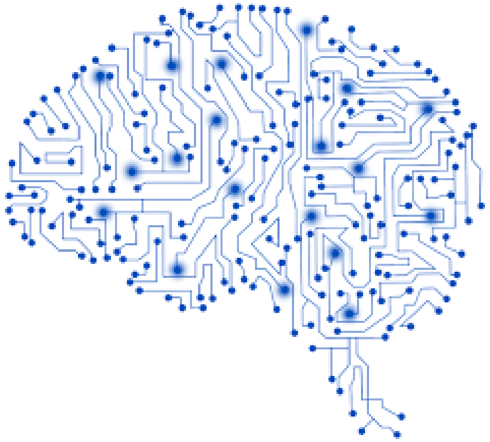
University of Minho
School of Engineering



Aprendizagem Profunda

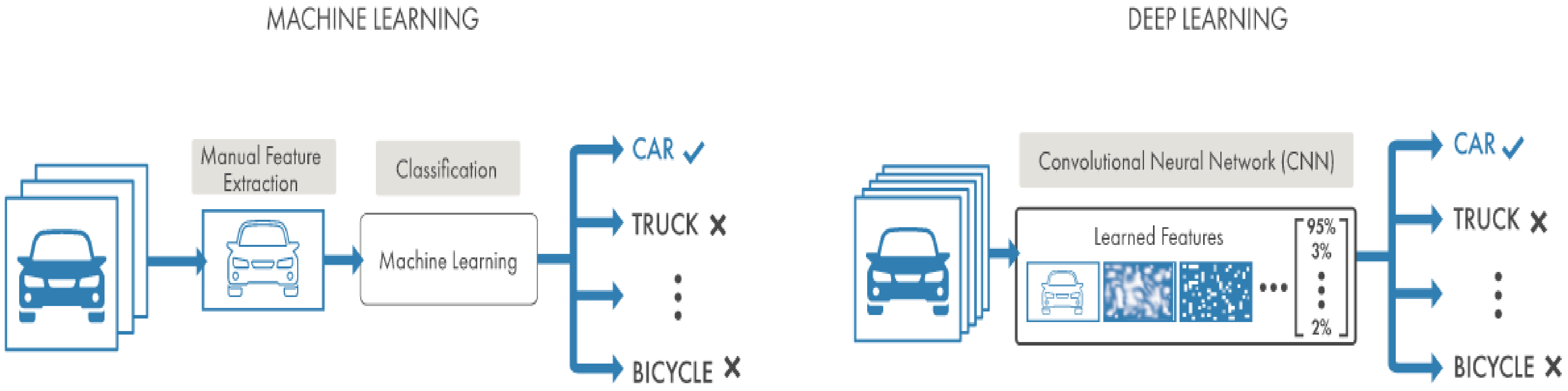
Convolutional Neural Networks

AP @ MEI/4º ano – 2º Semestre
Victor Alves

Content

- Frameworks
- Convolutions
- CNN (Layers)
- Train process

Motivation


MACHINE LEARNING
Manual Feature Extraction
Classification
Machine Learning
CAR ✓
TRUCK ✗
BICYCLE ✗
DEEP LEARNING
Convolutional Neural Network (CNN)
Learned Features
95%
3%
2%
CAR ✓
TRUCK ✗
BICYCLE ✗

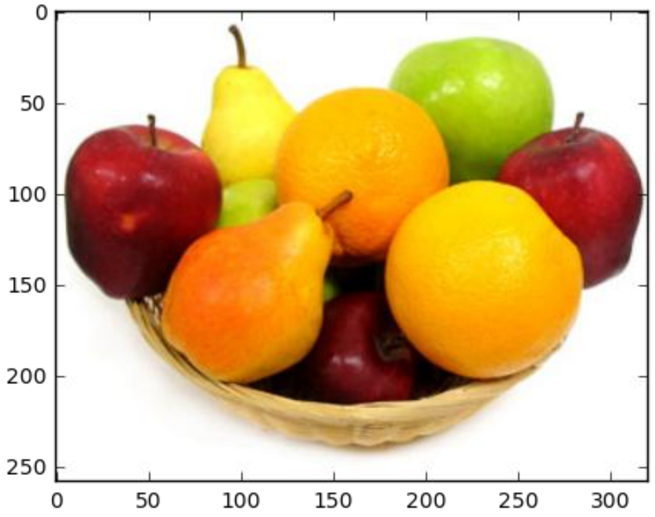
Frameworks – Core Components

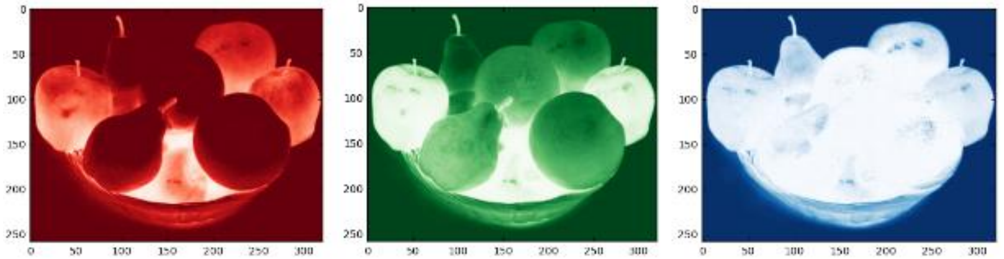
Fundamental components of any **DL framework**:

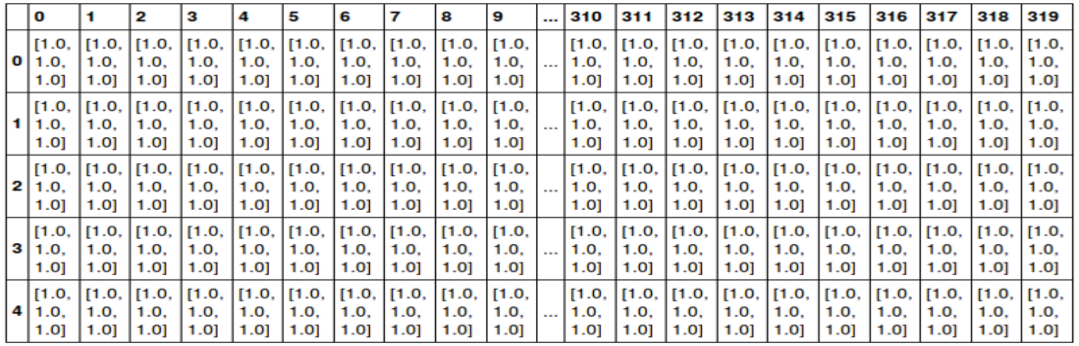
- ❑ **The Tensor Object**
- ❑ **Operations on the Tensor object**
- ❑ **Graph computing and optimisation**
- ❑ Automatic **differentiation tools**
- ❑ Extensions **BLAS** / **cuBLAS** and **cuDNN**

The Tensor Object

At the core of any **framework** is the **Tensor object**. A tensor is a generalisation of an n-dimensional matrix (like numpy's ndarrays). For example, let's consider a 258 x 320 (height x width) colour (RGB) bitmap image. This is a 3-dimensional tensor (**height, width, colour channels**).





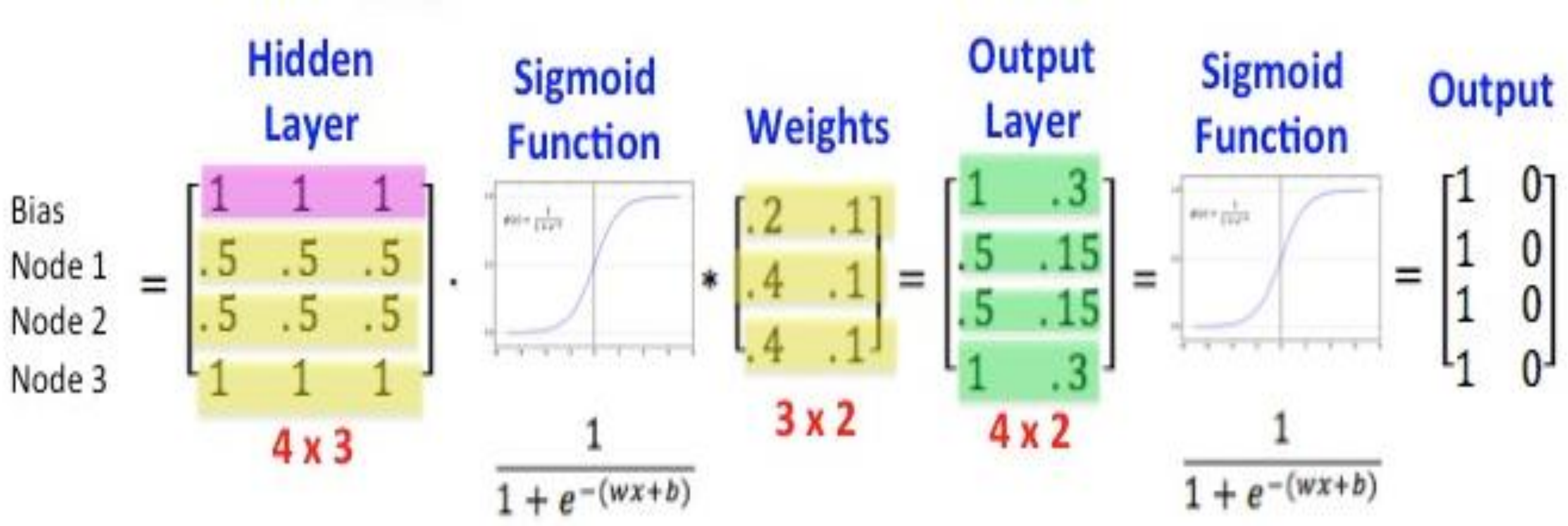


	0	1	2	3	4	5	6	7	8	9	...	310	311	312	313	314	315	316	317	318	319
0	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	...	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]
1	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	...	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]
2	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	...	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]
3	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	...	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]
4	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	...	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]

Analogously a set of 100 images can be represented by a 4-dimensional tensor (**image id, height, width, colour channels**).

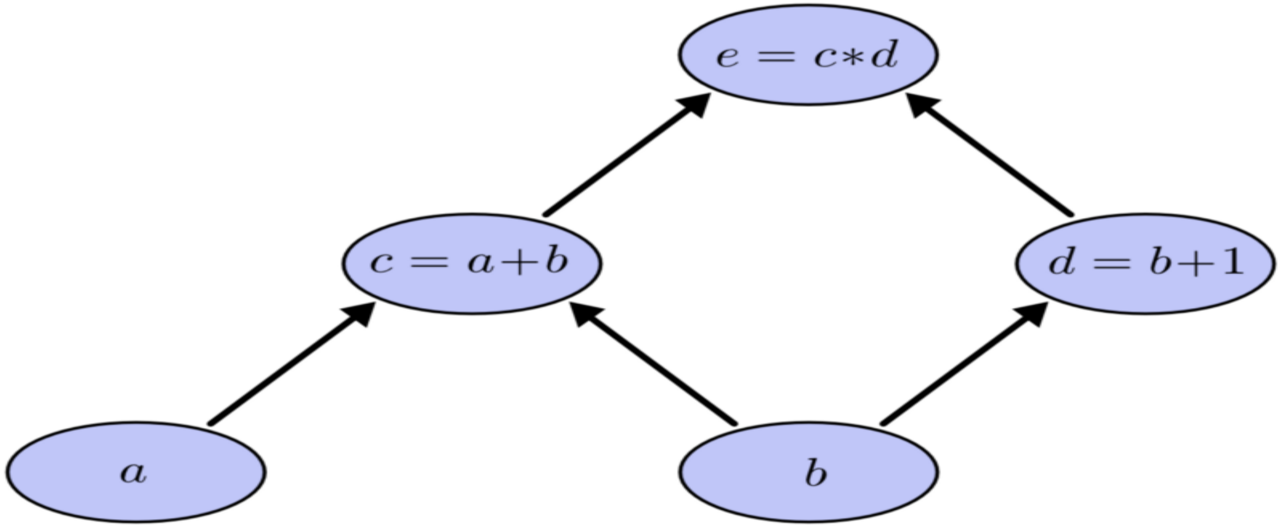
Operations on the Tensor object

We can look at a neural network as a series of operations performed on the input tensor to produce a result. Learning is achieved by correcting the errors between the obtained result and the expected one. These operations range from simple matrix multiplication to more complicated operations like **convolutions**, **pooling** or **LSTMs**.



Graph computing and optimisation

Until now we would have to consider classes to represent tensors and operations on them. The power of neural networks depends on their ability to chain multiple operations in order to highlight their learning properties. It is therefore necessary to provide the framework with the ability to **compute over graphs** and their **optimization**.



Automatic Differentiation

Another advantage of having the ability to compute over graphs is that it makes the computation of the gradients used in the learning phase a modular and simple computation process. This is thanks to the chain rule in differential calculus that allows the computation of a composition of functions in a systematic way. Considering neural networks as a composition of nonlinearities that give rise to more complex functions, the differentiation of these functions can be seen as the traversal of a graph from the output to the input.

Symbolic Differentiation or **Automatic Differentiation** is a programmable mode in which gradients can be computed using computation over graphs.

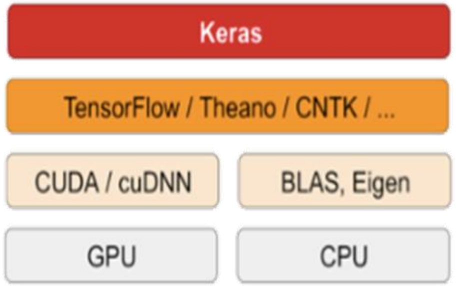
Extensions BLAS / cuBLAS e cuDNN

- ❑ Although these extensions are not mandatory to have a fully functional framework, the problem arises from the fact that usually the implementation is done in a high level programming language (Java / Python / Lua), inherently leading to a limit on the processing speed that can be achieved. This happens because even in the simplest of operations a high level language always takes more time (CPU cycles) than when executed in a low level language.

- ❑ **BLAS** or **B**asic **L**inear **A**lgebra **S**ubprograms are a collection of optimized functions for matrix processing, initially strict in Fortran. They can be used to perform operations on tensors achieving substantially higher processing speeds. There are many alternatives such as Intel **MKL** or **ATLAS** that perform similar functions.

- ❑ The **BLAS** package optimizes by assuming the instructions will be executed on a **CPU**. In the case of deep learning this is generally not the case and **BLAS** may not fully take advantage of the parallelism provided by GPUs. To solve this issue NVIDIA developed **cuBLAS** which is optimized for **GPUs**. It is now included in the **CUDA** toolkit. **cuDNN** is a library built using **cuBLAS** that provides operations optimized specifically for neural networks.

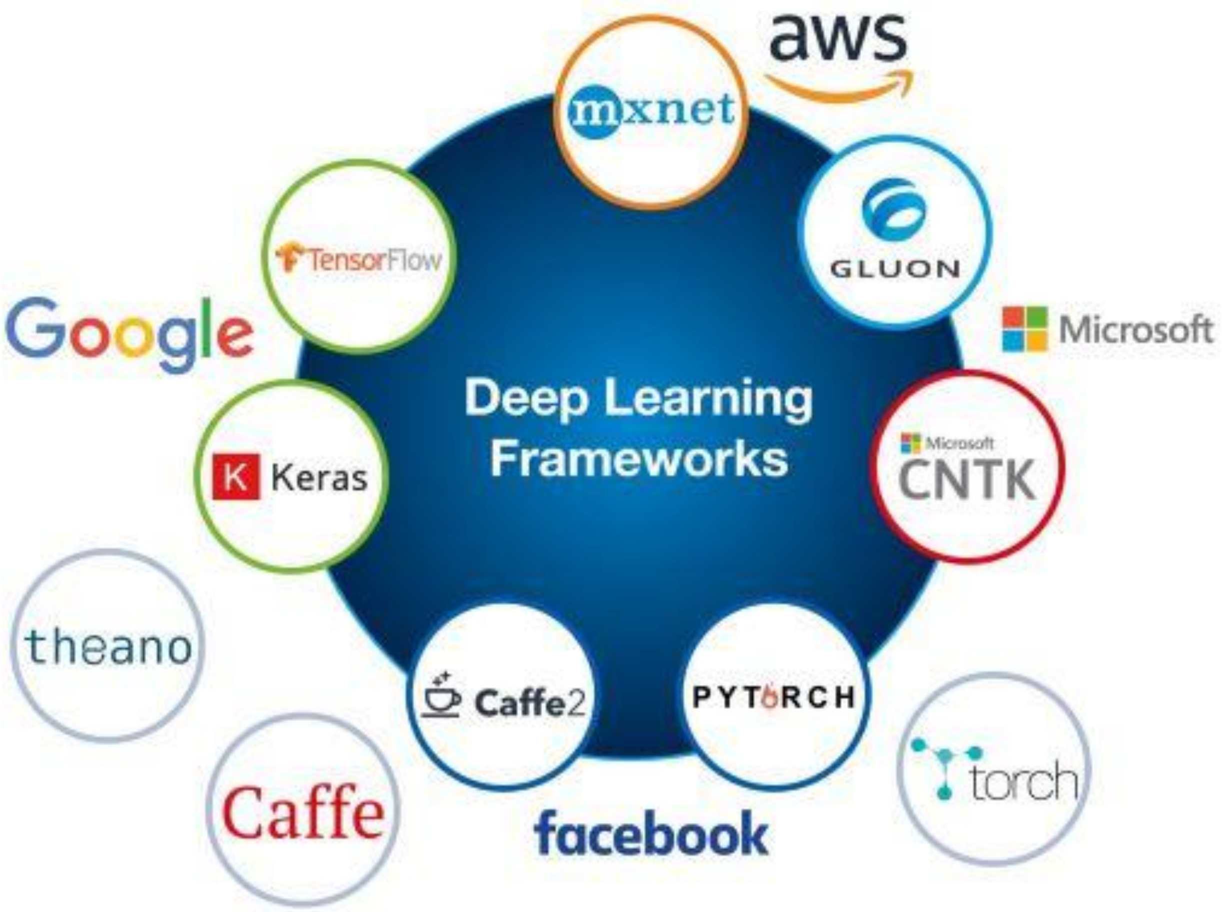
- ❑ For **AMD** there are similar libraries (opted for open source: ROCm).



