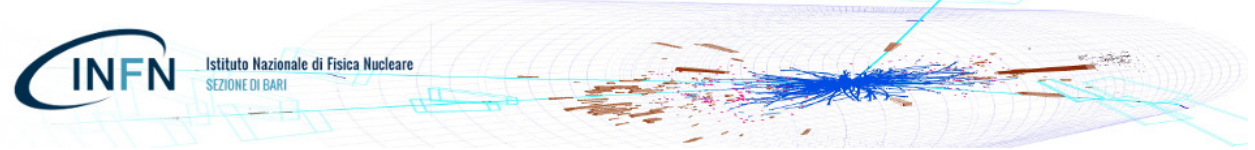
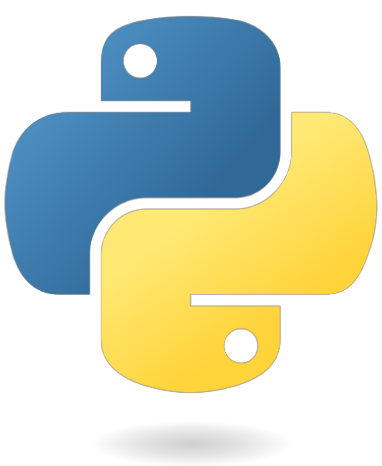




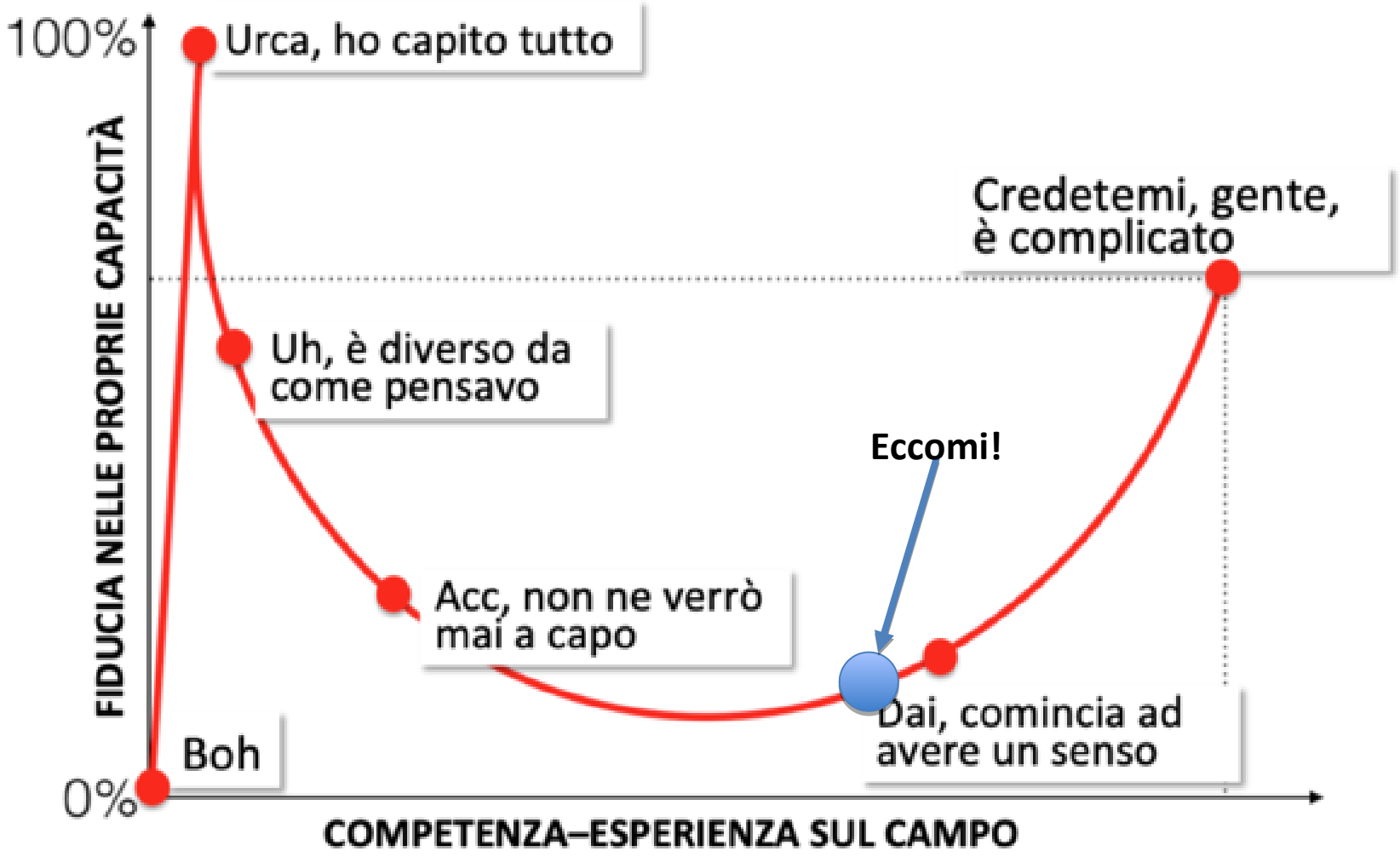
python™

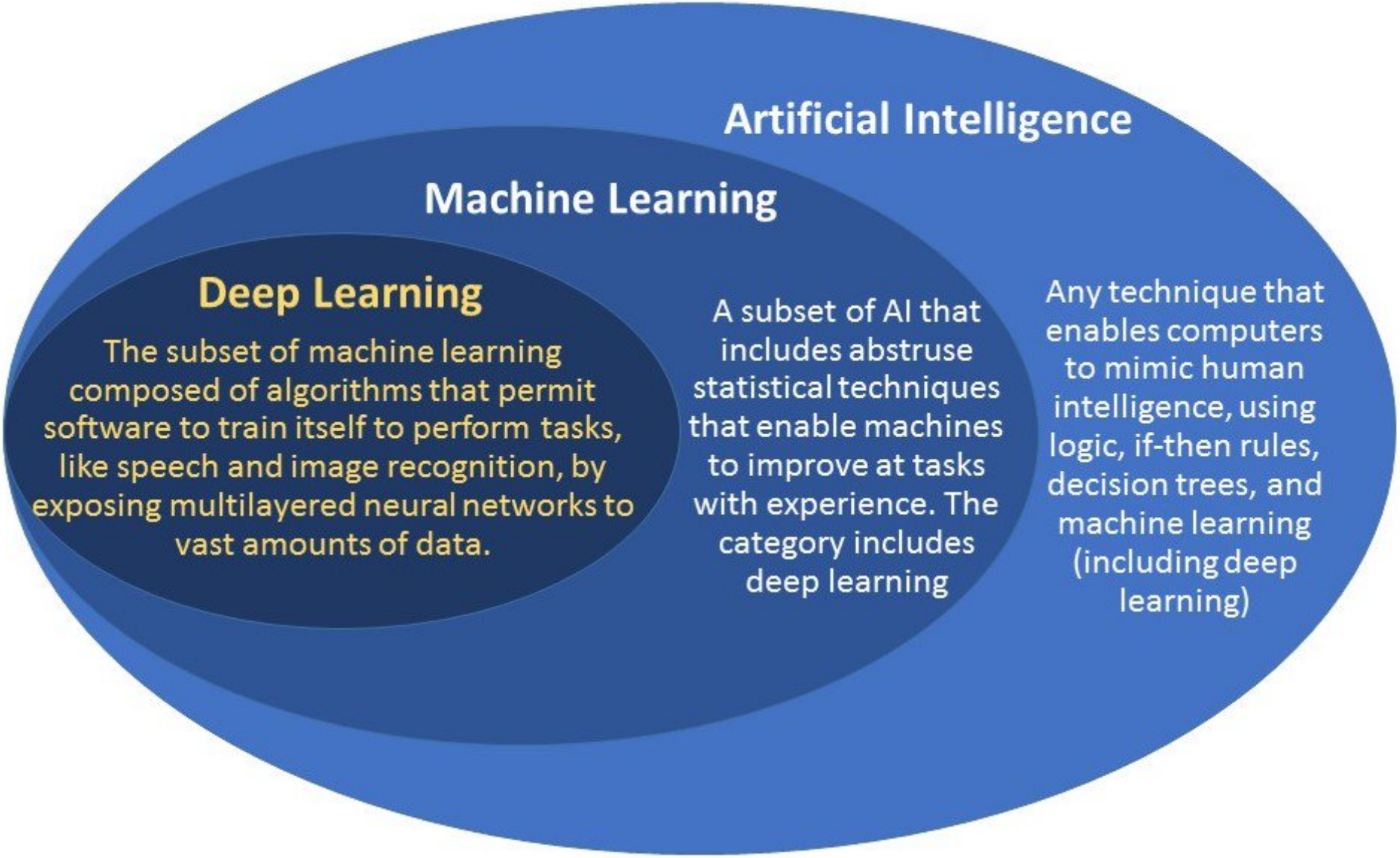
Big Data e Machine Learning

Ottobre 2021

Pandemic Edition!

EFFETTO DUNNING-KRUGER

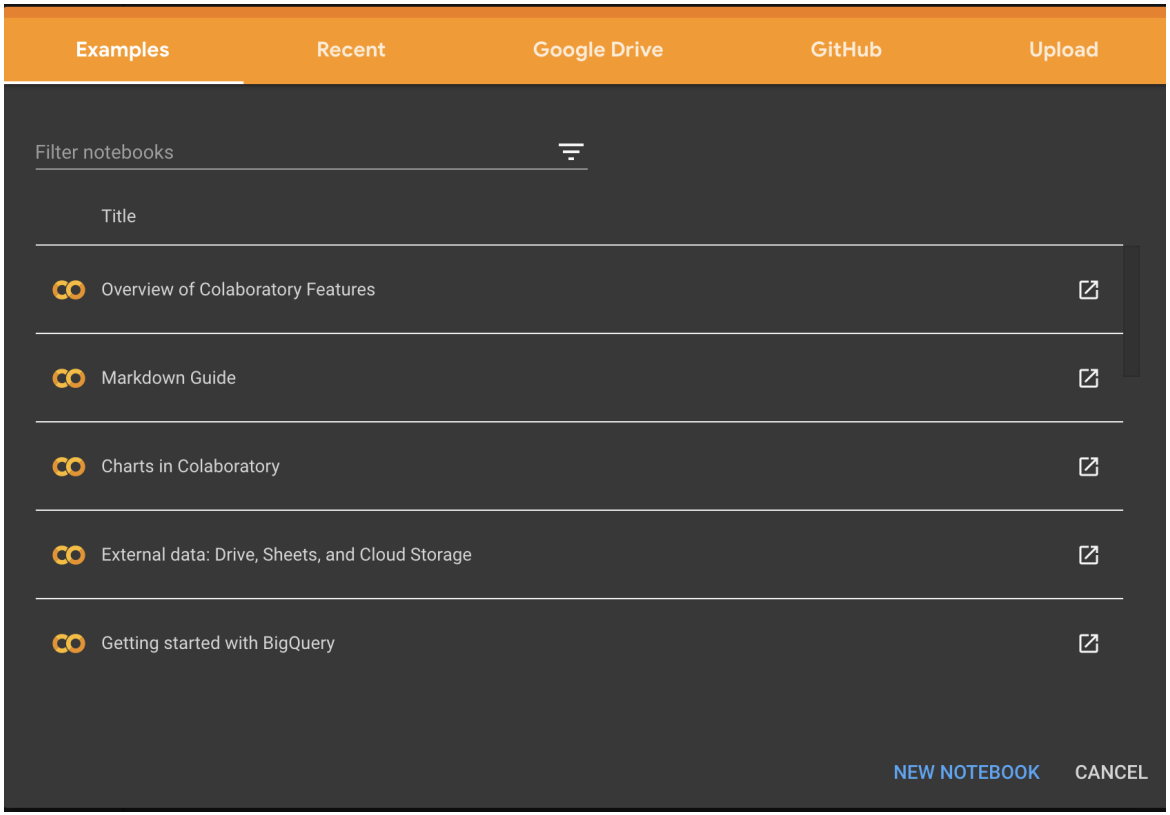




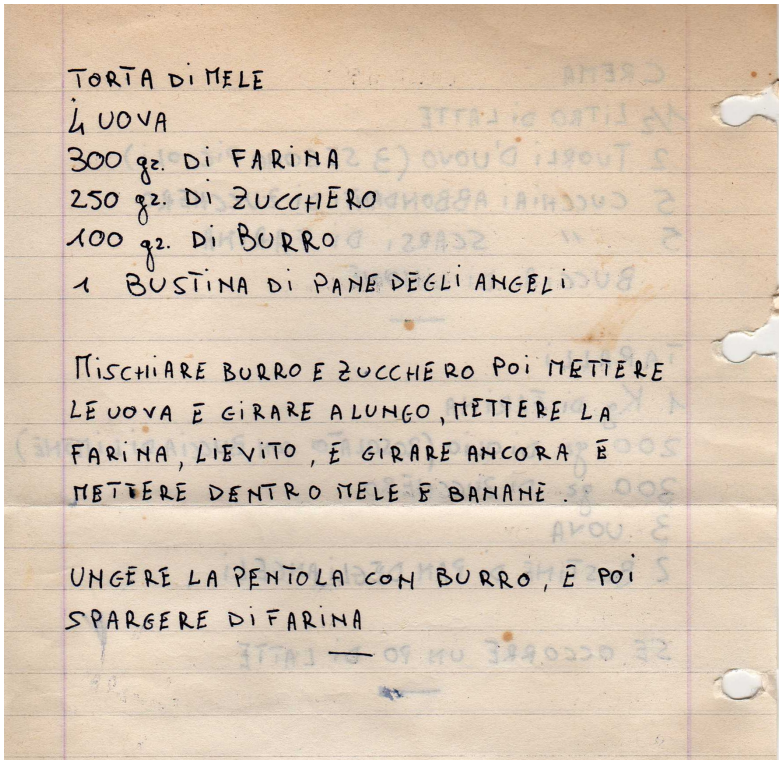
Picture credit:
https://twitter.com/deeplearn007/status/952833643386372096

Colab

https://colab.research.google.com/



An algorithm is a series of steps, that when performed in a specific order, produced your desired output.


TORTA DI MELE
4 UOVA
300 gr. DI FARINA
250 gr. DI ZUCCHERO
100 gr. DI BURRO
1 BUSTINA DI PANE DEGLI ANGELI

MISCHIARE BURRO E ZUCCHERO POI METTERE LE UOVA E GIRARE A LUNGO, METTERE LA FARINA, LIEVITO, E GIRARE ANCORA E METTERE DENTRO MELE E BANANE.

UNGERE LA PENTOLA CON BURRO, E POI SPARGERE DI FARINA

In Machine Learning, we don't specify the algorithm the way we did above.

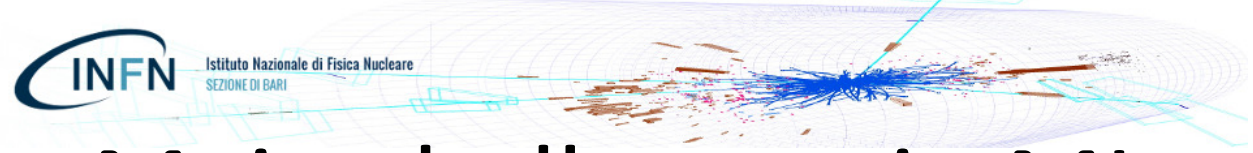
At a very high level, the task of Machine Learning is thus two-fold:

1.Find the best template (algorithm) that is best suited for the task. Deep Learning is simply a subset of the templates we get to choose from that have proven effective across many tasks.

2.Use the data to figure out what are the best numbers to fill up the template. This is the "Learning" part of "Machine Learning".

IS IT POSSIBLE TO REDUCE EVERYTHING TO A RIGHT RECIPE?

https://medium.com/intuitive-deep-learning/intuitive-deep-learning-part-1a-introduction-to-neural-networks-d7b16ebf6b99

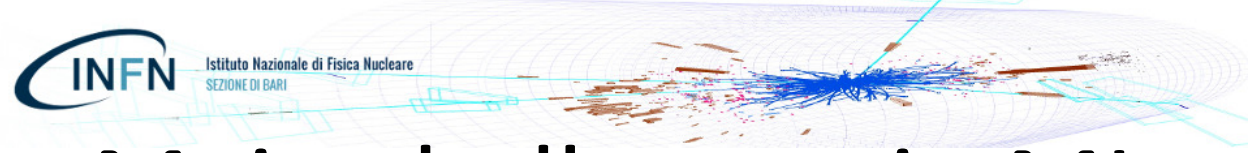


Main challenges in ML

BAD DATA

Insufficient quantity of Training Data
Non representative Training Data (we are talking about the representativeness of the problem)
It's crucial to use a training set that is representative of the cases you want to generalise to.
Poor Quality Data: *garbage in, garbage out*.
If the training data are full of errors, outliers, noise, the ML will not be able to detect any underlying pattern.
Irrelevant Features: success in a ML project largely depends on your ability to feed a training process with features in your data that enable an effective learning process.

This process is called **feature engineering**, and involves feature **selection** and feature **extraction.**



Main challenges in ML

BAD ALGOS

Overfitting: means that a model performs well on the training data, but it does not generalise well to new, previously unseen data. A serious issue.

The source: The more complex a model is (DNN, or just high-degrees polynomials) the more it is able to detect and describe subtle patterns in the training data. But if the training set is noisy, or if it is too small, then the model is likely to detect patterns in the noise itself.

The cures: Reducing the number of attributes, apply regularization, collect more training data

Underfitting: still a serious issue, but better. The model is too simple to learn.

The cures: Select a more powerful model, feature engineering, reducing regolarization.


