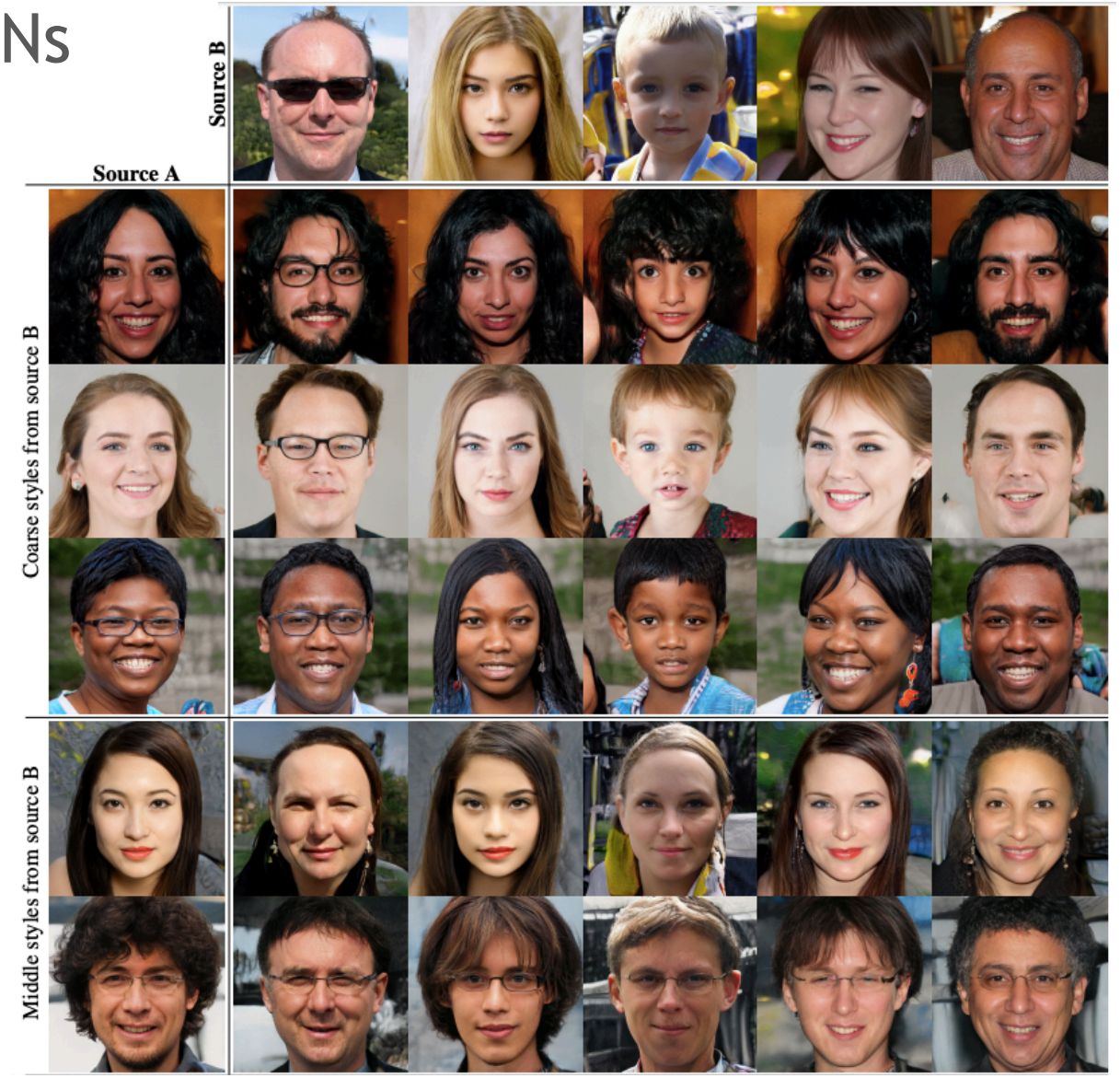


Figure 11. Uncurated set of images produced by our style-based generator (config F) with the LSUN CAR dataset at 512×384. FID computed for 50K images was 3.27.

Supplementary material - GANs

- **An impressive figure from the StyleGAN paper**



> *"What I cannot create, I do not understand"*
> — Richard Feynman, Nobel Prize in Physics, 1956

Supplementary material - GANs

- **Pose Guided Person Image Generation with GANs, 2017 (link)**