Frameworks

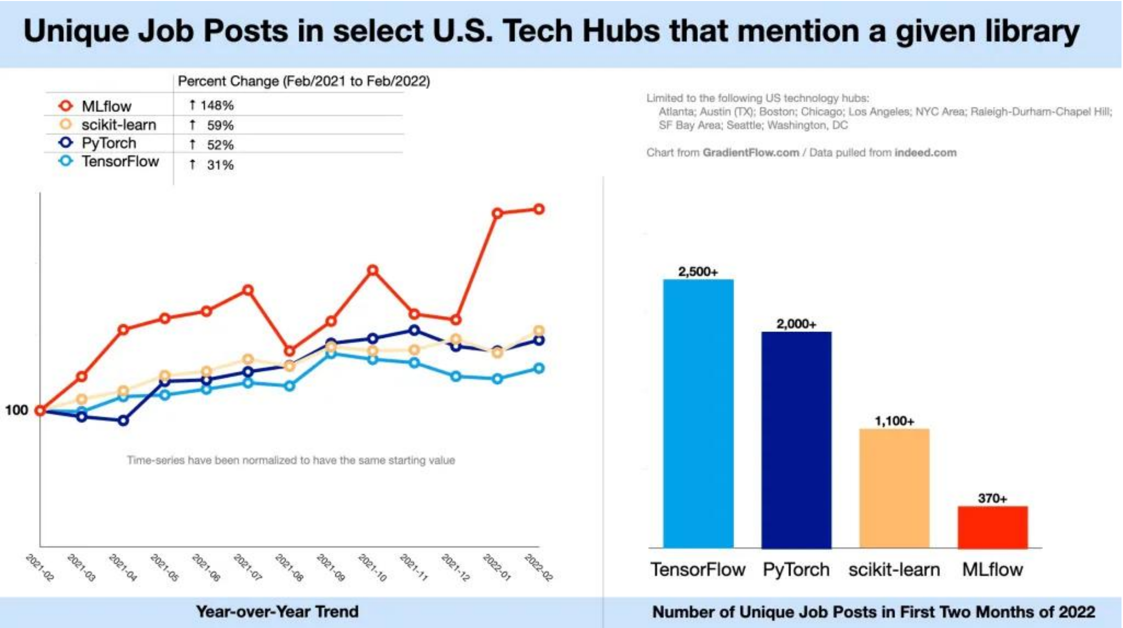
- ❑ **Theano** – Python library that allows you to define, optimize and calculate mathematical expressions with multidimensional arrays efficiently. It works with GPUs and efficiently performs differential calculations. (University of Montreal's lab, MILA)
- ❑ **Lasagne** – light library for building and training neural networks using **Theano**.
- ❑ **Blocks** – theano-based framework for building and training neural networks
- ❑ **TensorFlow** – open source library for numerical computation using graphs. (Google Brain team)
- ❑ **Keras** – Deep learning library for Python. Runs on Theano or TensorFlow.
- ❑ **MXNet** – deep learning framework designed for efficiency and flexibility. (Amazon)
- ❑ **PyTorch** - Flexible tensors and neural networks with strong GPU support . (Facebook Artificial Intelligence Research team (FAIR))
- ❑ **Torch** - (Ronan Collobert)
- ❑ **Caffe** - (Berkeley Vision and Learning Center)
- ❑ **CNTK** - (Microsoft)
- ❑ **Deeplearning4j** - (Skymind)

Frameworks



aws
mxnet
GLUON
Microsoft
Microsoft CNTK
PYTORCH
torch
facebook
Caffe2
Caffe
theano
Keras
Google
TensorFlow
Deep Learning Frameworks

Frameworks



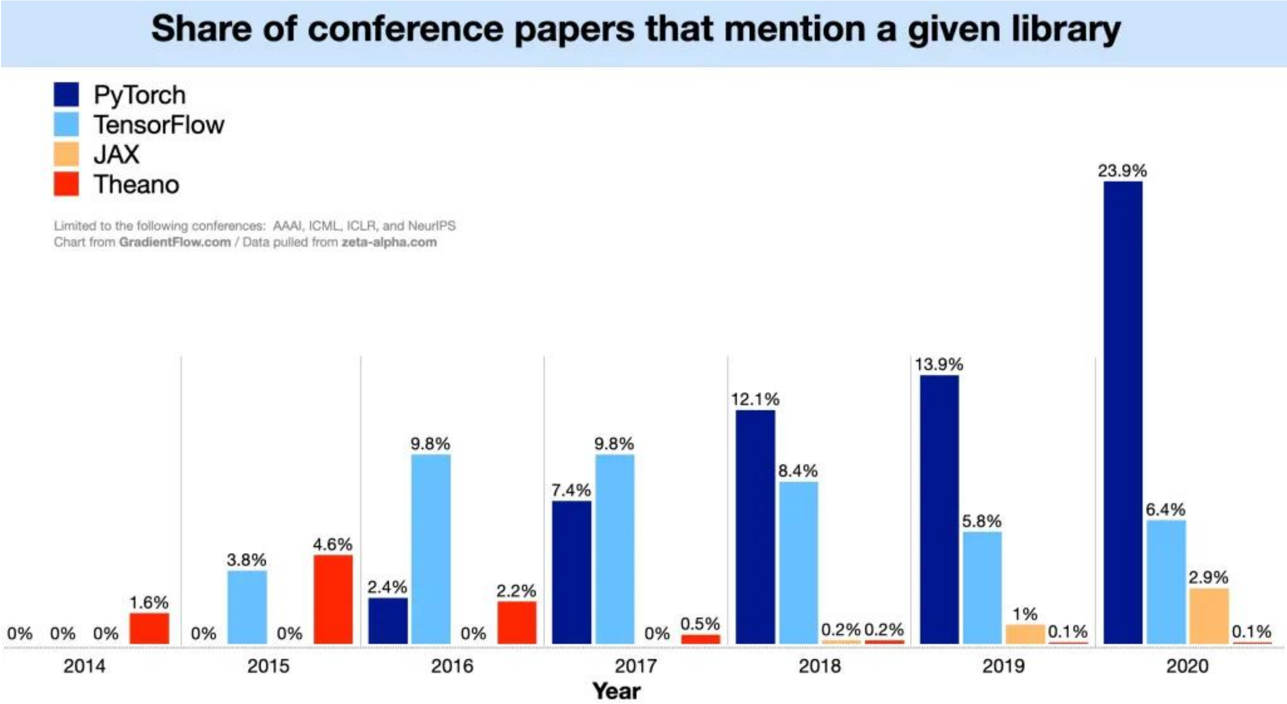
Unique Job Posts in select U.S. Tech Hubs that mention a given library

	Percent Change (Feb/2021 to Feb/2022)
MLflow	↑ 148%
scikit-learn	↑ 59%
PyTorch	↑ 52%
TensorFlow	↑ 31%

Limited to the following US technology hubs:
Atlanta; Austin (TX); Boston; Chicago; Los Angeles; NYC Area; Raleigh-Durham-Chapel Hill; SF Bay Area; Seattle; Washington, DC

Chart from **GradientFlow.com** / Data pulled from **indeed.com**

Time-series have been normalized to have the same starting value



Share of conference papers that mention a given library

Limited to the following conferences: AAAI, ICML, ICLR, and NeurIPS
Chart from **GradientFlow.com** / Data pulled from **zeta-alpha.com**

https://gradientflow.com/supercharging-your-data-and-ai-platforms

Convolutions

A convolution is a mathematical operation of two matrices:

3	0	1	2	7	4
1	5	8	9	3	1
2	7	2	5	1	3
0	1	3	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

*

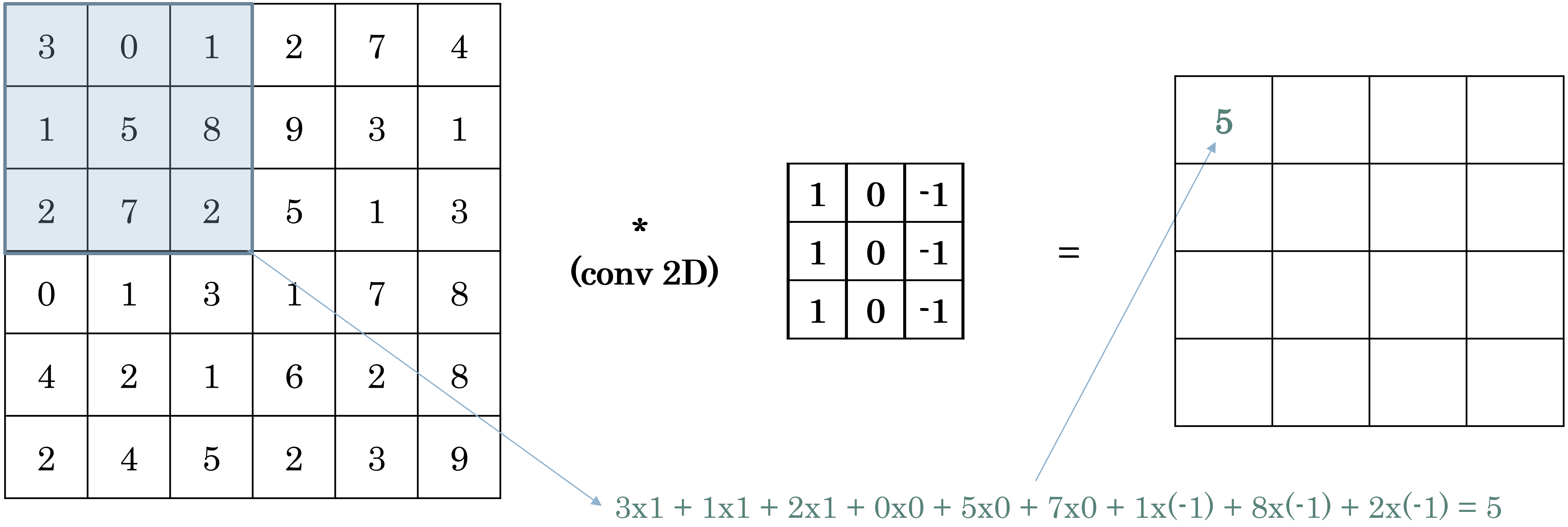
(conv 2D)

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

=

Convolutions

Overlays the 2nd matrix (filter) to the first one in each position and multiplies the elements in each position by adding the result of the multiplications



3	0	1	2	7	4
1	5	8	9	3	1
2	7	2	5	1	3
0	1	3	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

*****
(conv 2D)

1	**0**	**-1**
1	**0**	**-1**
1	**0**	**-1**

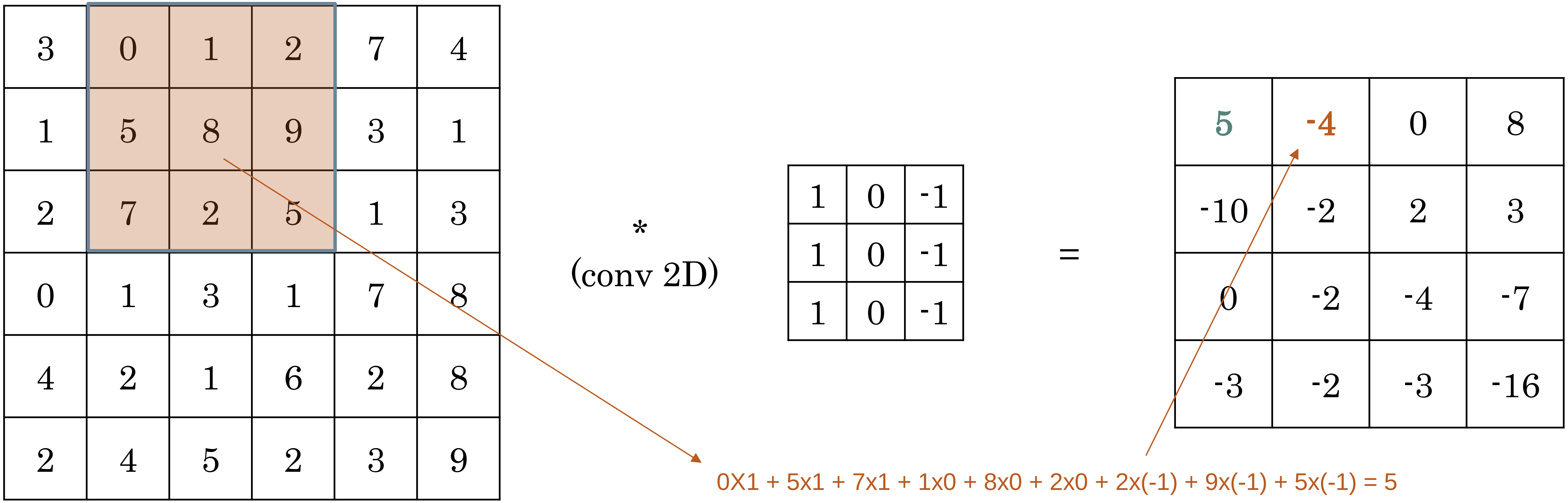
=

5			

3x1 + 1x1 + 2x1 + 0x0 + 5x0 + 7x0 + 1x(-1) + 8x(-1) + 2x(-1) = 5

Convolutions

Overlays the 2nd matrix (filter) to the first one in each position and multiplies the elements in each position by adding the result of the multiplications



3	0	1	2	7	4
1	5	8	9	3	1
2	7	2	5	1	3
0	1	3	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

*
(conv 2D)

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

=

5	-4	0	8
-10	-2	2	3
0	-2	-4	-7
-3	-2	-3	-16

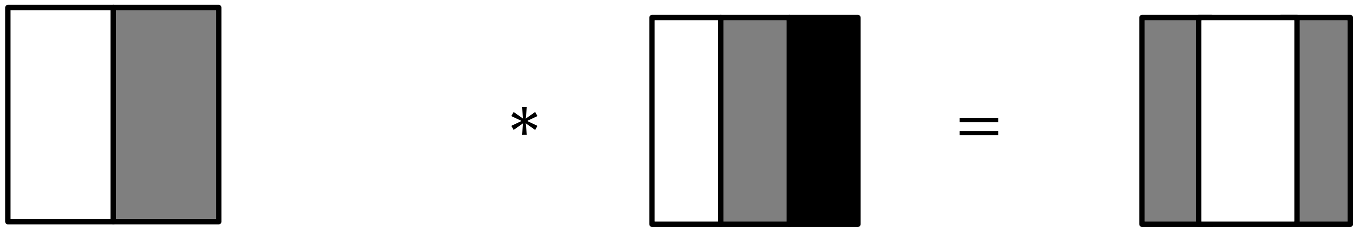
0X1 + 5x1 + 7x1 + 1x0 + 8x0 + 2x0 + 2x(-1) + 9x(-1) + 5x(-1) = 5

Convolutions as image filters

In image processing, convolutions applied to various areas of the image, act as filters that can detect certain patterns

Examples are convolutions that can act as vertical or horizontal “edge” detector

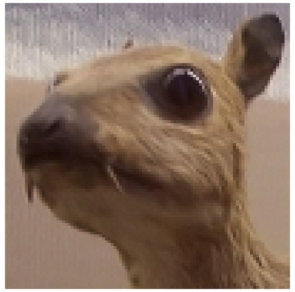
$$\begin{array}{|c|c|c|c|c|c|}\hline 10 & 10 & 10 & 0 & 0 & 0 \\ \hline 10 & 10 & 10 & 0 & 0 & 0 \\ \hline 10 & 10 & 10 & 0 & 0 & 0 \\ \hline 10 & 10 & 10 & 0 & 0 & 0 \\ \hline 10 & 10 & 10 & 0 & 0 & 0 \\ \hline 10 & 10 & 10 & 0 & 0 & 0 \\ \hline \end{array} * \begin{array}{|c|c|c|}\hline 1 & 0 & -1 \\ \hline 1 & 0 & -1 \\ \hline 1 & 0 & -1 \\ \hline \end{array} = \begin{array}{|c|c|c|c|}\hline 0 & 30 & 30 & 0 \\ \hline 0 & 30 & 30 & 0 \\ \hline 0 & 30 & 30 & 0 \\ \hline 0 & 30 & 30 & 0 \\ \hline \end{array}$$



Exemple by Andrew Ng

Convolutions as image filters

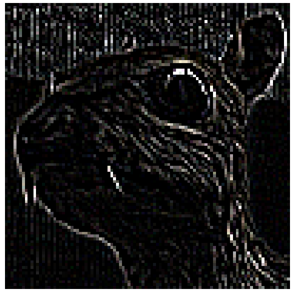
Input image



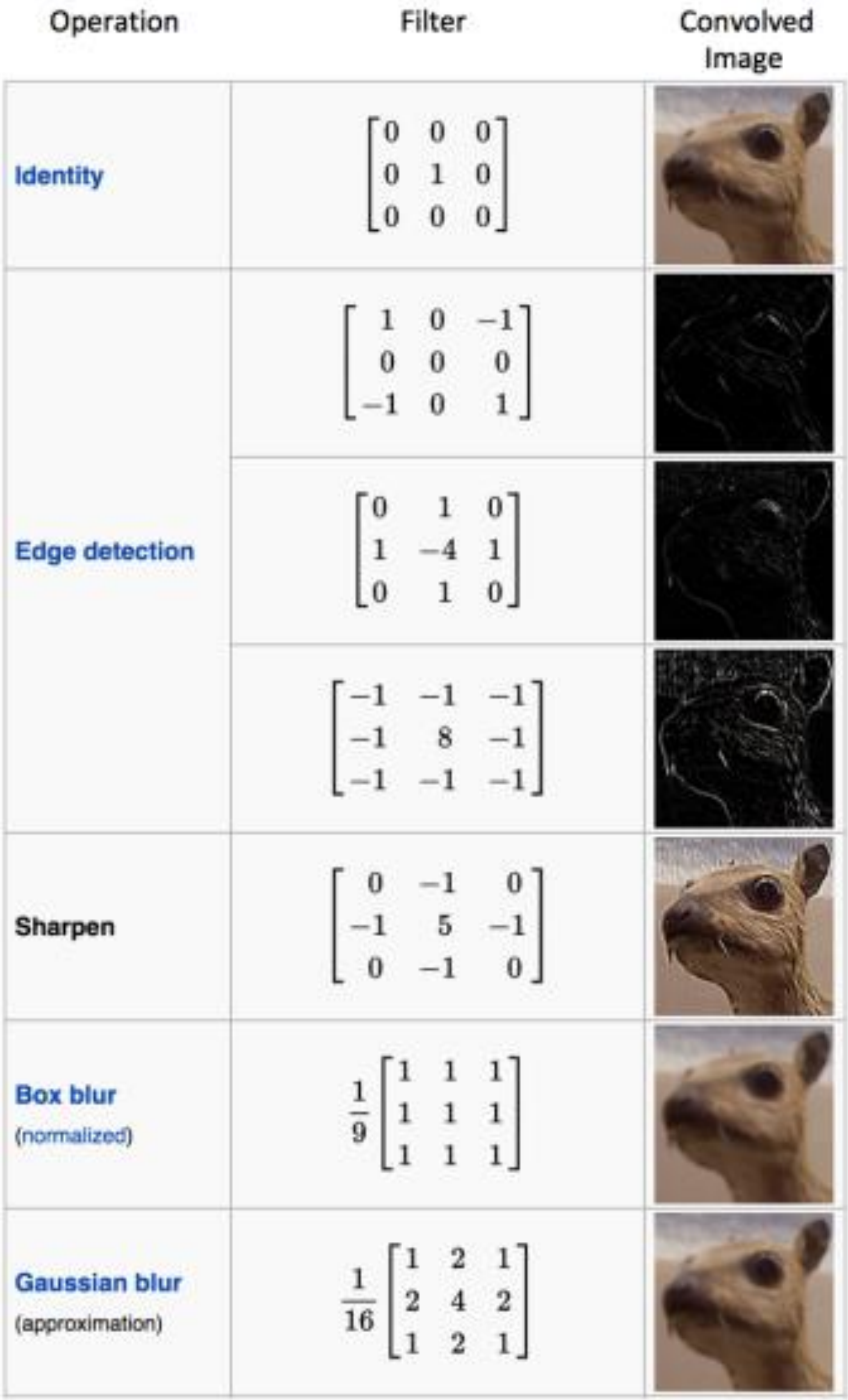
Convolution Kernel

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Feature map



In practice, **CNN** learn the values of these filters by itself during the training process. Although it is necessary to specify parameters such as “number of filters”, “filter size”, “network architecture”, etc. before starting the training process. The more filters we have, the more features of the image are extracted and the better the network becomes in recognizing patterns in the image.



Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Convolutional Neural Networks

CNN (Convolutional Neural Networks) are *deep learning* models in which the various layers of the network differ from the fully connected ones, consisting of **convolutional** and “**pooling**” layers that receive inputs and generate outputs that are 3D tensors for each example.

These are models tipically applied on images where 3D tensors represent height, width and depth (or “channel”). In the initial layer, these can be dimensions of an image (e.g. RGB will have 3 channels); in subsequent layers, each channel acts as a filter on the previous layer.

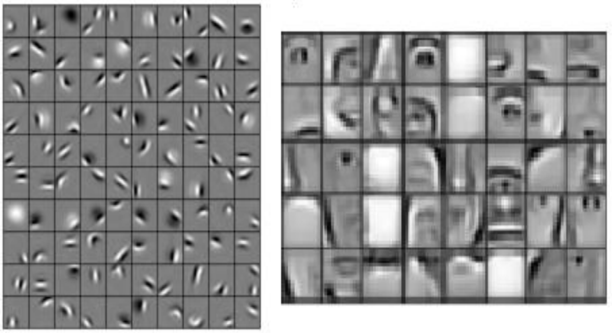
The introduction of these models represented a significant improvement in the way images are processed and classified.

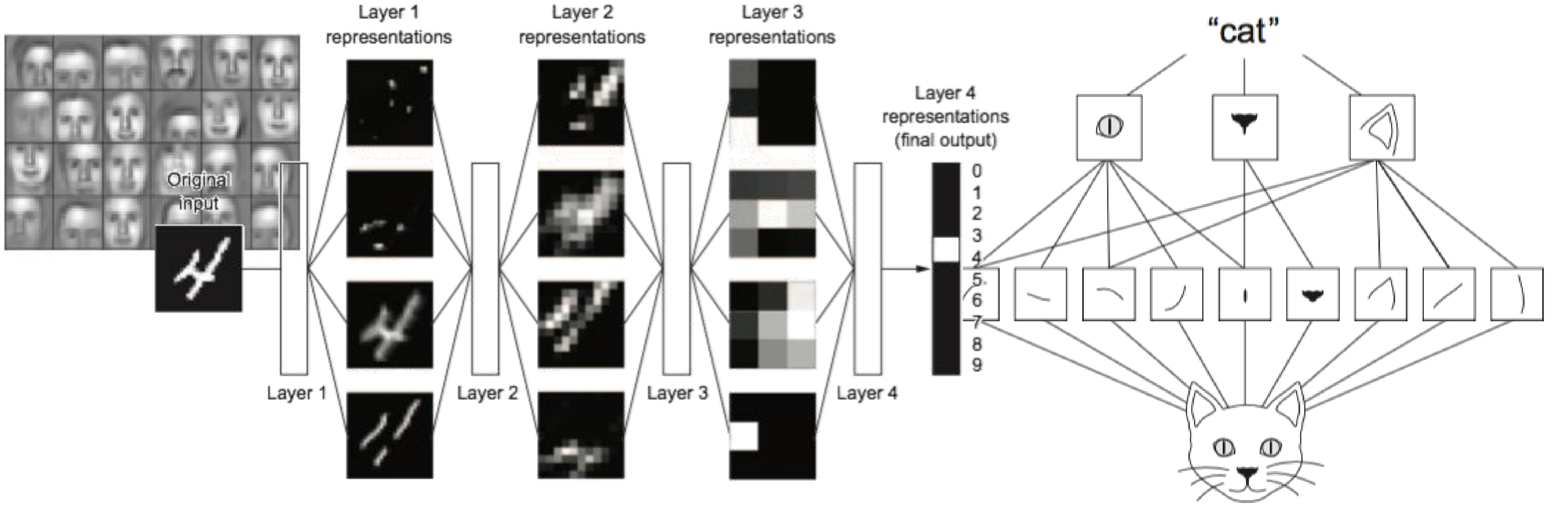
Learning in convolutional neural networks

Data processing in convolutional layers work on image regions, i.e. a matrix of small pixels (e.g. 3x3)

Allows learning patterns that can be recognized in various parts of the matrix (e.g. image) – invariant to translations

Allow learning in hierarchical way, with successive layers learning more complex and abstract concepts





CONVOLUTIONAL NEURAL NETS

Convolution

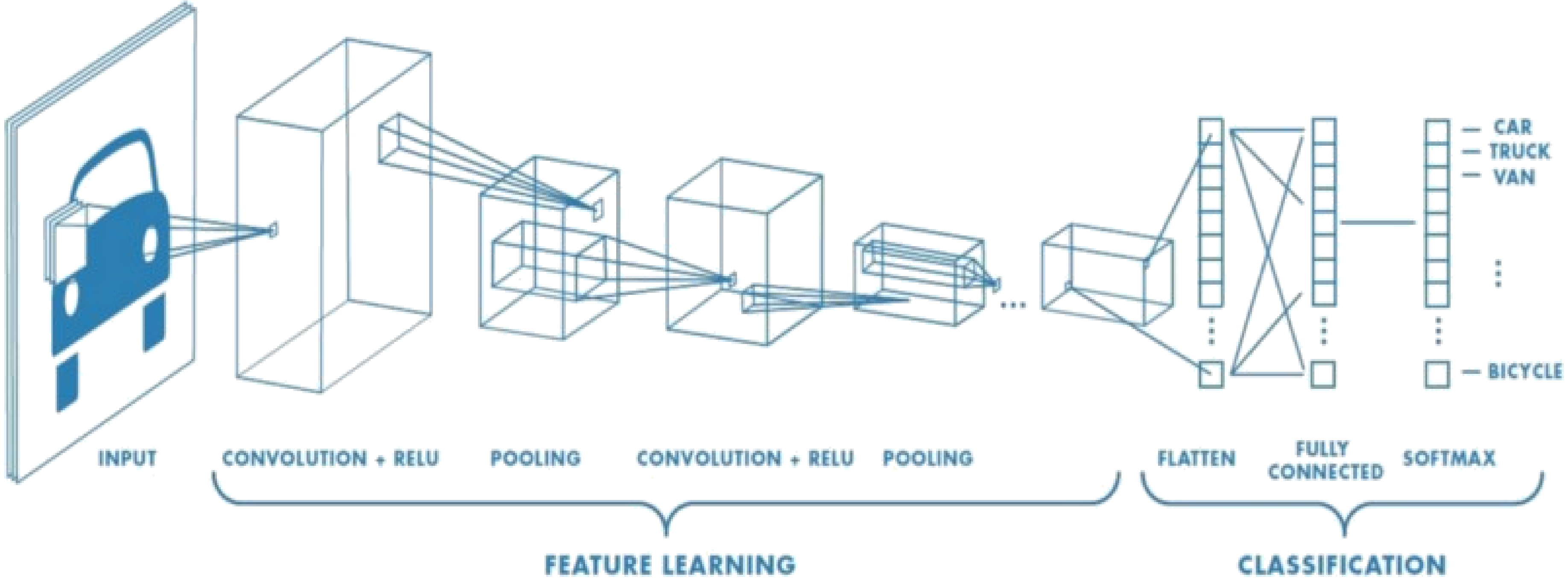
Convolution can be seen as a sliding window on top of a matrix representing an image. It is thus able to mimic the functioning of the biological visual cortex.

Convolutional Neural Networks (ConvNets or CNNs) are a category of neural networks that have proven to be very efficient in areas such as computer vision. ConvNets have been successful in applications such as:

- Image Classification – Search Engines, Social Media, Recommender Systems;
- Face Recognition Applications include Social Media, Identification, and Surveillance;
- Medical Image Computing – Predictive Analytics, Healthcare Data Science

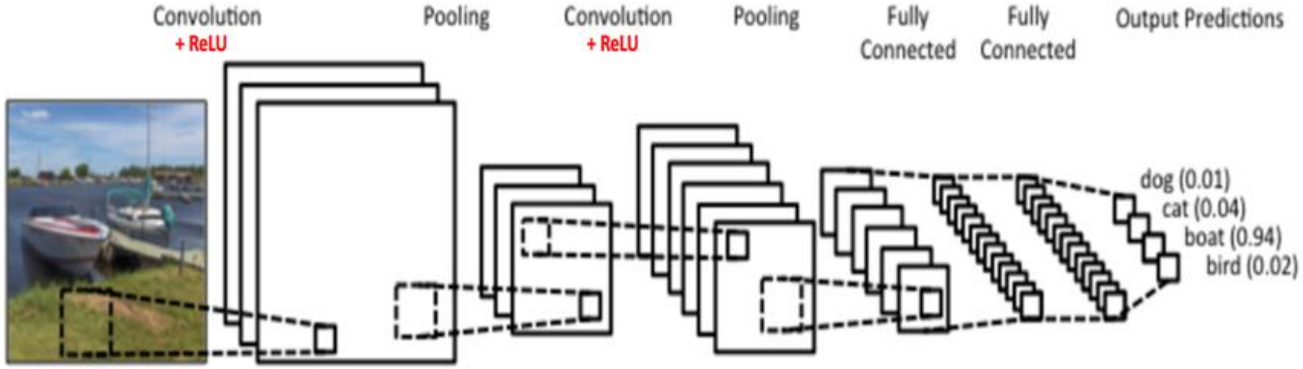
CNN ARCHITECTURE

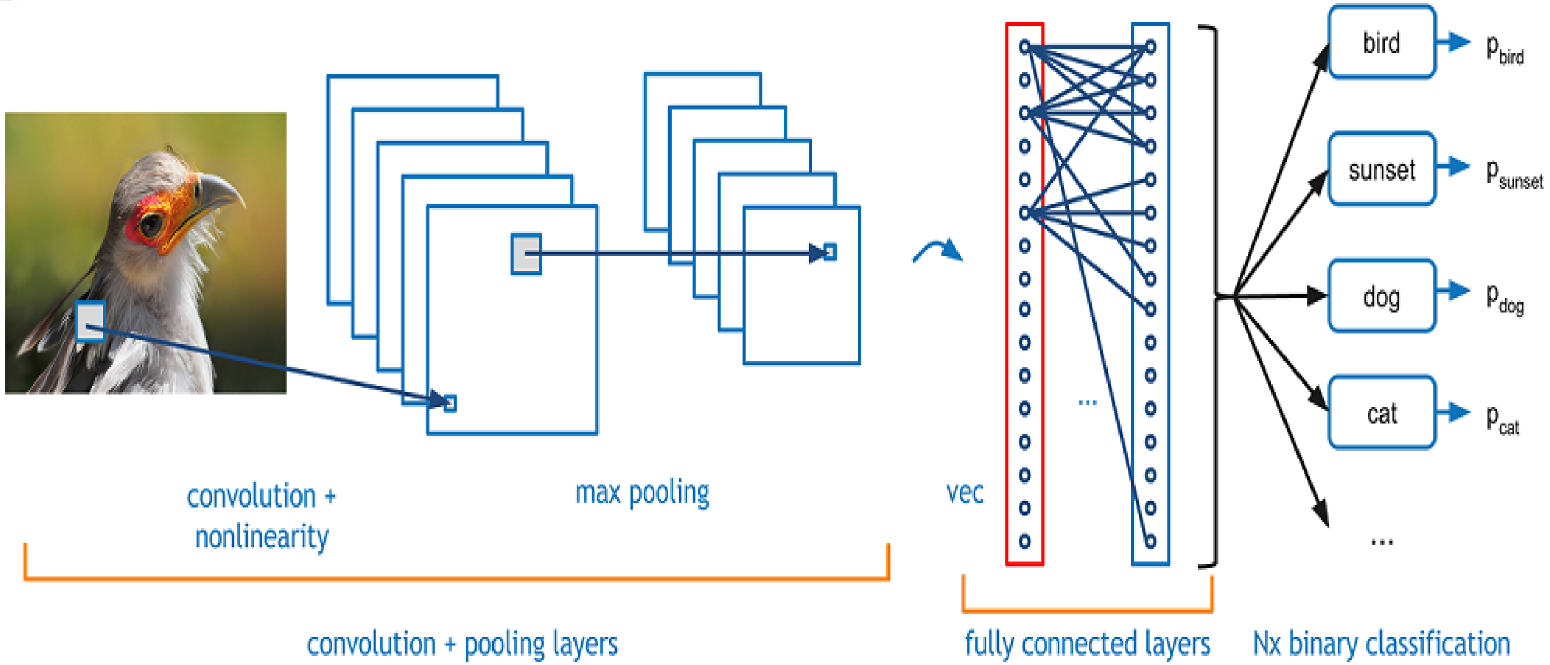
In a **CNN** we generally have the following entities: **Input** , **Filters** (or Kernels),**Convolutional Layer**, **Activation Layer**, **Pooling Layer, Batch Normalization layer** and **Fully Connected Layer**. Combining these layers in different permutations and some parameterisation gives us different deep learning architectures.



https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148

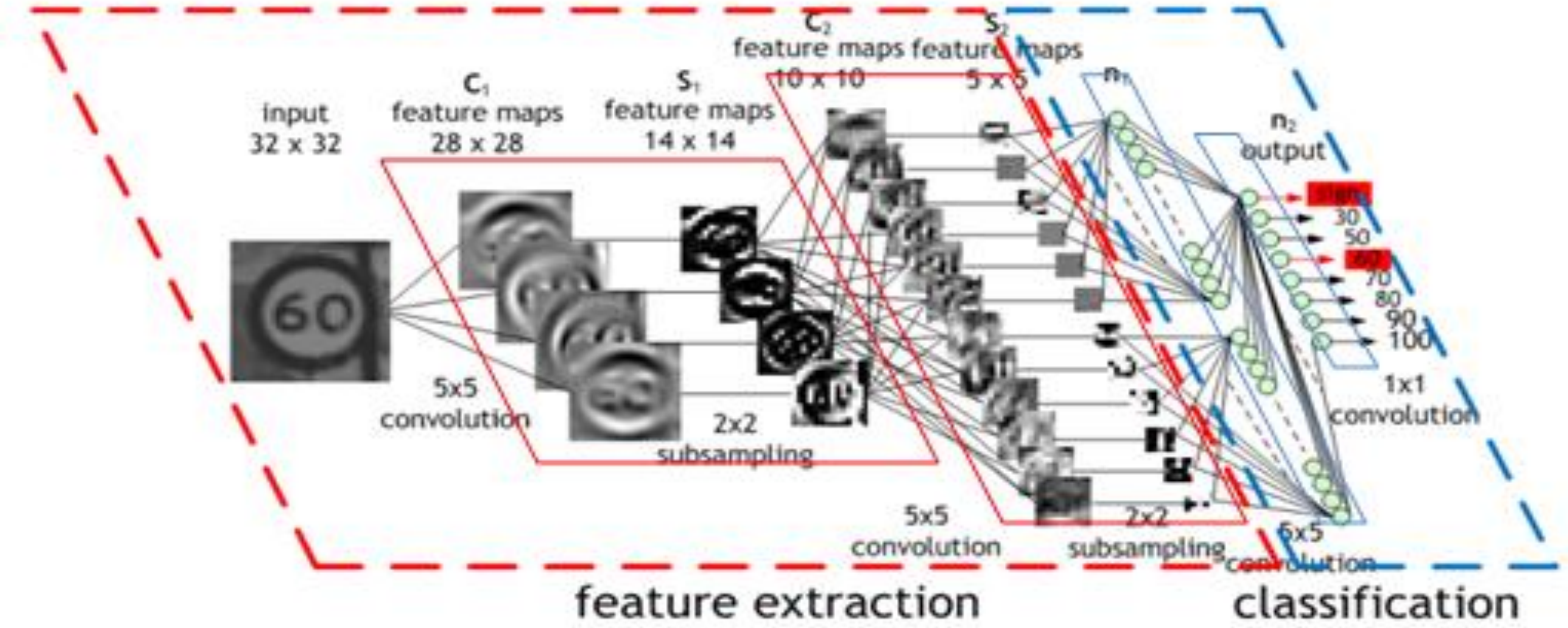
CNN ARCHITECTURE





CNN ARCHITECTURE

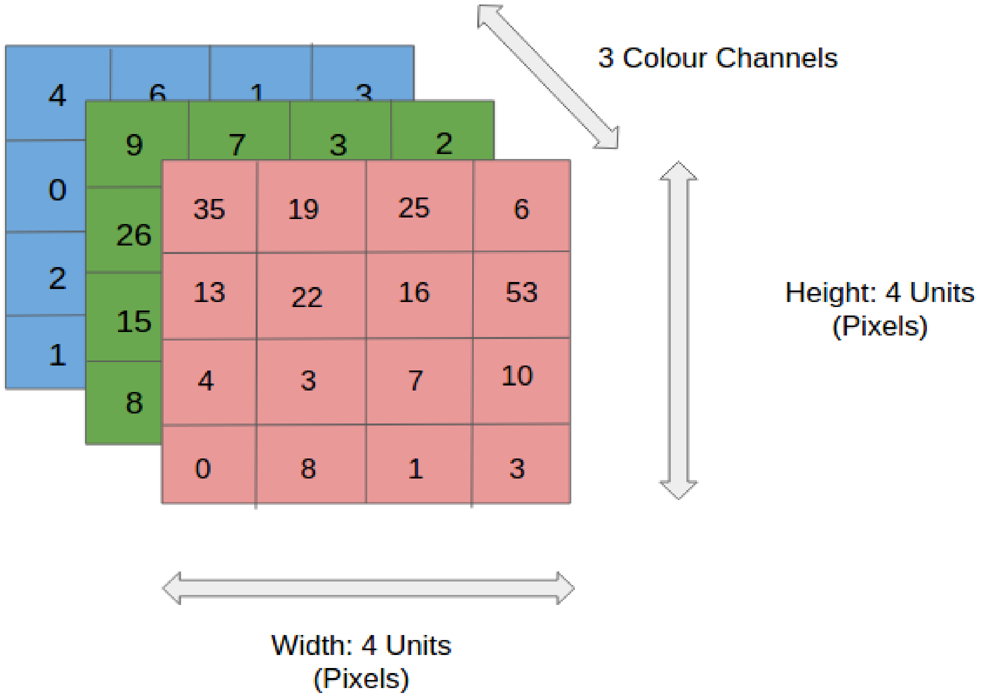
In the figure, the image is filtered by 4 5×5 **convolutional kernels** that give rise to 4 **feature maps**, these **feature maps** are reduced by **max pooling**. The next layer applies 12 5×5 **convolutional kernels** to these images and makes a new reduction. The final layer is **fully connected** where all **generated features** are combined and used for classification (**logistic regression**).