A GOOD EXAMPLE OF
OVERFITTING

Data

X	Y	Z	Class
1	2	2	A
1	2	3	B
1	4	3	B
4	2	6	A
3	1	1	A
2	4	4	B
5	2	1	A

Training set

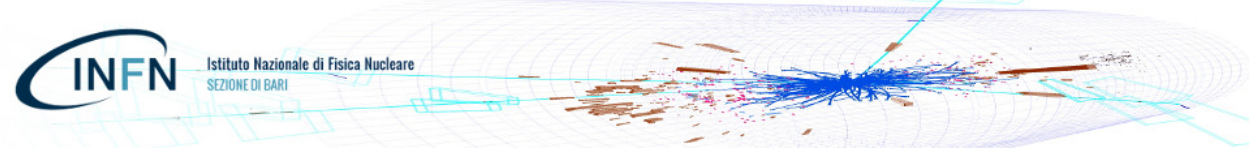
X	Y	Z	Class
1	2	2	A
1	2	3	B
1	4	3	B
4	2	6	A

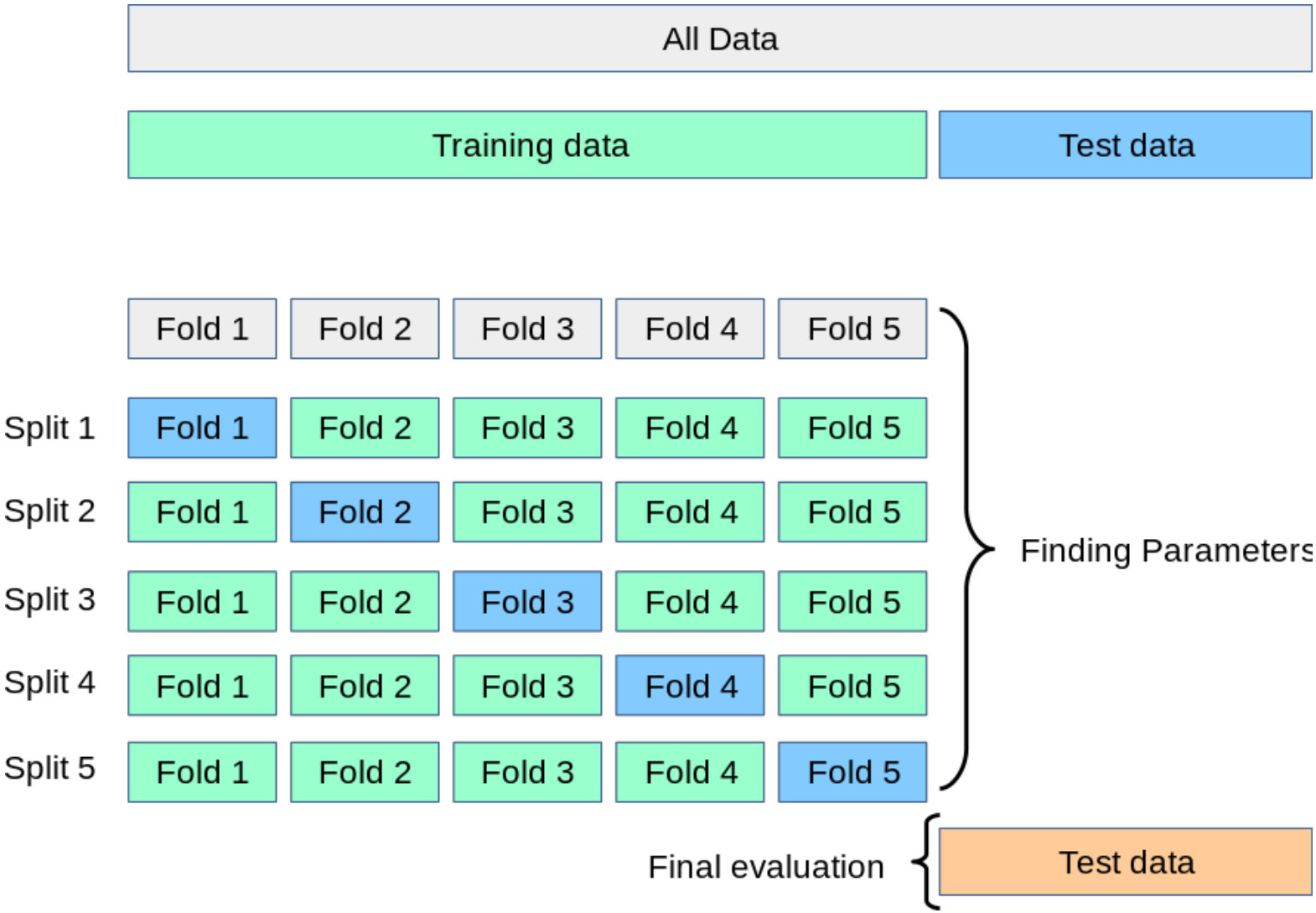
Test set

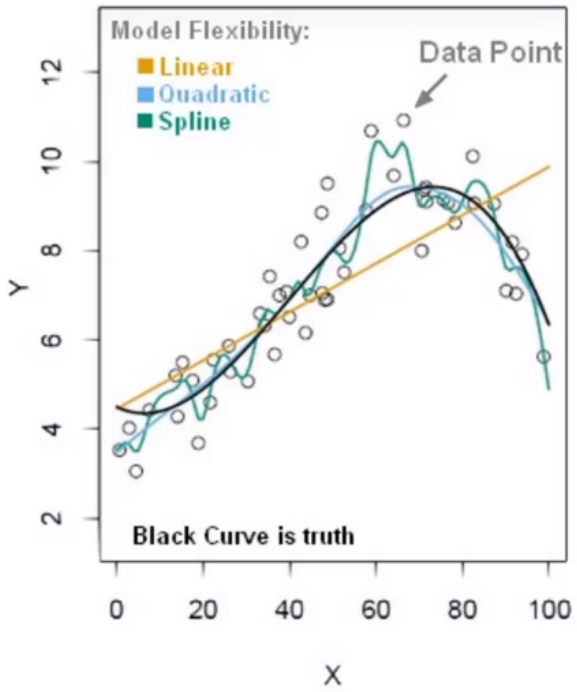
X	Y	Z	Class
3	1	1	A
2	4	4	B
5	2	1	A

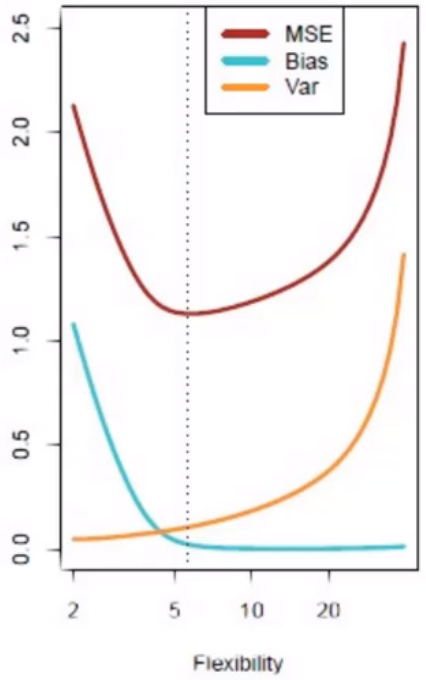
Model

X	Y	Z	Class	Preds
3	1	1	A	B
2	4	4	B	B
5	2	1	A	A




All Data
Training data
Test data
Fold 1
Fold 2
Fold 3
Fold 4
Fold 5
Split 1
Split 2
Split 3
Split 4
Split 5
Finding Parameters
Final evaluation
Test data





Bias *is the algorithm's tendency to consistently learn the wrong thing by not taking into account all the information in the data (underfitting).*

Variance *is the algorithm's tendency to learn random things irrespective of the real signal by fitting highly flexible models that follow the error/noise in the data too closely (overfitting).*

n=165	Predicted: NO	Predicted: YES	
Actual: NO	TN = 50	FP = 10	60
Actual: YES	FN = 5	TP = 100	105
	55	110	

Basic Terminology:

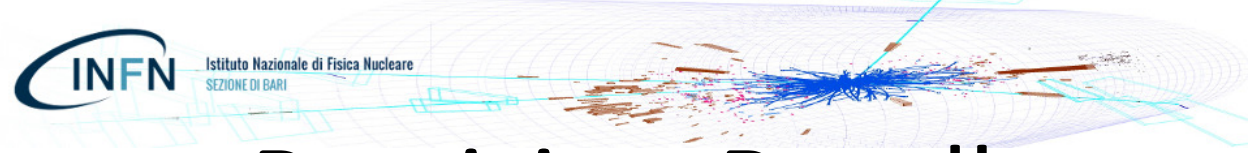
- True Positives (TP)
- True Negatives (TN)
- False Positives (FP)
- False Negatives (FN)

Accuracy:

- Overall, how often is it **correct**?
- (TP + TN) / total = 150/165 = 0.91

Misclassification Rate (Error Rate):

- Overall, how often is it **wrong**?
- (FP + FN) / total = 15/165 = 0.09



Precision-Recall

Precision = TP/(TP+FP) = the ability of the classifier NOT TO label as positive a sample that is negative = Positive Predictive Value PPV

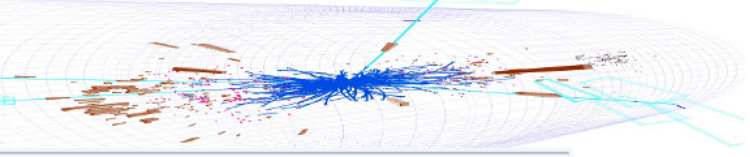
Note that (TP+FP) are all the predicted P
Among **all** patients **predicted** to have cancer, how many actually have it?
Should be as high as possible (FP->0)! *FP=Type 1 error*

Recall = TP/(TP+FN) = the ability of the classifier to FIND ALL the positive samples = True Positive Rate TPR = **Sensitivity (TP/P)**

Note that (TP+FN) are all the actual P
Among **all** patients that **actually** have cancer, how many did we predict to have it?
Should be as high as possible (FN->0)! *FN=Type 2 error*

Specificity = TN/(TN+FP) = the ability of the classifier to FIND ALL the negative samples = **(TN/N)**

In a practical situation when precision increases then recall reduces, and viceversa (**precision/recall tradeoff**)



sensitivity, recall, hit rate, or true positive rate (TPR)

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{P}} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} = 1 - \mathrm{FNR}$$

specificity, selectivity or true negative rate (TNR)

$$\mathrm{TNR} = \frac{\mathrm{TN}}{\mathrm{N}} = \frac{\mathrm{TN}}{\mathrm{TN} + \mathrm{FP}} = 1 - \mathrm{FPR}$$

precision or positive predictive value (PPV)

$$\mathrm{PPV} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} = 1 - \mathrm{FDR}$$

negative predictive value (NPV)

$$\mathrm{NPV} = \frac{\mathrm{TN}}{\mathrm{TN} + \mathrm{FN}} = 1 - \mathrm{FOR}$$

miss rate or false negative rate (FNR)

$$\mathrm{FNR} = \frac{\mathrm{FN}}{\mathrm{P}} = \frac{\mathrm{FN}}{\mathrm{FN} + \mathrm{TP}} = 1 - \mathrm{TPR}$$

fall-out or false positive rate (FPR)

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{N}} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}} = 1 - \mathrm{TNR}$$

false discovery rate (FDR)

$$\mathrm{FDR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TP}} = 1 - \mathrm{PPV}$$

false omission rate (FOR)

$$\mathrm{FOR} = \frac{\mathrm{FN}}{\mathrm{FN} + \mathrm{TN}} = 1 - \mathrm{NPV}$$

prevalence threshold (PT)

$$\mathrm{PT} = \frac{\sqrt{\mathrm{TPR}(-\mathrm{TNR} + 1)} + \mathrm{TNR} - 1}{(\mathrm{TPR} + \mathrm{TNR} - 1)} = \frac{\sqrt{\mathrm{FPR}}}{\sqrt{\mathrm{TPR}} + \sqrt{\mathrm{FPR}}}$$

threat score (TS) or critical success index (CSI)

$$\mathrm{TS} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN} + \mathrm{FP}}$$

MMWR Morb Mortal Wkly Rep. 2021 May 14;70(19):702-706. doi: 10.15585/mmwr.mm7019a3.

Diagnostic Performance of an Antigen Test with RT-PCR for the Detection of SARS-CoV-2 in a Hospital Setting - Los Angeles County, California, June-August 2020

Auguste Brihn, Jamie Chang, Kelsey OYong, Sharon Balter, Dawn Terashita, Zach Rubin, Nava Yeganeh

PMID: 33983916 PMCID: PMC8118154 DOI: 10.15585/mmwr.mm7019a3

Free PMC article

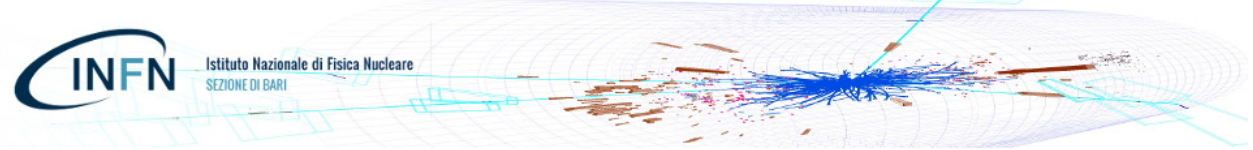
Among 1,732 paired samples from asymptomatic patients, the antigen test **sensitivity was 60.5%, and specificity was 99.5%** when compared with RT-PCR. Among 307 symptomatic persons, sensitivity and specificity were 72.1% and 98.7%, respectively.

Sensitivity/recall – how good a test is at detecting the positives. A test can cheat and maximize this by always returning “positive”

Specificity – how good a test is at detecting the negatives. A test can cheat and maximize this by always returning “negative”

Type I error
(false positive)

Type II error
(false negative)



High Precision

In a streaming platform you have *restricted mode* to restrict the violent and adult videos for the kids.

You want that an automated AI model flags as POSITIVE any video that is suitable for kids. So your focus is… **HIGH PRECISION (TP/(TP+FP))**, because if model has classified the video "good for kids" it must be safe to watch by kids, and this can be done by **reducing the false positive FP to 0**. (no type 1 error)

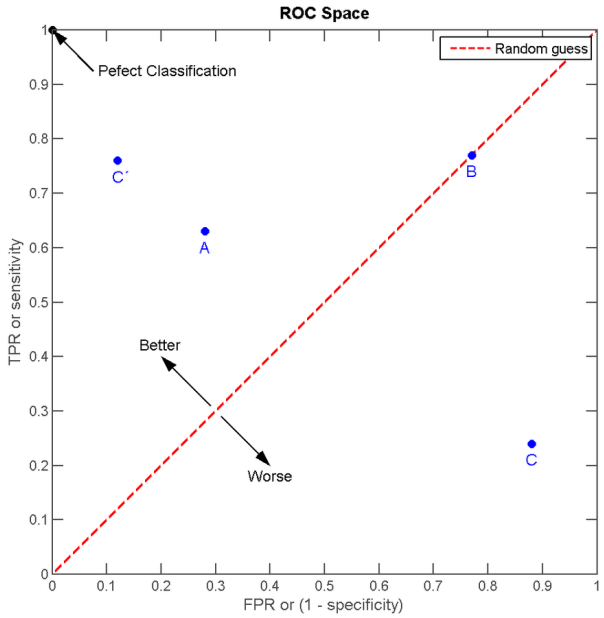
You can safely admit a false negative - a film for kids misclassified as for adult – (or a specificity very low) but not the opposite.

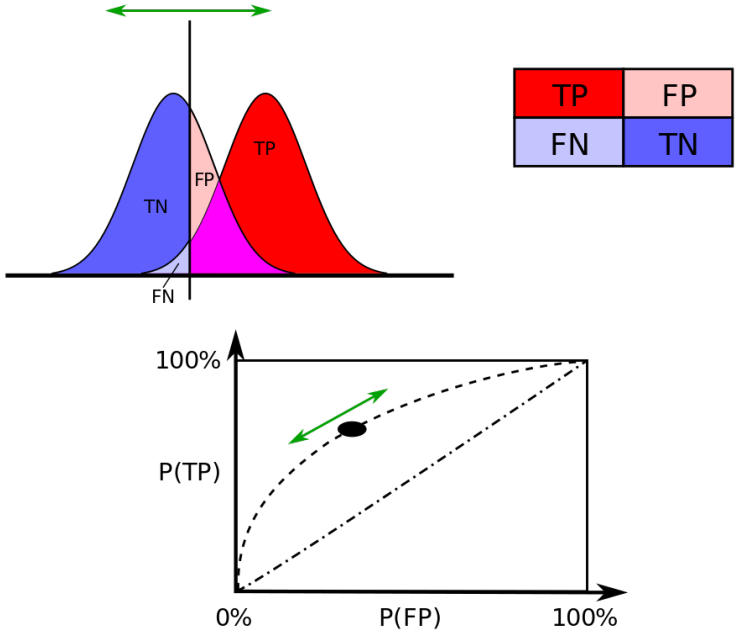
High Recall/Sensitivity

You are creating a model to detect if a patient is having COVID disease or not. In this case the aim of the model is to have **HIGH RECALL {TP/(TP+FN)}**: you absolutely do not want that a COVID patient is classified as Negative, so **you want FN=0**. (no type 2 error)

You can "safely" (well, a sort of…) admit a false positive – a healthy patient misclassified as infected.

In binary classification, the class prediction for each instance is often made based on a continuous random variable X, which is a "score" computed for the instance (e.g. the estimated probability in logistic regression). Given a threshold parameter T, the instance is classified as "positive" if X>T, and "negative" otherwise. The ROC curve plots parametrically TPR(T) versus FPR(T) with T as the varying parameter.
A perfect classifier has FP=FN=0, so TPR=TNR=1 and FPR=0

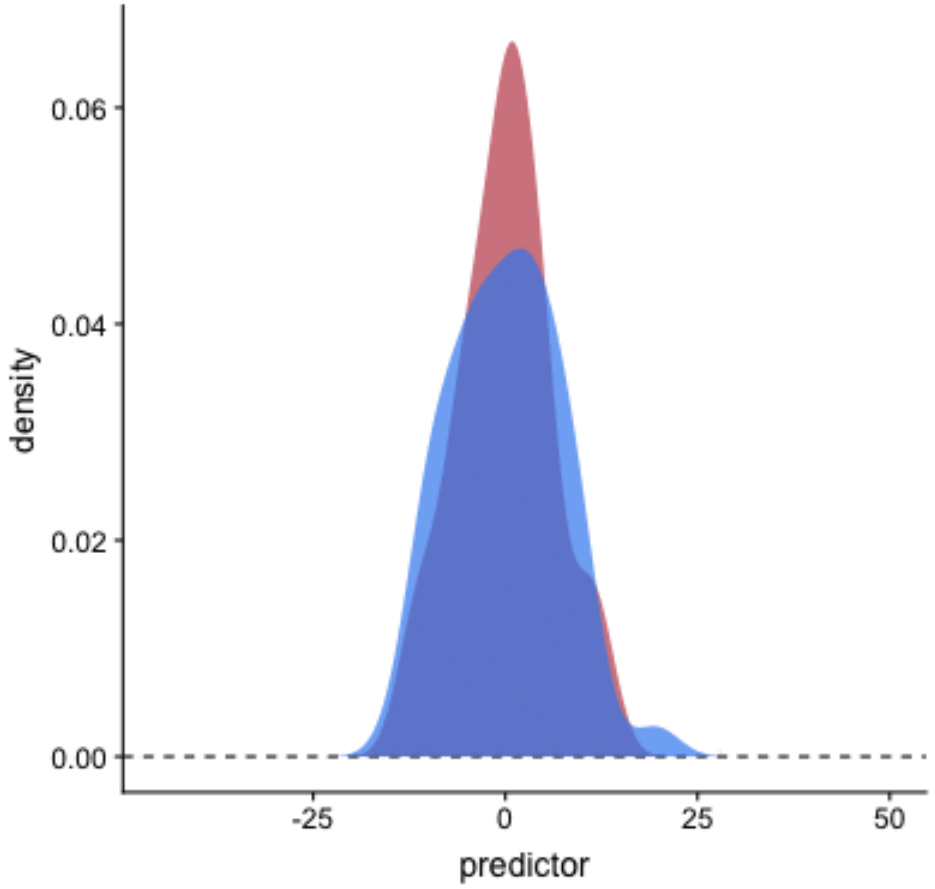


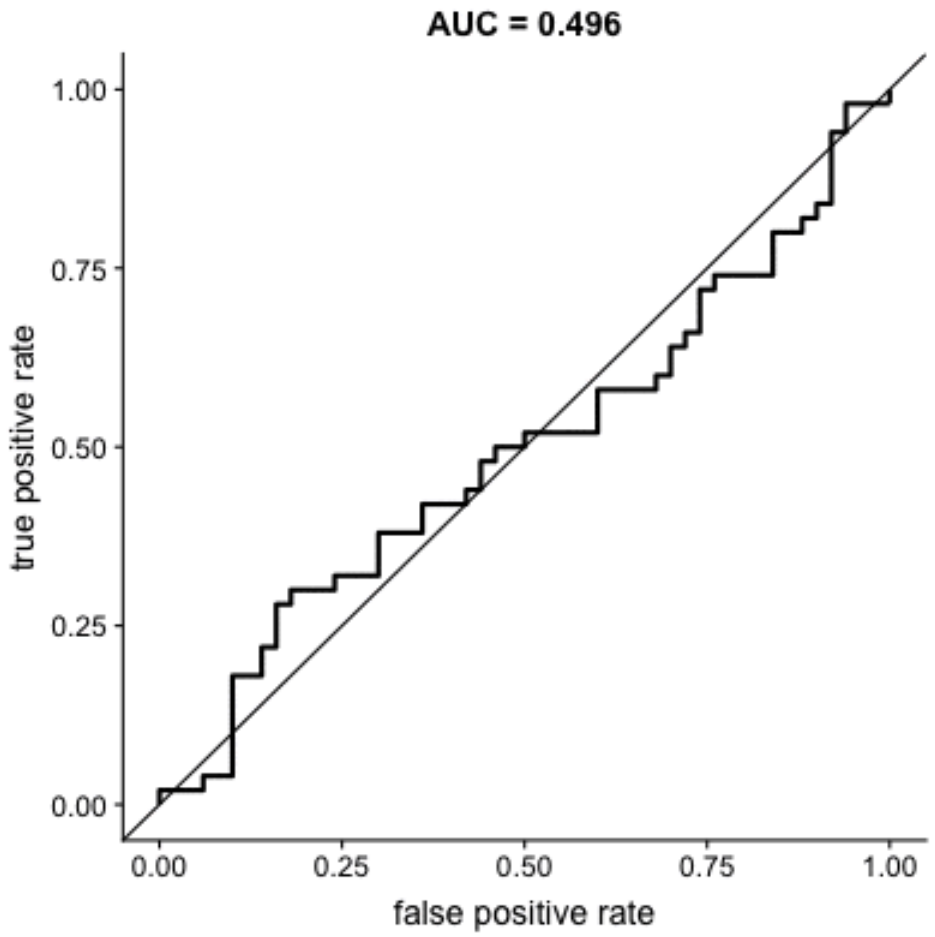


When using normalized units, the area under the curve (often referred to as simply the **AUC**) is equal to the probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one (assuming 'positive' ranks higher than 'negative'). The AUC is used to compare classifiers.

https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5

The shape of an ROC curve changes when a model changes the way it classifies the two outcomes. The animation starts with a model that cannot tell one outcome from the other, and the two distributions completely overlap (essentially a random classifier).
As the two distributions separate, the ROC curve approaches the left-top corner, and the AUC value of the curve increases. When the model can perfectly separate the two outcomes, the ROC curve forms a right angle and the AUC becomes 1.