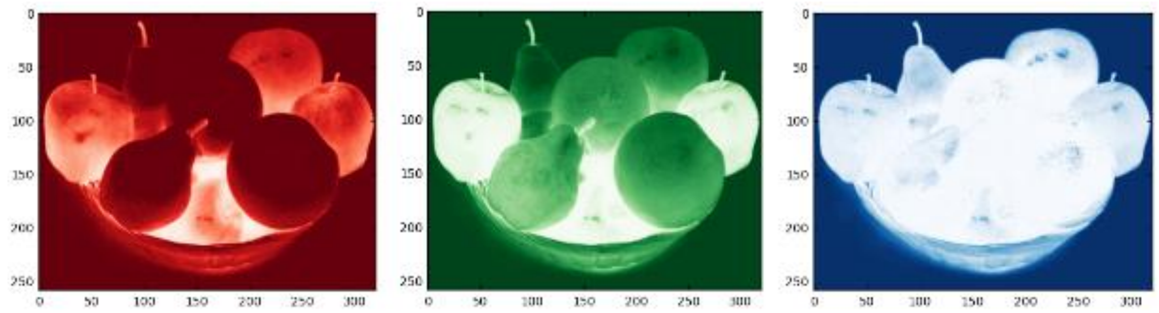


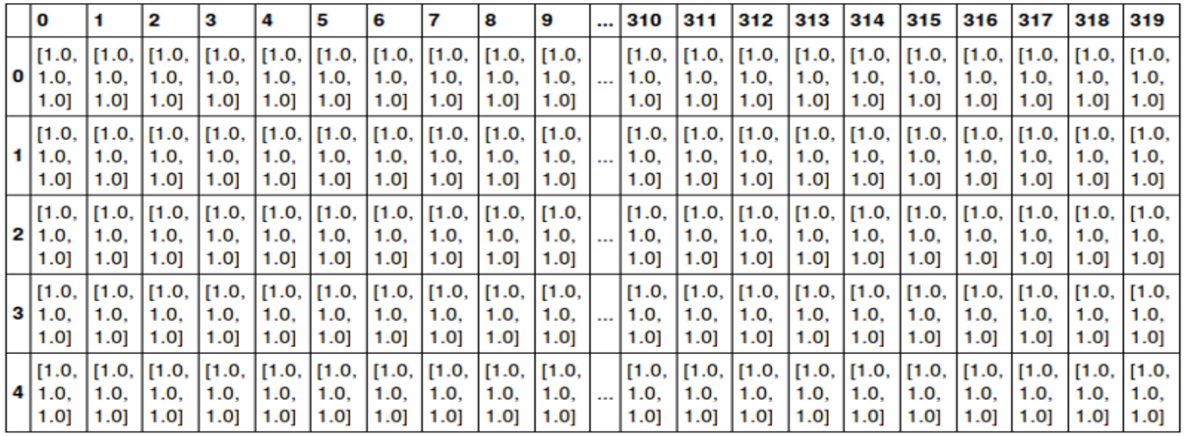
INPUT LAYER - TENSOR

Input Layer

The usual input for a CNN is an n-dimensional matrix i.e. a tensor. In the case of a colour image we generally have 3 dimensions - *width*, *height* and colour channels (*depth*).



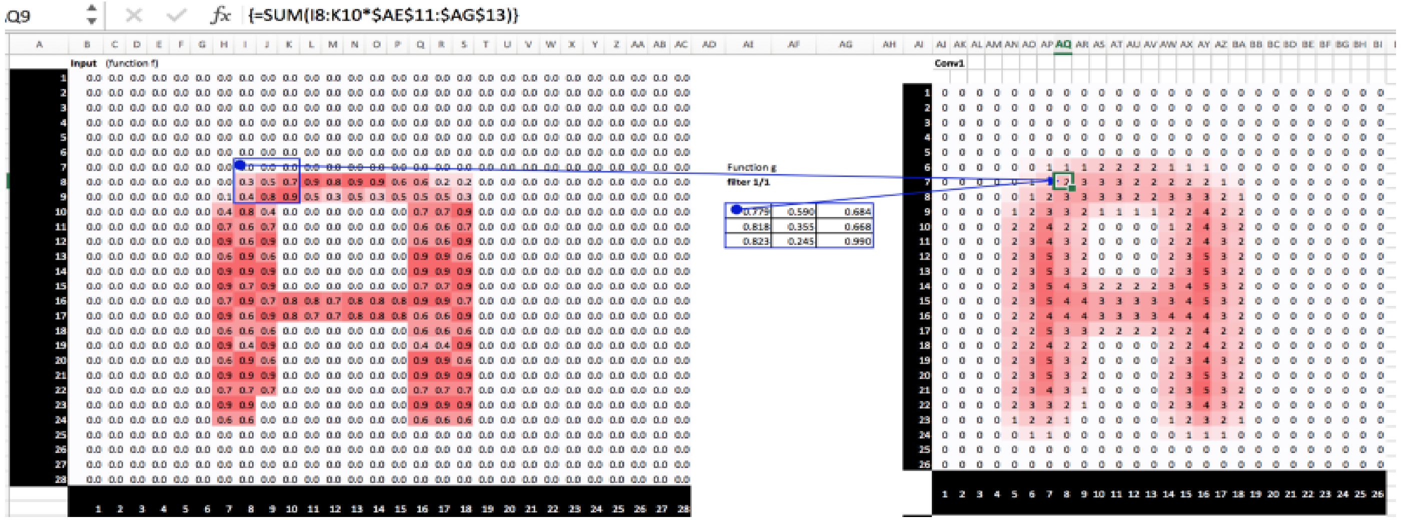




	0	1	2	3	4	5	6	7	8	9	...	310	311	312	313	314	315	316	317	318	319
0	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	...	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]
1	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	...	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]
2	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	...	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]
3	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	...	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]
4	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	...	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]

CONVOLUTIONAL LAYER

Convolutional Layer
This is the layer resulting from the scalar product of the input tensor by the filter (called **kernel**, **Patch Filters** or **Convolution Kernels**). The resulting matrices are usually called **Feature Map**, **Convolved Feature** or **convolutional matrix**.



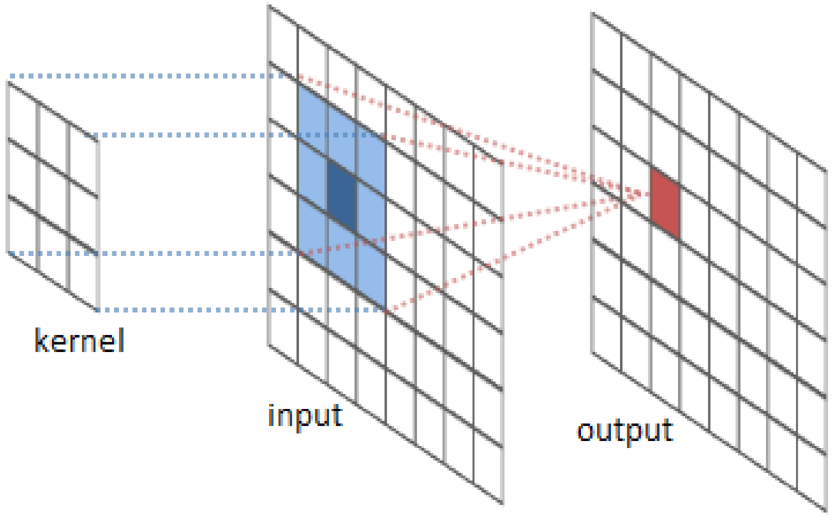
```
Q9    fx  {=SUM(I8:K10*$AE$11:$AG$13)}
```

Input (function f)

Function g
filter 1/1

0.779	0.590	0.684
0.818	0.355	0.668
0.823	0.245	0.990

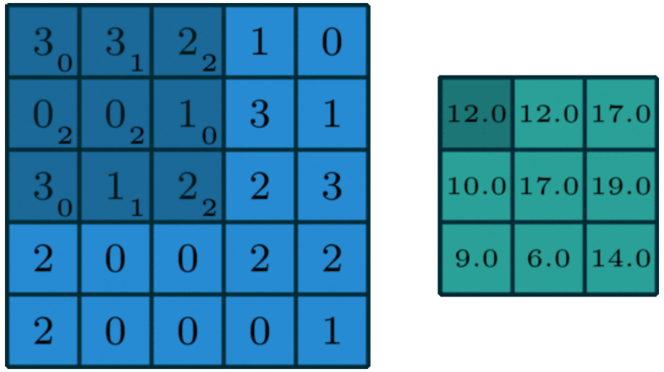
Conv1



Convolutional Layer - Kernels

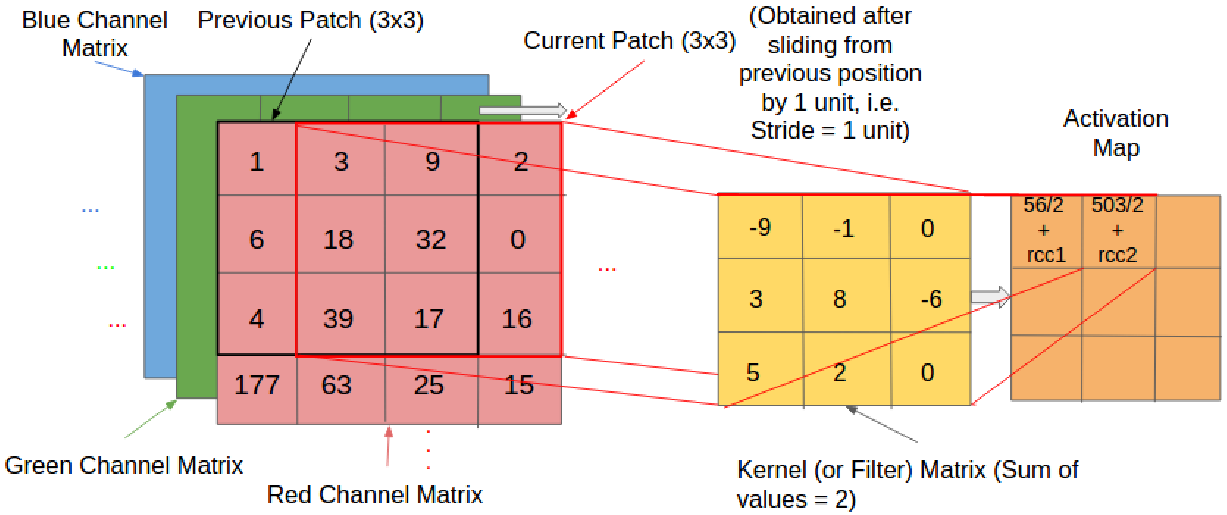
Filters or **Kernels**

A filter is applied by sliding to all the positions of a tensor giving as a result a new value (weighted sum) of the values to which it is applied.



3_0	3_1	2_2	1	0
0_2	0_2	1_0	3	1
3_0	1_1	2_2	2	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0



Blue Channel Matrix
Previous Patch (3x3)
Current Patch (3x3)
(Obtained after sliding from previous position by 1 unit, i.e. Stride = 1 unit)
Activation Map
...
...
...
...
Green Channel Matrix
Red Channel Matrix
Kernel (or Filter) Matrix (Sum of values = 2)

1	3	9	2
6	18	32	0
4	39	17	16
177	63	25	15

-9	-1	0
3	8	-6
5	2	0

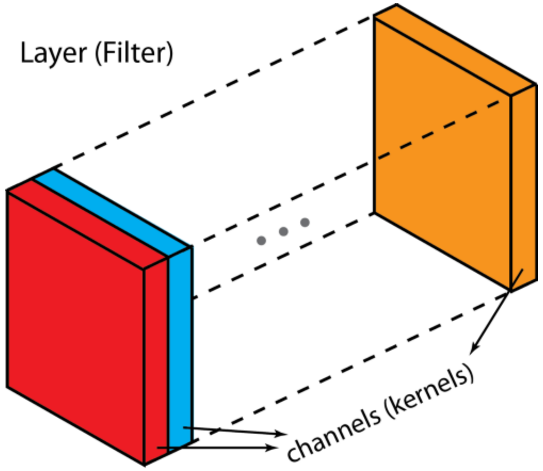
56/2 + rcc1	503/2 + rcc2	

Convolutional Layer - Concepts

layer – can have multiple **channels** (**feature maps**).
RGB Images – is an **input layer** with **3 channels**.
Channel (**feature maps**) – describes the structure of a layer.

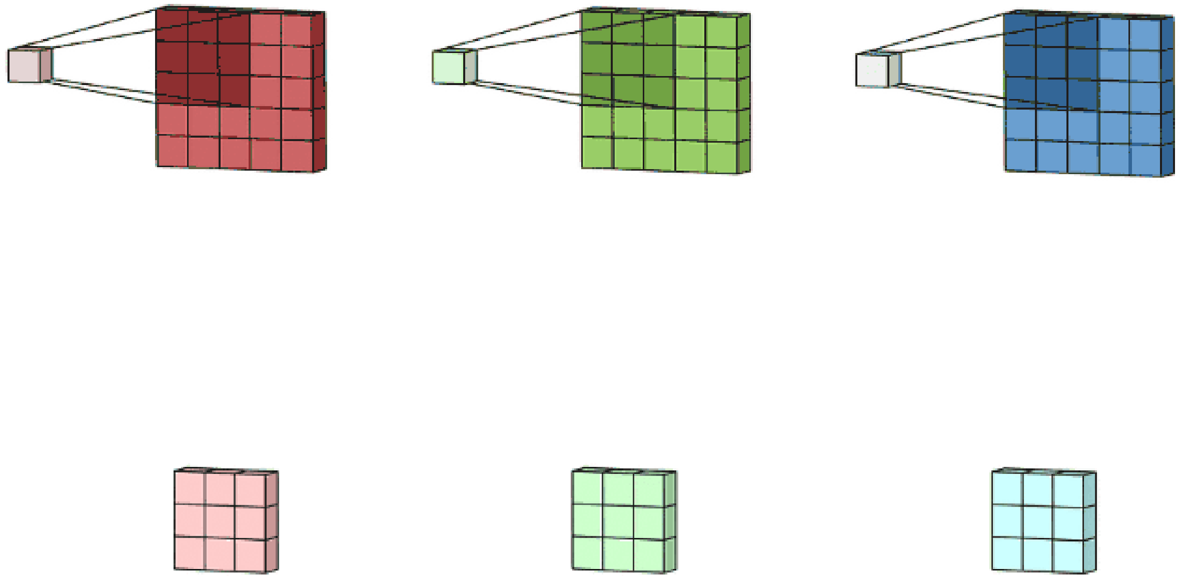
Kernel – describes the structure of a **filter** (receptive field = kernel size).
Kernel – is not quite the same thing as **filter.**
Kernel – refers to a 2D matrix of weights (learned via gradient descent)
each **kernel** is unique by highlighting different **features** of the input **channel**

Filter - refers to 3D structures of multiple **Kernels**.
Filter – if it is 2D then it is the same as **kernel**.
Filter – if it is 3D, which is the case for most DL convolutions, a **filter** is a collection of **kernels**



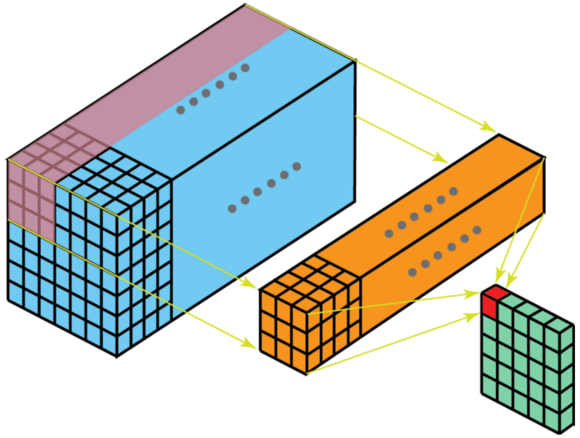
CONVOLUTIONAL LAYER - MULTICHANNEL(RGB)

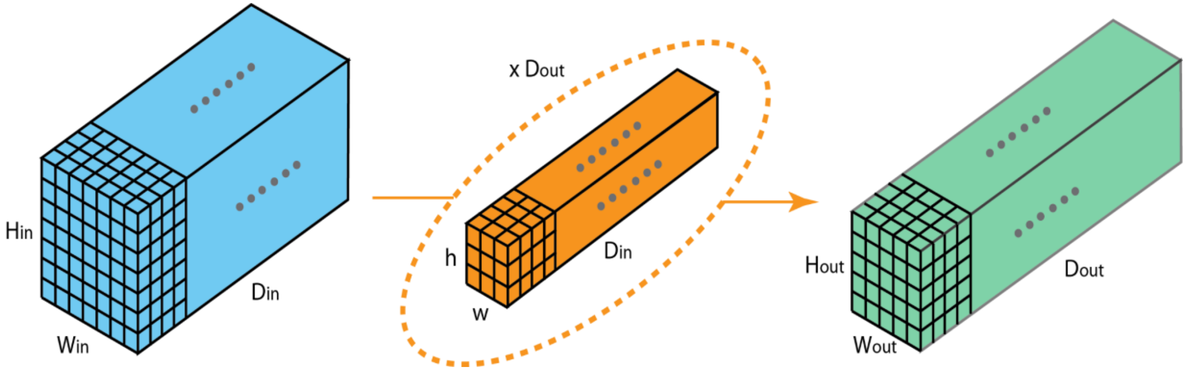
Example for an input **layer** of dimension 5x5 with 3 **channels** to which a 3x3 **filter** with 3 **kernels** is applied



Convolutional Layer - multichannel

In **multichannel** we have a 3D **filter** sliding across the input layer. The **input layer** and the **filter** have the same depth (**depth** = **channel number** = **kernel number**).
In **2D convolutions** the 3D filter moves in 2 directions only. In **3D convolutions** it moves in 3 directions.
In each position, the dot product and the sum is performed, which results in a single number, that is, in the end we obtain a **single output channel**.