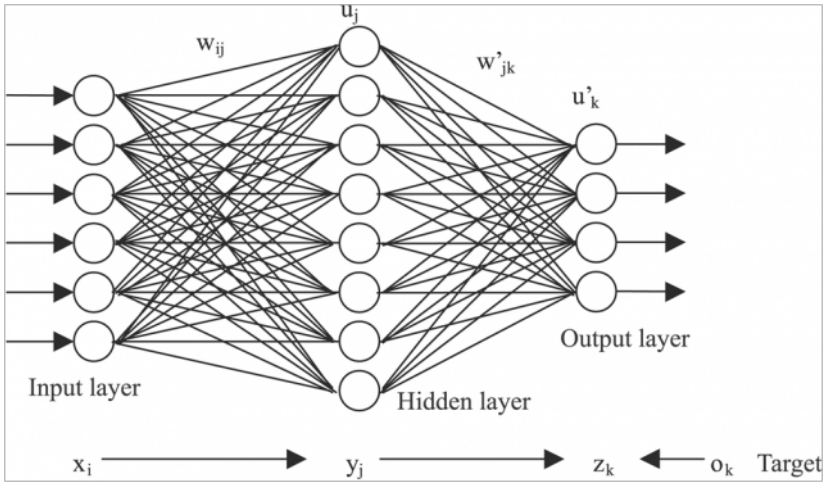


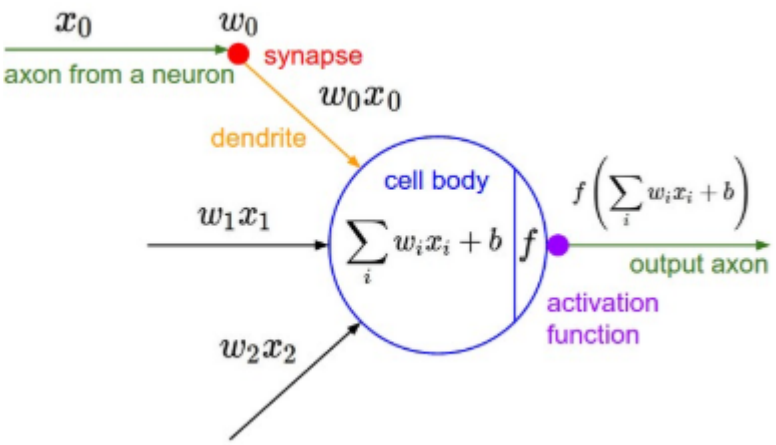



Deep Learning
ME
Not owning
a GPU

Feedforward Neural Networks

Many applications of deep learning use feedforward neural network architectures, which learn to map a fixed-size input (for example an image) to a fixed-size output (for example a probability for each of several categories

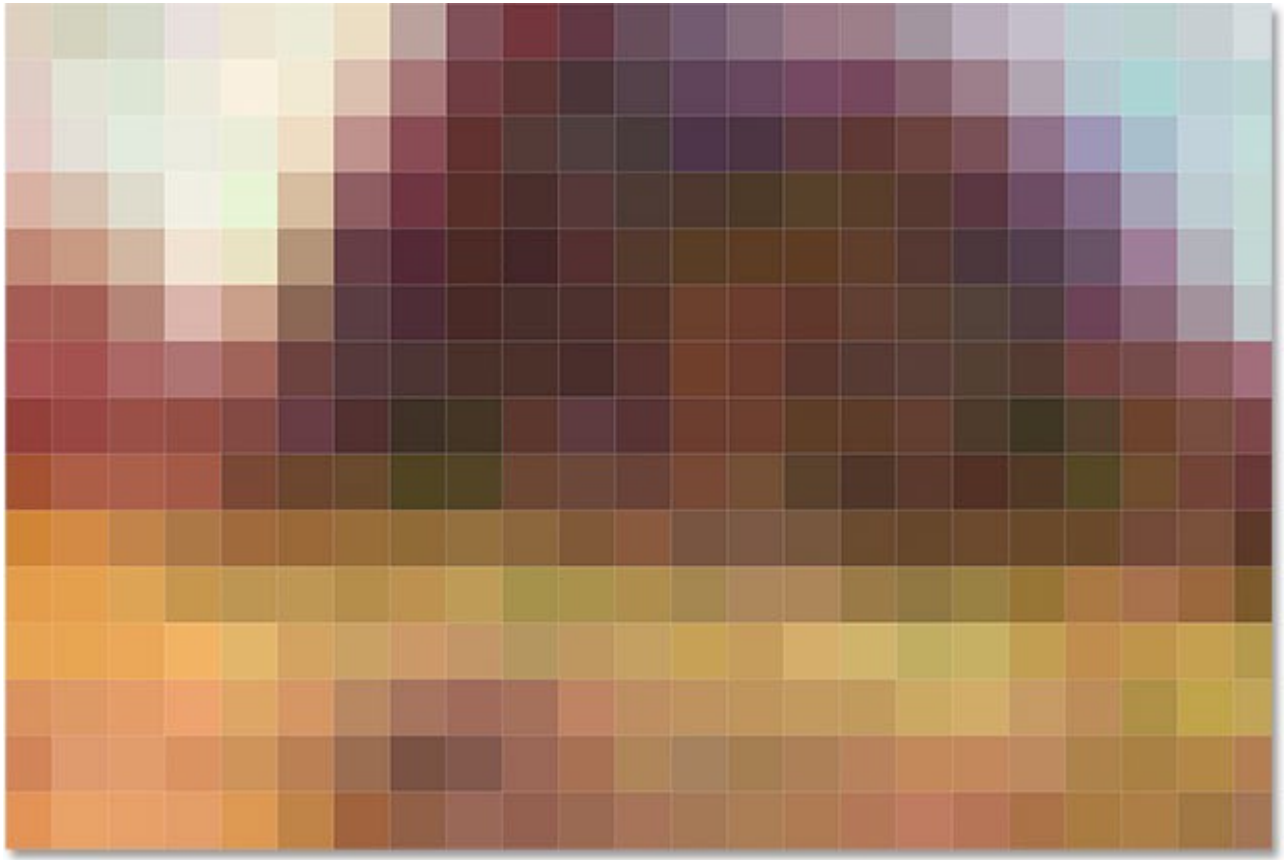




To go from one layer to the next, a set of units compute a weighted sum of their inputs from the previous layer, and pass the result through a non linear function.

The **universal approximation theorem** shows that a neural network with a single hidden layer with a sufficient (and finite) number of neurons can approximate any continuous function.

CNN



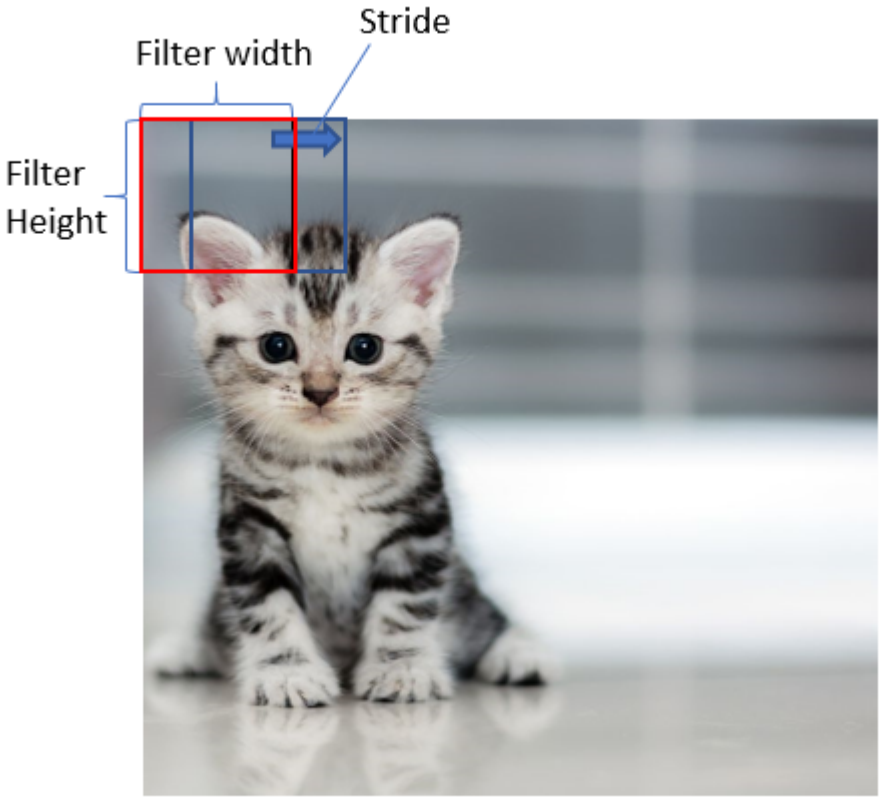
Images are made out of pixels (RGB) arranged within a 2-D (3-D) array.

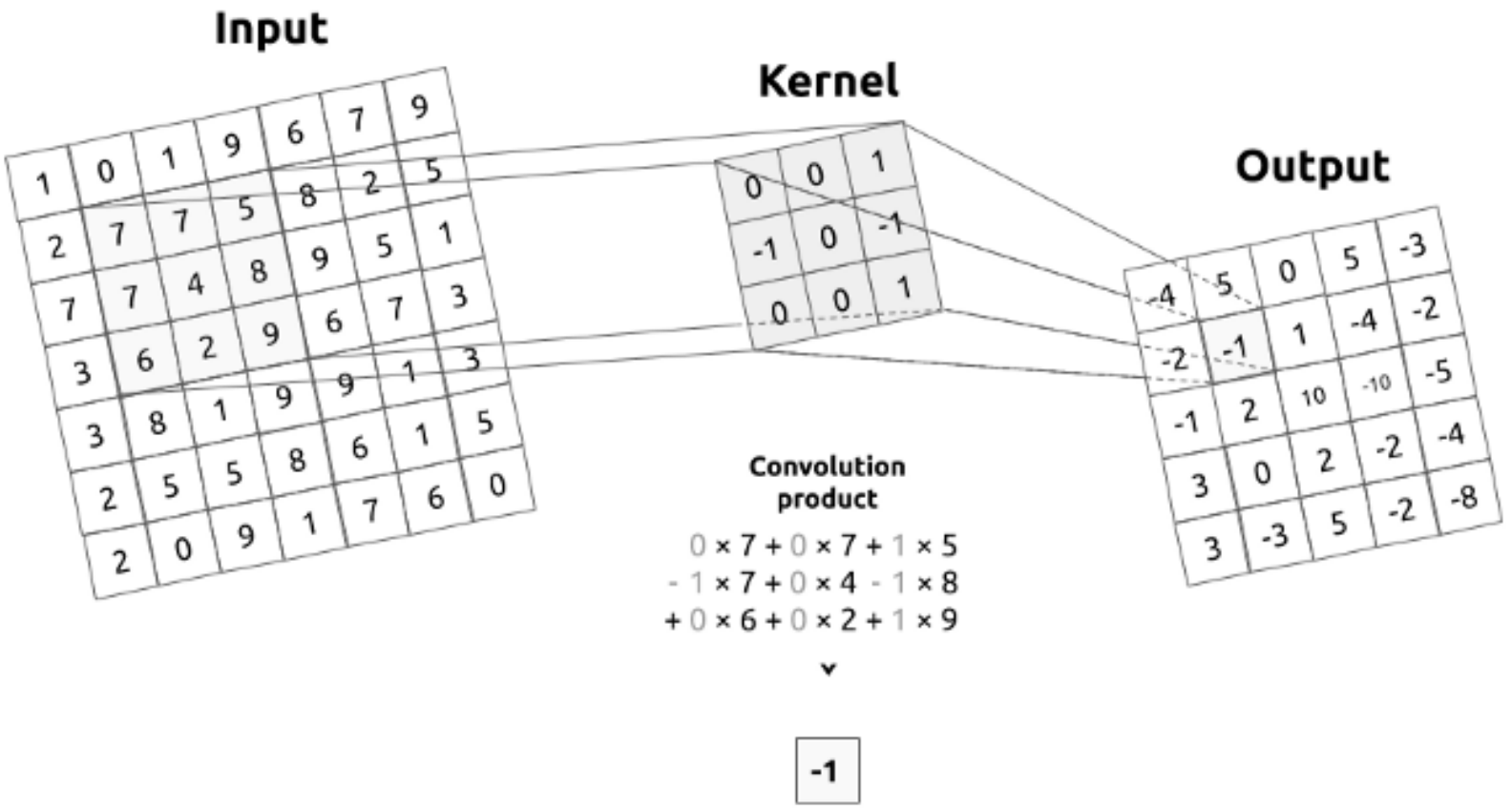
Image taken from
https://www.photoshopessentials.com/basics/pixels-image-size-resolu

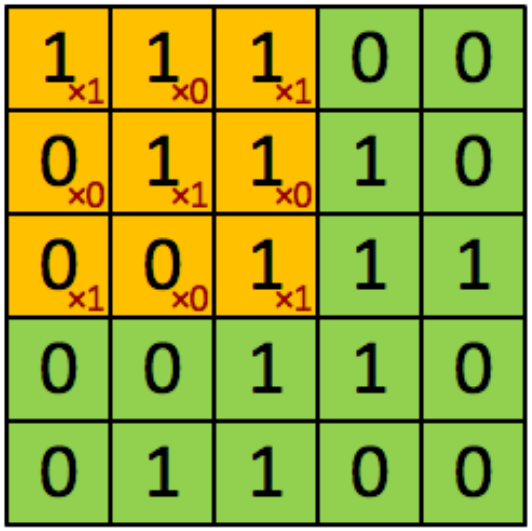
64 Mpixels = 192M features = 192M parameters on the first layer!

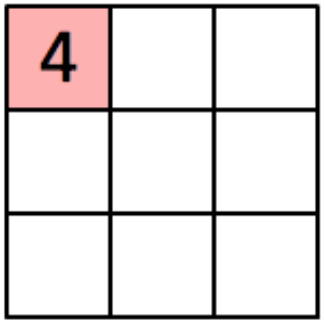
256x256 pixels = 196.608 features

So… EACH neuron has 196.609 parameters, from which the network must be able to learn to identify an object. Plus MORE: the object can be everywhere in the image!

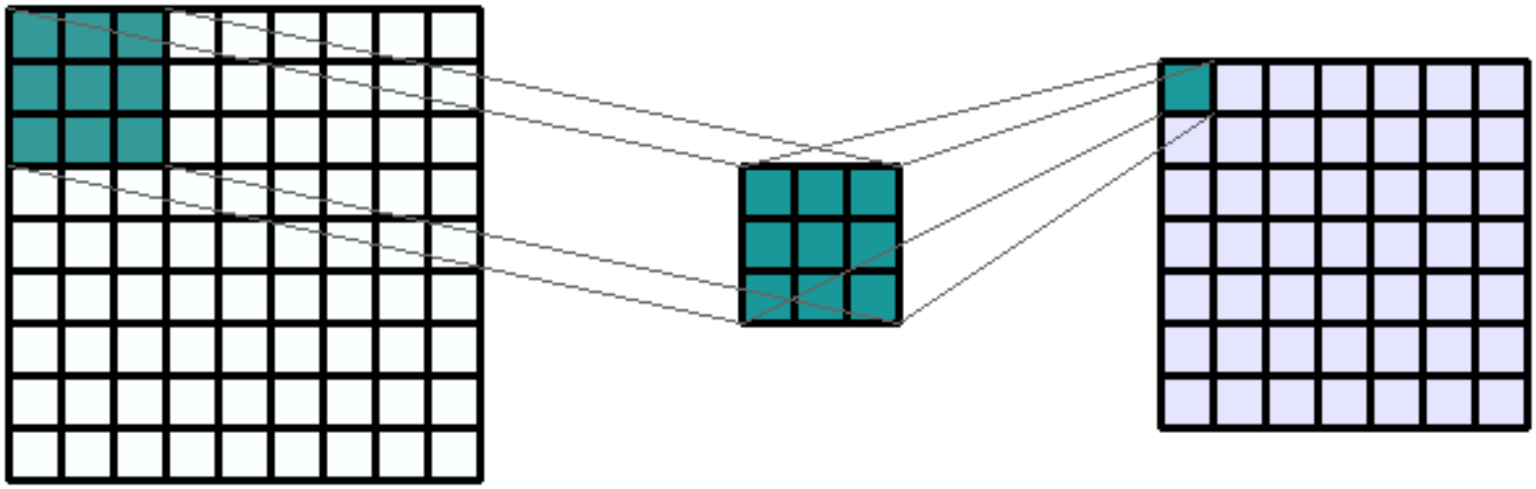



Input
Kernel
Output
Convolution product
0 × 7 + 0 × 7 + 1 × 5
- 1 × 7 + 0 × 4 - 1 × 8
+ 0 × 6 + 0 × 2 + 1 × 9
-1



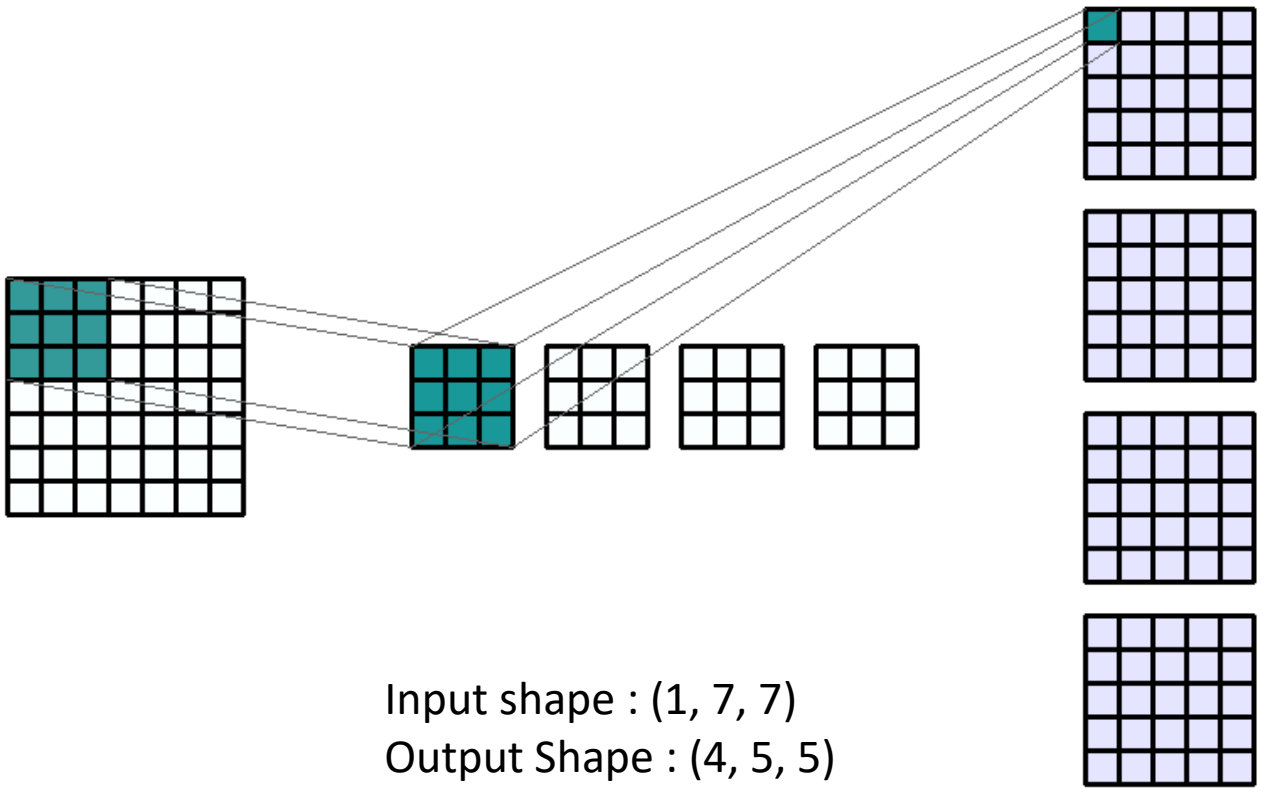


In the demonstration, the green section resembles a **5x5x1 input image, I**.
The element involved in carrying out the convolution operation in the first part of a Convolutional Layer is called the **Kernel/Filter, K**, represented in the color yellow. We have selected **K as a 3x3x1 matrix.**

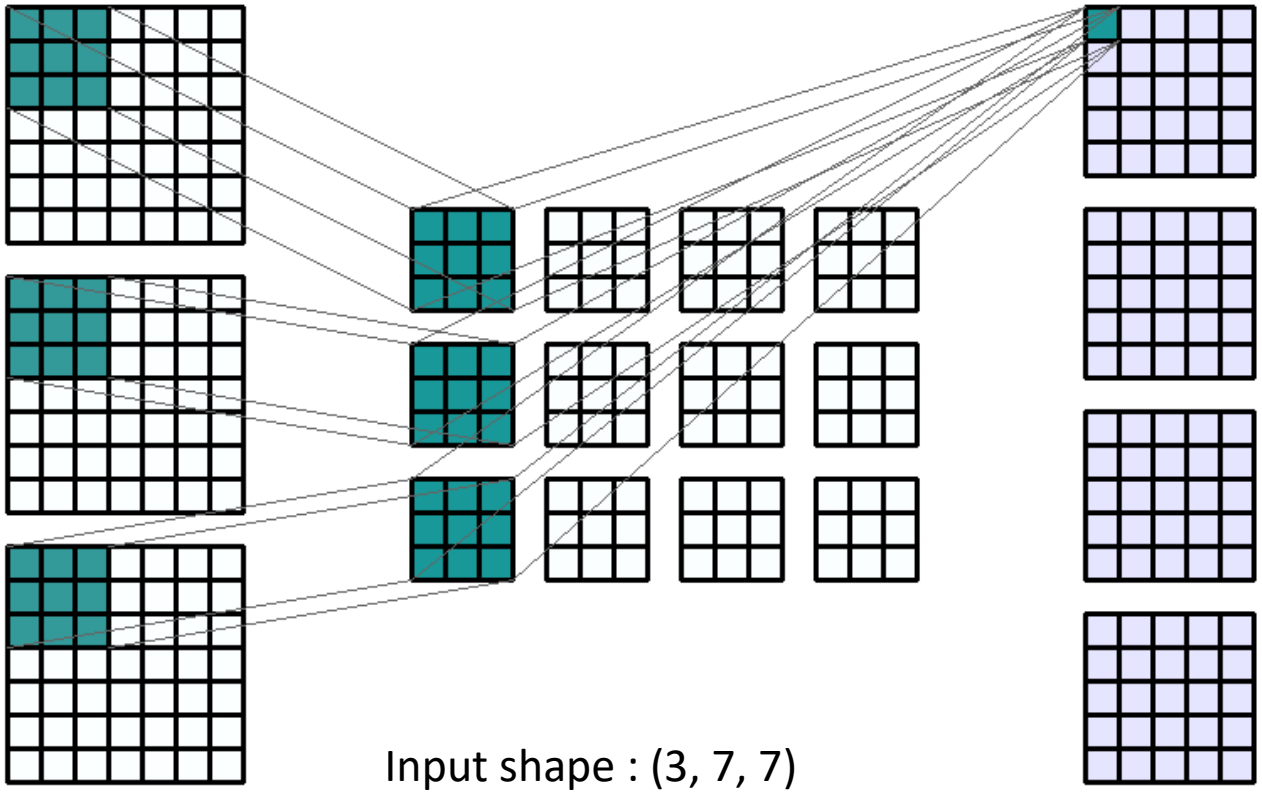


Input shape : (1, 9, 9)
Output Shape : (1, 7, 7)
K : (3, 3)

In a Conv2d, the trainable elements are the values that **compose the kernels**.
So for our 3 by 3 convolution kernel, we have 3*3=9 trainable parameters (+bias).

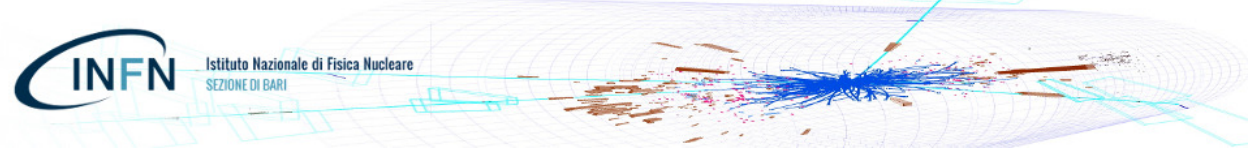


With a layer we can **perform similar operations** at the same time. If we want to apply 4 different filters of the same size to an input channel, then we will have 4 output channels. These channels are the result of 4 different filters, resulting from 4 **distinct kernels**.



Input shape : (3, 7, 7)
Output Shape : (4, 5, 5)
K : (3, 3)

RGB



0	0	0	0	0	0	...
0	156	155	156	158	158	...
0	153	154	157	159	159	...
0	149	151	155	158	159	...
0	146	146	149	153	158	...
0	145	143	143	148	158	...
...	...	...	...	...	...	...

Input Channel #1 (Red)

0	0	0	0	0	0	...
0	167	166	167	169	169	...
0	164	165	168	170	170	...
0	160	162	166	169	170	...
0	156	156	159	163	168	...
0	155	153	153	158	168	...
...	...	...	...	...	...	...

Input Channel #2 (Green)

0	0	0	0	0	0	...
0	163	162	163	165	165	...
0	160	161	164	166	166	...
0	156	158	162	165	166	...
0	155	155	158	162	167	...
0	154	152	152	157	167	...
...	...	...	...	...	...	...

Input Channel #3 (Blue)

-1	-1	1
0	1	-1
0	1	1

Kernel Channel #1

1	0	0
1	-1	-1
1	0	-1

Kernel Channel #2

0	1	1
0	1	0
1	-1	1

Kernel Channel #3

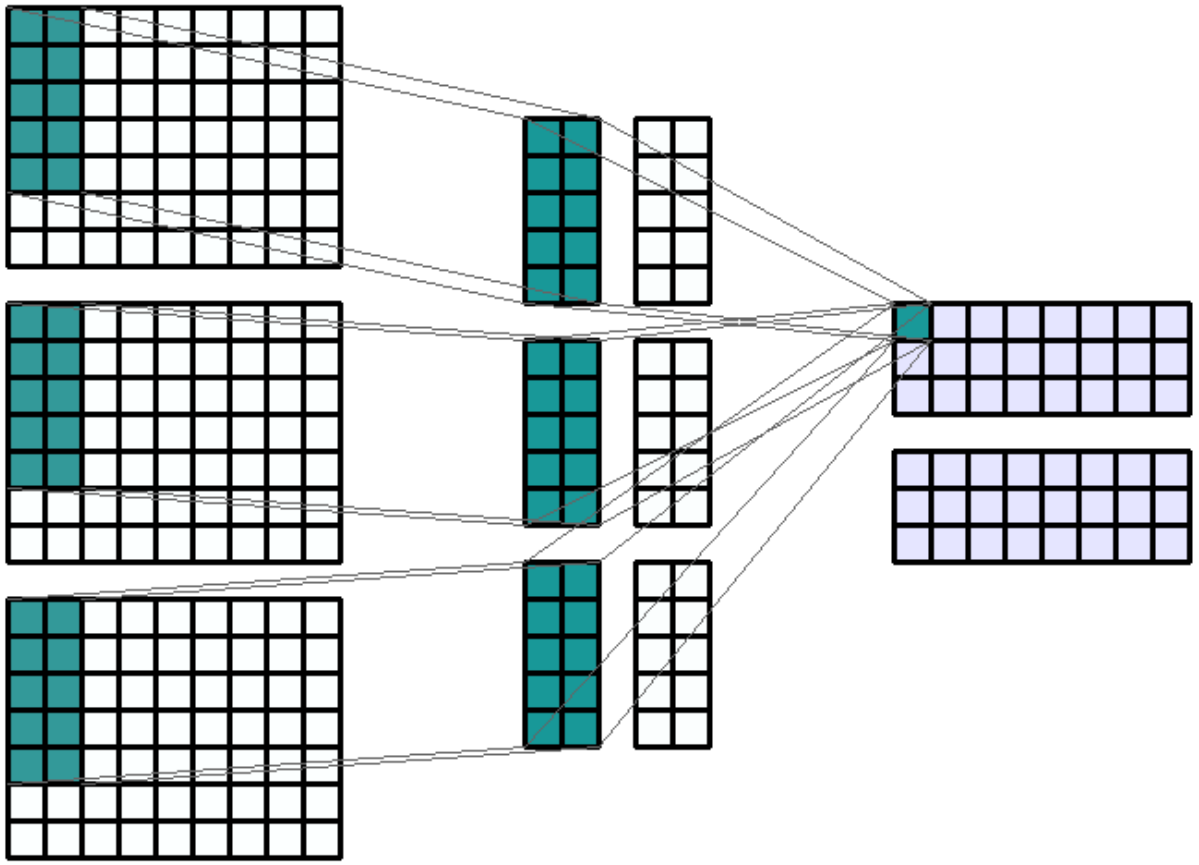
$308 + (-498) + 164 + 1 = -25$

Bias = 1

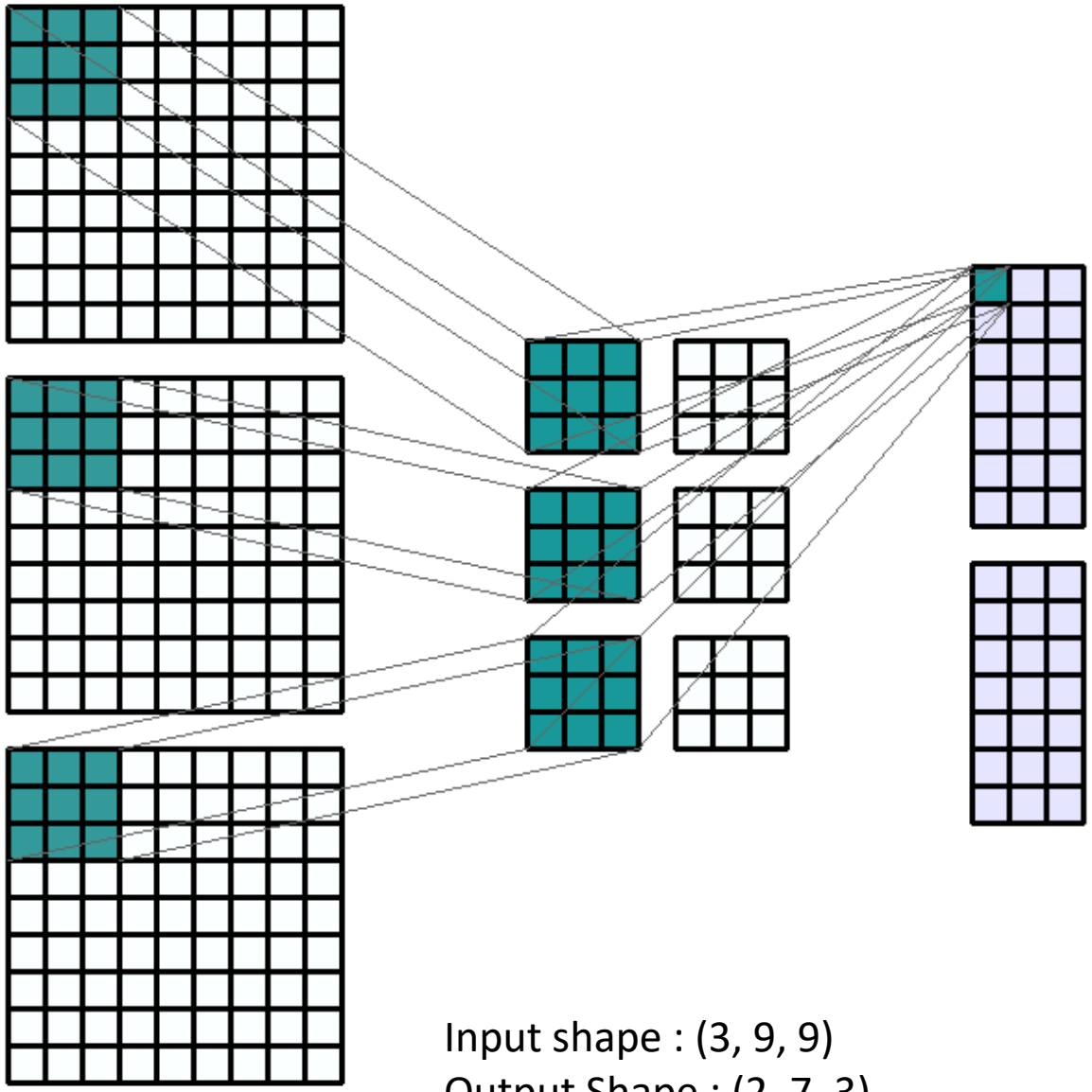
Output

-25				...
				...
				...
				...
...	...	...	...	...

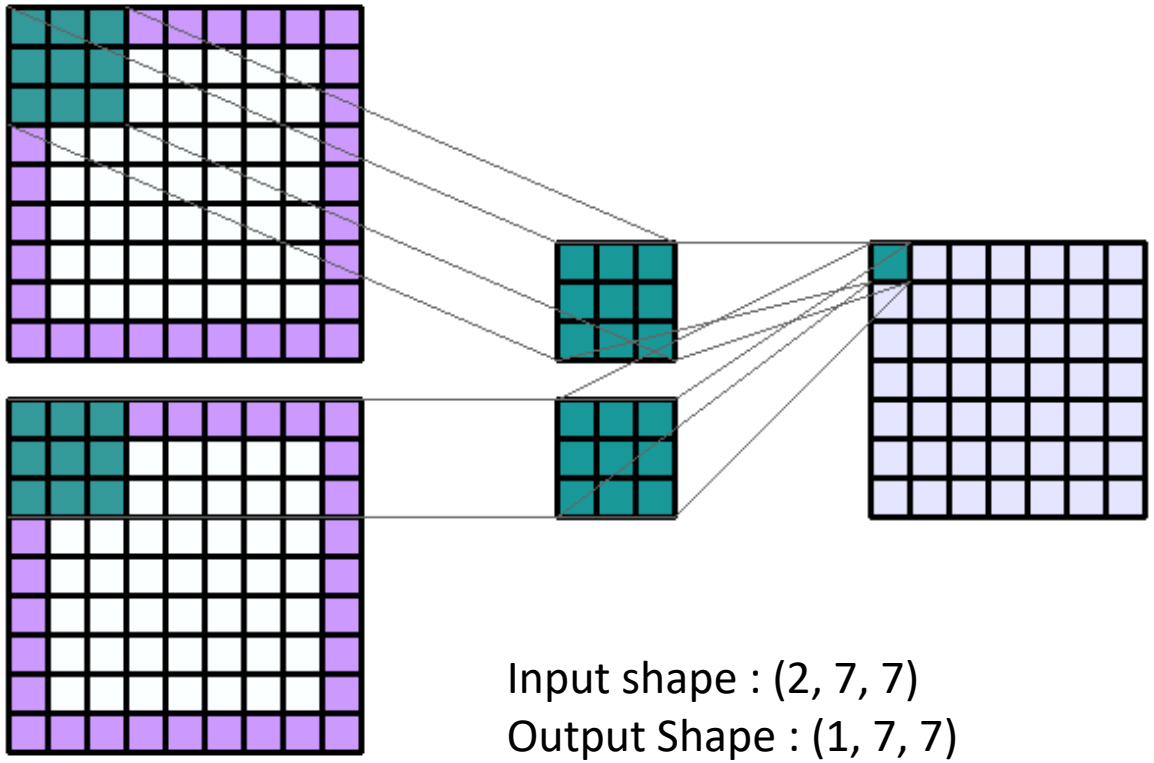
We always assume that we do not split up the image by its depth (or the channels), only the width and height. So if we specify the filter size, the number of parameters in our neuron is filter_width * filter_height * input_depth + 1



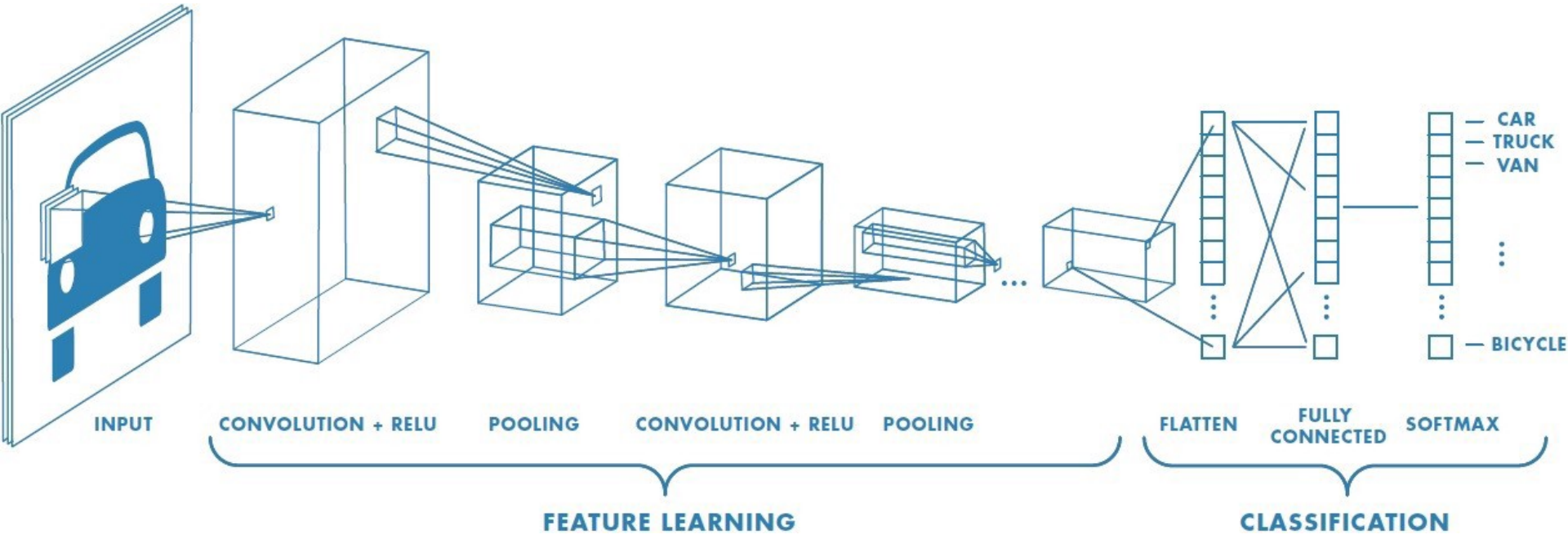
Input shape : (3, 7, 9)
Output Shape : (2, 3, 9)
K : (5, 2)



Input shape : (3, 9, 9)
Output Shape : (2, 7, 3)
K : (3, 3)
S : (1, 3)



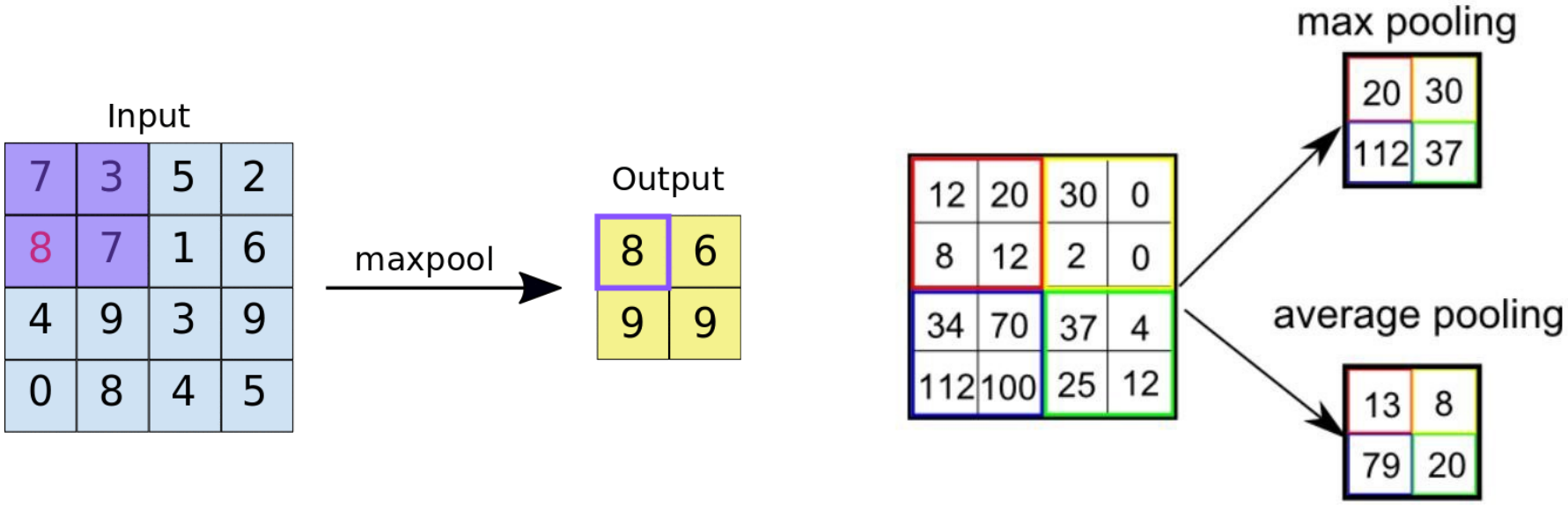
Input shape : (2, 7, 7)
Output Shape : (1, 7, 7)
K : (3, 3)



A **Convolutional Neural Network (ConvNet/CNN)** is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/ objects in the image and be able to differentiate one from the other.