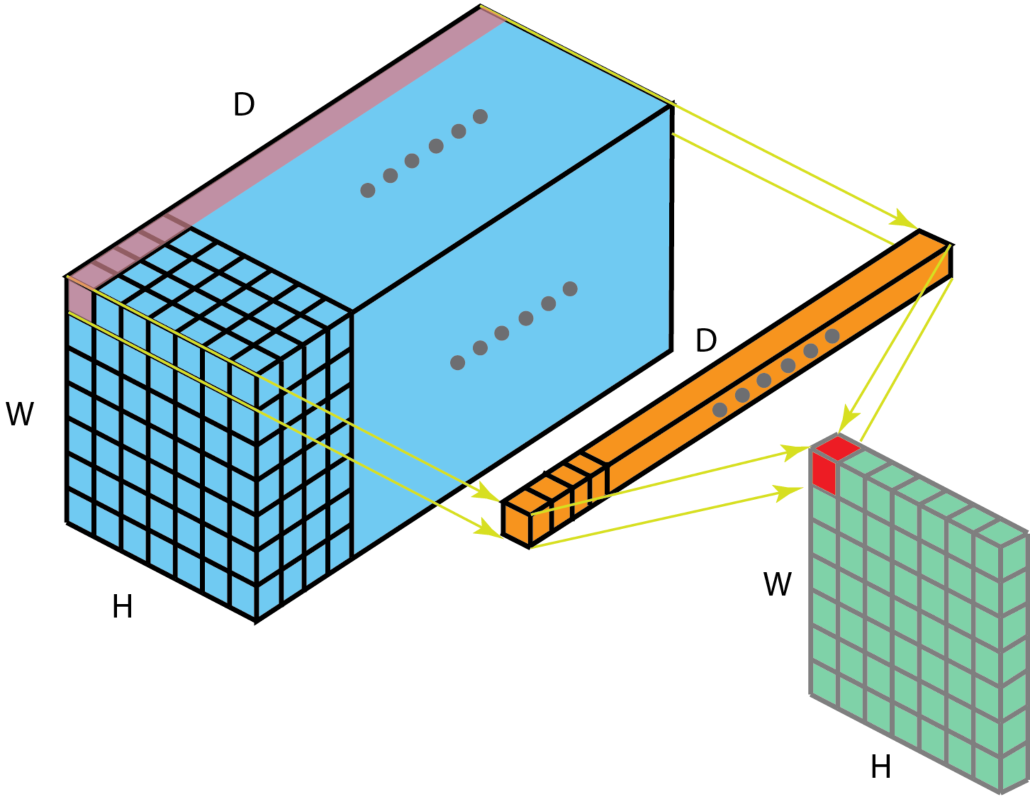




So we can make transitions between **layers** with different depths.
If we have an input layer with **D***in* channels, and we want to get an output layer with **D***out* channels, we just apply to the input layer, **D***out* filters.
Each **filter** has **D***in* kernels and will give one output channel.
After applying **D***out* **filters**, we get **D***out* **channels**, which are stacked to give the **output layer**.

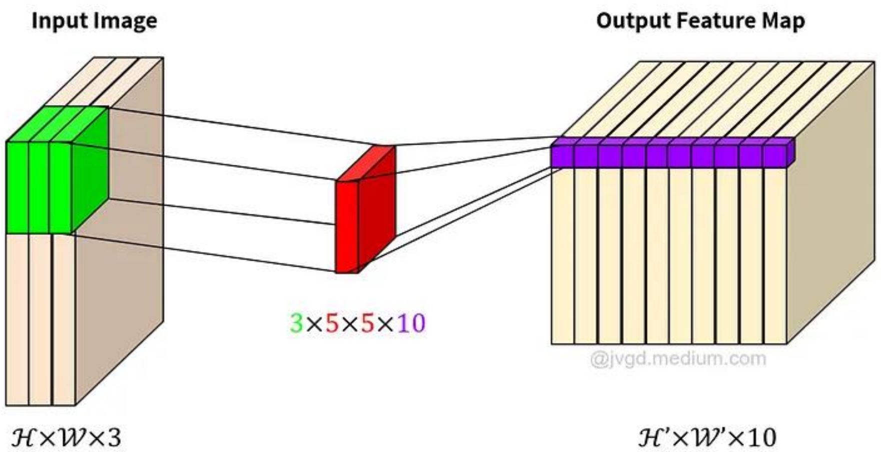
Convolutional Layer – 1x1

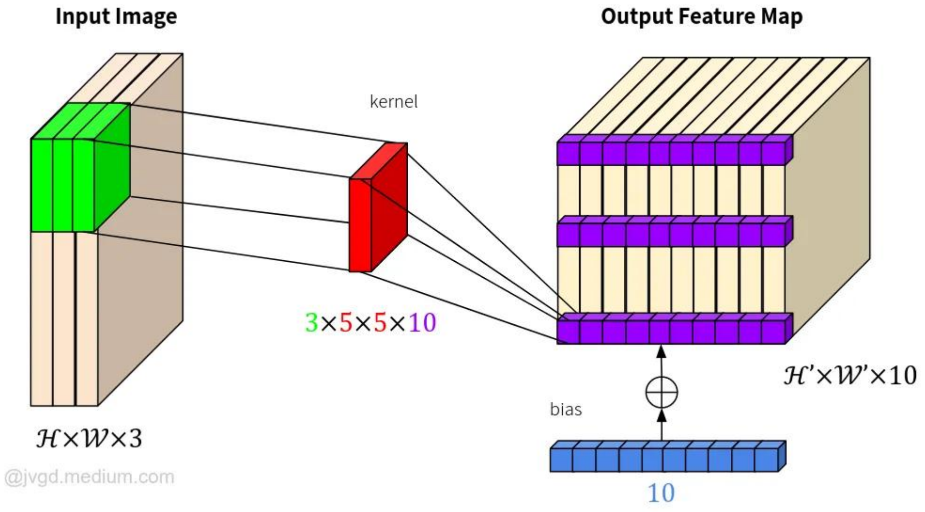
In cases with **layers** with a depth greater than one (**channels>1**) it allows a **dimensional reduction**. In the case of the figure, applying the *1x1xD* filter we obtain a channel with dimensions **HxWx1**. Applying ***N* filters** we get an **output layer** with dimension ***HxWxN***



Convolutional Layer – Torch conv2D

```
conv_layer = nn.Conv2d(in_channels=3, out_channels=10, kernel_size=5)

print(conv_layer.weight.shape)      # torch.Size([10, 3, 5, 5])

print(conv_layer.bias.shape)        # torch.Size([10])

print(sum(param.numel() for param in conv_layer.parameters()))

# 3x5x5x10+10 = 760
```


Input Image
Output Feature Map
3×5×5×10
@jvgd.medium.com
$\mathcal{H}\times\mathcal{W}\times3$
$\mathcal{H}'\times\mathcal{W}'\times10$

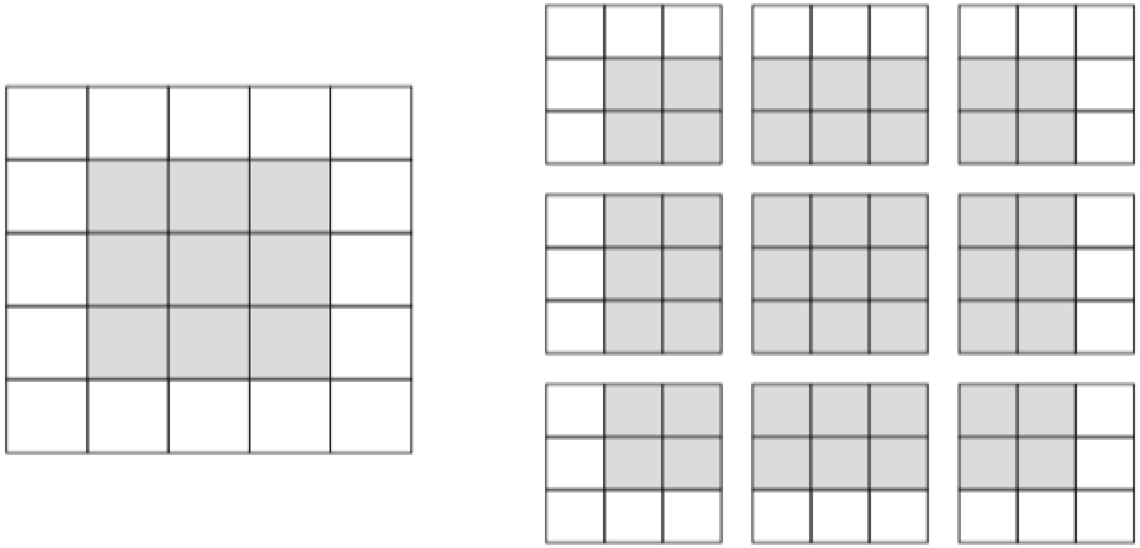

Input Image
Output Feature Map
kernel
3×5×5×10
$\mathcal{H}'\times\mathcal{W}'\times10$
bias
$\mathcal{H}\times\mathcal{W}\times3$
@jvgd.medium.com
10

https://towardsdatascience.com/pytorch-conv2d-weights-explained-ff7f68f652eb

Layer dimensions: padding and stride

Dimensions of convolutional layers (length, width) tend to decrease given valid window positions

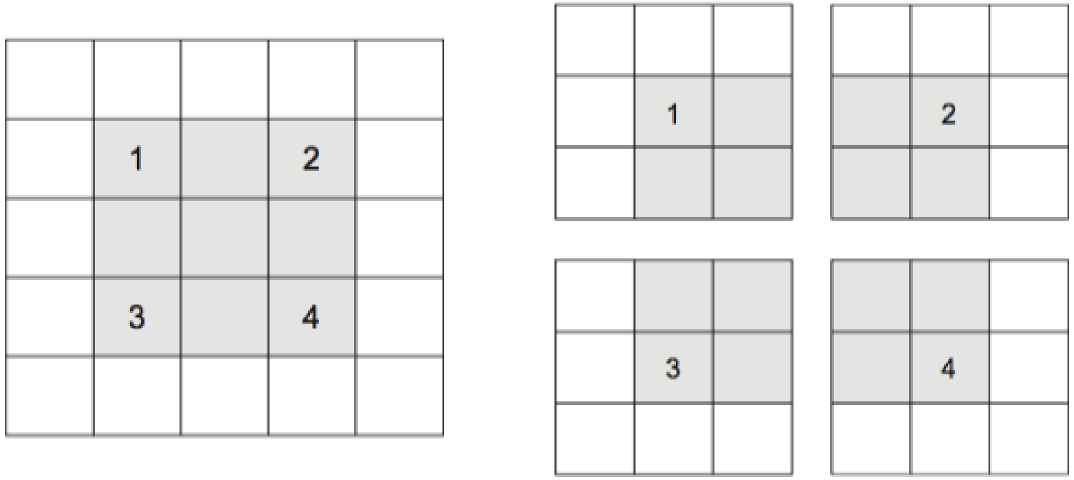
You can use **padding** to keep the same dimensions (consider extra rows and columns with values equal to zero)



The stride parameter controls the “intervals” between convolutions

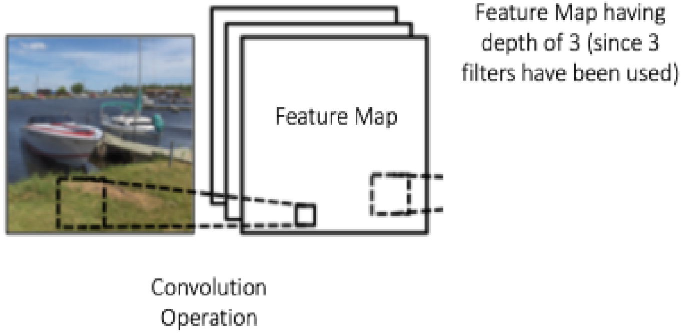
Example with stride = 2 ====>

Used by default in max pooling layers



CONVOLUTIONAL LAYER - HYPERPARAMETERS

Depth (hyperparameter)
Corresponde ao numero de filtros a utilizar na operação de convolução. No caso da figura estamos a utilizar 3 produzindo desse modo 3 **feature maps** distintos. Pode-se olhar para estes 3 **feature maps** como uma stack de matrizes 2D. A **depth** desta camada é portanto de 3.


Feature Map having depth of 3 (since 3 filters have been used)
Feature Map
Convolution Operation

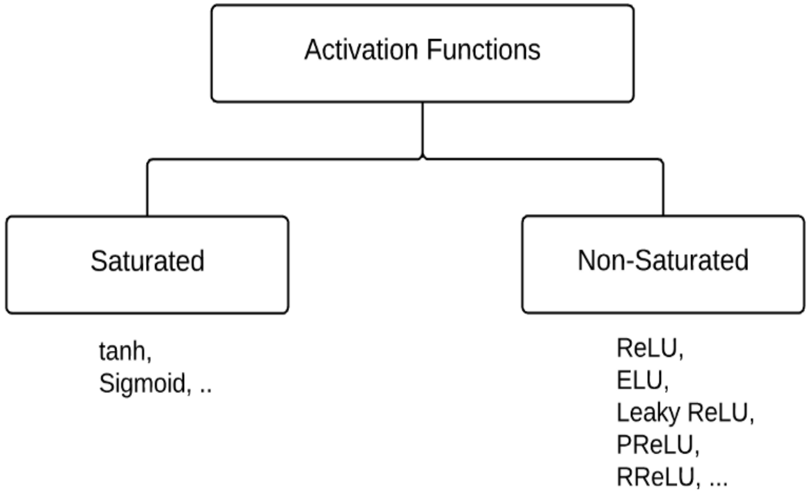
Kernal size (hyperparameter)
Corresponde ao **field of view** da convolução. Também chamado **Receptive Field**

Stride (hyperparameter)
É o número de pixéis que utilizamos em cada deslizamento do filtro sobre a matriz. Quando o **stride** é 1 então fazemos o deslizamento 1 pixel de cada vez. Quando é 2 o filtro salta 2 pixéis de cada vez. Em resultado, quanto maior for o **stride** menor é a dimensão do **feature map**.

Zero-padding (hyperparameter)
Por vezes é conveniente simular a matriz de entrada com zeros em volta de toda a borda. De modo a que ao aplicar os filtros a matriz resultante tenha a dimensão da matriz de entrada. Acrescentar zeros à volta da matriz de entrada também se chama **wide convolution**, e a sua não utilização chama-se **narrow convolution**.

Activation Layer

Activation Layer
As funções de activação podem ser classificadas em 2 categorias: Saturadas e Não-Saturadas



A vantagem da utilização de funções de activação não-saturadas são:

- ❑ Resolver o problema denominado “**exploding/vanishing gradient**”.
- ❑ Acelerar a velocidade de convergência.

Sigmoid - Recebe um valor real e produz um valor entre [0,1]: $\sigma(x) = 1 / (1 + \exp(-x))$

tanh - Recebe um valor real e produz um valor entre [-1, 1]: $\tanh(x) = 2\sigma(2x) - 1$

ReLU - **Re**ctified **L**inear **U**nit. **ReLU** coloca todos os valores negativos a zero e os positivos ficam como estão. **ReLU** é aplicado depois da convolução, tratando-se portanto de uma função de activação não-linear tal como **tanh** ou **sigmoid**.

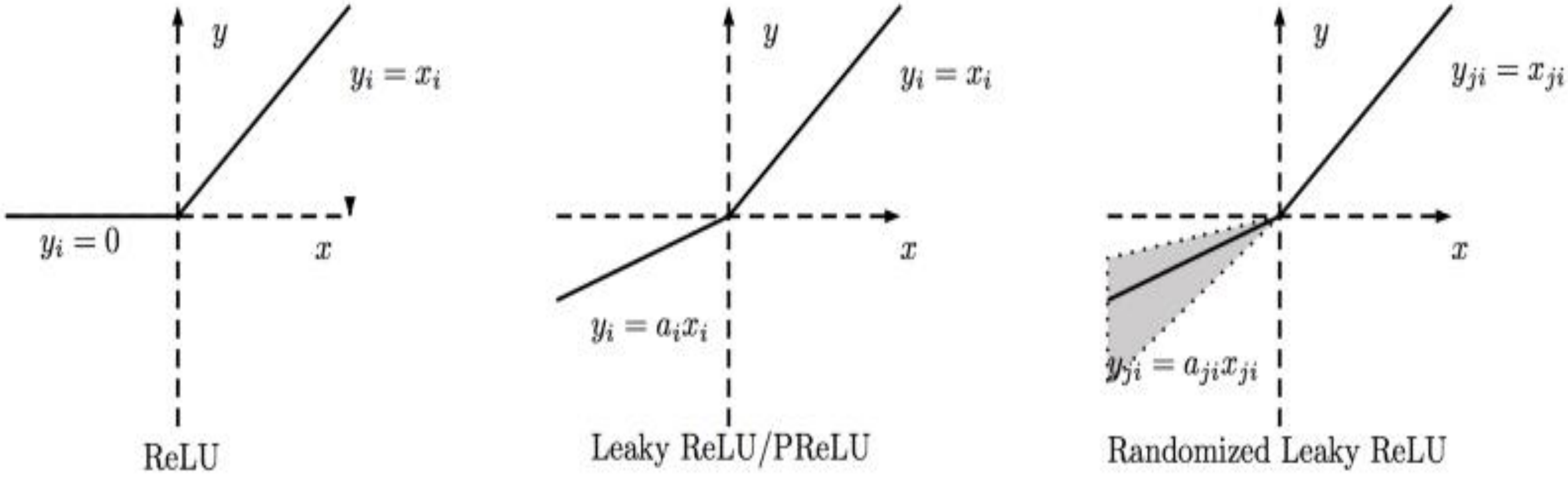
ELUs - **E**xponential **L**inear **U**nits tentam colocar a média das activações mais perto do zero o que acelera a aprendizagem. ELUs também evitam o **vanishing gradient** devido à identidade nos valores positivos. Está demonstrado que ELUs conseguem maior acurácia na classificação do que ReLUs.

Leaky ReLUs - Ao contrario da **ReLU**, em que a parte negativa passa a zero, a **leaky ReLU** atribui um valor.

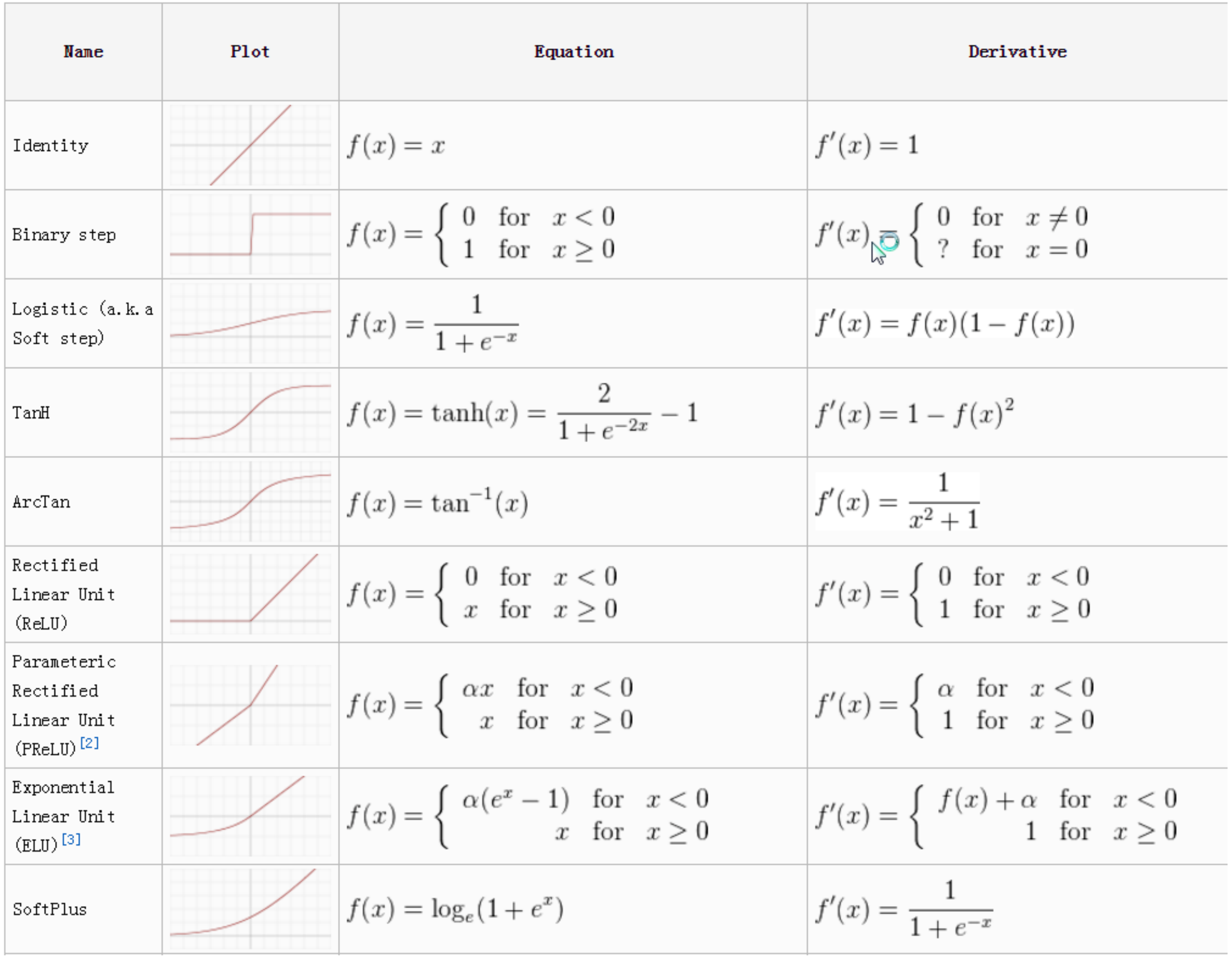
ACTIVATION LAYER

PReLU - Parametric **Re**ctified **L**inear **U**nit**.** É considerada uma variante à **Leaky ReLU**. Na **PReLU**, o declive da parte negativa é calculada a partir dos dados (através do backpropagation) em vez de ser pré-definida. Os autores defendem que foi o factor chave na ultrapassagem do desempenho dos humanos na classificação da ImageNet.