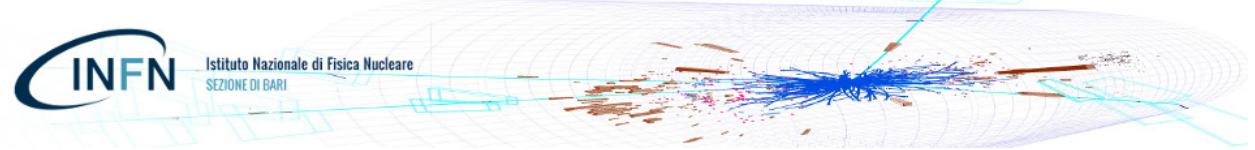
A ConvNet is able to **successfully capture the Spatial and Temporal dependencies** in an image through the application of relevant filters. The architecture performs a better fitting to the image dataset due to the reduction in the number of parameters involved and reusability of weights.

The pooling layer applies to each individual depth channel separately. This is unlike the conv layer, which combines inputs from all the depth channels.



Pooling layer is responsible for reducing the spatial size of the Convolved Feature.

This is to **decrease the computational power required to process the data** through dimensionality reduction.
Furthermore, it is useful for **extracting dominant features** which are rotational and positional invariant, thus maintaining the process of effectively training of the model.



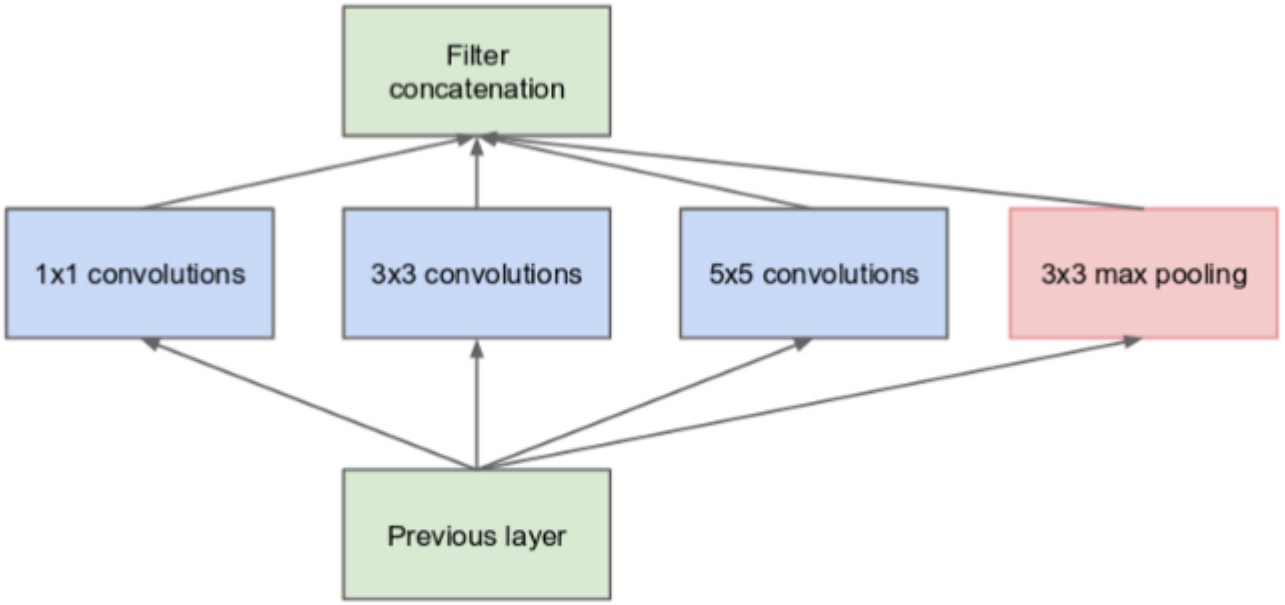
http://scs.ryerson.ca/~aharley/vis/conv/
http://scs.ryerson.ca/~aharley/vis/conv/flat.html
https://www.cs.ryerson.ca/~aharley/vis/fc/

This network has 1024 nodes on the bottom layer (corresponding to pixels), six 5x5 (stride 1) convolutional filters in the first hidden layer, followed by sixteen 5x5 (stride 1) convolutional filters in the second hidden layer, then three fully-connected layers, with 120 nodes in the first, 100 nodes in the second, and 10 nodes in the third. The convolutional layers are each followed by downsampling layer that does 2x2 max pooling (with stride 2).

The networks were trained on an augmented version of MNIST, so they excel at categorizing centred upright numbers. The networks were trained in a custom neural network implementation in MATLAB; the math for the visualizations was written in Javascript; the visualization was created in WebGL.

Inception

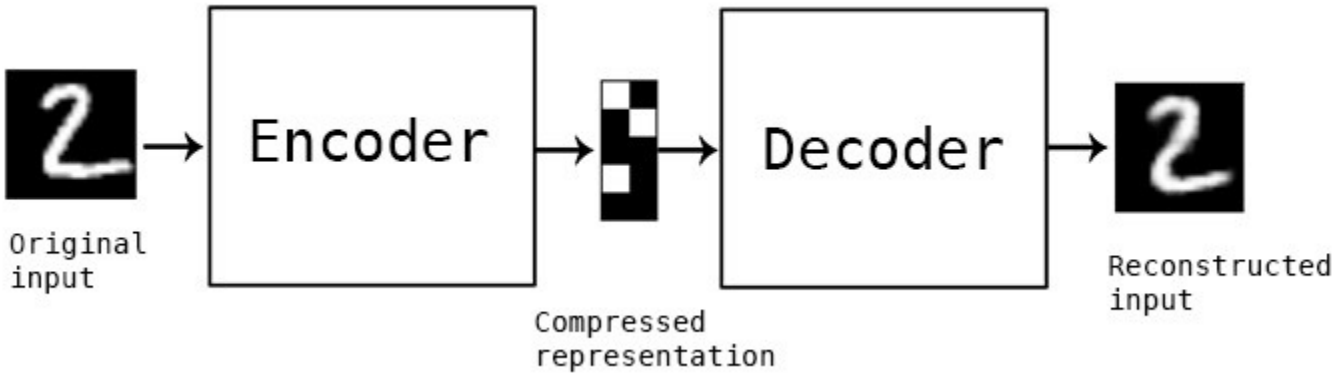

A FILM BY CHRISTOPHER NOLAN
INCEPTION
FROM THE DIRECTOR OF THE DARK KNIGHT
WARNER BROS. PICTURES PRESENTS
IN ASSOCIATION WITH LEGENDARY PICTURES A SYNCOPY PRODUCTION A FILM BY CHRISTOPHER NOLAN LEONARDO DiCAPRIO "INCEPTION" KEN WATANABE JOSEPH GORDON-LEVITT
MARION COTILLARD ELLEN PAGE TOM HARDY CILLIAN MURPHY TOM BERENGER AND MICHAEL CAINE MUSIC BY HANS ZIMMER EDITED BY LEE SMITH, A.C.E.
PRODUCTION DESIGNER GUY HENDRIX DYAS DIRECTOR OF PHOTOGRAPHY WALLY PFISTER, A.S.C. EXECUTIVE PRODUCERS CHRIS BRIGHAM THOMAS TULL PRODUCED BY EMMA THOMAS CHRISTOPHER NOLAN
PG-13 PARENTS STRONGLY CAUTIONED
LEGENDARY PICTURES
SYNCOPY
WRITTEN AND DIRECTED BY CHRISTOPHER NOLAN
inceptionmovie.com
WARNER BROS. PICTURES

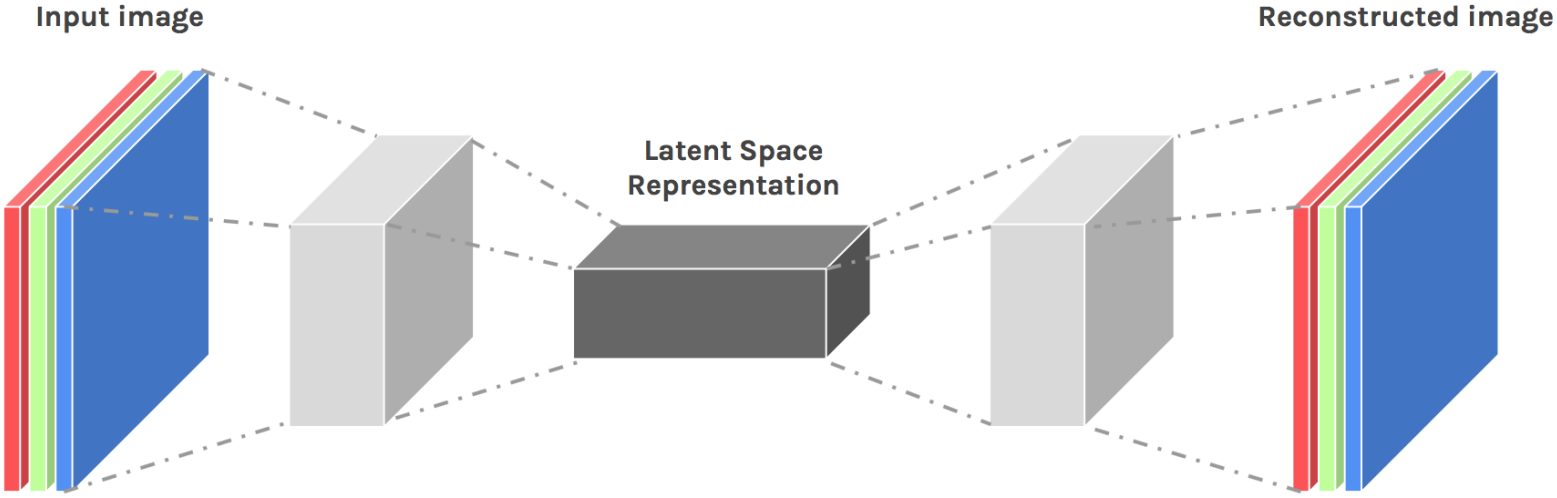


(a) Inception module, naïve version

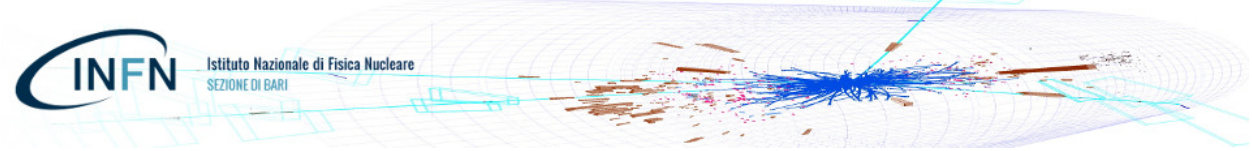
Autoencoder

Autoencoder is an **unsupervised** artificial neural network that learns how to efficiently compress and encode data, then learns how to reconstruct the data back **from** the reduced encoded representation **to** a representation that is as close to the original input as possible.



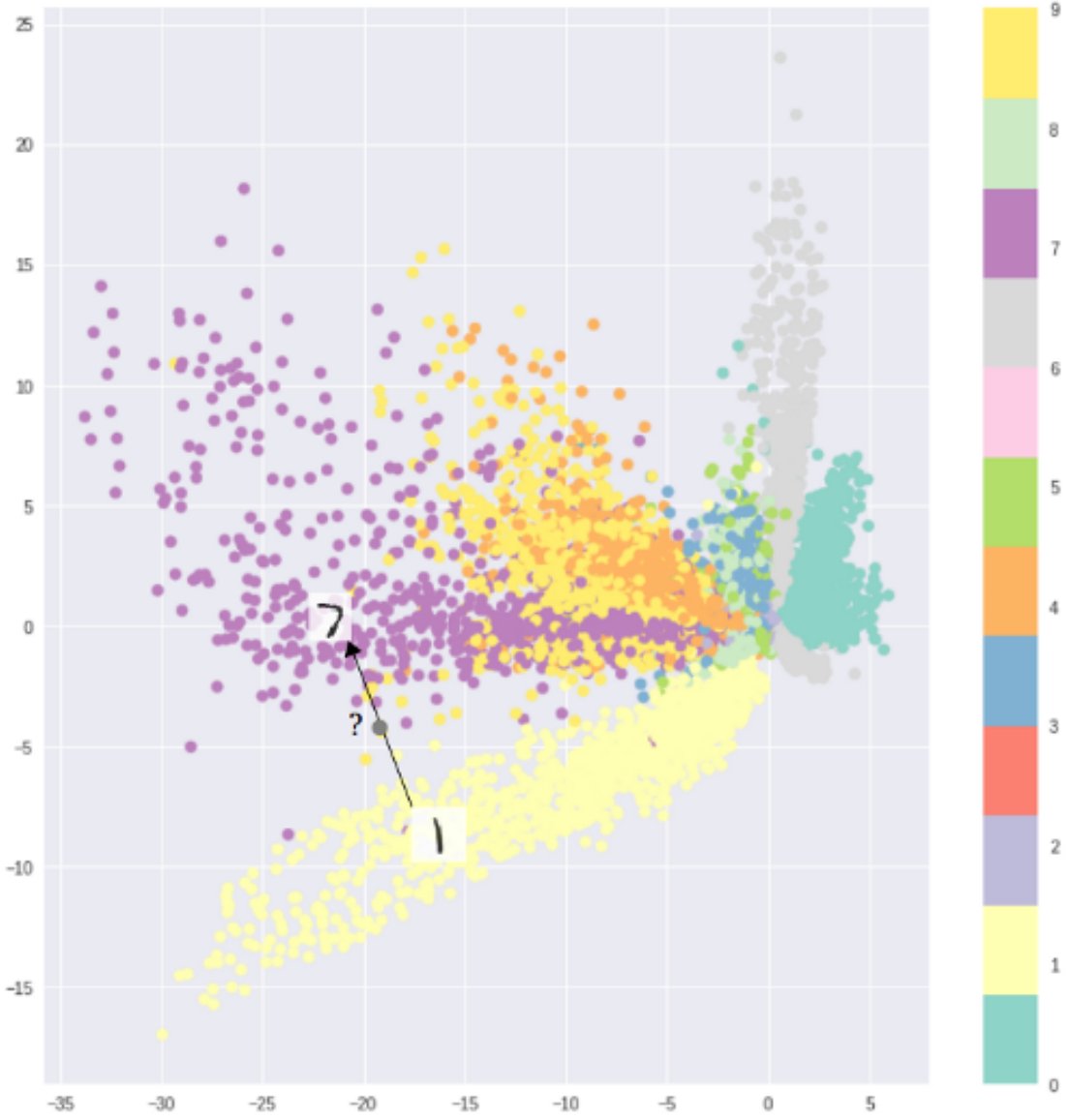


Tipical uses: anomaly detection, noise reduction, full image colorization, latent space clustering, generating high resolution images.)

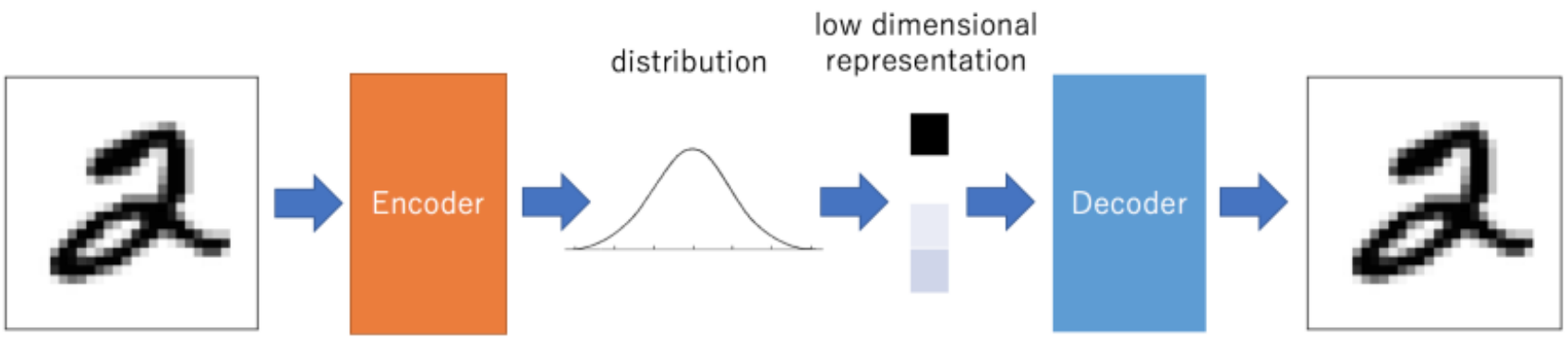


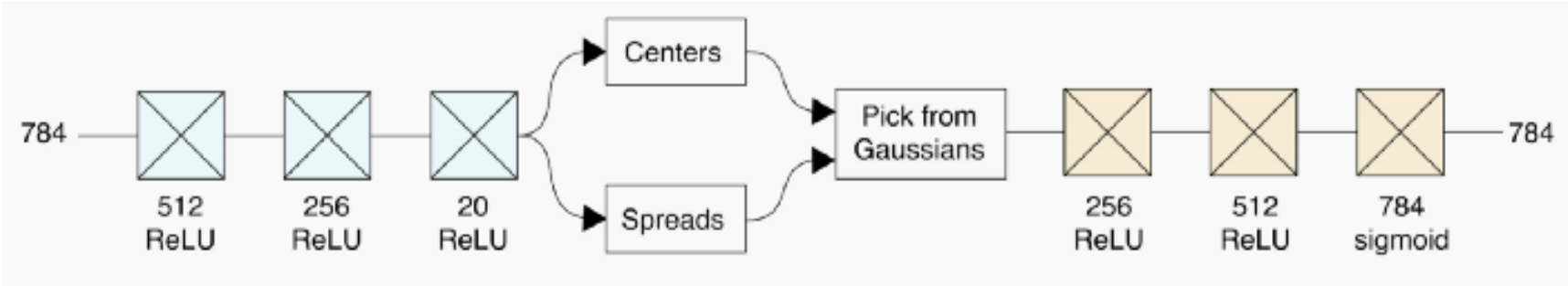
But why would I train a neural network just to output an image or data that is exactly the same as the input??

COLAB!



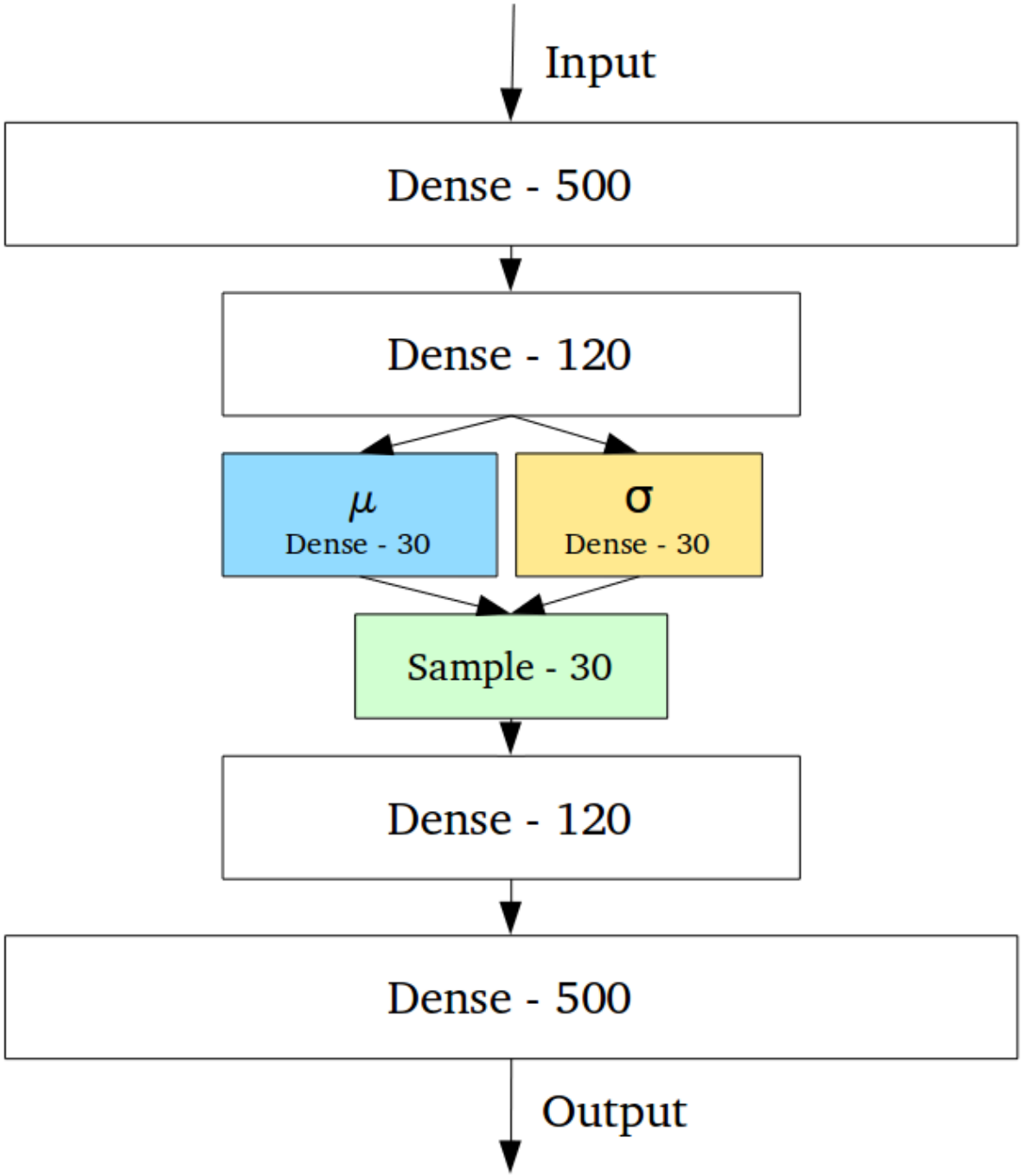
- Gaps in the latent space: unrealistic output
- Separability in the latent space
- Discrete latent space



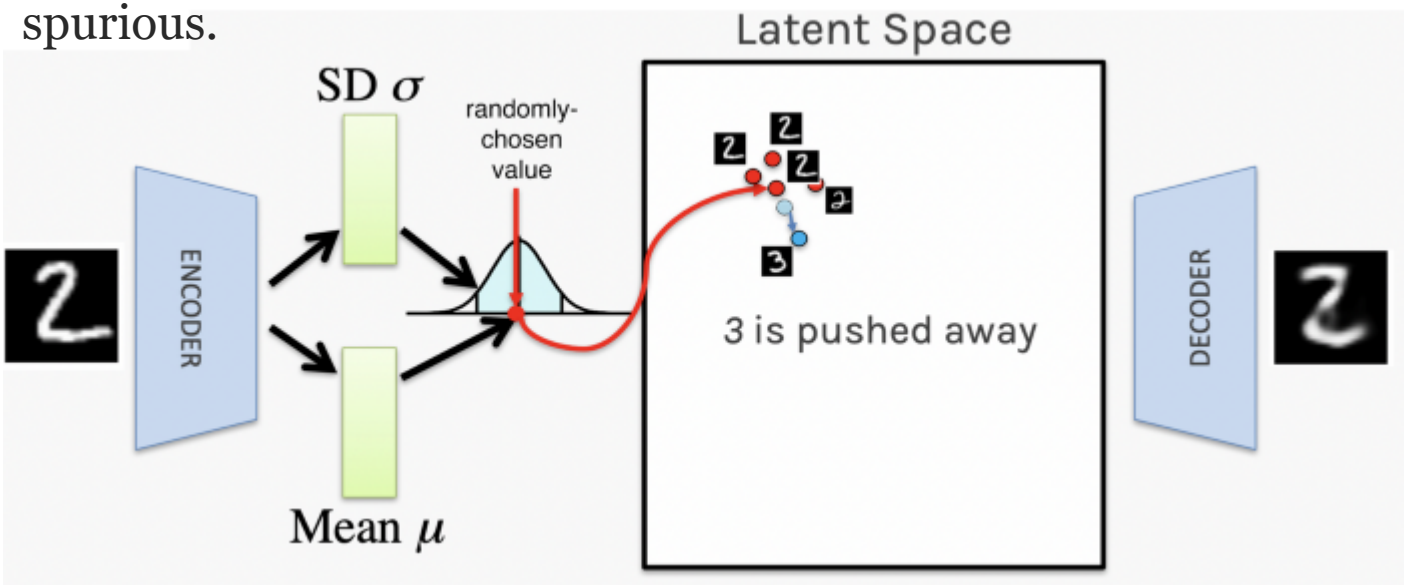


If we add a constraint on the encoding network, that forces it to generate latent vectors that roughly follow a unit gaussian distribution we have a **variational autoencoder.**

Generating new images is now easy: all we need to do is sample a latent vector from the unit gaussian and pass it into the decoder.

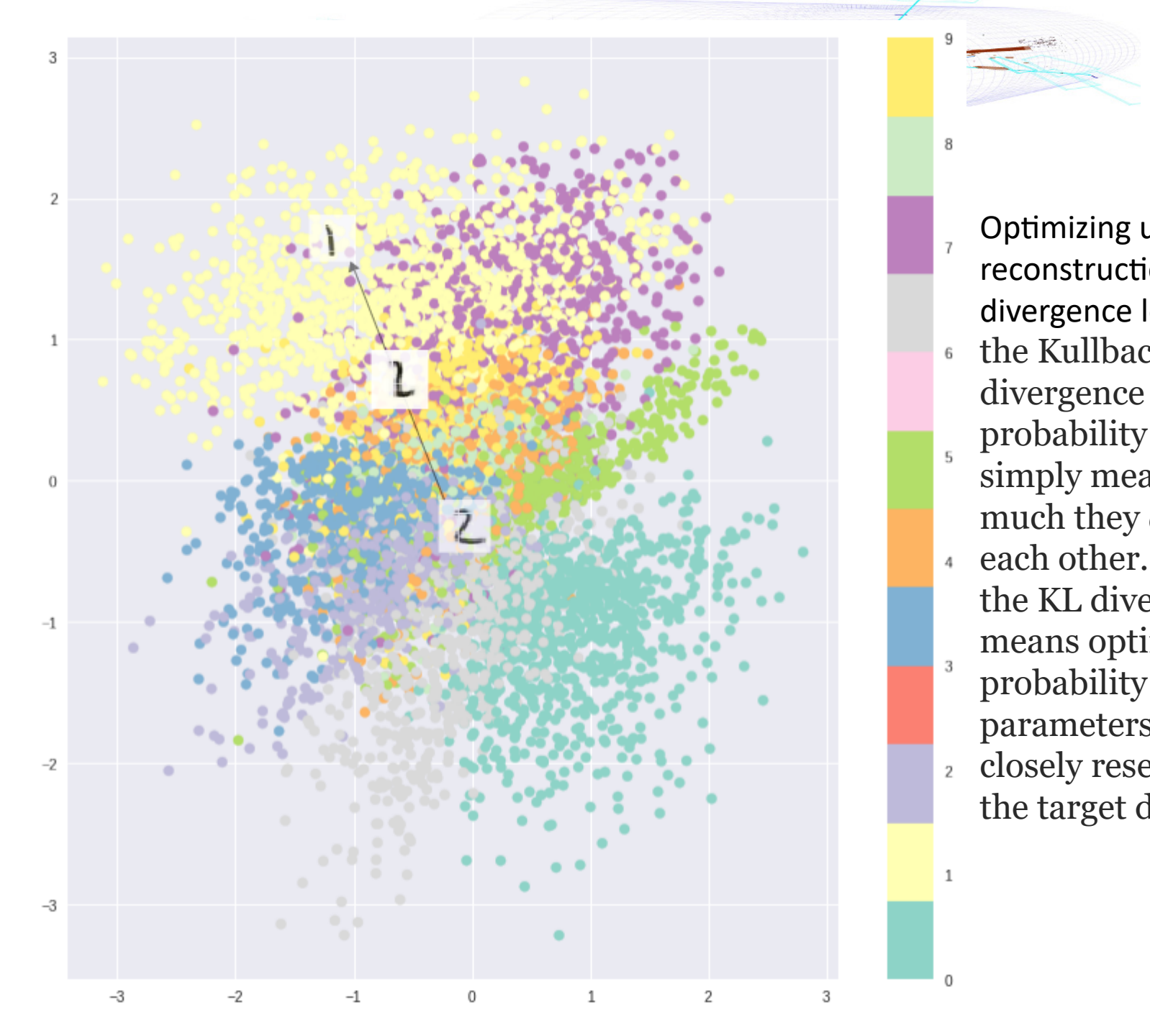

Input
Dense - 500
Dense - 120
μ
Dense - 30
σ
Dense - 30
Sample - 30
Dense - 120
Dense - 500
Output

We see that our values of 2's begin to cluster together, whilst the value 3 gradually becomes pushed away. This is useful as it means the network does not arbitrarily place characters in the latent space, making the transitions between values less spurious.



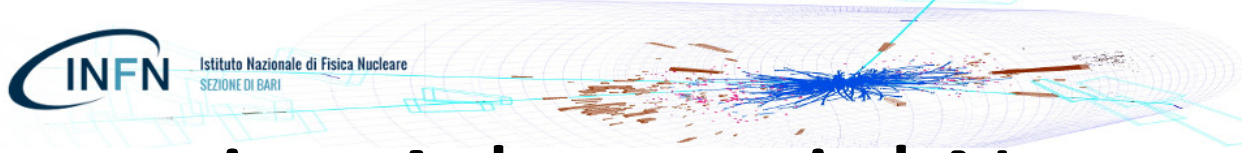
We train the autoencoder using a set of **images to learn our mean and standard deviations within the latent space**, which forms our data generating distribution. Next, when we want to generate a similar image, we sample from one of the centroids within the latent space, **distort it slightly** using our standard deviation and some random error, and then pass this through the decoder network.

COLAB!



Optimizing using both reconstruction loss and KL divergence loss

the Kullback–Leibler divergence between two probability distributions simply measures how much they *diverge* from each other. Minimizing the KL divergence here means optimizing the probability distribution parameters **(μ** and **σ)** to closely resemble that of the target distribution.

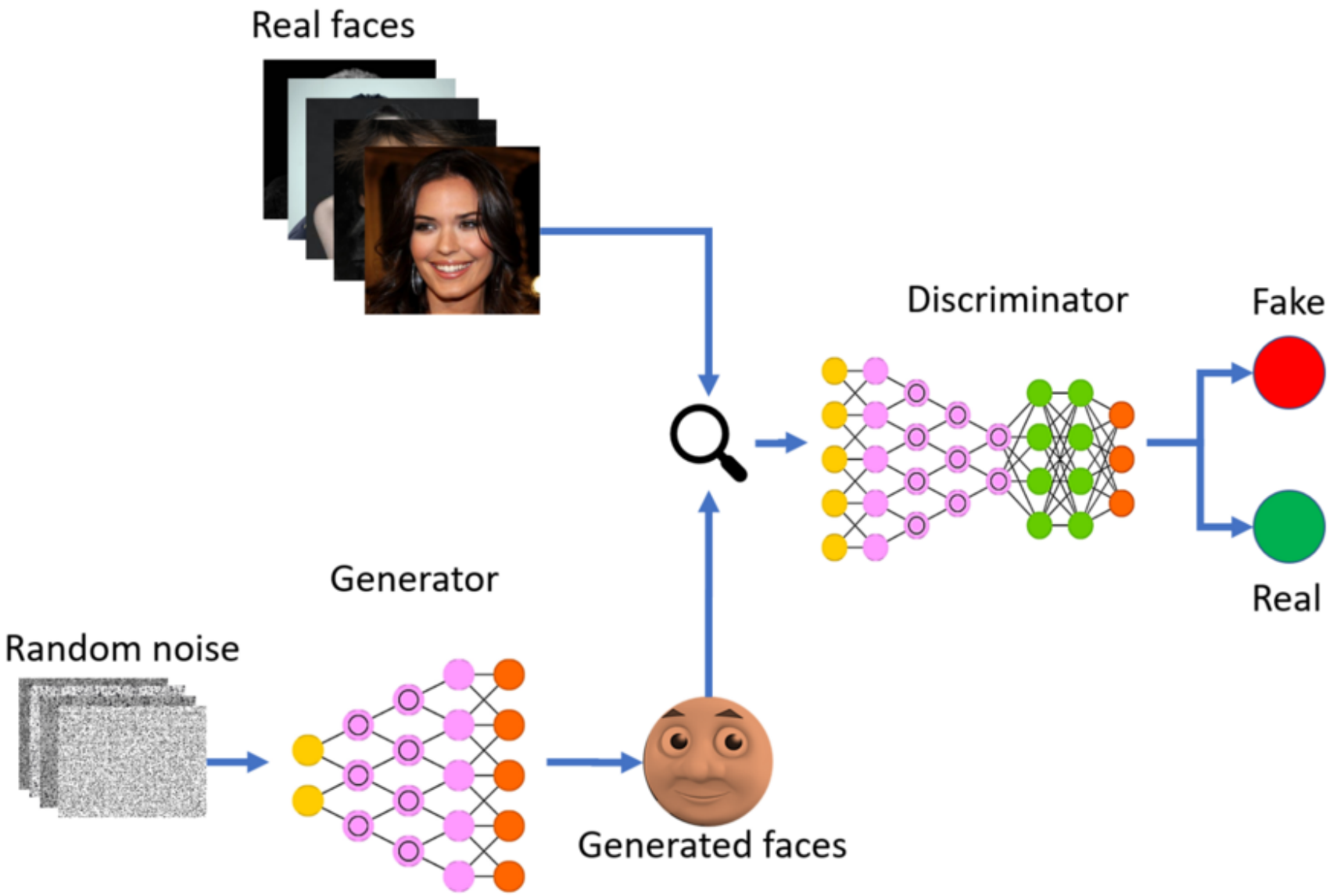


Generative Adversarial Networks

The GAN model architecture involves two sub-models: a *generator model* for generating new examples and a *discriminator model* for classifying whether generated examples are real, from the domain, or fake, generated by the generator model.

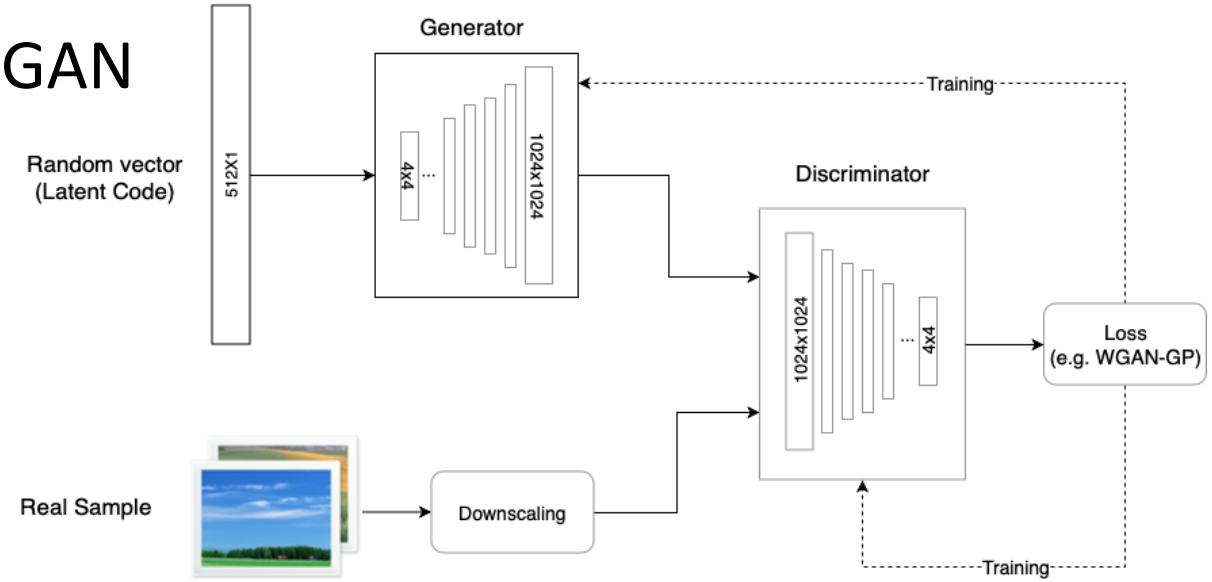
- **Generator**. Model that is used to generate new plausible examples for the problem domain.
- **Discriminator**. Model that is used to classify examples as real (*from the domain*) or fake (*generated*).

Generative adversarial networks are based on a game theoretic scenario in which the generator network must compete against an adversary. The generator network directly produces samples. Its adversary, the discriminator network, attempts to distinguish between samples drawn from the training data and samples drawn from the generator.



https://www.kdnuggets.com/2020/03/generate-realistic-human-face-using-gan.html

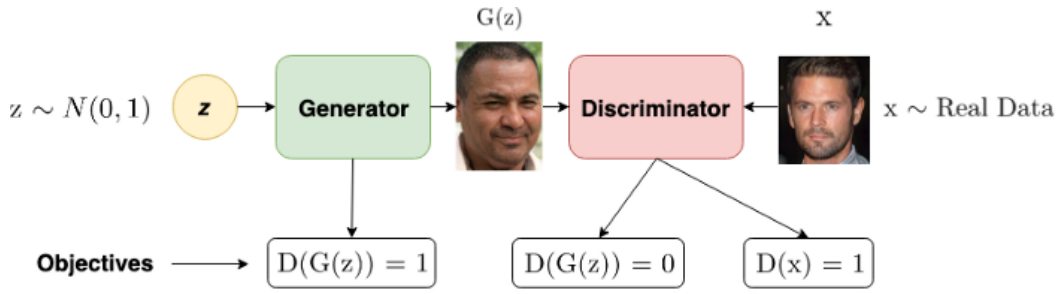
GAN



1.The **discriminator** wants to *maximize* the probability of the real data being identified as “real” and the generated data being identified as “fake”.

2.The **generator** wants to *minimize* the probability that the discriminator identifies its generated data as “fake”.

- The generator takes in random numbers and returns an image.
- This generated image is fed into the discriminator alongside a stream of images taken from the actual, ground-truth dataset.
- The discriminator takes in both real and fake images and returns probabilities, a number between 0 and 1, with 1 representing a prediction of authenticity and 0 representing fake.



At convergence, the generator’s samples are indistinguishable from real data, and the discriminator outputs 1/2 everywhere. The discriminator may then be discarded.

https://www.thispersondoesnotexist.com/

Why Generative Adversarial Networks?

https://www.louisbouchard.ai/deepfakes-in-5-minutes/

https://github.com/iperov/DeepFaceLab

RNN

Recurrent neural networks (RNN) are a class of neural networks that is powerful for modeling sequence data such as time series or natural language.

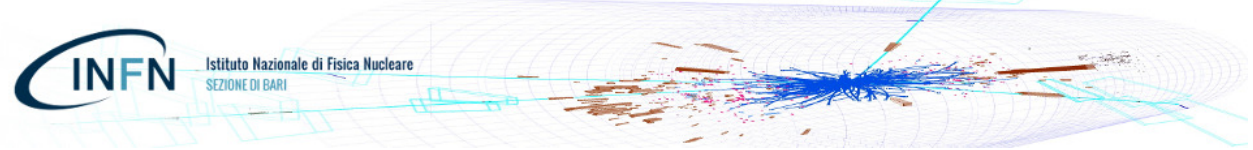
Schematically, a RNN layer uses a for loop to iterate over the timesteps of a sequence, while maintaining an internal state that encodes information about the timesteps it has seen so far.

There are three built-in RNN layers in Keras:

keras.layers.SimpleRNN, a fully-connected RNN where the output from previous timestep is to be fed to next timestep.

keras.layers.LSTM, first proposed in Hochreiter & Schmidhuber, 1997.

keras.layers.GRU, first proposed in Cho et al., 2014.



RNNs are widely used in the following domains/ applications:

- Prediction problems.
- Language Modelling and Generating Text.
- Machine Translation.
- Speech Recognition.
- Generating Image Descriptions.
- Video Tagging.
- Text Summarization.
- Call Center Analysis.