RLReLU - Randomized **L**eaky **Re**ctified **L**inear **U**nit**.** É também uma variante da **Leaky ReLU**. Na **RLReLU**, o declive da parte negativa é aleatorio numa parte do treino, e depois fixa no teste. A ideia é que no treino o declive **a**ij é um número aleatório retirado de uma distribuição uniforme.

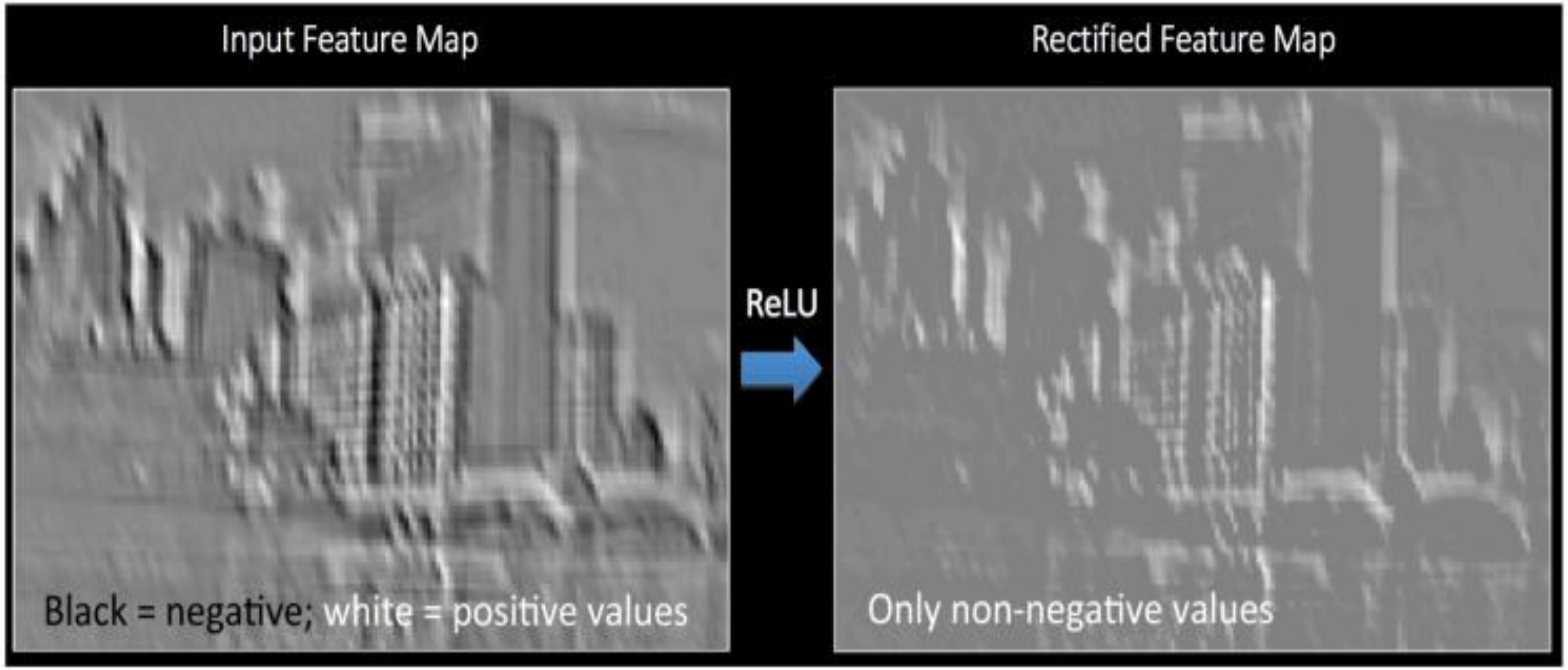


Activation Layer



Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU)[2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU)[3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

Activation Layer



Noisy Activation functions - Tratam-se de funções estendidas de modo a incluir ruido Gaussiano.

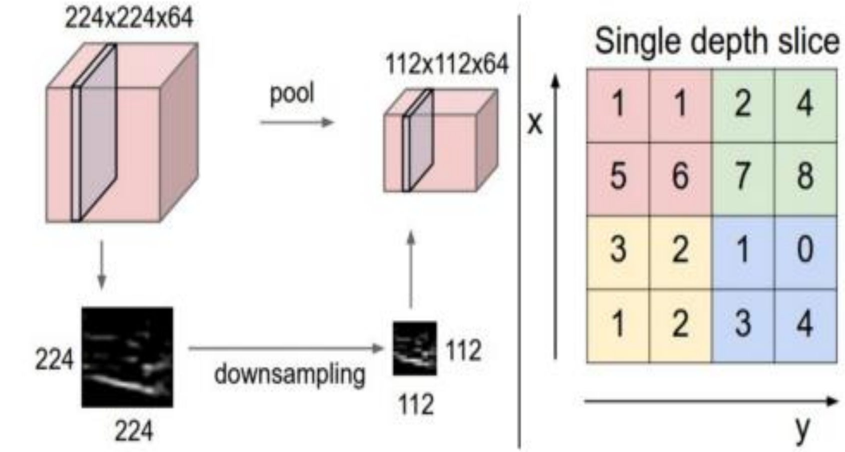
Pooling layers

The objective of the **pooling layer** is to progressively decrease the spatial dimension of the matrix in order to reduce the amount of parameters and computation volume in the network, also controlling overfitting. The Pooling Layer operates independently in each depth feature map, changing its size using operations like MAX, SUM or Average. That is:

- Used as a way to reduce the size of convolutional layers through a sampling process
- Extracts windows from the previous layer and calculates a single value for each window, typically considering the application of the **max** function (fit only takes the largest value in the window)
- Typical values for windows are 2 x 2
- These layers have no parameters to learn
- Pooling layers promote “hierarchical” learning

Pooling Layer

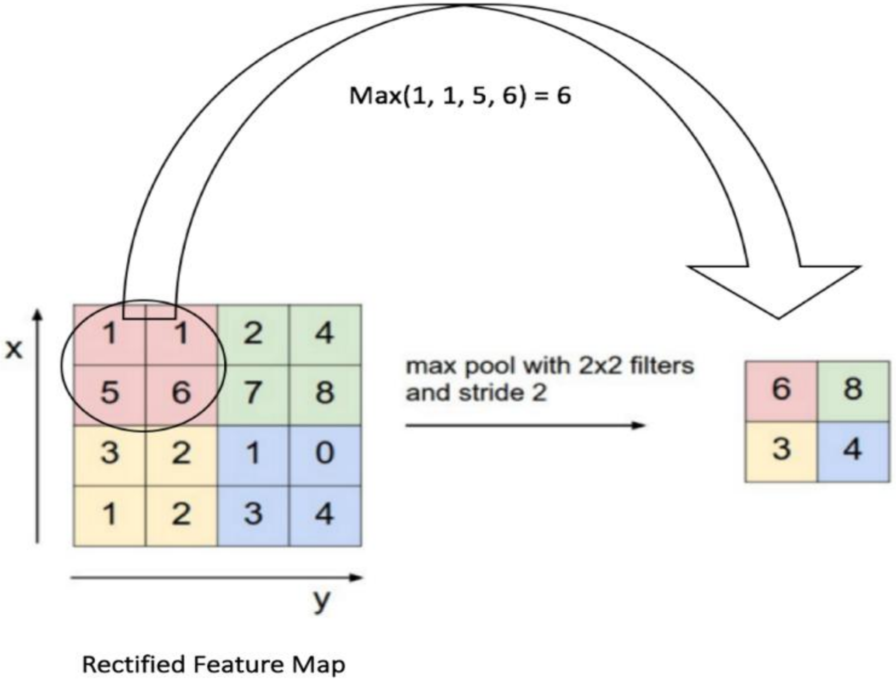
One of the most usual configurations for the **pooling layer** are 2x2 size filters applied with **stride** of 2 reducing each **feature map** to half in each dimension thus ignoring 75% of the activations. Each **MAX** operation will thus result in a maximum of 4 numbers. The depth dimension remains unchanged.



1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

6	8
3	4

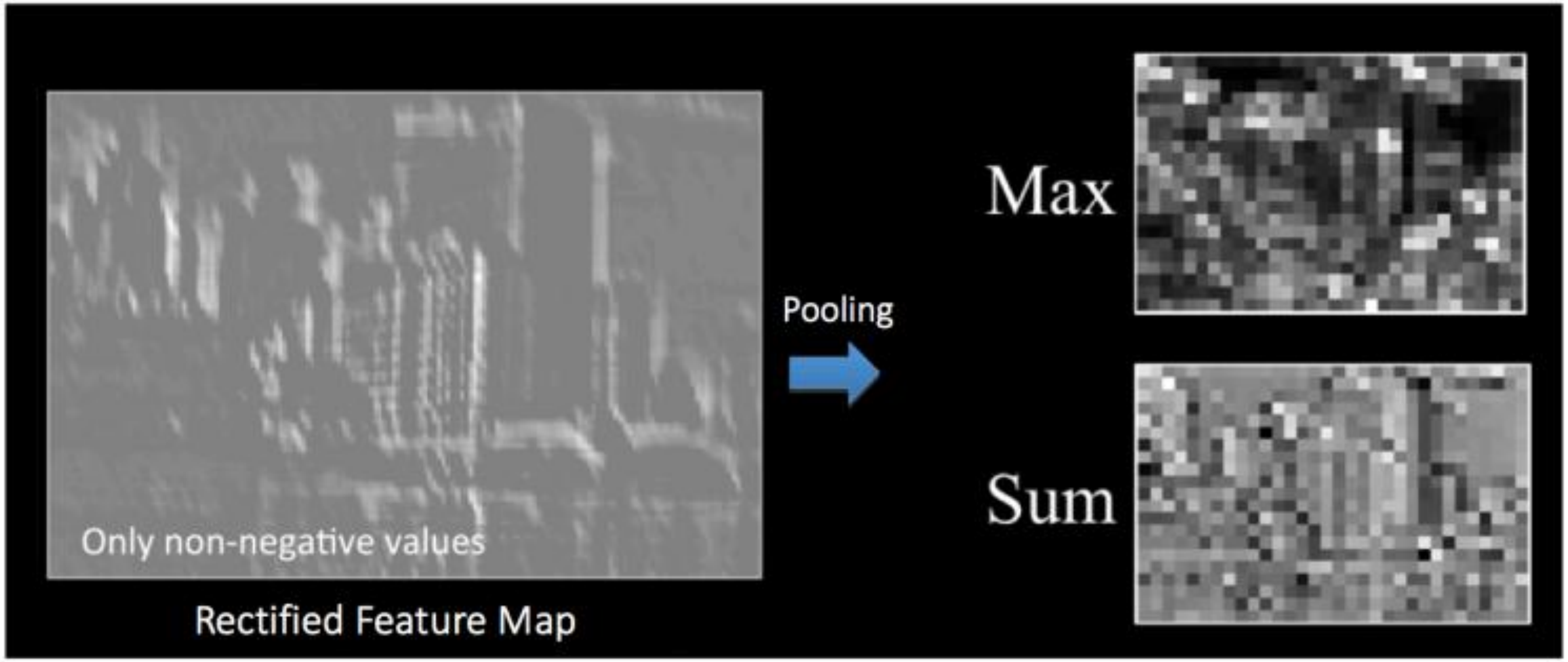
Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. **Left:** In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. **Right:** The most common downsampling operation is max, giving rise to **max pooling**, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).



1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

6	8
3	4

Pooling Layer



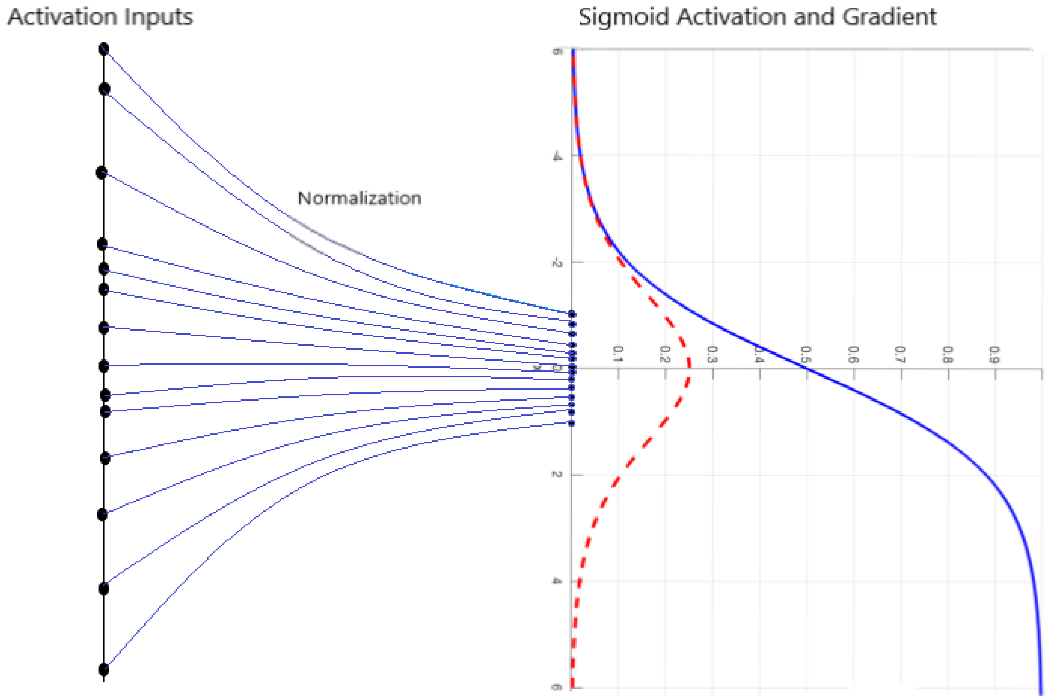
BATCH NORMALIZATION LAYER

Normalization is the process in which a data set is subtracted from each element the mean value of this set and the result is divided by the standard deviation of the data set. By doing this we put all values on the same scale. Generally with **images** we don't worry about dividing by the standard deviation, simply subtracting the mean.

Batch Normalization layer

This is an effective way to normalize each intermediate layer including weights and activation functions. There are 2 main advantages to using **batchnorm**:

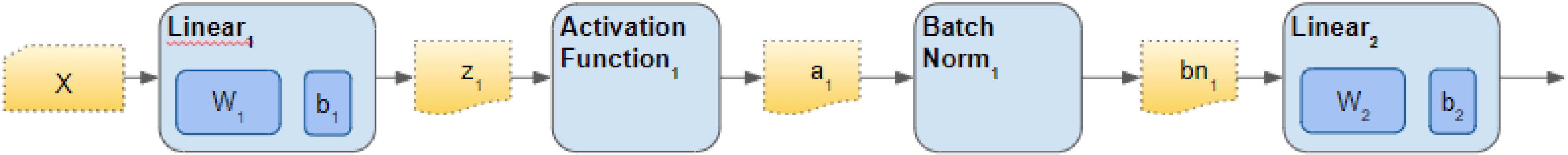
- Adding **batchnorm** to a model can result in **10x or more improvement in training speed** .
- As **normalization** drastically reduces the possibility that a small number of **outlying inputs** will exaggeratedly influence training, it also tends to reduce **overfitting**.
- By default, you should always include batch normalization



? BatchNorm Before or after Activation ?

Conv - BatchNorm - Activation – Pool

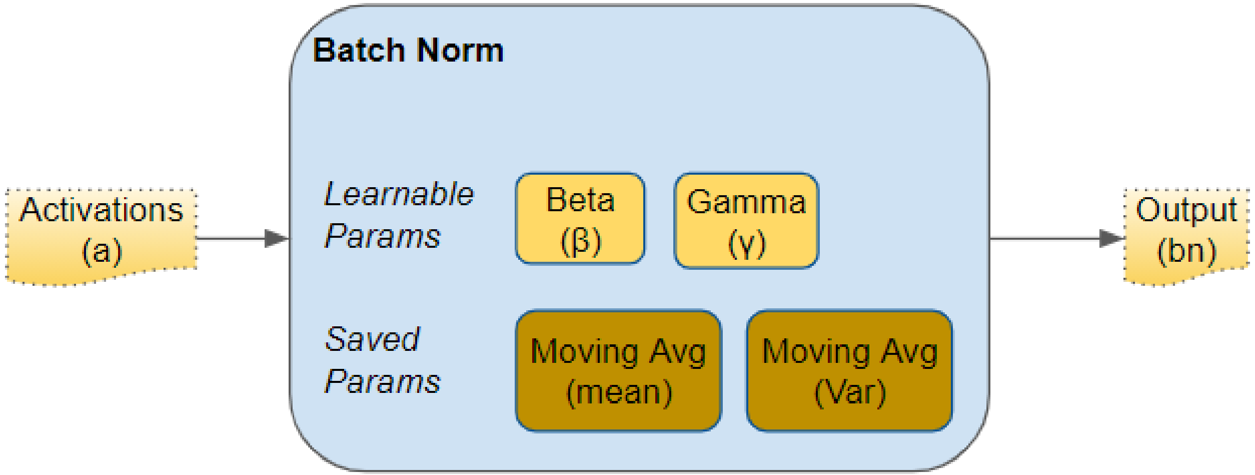
BATCH NORMALIZATION LAYER



Normalization won't prevent SGD from trying to create an imbalanced weight again and again during the next back-propagations.