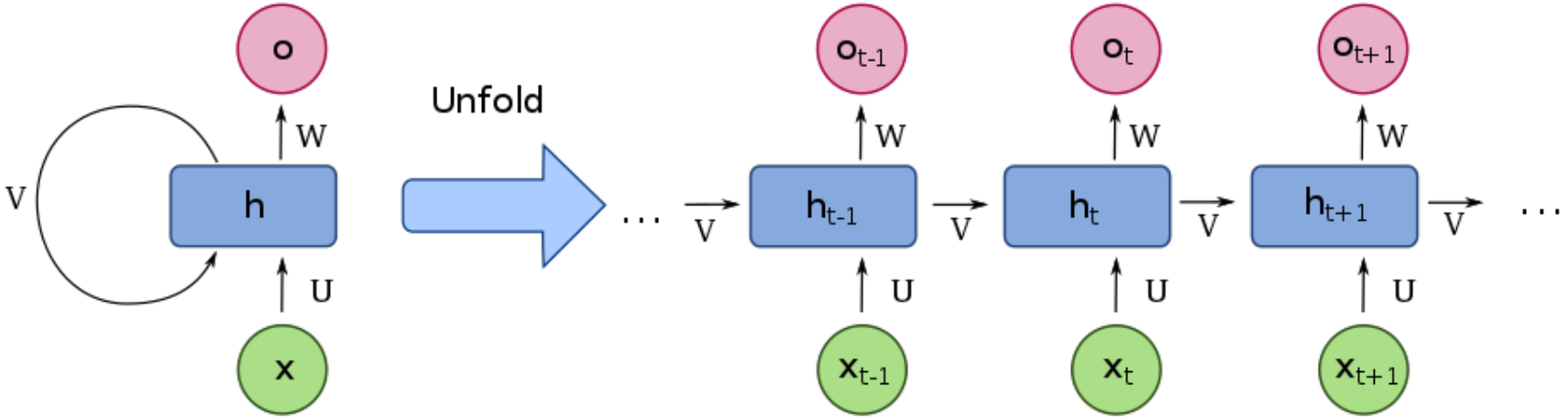
"In summary, you don't need to understand everything about the specific architecture of an LSTM cell; as a human, it shouldn't be your job to understand it. Just keep in mind what the LSTM cell is meant to do: allow past information to be reinjected at a later time."

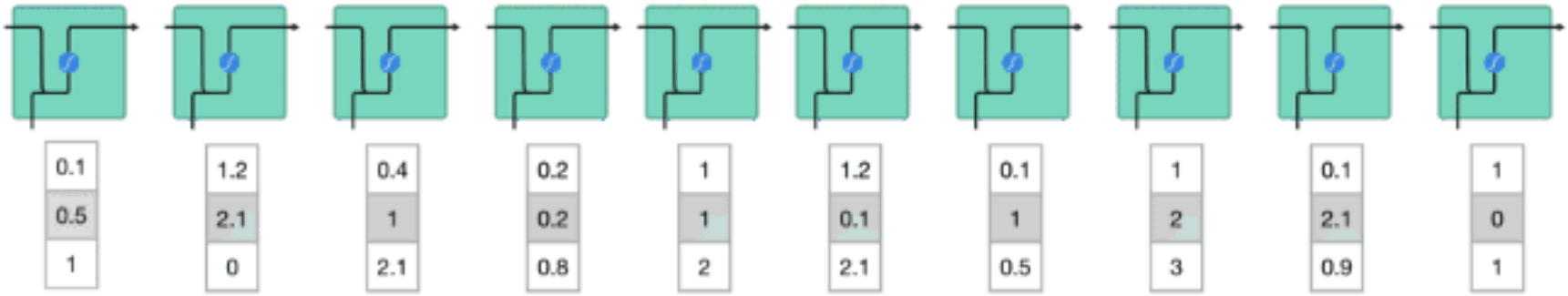
Francois Chollet *(author of Keras) - Deep Learning with Python*

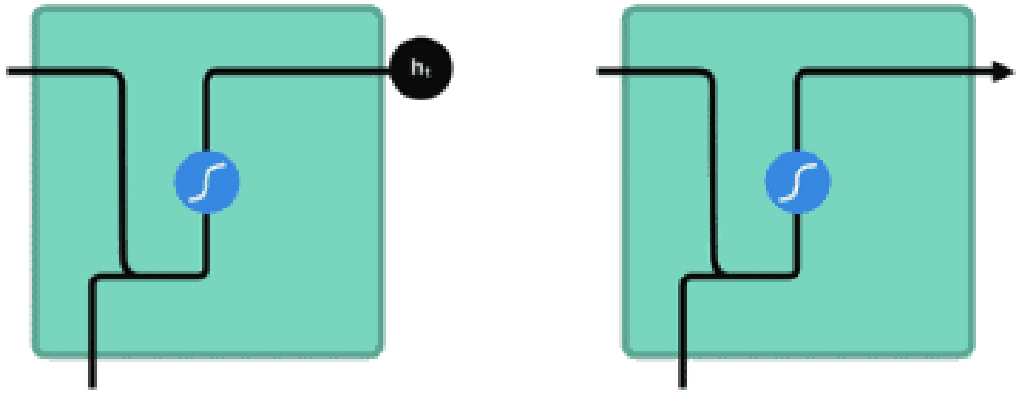
At a high level, a **recurrent neural network (RNN)** processes sequences — whether daily stock prices, sentences, or sensor measurements — one element at a time while retaining a memory (called a state) of what has come previously in the sequence.

Recurrent means the output at the current time step becomes the input to the next time step. At each element of the sequence, the model considers not just the current input, but what it remembers about the preceding elements.

A RNN is designed to mimic the human way of processing sequences: we consider the entire sentence when forming a response instead of words by themselves

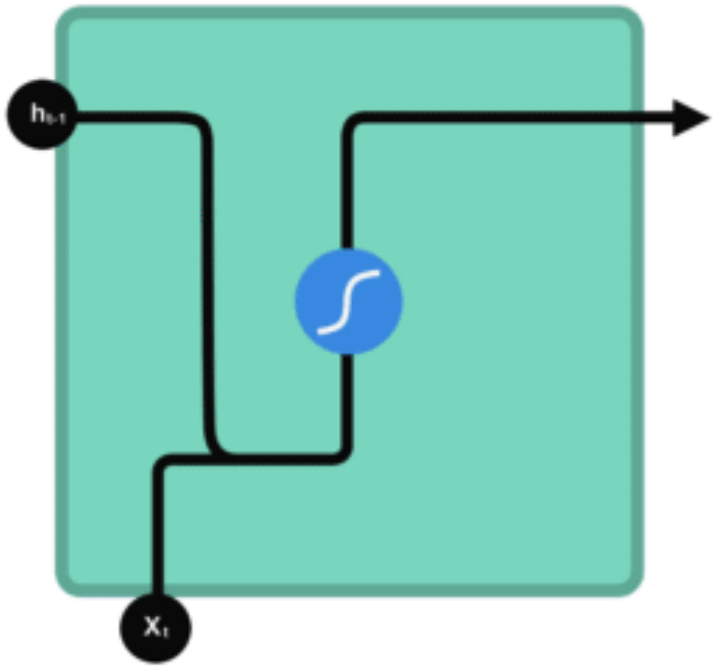






Tanh function

h_t hidden state (memory)