Batch Norm extends normalization with two additions:

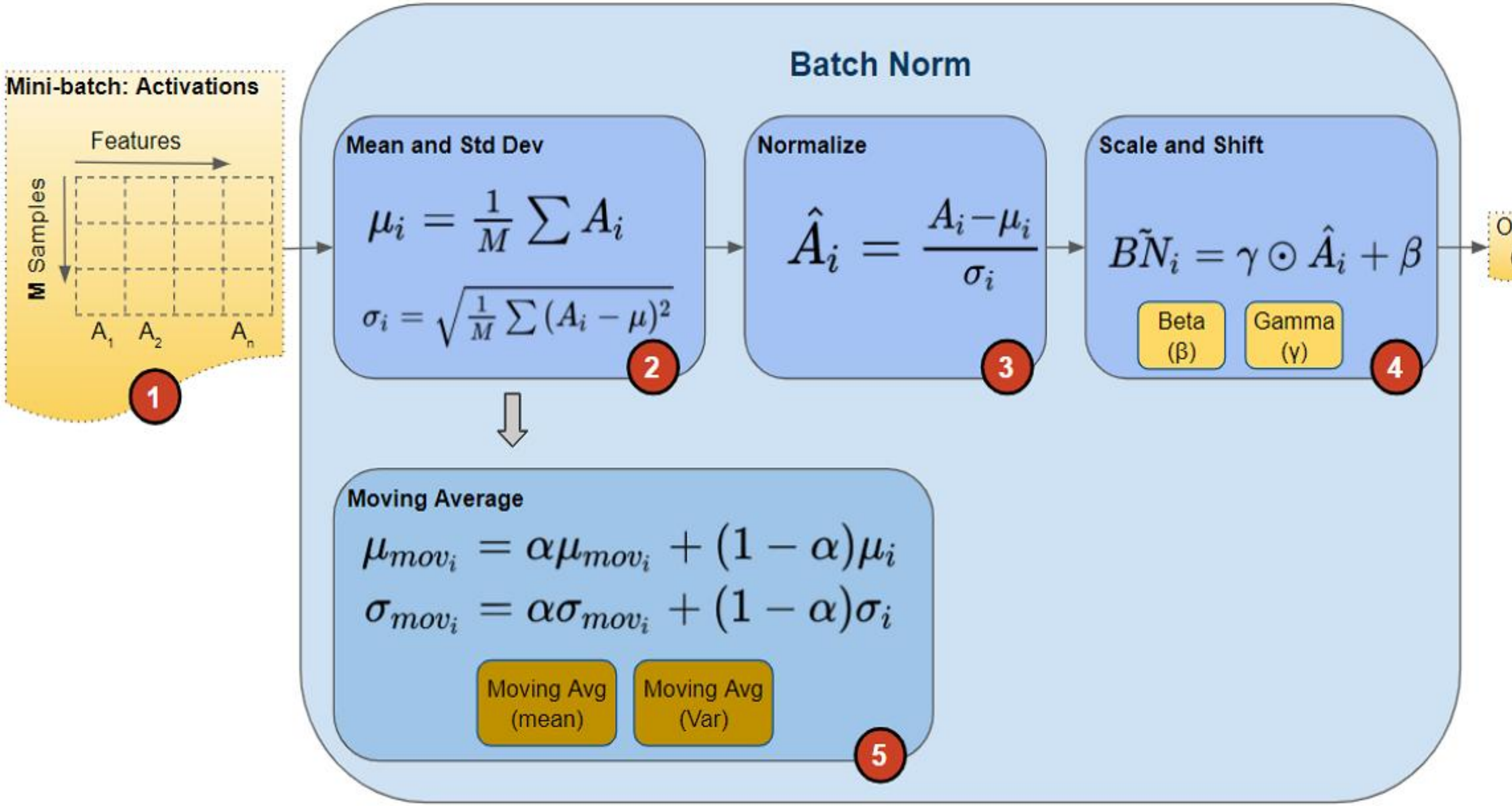
- after normalizing the activation layer, multiply the outputs by an arbitrarily set parameter, and add to that value an additional parameter, therefore setting a new standard deviation and mean;
- Make all four of these values (the mean and standard deviation for normalization, and the two new parameters to set arbitrary mean and standard deviations after the normalization) **trainable**.



https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739

BATCH NORMALIZATION LAYER

1. **Activations**
2. **Calculate Mean and Variance**
3. **Normalize**
4. **Scale and Shift** - Unlike the input layer, which requires all normalized values to have zero mean and unit variance, Batch Norm allows its values to be shifted (to a different mean) and scaled (to a different variance). It does this by multiplying the normalized values by a factor, gamma, and adding to it a factor, beta that are learnable parameters allowing to shift and scale to get the best results
5. **Moving Average** - This value is saved as part of the layer's state, for use during the Inference phase. The Moving Average calculation uses a scalar 'momentum' denoted by alpha (**hyperparameter**).

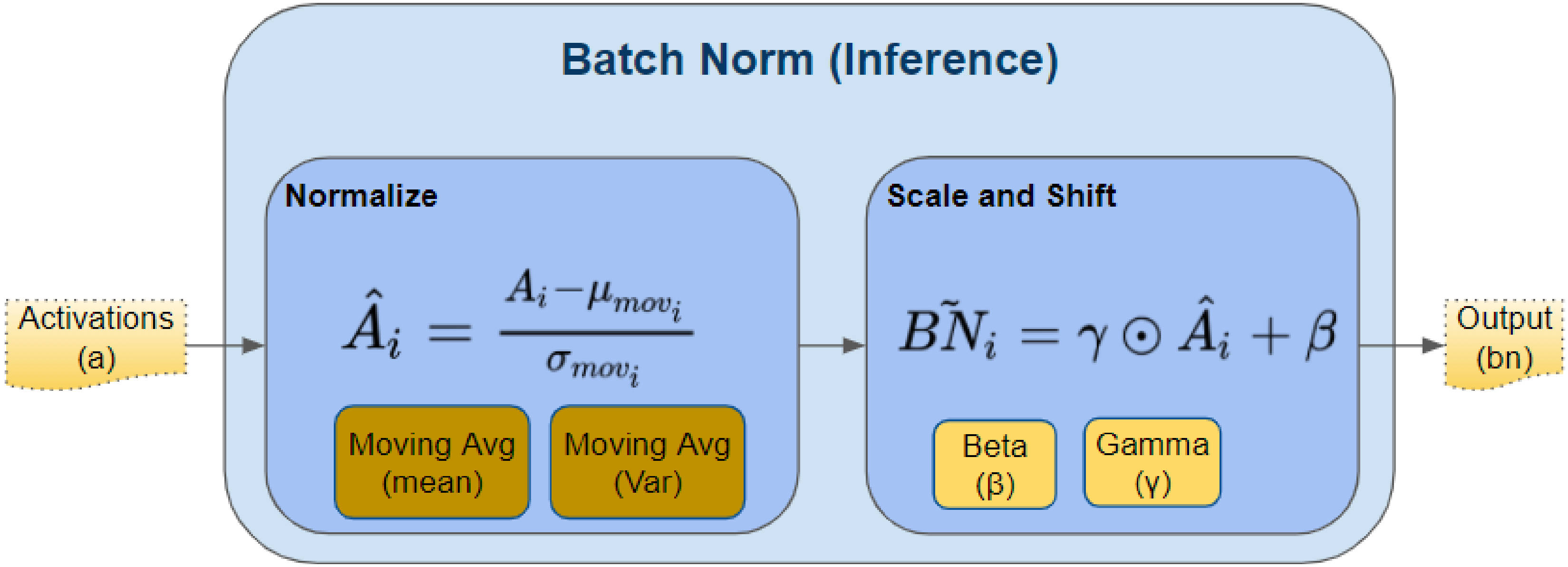


Mini-batch: Activations
Features
M Samples
A_1 A_2 A_n
1
Batch Norm
Mean and Std Dev
$\mu_i = \frac{1}{M}\sum A_i$
$\sigma_i = \sqrt{\frac{1}{M}\sum(A_i - \mu)^2}$
2
Normalize
$\hat{A}_i = \frac{A_i - \mu_i}{\sigma_i}$
3
Scale and Shift
$B\tilde{N}_i = \gamma \odot \hat{A}_i + \beta$
Beta (β)
Gamma (γ)
4
Moving Average
$\mu_{mov_i} = \alpha\mu_{mov_i} + (1-\alpha)\mu_i$
$\sigma_{mov_i} = \alpha\sigma_{mov_i} + (1-\alpha)\sigma_i$
Moving Avg (mean)
Moving Avg (Var)
5
O

BATCH NORMALIZATION LAYER

Batch Norm during Inference

During training, Batch Norm starts by calculating the mean and variance for a mini-batch. However, during Inference, we have a single sample, not a batch. This is why we need the two Moving Average parameters come in — the ones that we calculated during training and saved with the model. We use those saved mean and variance values for the Batch Norm during Inference.



FULLY CONNECTED LAYER

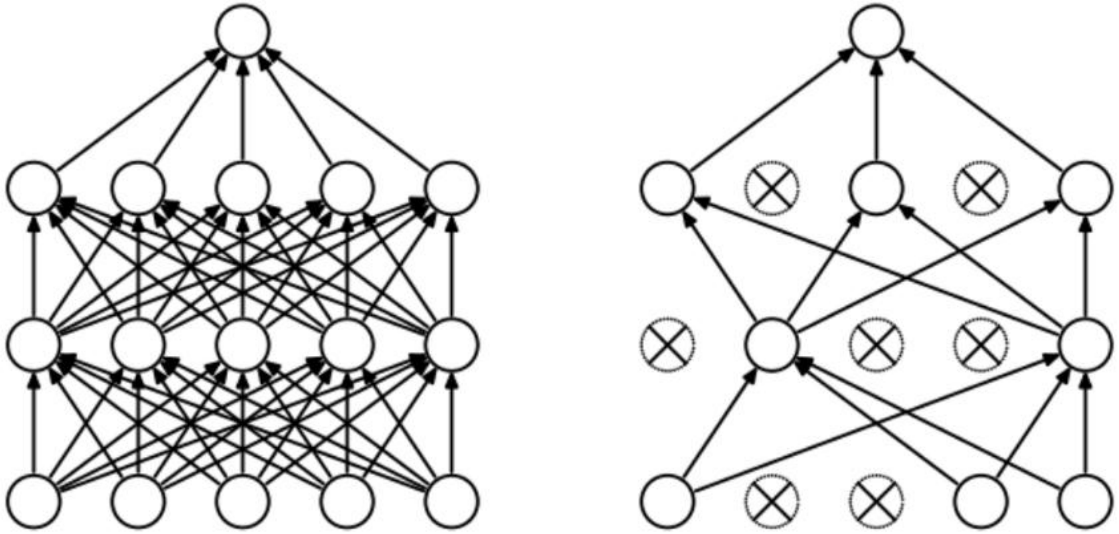
Fully Connected layer

This final layer is traditionally a Multi Layer Perceptron (MLP) that uses the **softmax** activation function in the output layer. The term "**Fully Connected**" implies that each neuron in the previous layer is connected to all neurons in the next layer. The **softmax** function is a generalization of the logistic function that "squashes" a K-dimensional vector of arbitrary real values into values between 0 and 1 that sum to the sum of 1 (i.e. probabilities).

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

TREINO "A BETTER MODEL"

Dropout, (only happens during training) occurs after activation and randomly sets activations to zero. It corresponds to erasing parts of the neuronal network. This deletion of the information that was learned will prevent **overfitting**, allowing the network to **generalize**. The **dropout rate** is specified at the beginning and depends on the problem and the results achieved during training.



(a) Standard Neural Net

(b) After applying dropout.

TRAINING PROCESS

Training process of a CNN:

1: initialize all filters and parameters/weights with random values

2: The net takes a training image as input, runs through forward propagation (convolution, ReLU and pooling operations in addition to forward propagation in the Fully Connected layer) and outputs the probabilities for each class. Obs. Since in the first case the weights were random, the probabilities will also be random.

3: Calculate the total error in the output layer: Total Error = ∑ ½ (target probability - output probability)

4: Use Backpropagation to calculate error gradients for all weights in the network and use gradient descent to adjust all filters/weights and parameter values to minimise the output error. The weights are adjusted in proportion to their contribution to the total error. If the same image was processed the probabilities would be virtually right. This would mean that the network would have learned to classify that particular case. Parameters like number of filters, size of filters, network architecture etc. have been predefined and are not changed during training. Only the values of filter matrix and link weights are adjusted.

5: Repeat steps 2-4 with all the images in the training set.

Training process "A Better Model"

Underfitting
This is a model that lacks the necessary complexity to correctly capture the complexity inherent to the problem it is intended to solve. **You can recognize this situation when the error is too large, both in the training cases and in the test (validation) cases**.

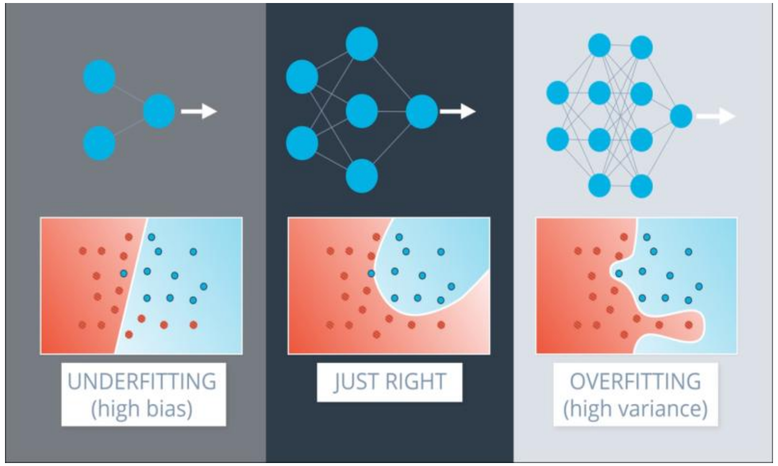
Overfitting
This is a model that is using too many parameters and has been trained too much.
Specifically, it has learned to identify exactly each case in the training set, becoming so specific that it cannot generalize to similar images.
You can recognize this situation when the error in the training cases is much smaller than in the test (validation) cases.

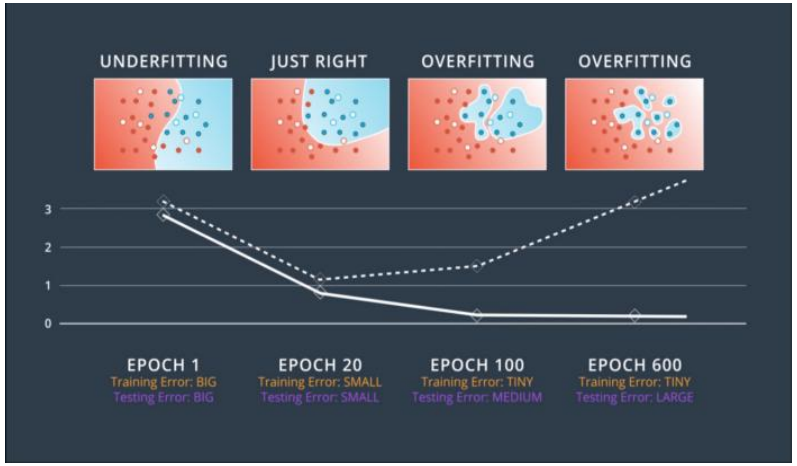
Generally there are several measures you can take to reduce **overfitting**:

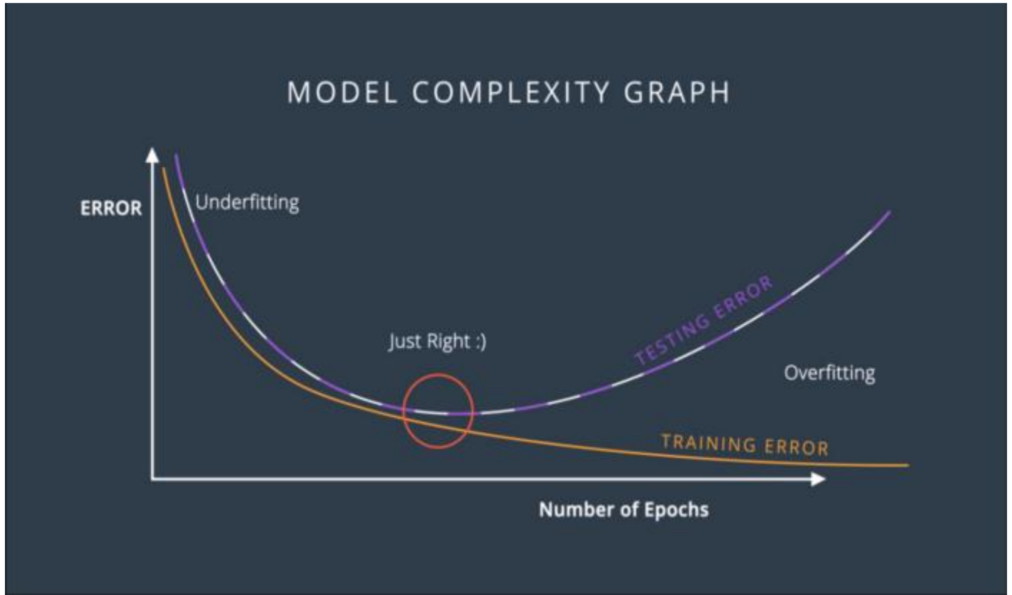
- Adding **more cases** to the training set
- Use architectures that have been shown to generalize well
- Reduce the complexity of the **network architecture**
- Use **data augmentation**
- Add normalization (**Batch Normalization layer**)
- Add Dropout (**Dropout layer**)

ATTENTION: Most of the **computation time** is spent on **convolutional layers**,
while most of the **memory** is spent on **dense layers.**

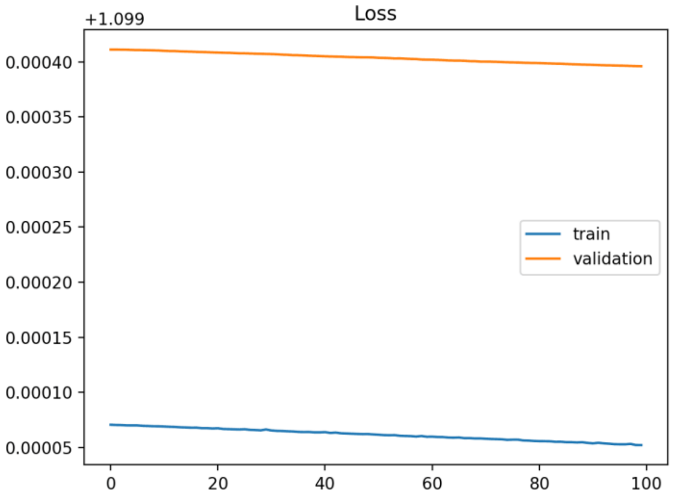
TRAINING PROCESS "A BETTER MODEL"







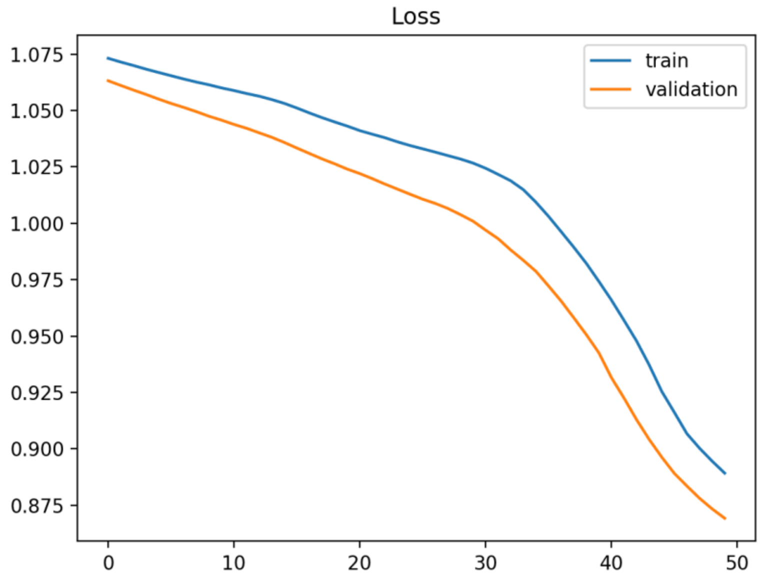
TRAINING PROCESS "LEARNING CURVES"



1: Example of Training Learning Curve Showing An **Underfit Model** That **Does Not Have Sufficient Capacity**. It may show a **flat line** or **noisy values** of **relatively high loss**, indicating that the model was unable to learn the training dataset at all.

- **Add more observations**. You may not have enough data for the existing patterns to become strong signals.
- **Add more features**. Occasionally our model is under-fitting on the grounds that the feature items are insufficient.
- **Reduce any regularization on the model**. If you have explicit regularization parameters specified (i.e. dropout, weight regularization), remove or reduce these parameters.
- **Increase model capacity**. Your model capacity may not be large enough to capture and learn existing signals.

TRAINING PROCESS "LEARNING CURVES"

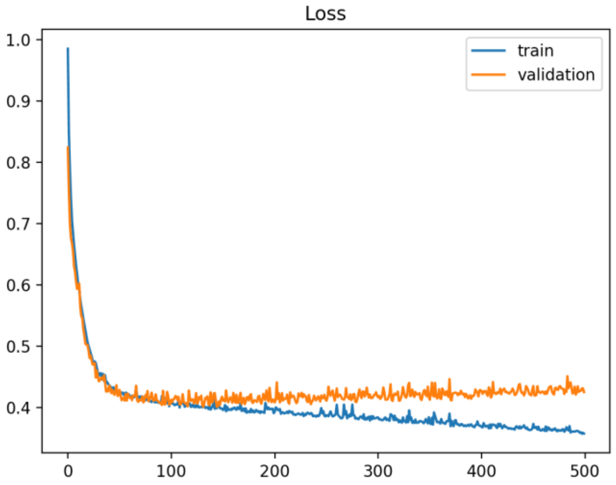


2: Example of Training Learning Curve Showing an **Underfit Model** That **Requires Further Training**. An underfit model may also be identified by a training loss that is decreasing and continues to decrease at the end of the plot. This indicates that the model is capable of further learning and possible further improvements and that the training process was halted prematurely.

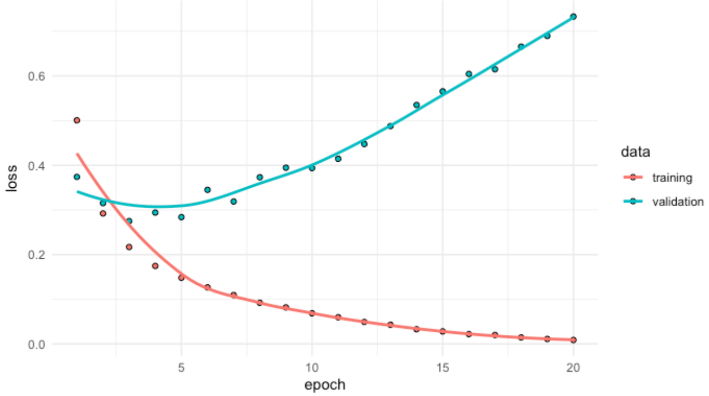
- **Increase the number of epochs** until the validation curve has stopped improving. This is a good time to crank up the epochs and add an **early stopping callback (**use Lightning or Ignite, with Neptune for learning curves) to identify how many epochs are required.

- If it is taking a long time to reach a minimum for the validation curve, **increase the learning rate** to speed up the gradient traversal and also add a **callback (**scheduler) **to automatically adjust the learning rate**.

Training process "Learning Curves"



Example of learning curve showing an overfit model with too large of a capacity and learning rate.



3: - Example of Train and Validation Learning Curves Showing an **Overfit Model**. The inflection point in validation loss may be the point at which training could be halted as experience after that point shows the dynamics of overfitting.

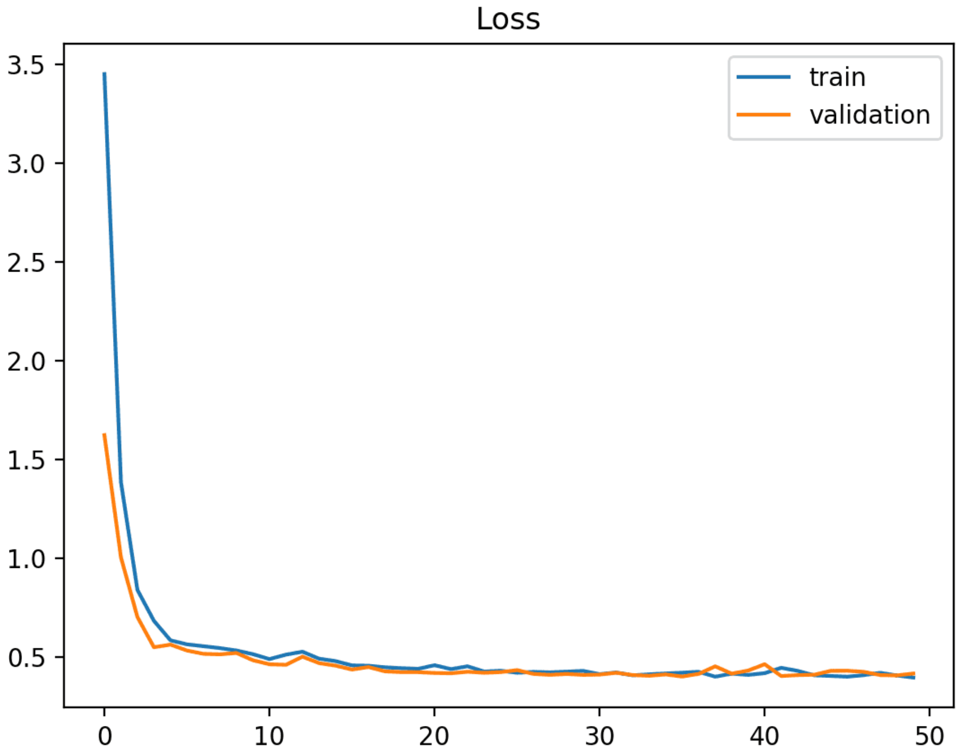
- Regularize how quickly the model learns by **reducing the learning rate**. Add a **callback** to automatically reduce the learning rate as the validation loss plateaus.

- Regularize model capacity by **reducing the number and/or size of the hidden layers**.

- Regularize the weights to constrain the complexity of the network.

- Regularize happenstance patterns by **adding dropout** to minimize the chance of fitting patterns to noise in the data.

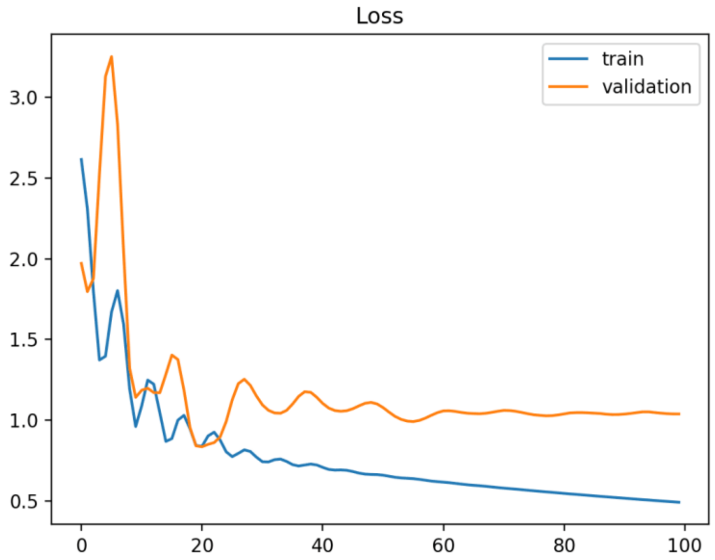
TRAINING PROCESS "LEARNING CURVES"



4: -Example of Train and Validation Learning Curves Showing a **Good Fit**

.

Training process "Learning Curves"



Diagnosing Unrepresentative Datasets

1 - Example of Train and Validation Learning Curves Showing a **Training Dataset That May Be too Small Relative to the Validation Dataset**. This situation can be identified by a learning curve for training loss that shows improvement and similarly a learning curve for validation loss that shows improvement**, but a large gap remains between both curves**

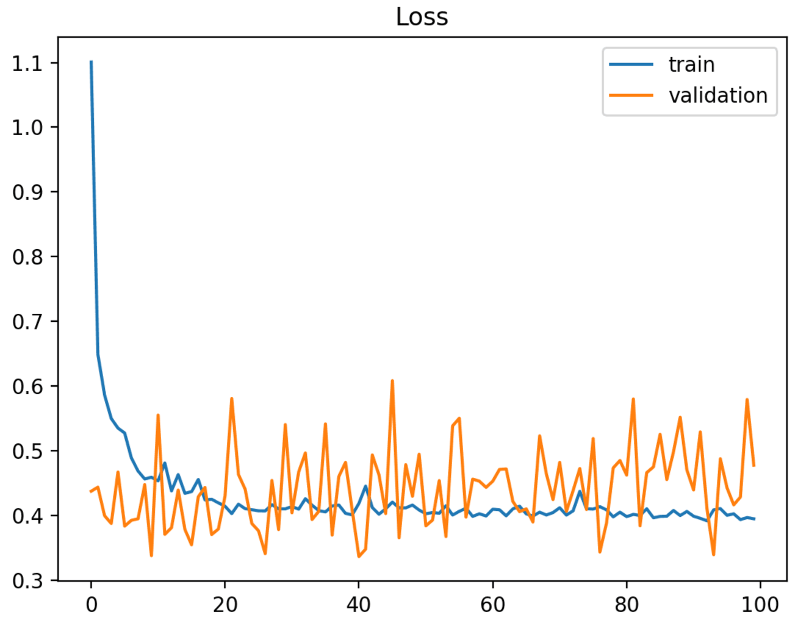
- **Add more observations**. You may not have enough data to capture patterns present in both the training and validation data.

- If using CNNs incorporate **data augmentation** to increase feature variability in the training data.

- Make sure that you are **randomly sampling observations** to use in your training and validation sets. If your data is ordered by some feature (i.e. neighbourhood, class) then you validation data may have features not represented in your training data.

- Perform **cross-validation** so that all your data has the opportunity to be represented in both the training and validation sets.

TRAINING PROCESS "LEARNING CURVES"



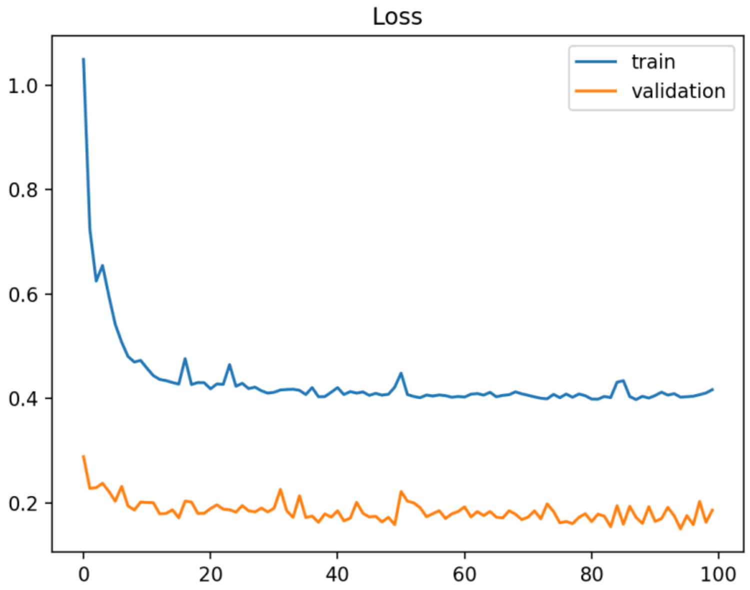
Diagnosing Unrepresentative Datasets

2 – Example of Train and Validation Learning Curves Showing a **Validation Dataset That May Be too Small Relative to the Training Dataset**. This case can be identified by a learning curve for training loss that looks like a good fit (or other fits) and a learning curve for validation loss that shows noisy movements around the training loss.

- Add **more observations** to your validation dataset.

- If you are limited on the number of observations, perform **cross-validation** so that all your data has the opportunity to be represented in both the training and validation sets.

TRAINING PROCESS "LEARNING CURVES"



Diagnosing Unrepresentative Datasets

3 - Example of Train and Validation Learning Curves Showing a **Validation Dataset That Is Easier to Predict Than the Training Dataset**. It may also be identified by a validation loss that is lower than the training loss.

- Check to **make sure duplicate observations do not exists** across training and validation datasets.

- Check to make sure there is no **information leakage** across training and validation datasets.

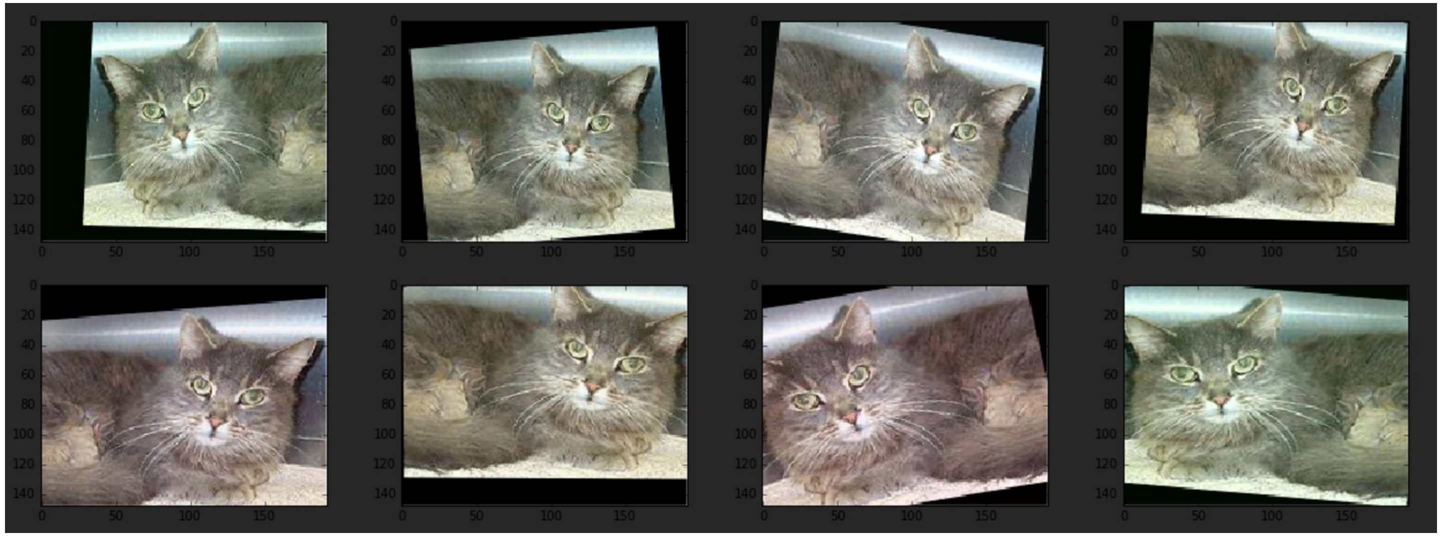
- Make sure that you are **randomly sampling observations** to use in your training and validation sets so that feature variance is consistent across both sets.

- Perform **cross-validation** so that all your data has the opportunity to be represented in both the training and validation sets.
- Dropout ?

TRAINING PROCESS "A BETTER MODEL"

Data Augmentation
It was noted that overfitting is the result of overlearning the specificity of the training cases, failing to generalise to similar images.

Data augmentation simply causes changes in each **batch** of our images. It does this through operations like **flipping**, **hue changes**, **stretching**, **shearing**, **rotation**, etc. doing this in a way that makes sense (for example, it doesn't make sense to flip an image with a dog vertically). Current frameworks usually have a **data-augmentation batch generator (**transformers**)**. Unfortunately there is no way to know in advance the best parameterization, only by experimenting.



CONCEPTS

Epoch
A forward pass and a backward pass in all training cases.

Batch size
Number of training cases used in each forward and backward pass . The larger the batch size, the more memory is required.

Iterations
Number of passes, each pass using the number of batch size cases. Note that one pass = one forward pass + one backward pass.
Example: If we have 1000 training cases, and the **batch size** is 500, then it will take 2 **iterations** to complete 1 **epoch**.

learning rate
parameter chosen by the programmer that will determine the size of the steps to be taken in updating the network's learning parameters during the training process. A larger value of learning rate can mean less time for the model to converge to the optimal weights, but it can also mean that they never reach the optimal value due to lack of accuracy.

$$w = w_i - \eta \frac{dL}{dW}$$

w = Weight
w_i = Initial Weight
η = Learning Rate

CNN - Good Practices

Input Receptive Field Dimensions – By default it is 2D for images, but it could be 1D for e.g. word in a sentence or 3D for video which adds the time dimension.

Receptive Field Size - The **patch** should be as small as possible, but large enough to detect features in the data. It is common to use 3×3 for small images and 5×5 or 7×7 or even larger for large images. Putting the 2×2 with the **stride** at 2 will discard 75% of the activations coming from the output of the previous layer.

Stride Width – Use the default value of 1. It's easier to understand and you don't need to padding because of the output outside the image border. On large images you can use 2.

Number of Filters – Filters are what detect the **features**. Generally, fewer filters are used in the input layer and more filters in the deeper layers.

Padding - set to zero (zero padding).

Pooling - A destructive or generalisation process to reduce **overfitting**.

Data Preparation: Always consider normalising the input data, both in image size and pixel values.

Pattern Architecture – It is normal to standardize the layers in the network architecture. It can be for example one or two convolution layers followed by a pooling layer. This pattern can then be repeated one or more times. Fully connected layers are only used at the end and may have more than one layer.

Dropout - CNNs have a habit of **overfitting**, even with **pooling layers**. **Dropout** can be used both between fully connected layers and after **pooling layers**.

References

https://gradientflow.com/supercharging-your-data-and-ai-platforms/
https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148
http://cs231n.github.io/convolutional-networks/
https://towardsdatascience.com/types-of-convolutions-in-deep-learning-717013397f4d
https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215
https://towardsdatascience.com/pytorch-conv2d-weights-explained-ff7f68f652eb
https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6
https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739
https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/
https://rstudio-conf-2020.github.io/dl-keras-tf/notebooks/learning-curve-diagnostics.nb.html