Tanh function

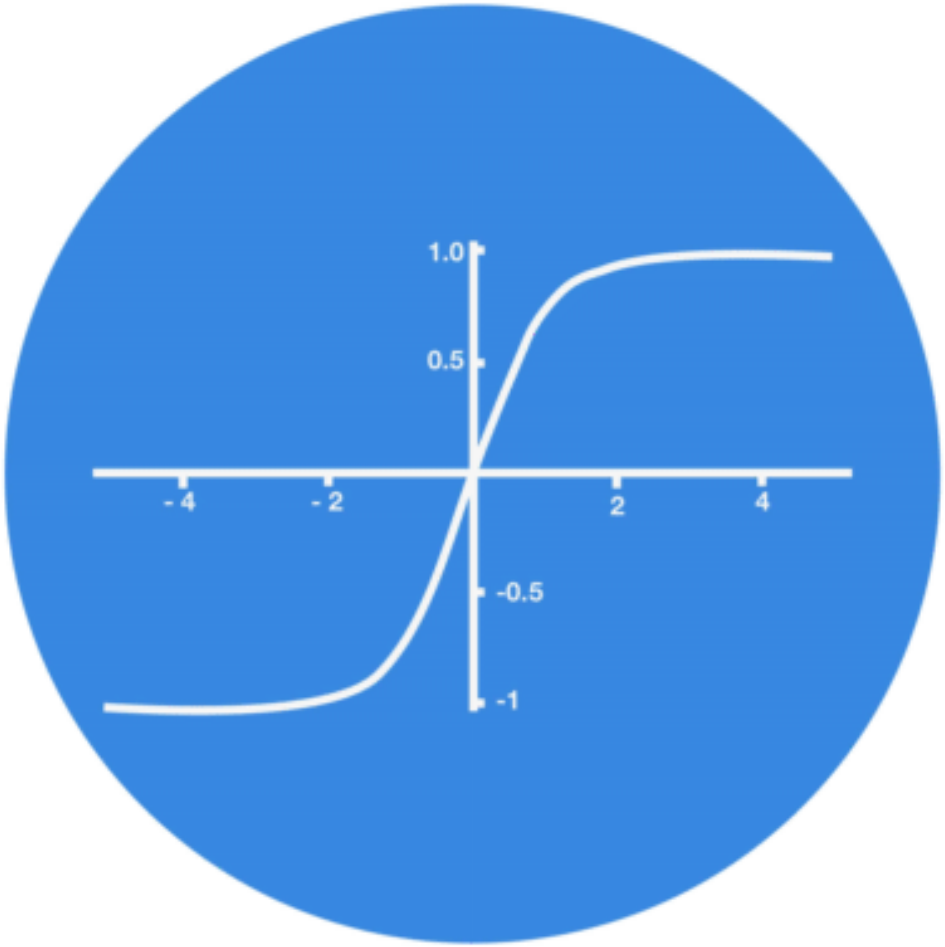
h_t new hidden state

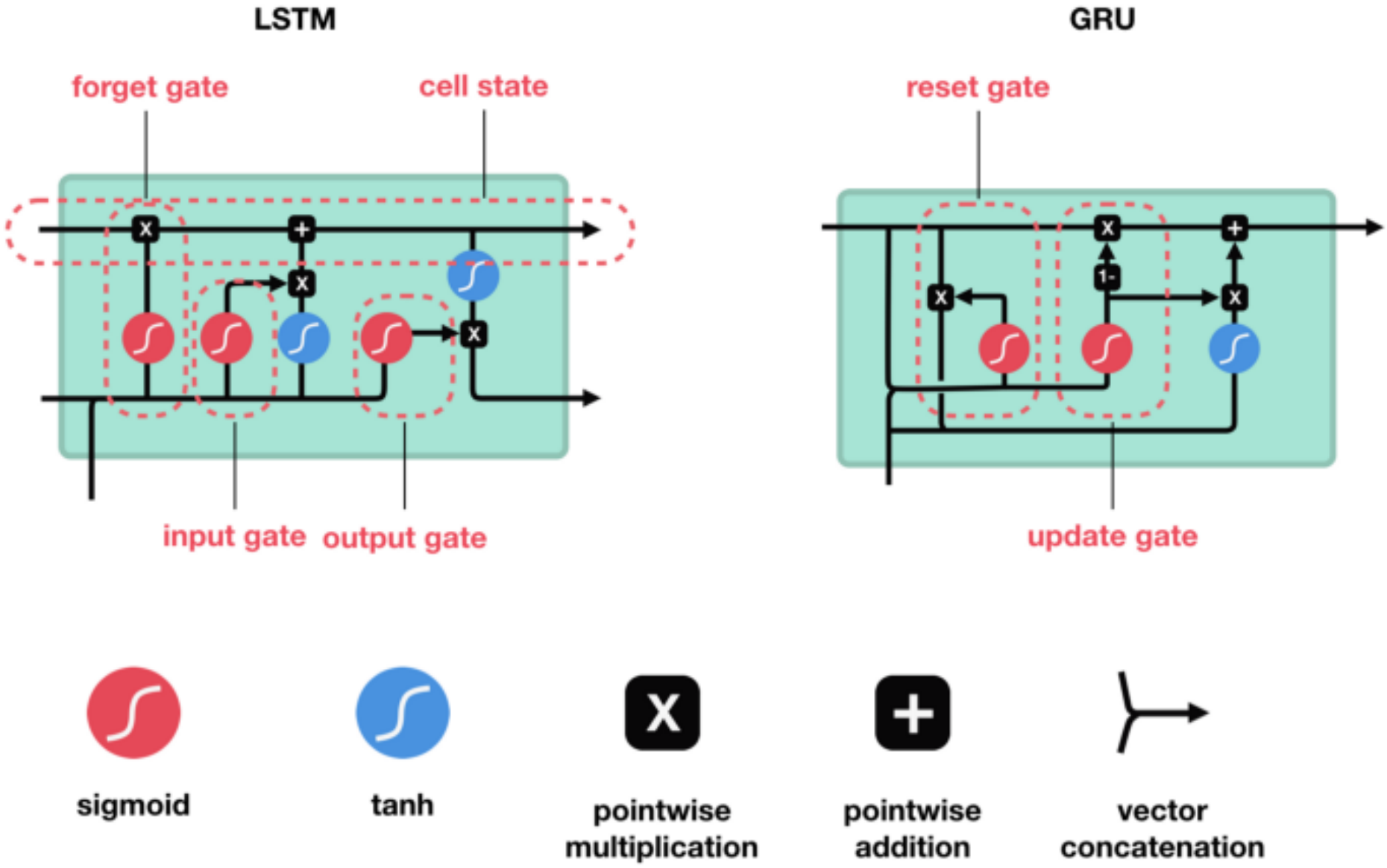
h_{t-1} previous hidden state

x_t input

concatenation

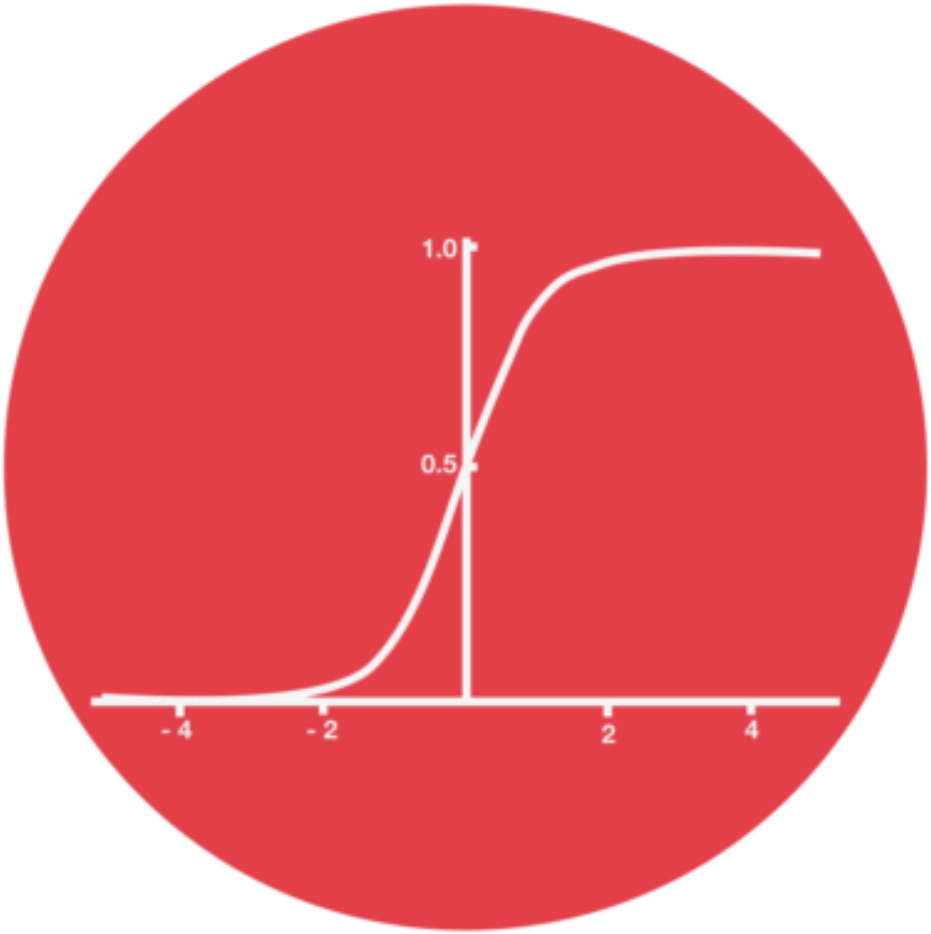
5
0.1
-0.5

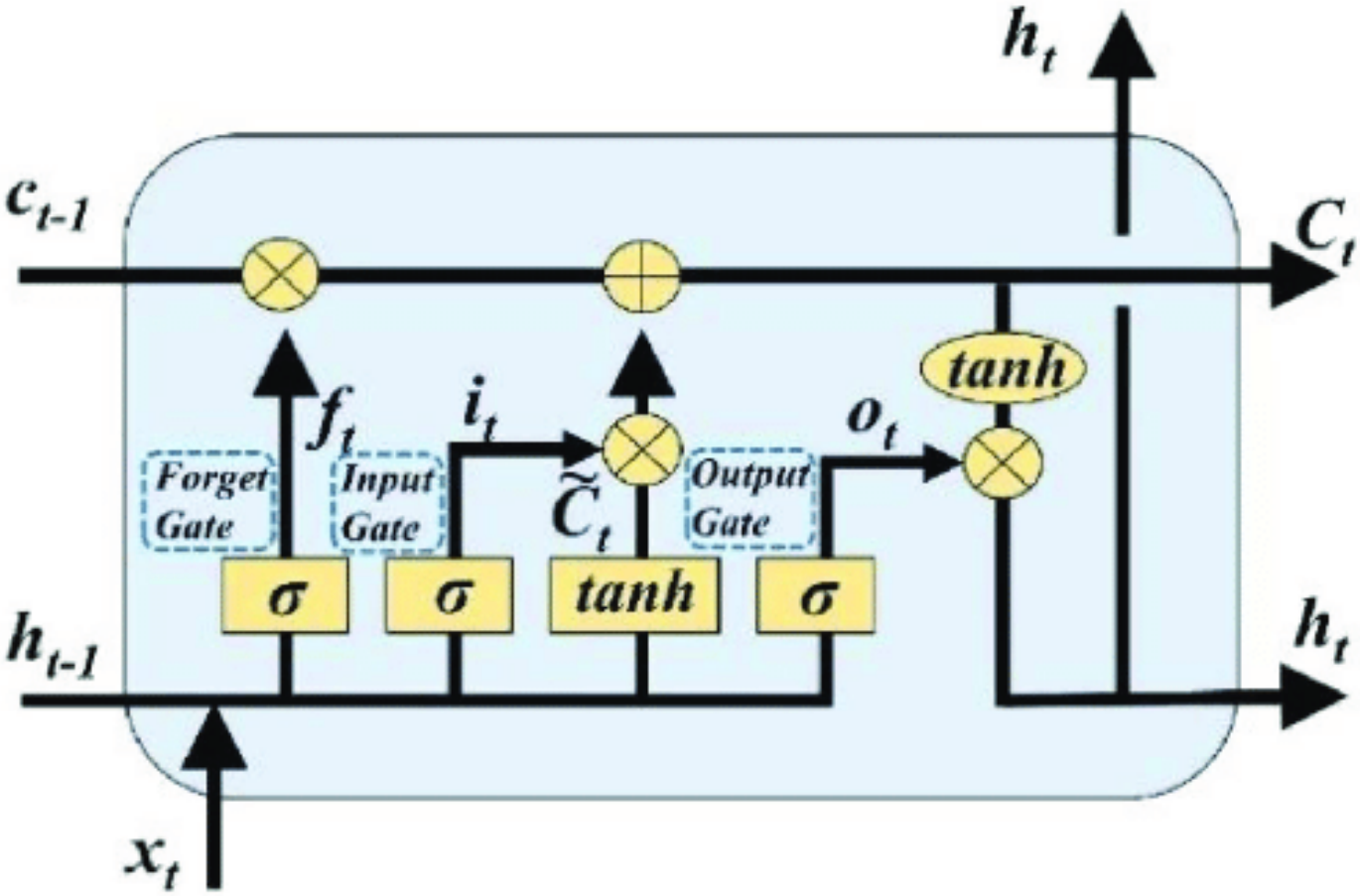


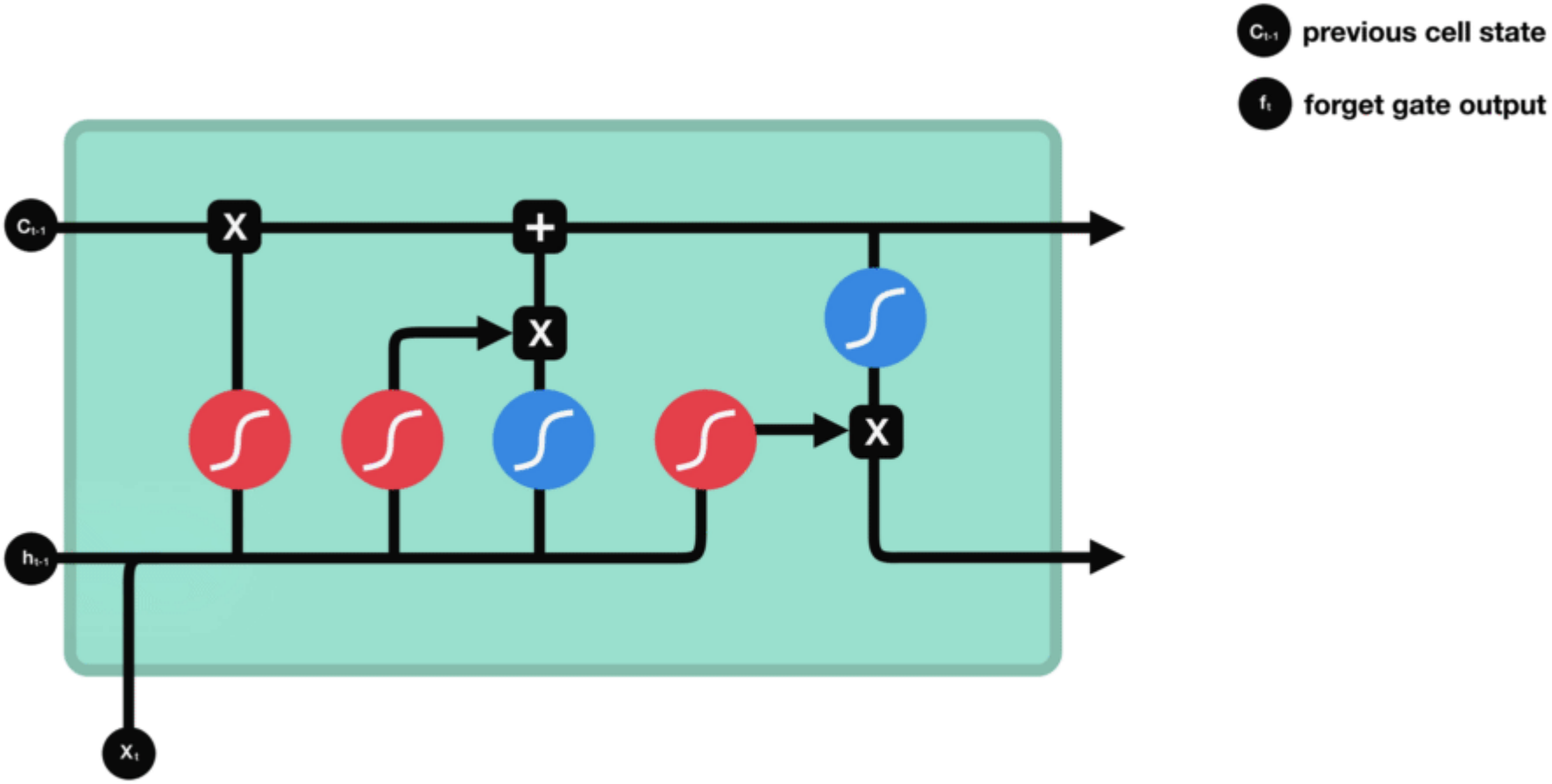


These gates can learn which data in a sequence is important to keep or throw away. By doing that, it can pass relevant information down the long chain of sequences to make predictions.

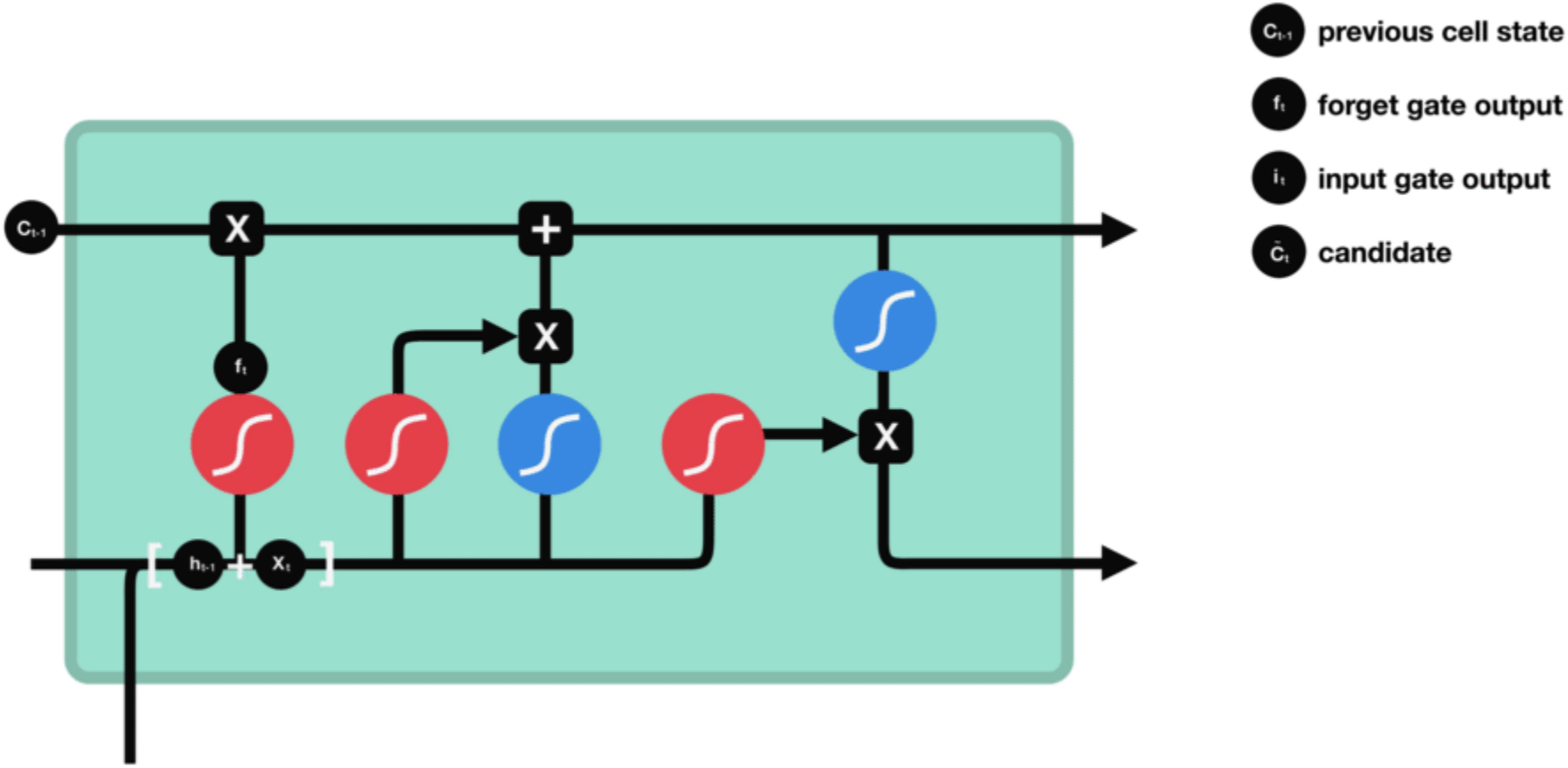
5
0.1
-0.5



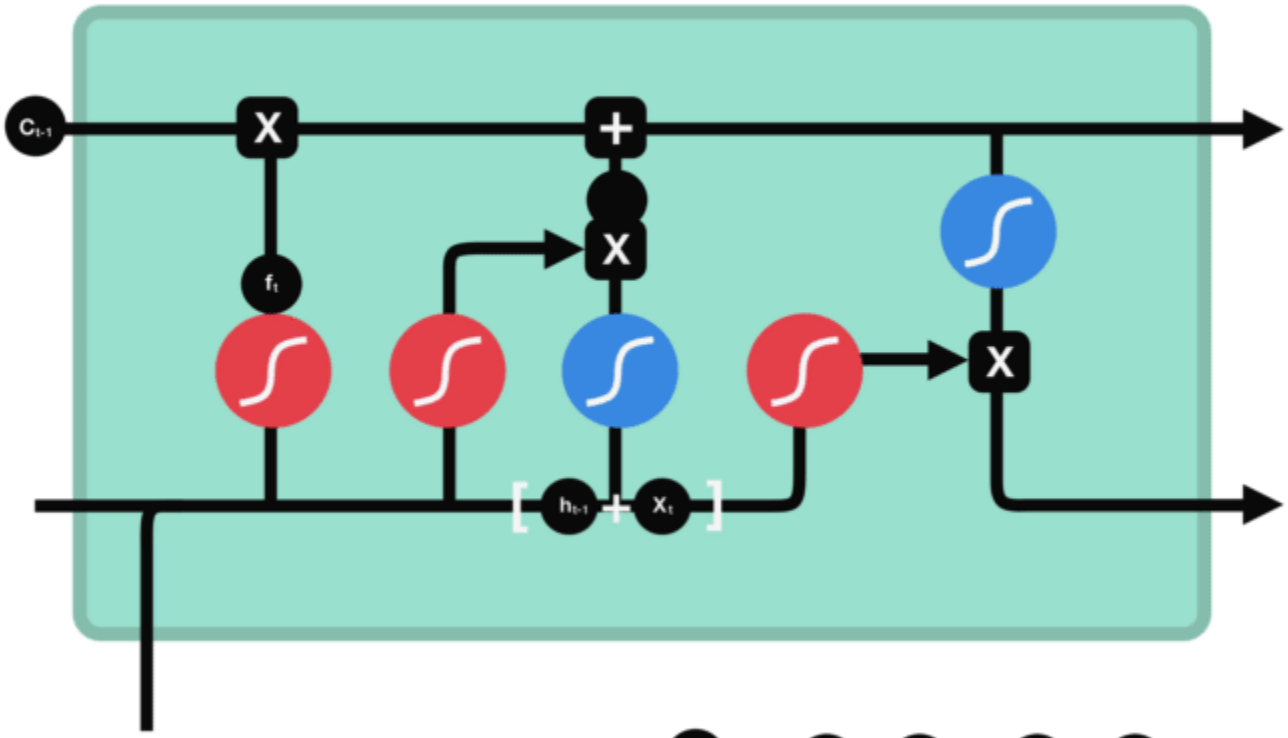

h_t
c_{t-1}
C_t
tanh
f_t
i_t
o_t
Forget Gate
Input Gate
$\tilde{C}_t$
Output Gate
σ
σ
tanh
σ
h_{t-1}
h_t
x_t



This gate decides what information should be thrown away or kept.



First, we pass the previous hidden state and current input into a sigmoid function. You also pass the hidden state and current input into the tanh function to squish values between -1 and 1 to help regulate the network. Then you multiply the tanh output with the sigmoid output.



C_{t-1} previous cell state

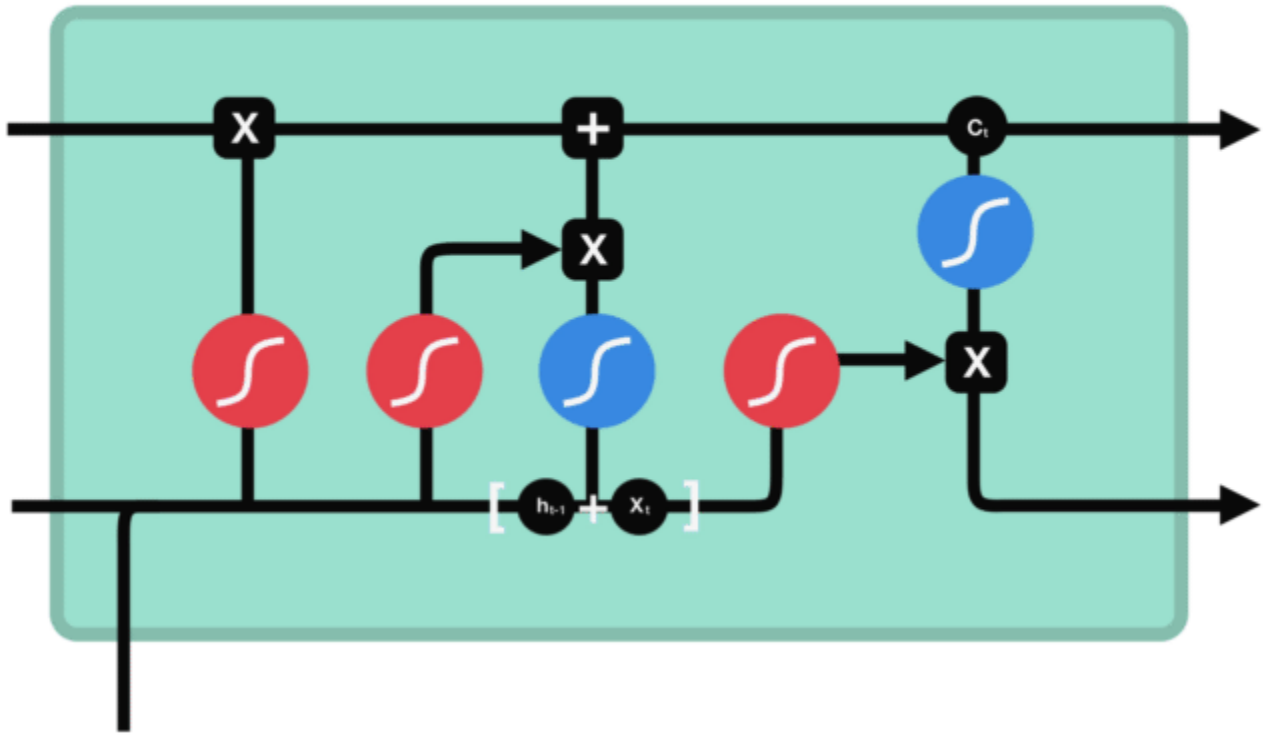
f_t forget gate output

i_t input gate output

$\tilde{C}_t$ candidate

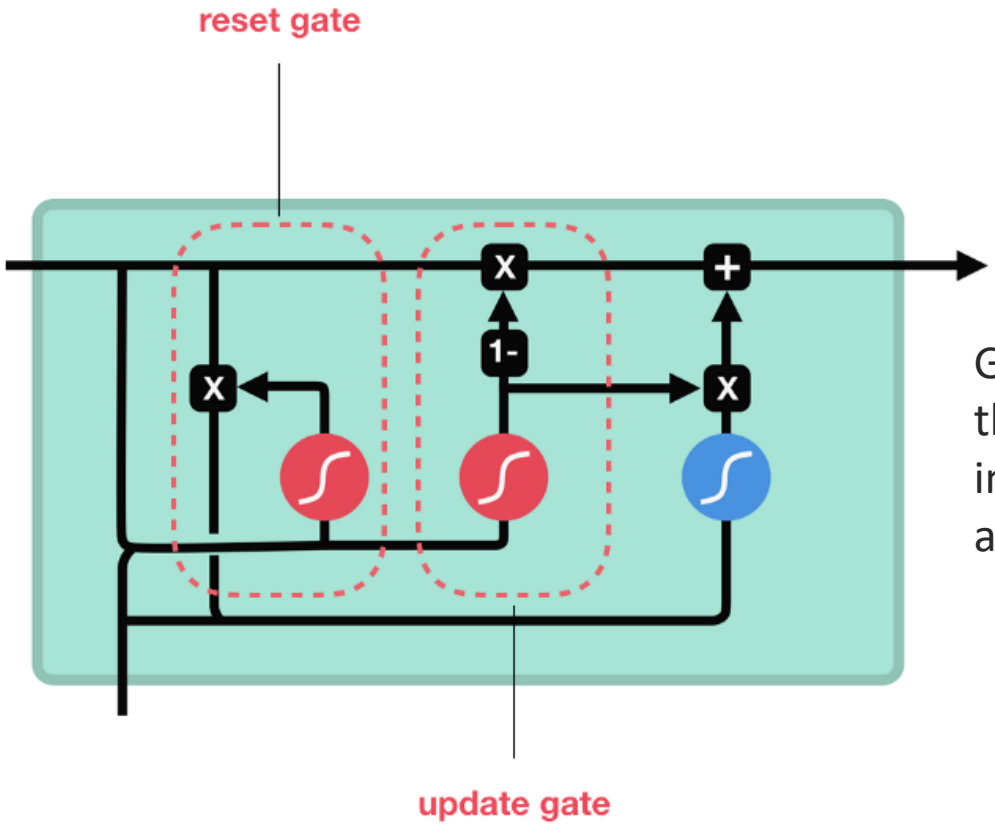
C_t new cell state

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

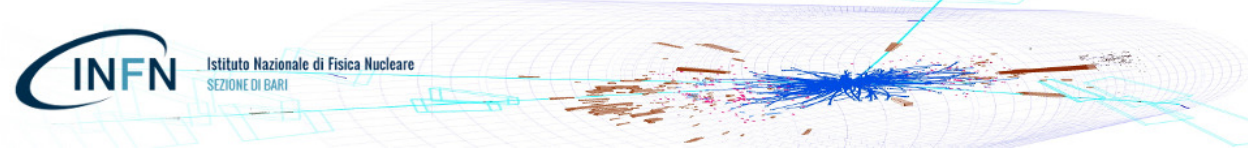


- c_{t-1} previous cell state
- f_t forget gate output
- i_t input gate output
- $\tilde{c}_t$ candidate
- c_t new cell state
- o_t output gate output
- h_t hidden state

GRU



GRU's got rid of the cell state and used the hidden state to transfer information. It also only has two gates, a reset gate and update gate.



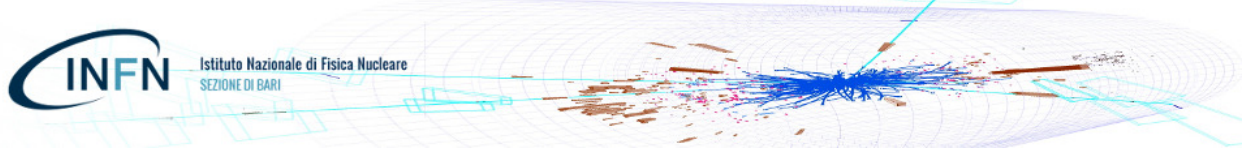
```
def LSTMCELL(prev_ct, prev_ht, input):
    combine = prev_ht + input
    ft = forget_layer(combine)
    candidate = candidate_layer(combine)
    it = input_layer(combine)
    Ct = prev_ct * ft + candidate * it
    ot = output_layer(combine)
    ht = ot * tanh(Ct)
    return ht, Ct



ct = [0, 0, 0]
ht = [0, 0, 0]

for input in inputs:
    ct, ht = LSTMCELL(ct, ht, input)
```

COLAB!



- https://towardsdatascience.com/conv2d-to-finally-understand-what-happens-in-the-forward-pass-1bbaafb0b148
- https://towardsdatascience.com/intuitive-deep-learning-part-2-cnns-for-computer-vision-472bbb2c8060
- https://towardsdatascience.com/a-simple-guide-to-the-versions-of-the-inception-network-7fc52b863202
- https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/
- https://towardsdatascience.com/auto-encoder-what-is-it-and-what-is-it-used-for-part-1-3e5c6f017726
- https://machinelearningmastery.com/how-to-develop-a-cnn-from-scratch-for-fashion-mnist-clothing-classification/
- https://stackabuse.com/autoencoders-for-image-reconstruction-in-python-and-keras/
- https://towardsdatascience.com/deep-inside-autoencoders-7e41f319999f
- https://www.pyimagesearch.com/2014/09/15/python-compare-two-images/
- https://towardsdatascience.com/convolutional-neural-networks-in-practice-406426c6c19a
- https://towardsdatascience.com/generating-images-with-autoencoders-77fd3a8dd368
- https://jaan.io/what-is-variational-autoencoder-vae-tutorial/
- https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/
- https://www.kdnuggets.com/2020/03/generate-realistic-human-face-using-gan.html
- https://machinelearningmastery.com/how-to-develop-a-generative-adversarial-network-for-an-mnist-handwritten-digits-from-scratch-in-keras/
- https://towardsdatascience.com/recurrent-neural-networks-by-example-in-python-ffd204f99470
- https://colah.github.io/posts/2015-08-Understanding-LSTMs/
- https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21
- https://www.retineuraliartificiali.net/deep-learning/keras-tutorial/keras-lstm-tutorial/
- https://towardsdatascience.com/predictive-analytics-time-series-forecasting-with-gru-and-bilstm-in-tensorflow-87588c852915
- https://www.knime.com/blog/use-deep-learning-to-write-like-shakespeare
- https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf