



# Visão por computador

Uma abordagem Deep Learning

**IMAGEM MÉDICA**

# INTRODUÇÃO

- Processamento de imagens é um processo onde a entrada do sistema é uma imagem e a saída é um conjunto de valores numéricos, que podem ou não compor uma outra imagem.
- Visão computacional procura emular a visão humana, portanto também possui como entrada uma imagem, porém, a saída é uma interpretação da imagem como um todo, ou parcialmente.



Placa:  
BRK 8558  
Veículo:  
Pajero 1995

Veículo em  
Imagem Escura

Após uma  
equalização de  
histograma, em  
nível de cinza,  
onde a placa do  
veículo pode ser  
lida

Informação da  
placa e do  
veículo no  
retângulo

# INTRODUÇÃO

- Processos de visão computacional geralmente iniciam com o processamento de imagens.
- Processamento ocorre em três níveis:
  - baixo-nível: operações primitivas (redução de ruído ou melhoria no contraste de uma imagem);
  - nível-médio: operações do tipo segmentação ou Classificação;
  - alto-nível: tarefas de cognição normalmente associadas com a visão humana.

**Universidade do Minho**

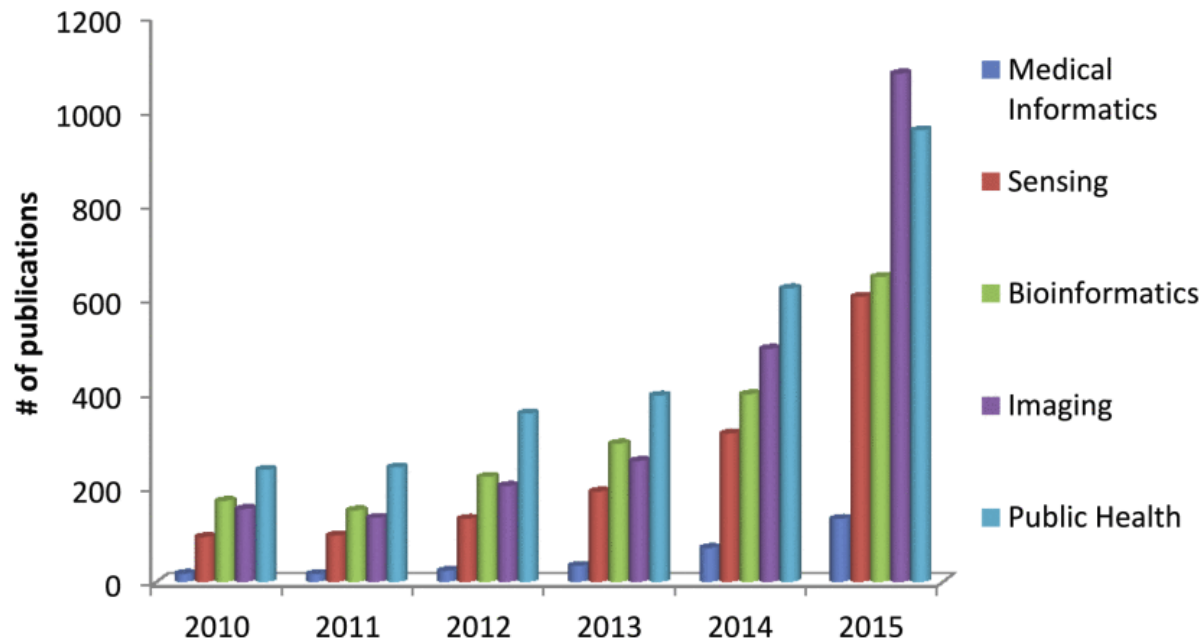
Escola de Engenharia/Departamento de Informática



# Deep Learning

# TENDENCIAS

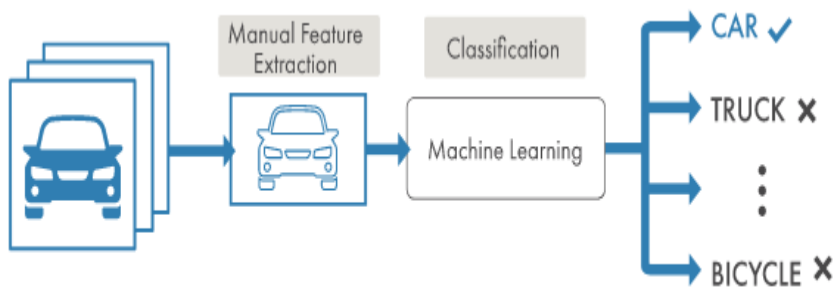
**Figure 2: Distribution of published papers that use deep learning in subareas of health informatics. Publication statistics are obtained from Google Scholar; the search phrase is defined as the subfield name with the exact phrase deep learning and at least one of medical or health appearing, e.g., “public health” “deep learning” medical OR health**



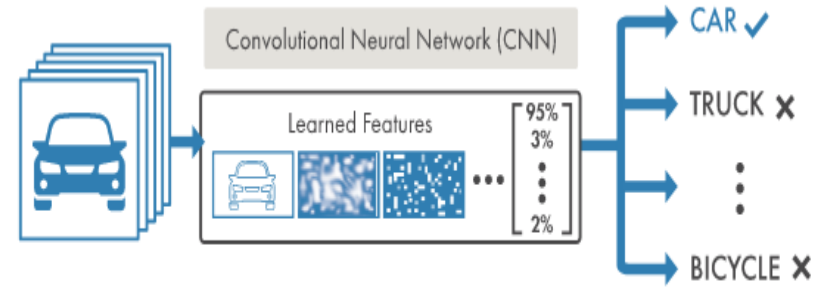
Source: Daniele Ravi et al, Deep Learning for Health Informatics, IEEE Jnl. for Biomedical and Health Informatics, vol. 21, pp.4-21, January 2017.

# CLASSIFICAÇÃO

MACHINE LEARNING



DEEP LEARNING



# MACHINE LEARNING

**Machine Learning** - Pertence à área da inteligência artificial e tem como objetivo conseguir que as máquinas tenham a capacidade de aprender sem programação explícita. Concretamente a ideia é que os programas uma vez desenvolvidos sejam capazes de evoluir e adaptar-se quando expostos a novos dados

Baseada nesta abordagem existem 3 métodos de aprendizagem de sistemas de Machine Learning:

- ❑ **Supervised Learning** – O sistema recebe um conjunto de casos classificados (training set) e pretende-se criar um modelo genérico relativo a esses casos de modo a poder actuar do mesmo modo em casos novos.
- ❑ **Unsupervised Learning** – O sistema recebe um conjunto de casos não classificados nos quais se pretende descobrir padrões. Geralmente o objetivo é descobrir padrões escondidos
- ❑ **Reinforcement Learning** – Ao sistema é pedido que efectue uma acção, sendo recompensada pela mesma.. O sistema deve aprender quais as acções que trazem maior recompensa numa determinada situação.



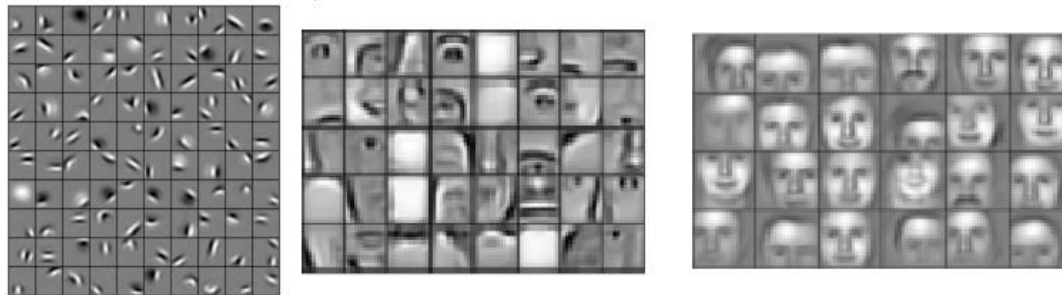
# DEEP LEARNING

## **Deep learning:**

- ❑ É um processo, tal como o *data mining*, que utiliza a arquitetura das redes neuronais na resolução de problemas.
- ❑ É uma sub-área do Machine Learning, sendo baseada em algoritmos para aprendizagem de múltiplos níveis de representação de modo a conseguir modelar relações complexas entre os dados. Características de nível mais elevado e conceitos acabam por ser definidos em termos de características mais simples. A esta hierarquia de características chamamos **Deep Architecture**.
- ❑ Também se pode denominar **Deep Structured Learning** or **Hierarchical Learning**.

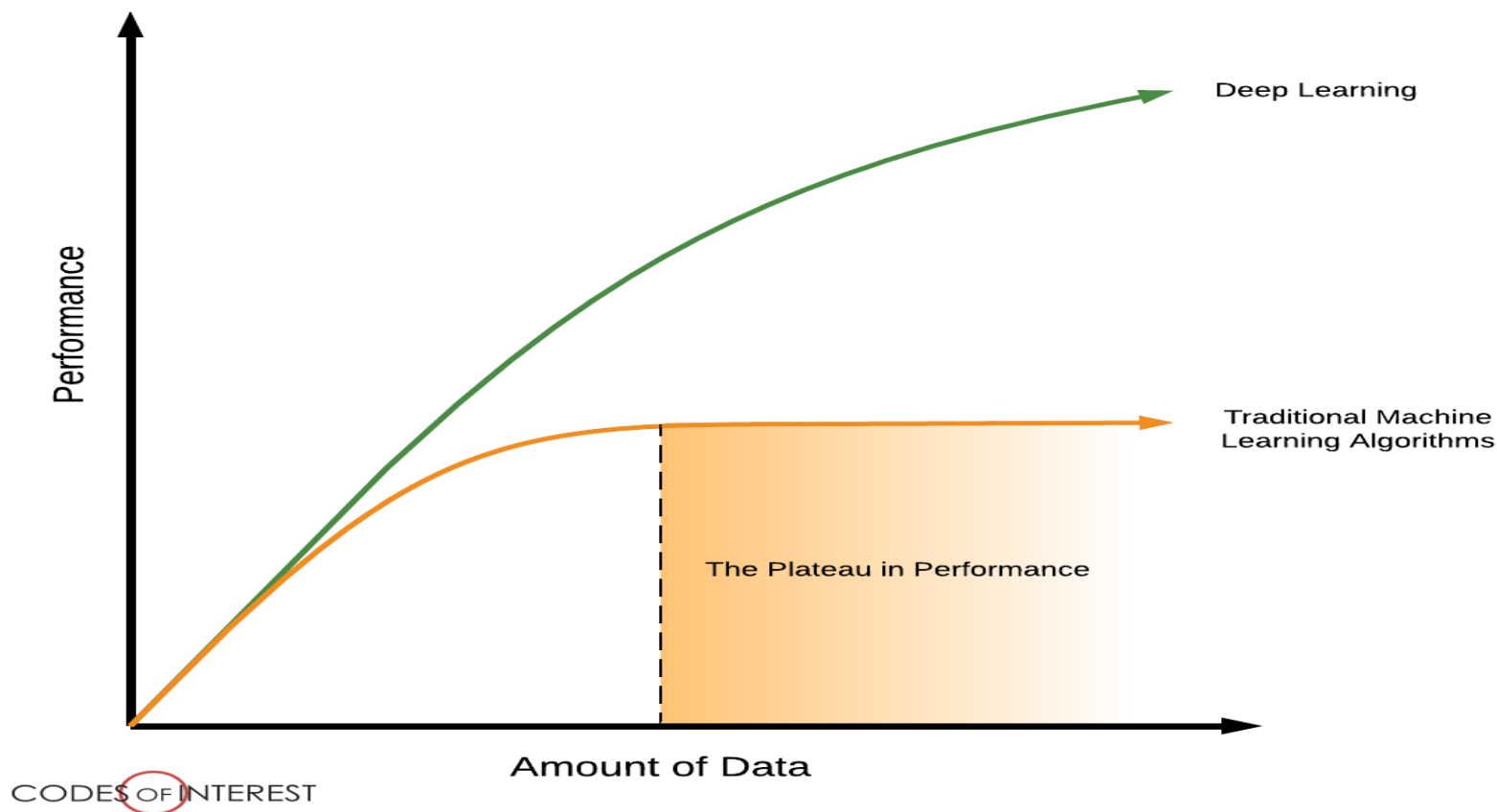
## **Como identificar se se trata de um modelo Deep Learning ou não:**

- Simplesmente se o modelo utilizar aprendizagem de características hierarquizadas, identificando características mais simples primeiro e depois a partir dessas identificar características mais complexas -e.g. utilizando filtros de convolução) então é um modelo Deep Learning. Caso contrário, independentemente dos níveis que tiver o modelo não é considerado Deep Learning.

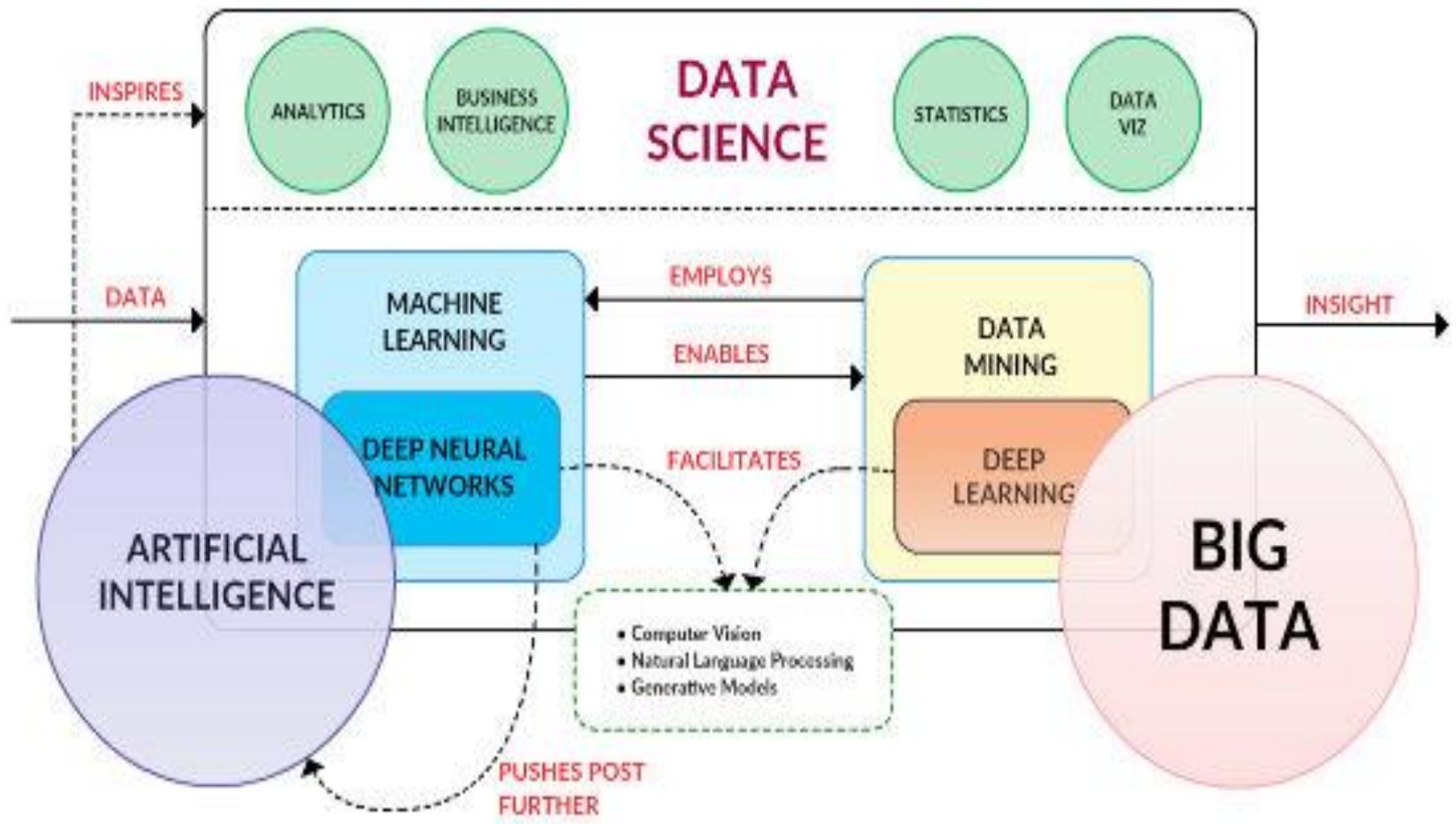


# DEEP LEARNING

Uma das características mais distintivas do Deep Learning é a sua escalabilidade, i.e. quantos mais dados, melhor desempenho tem. Ao contrario de outros algoritmos de **Machine Learning** que atingem o chamado **Plateau in Performance** muito mais depressa.



# DEEP LEARNING



By Matthew Mayo, KDnuggets

# DEEP LEARNING

## ATENÇÃO!

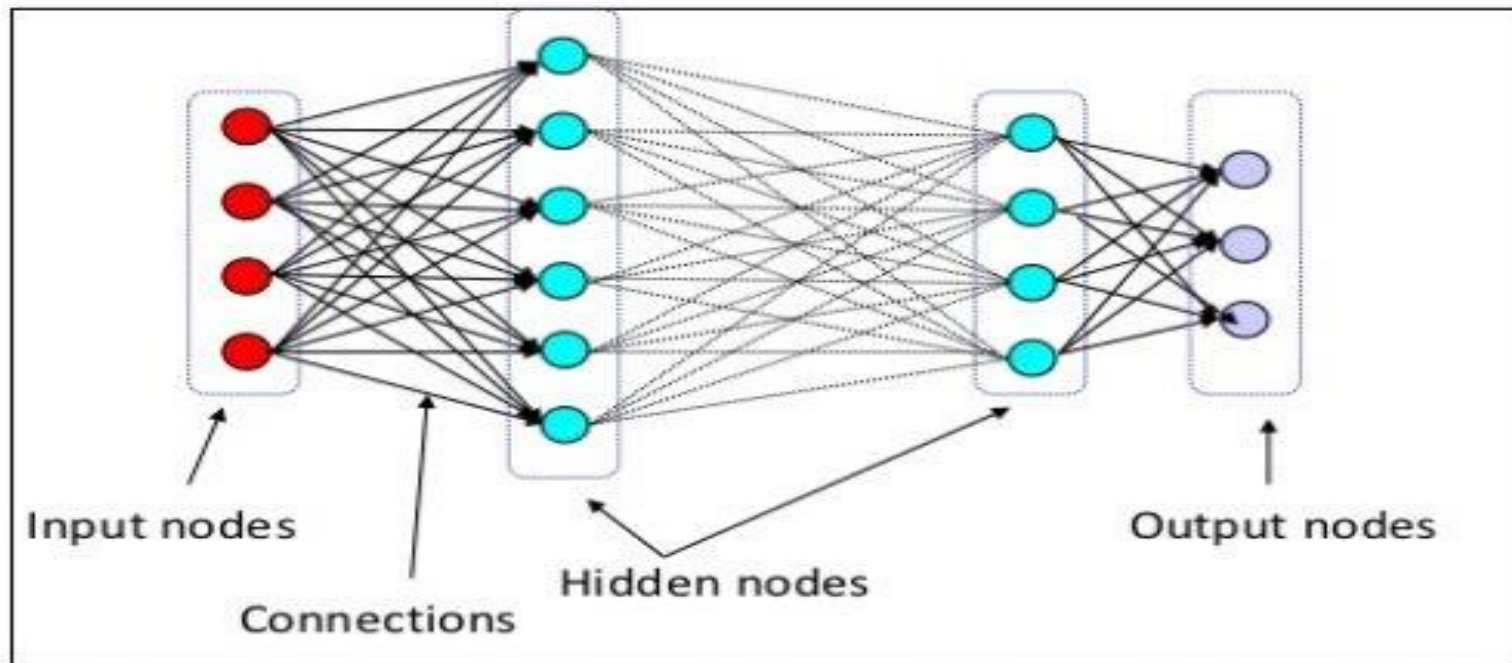
Embora tenha alcançado imenso sucesso nos últimos anos é preciso ter em atenção:

- ❑ - **Deep learning não é a solução para tudo.** – Não é a solução fácil que vai agora resolver todos os problemas.
- ❑ - **Não vai substituir todos os outros algoritmos de *machine learning*.**
- ❑ - **As expectativas devem ser contidas** – Sendo evidentes os sucessos obtidos em problemas de classificação em áreas como visão por computador e processamento de linguagem natural, o deep learning não escala para resolver todos os problemas do mundo.
- ❑ **Deep learning e inteligência artificial não são sinónimos;**
- ❑ **Deep learning é uma valiosa mais-valia para a área actualmente conhecida como data science.**

# ARTIFICIAL NEURAL NETWORKS

## Artificial Neural Networks (ANNs)

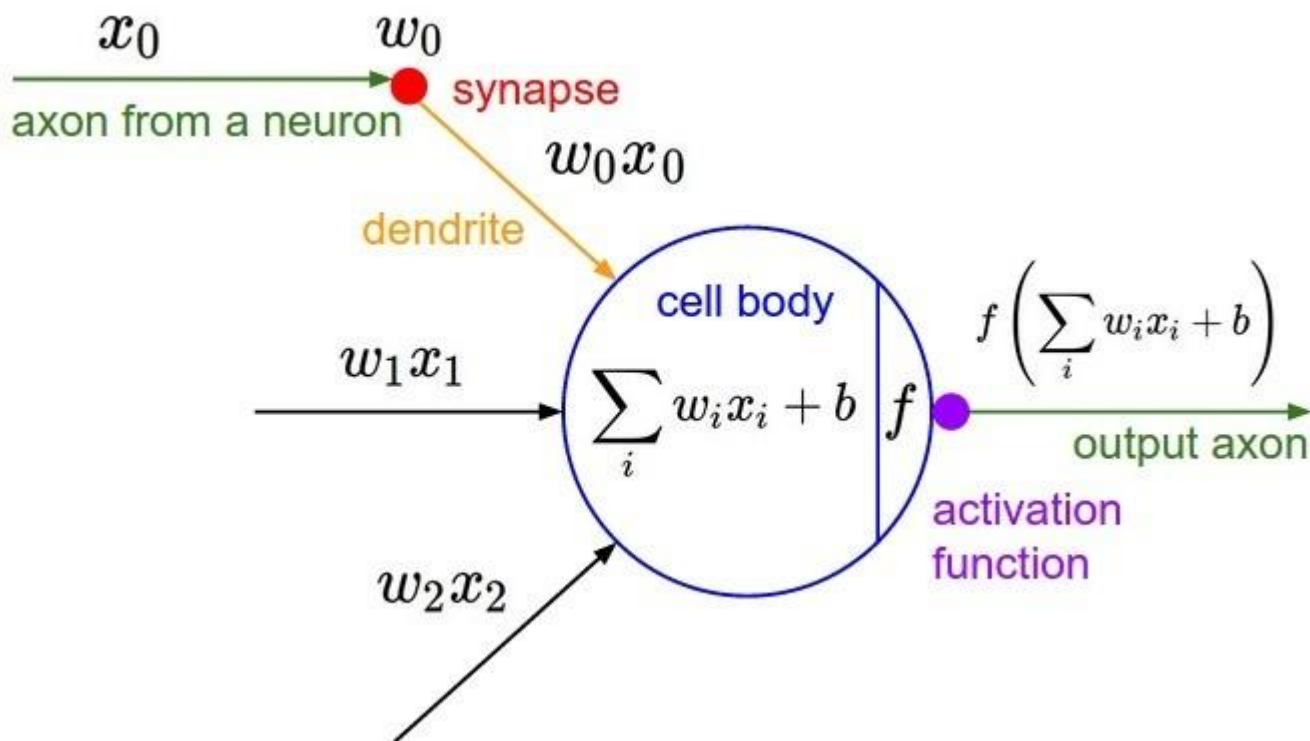
Arquitetura de machine learning originalmente inspirada no neurônio do cérebro biológico a partir da qual se dá o processo de aprendizagem. Concretamente possuem a capacidade de aproximar funções não lineares dos dados de entrada aos neurônios.



# ARTIFICIAL NEURAL NETWORKS

## Artificial Neural Networks (ANNs)

Arquitetura de machine learning originalmente inspirada no neurônio do cérebro biológico a partir da qual se dá o processo de aprendizagem. Concretamente possuem a capacidade de aproximar funções não lineares dos dados de entrada aos neurônios.



# ARTIFICIAL NEURAL NETWORKS

## Perceptron

Um **perceptron** é simplesmente um classificador linear binário. Os perceptrons recebem inputs e pesos associados (que representam a sua importância) e combinam-os para produzir o output o qual é utilizado para classificação.

## Feedforward Neural Network

Feedforward neural networks são a forma mais simples de uma arquitetura de rede neuronal nas quais as ligações não são cíclicas. É neste aspeto que divergem das redes neuronais recorrentes.

## Multilayer Perceptron (MLP)

Um *multilayer perceptron* (MLP) são vários perceptron ligados em camadas adjacentes formando uma simples **feedforward neural network**. Este multilayer perceptron tem a vantagem adicional de possuir funções de ativação não lineares o que não é o caso dos perceptrons (é uma rede feedforward).

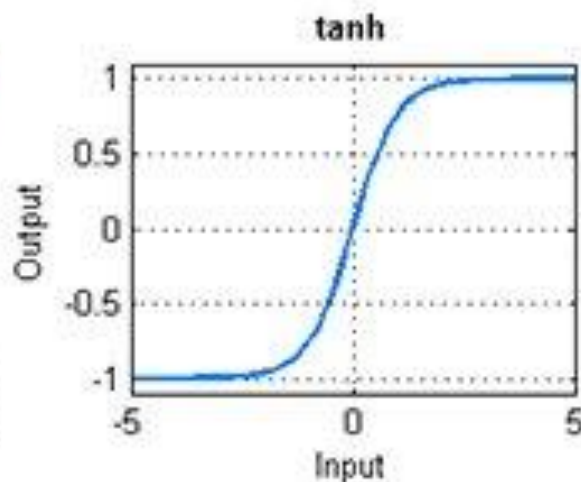
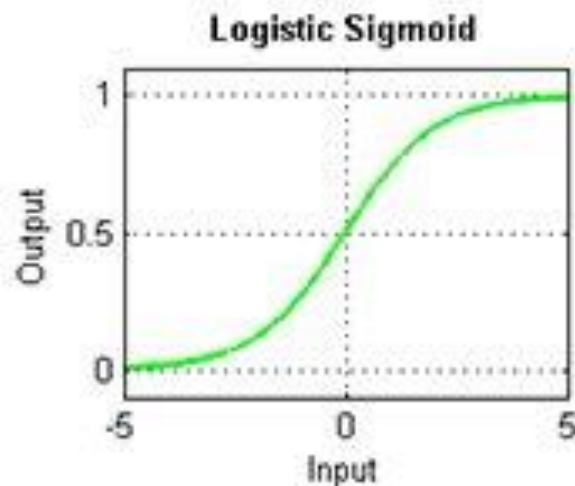
## Recurrent Neural Network

Esta característica de possuir ligações cíclicas que as distingue das FNN permite uma representação temporal interna, a qual fornece as características para poder processar sequencias permitindo tratar problemas como as previsões em series temporais, a fala e a escrita.

# ARTIFICIAL NEURAL NETWORKS

## Activation Function

Em redes neurais a função de ativação implementa a decisão relativamente aos limites do output combinando os valores das entradas pelos respectivos pesos. Como função de ativação podemos ter simplesmente a identidade (linear), o sigmoide (logistic, ou soft step) a hiperbólica (tangente) e outras. Para permitir a aprendizagem por backpropagation, a rede tem de utilizar funções de ativação que seja diferenciáveis.





# ARTIFICIAL NEURAL NETWORKS

## **Cost Function ou Loss Function**

A função de custo (loss) mede a diferença entre o valor de saída obtido e o pretendido. Um custo de zero significa que a rede foi treinada o mais possível. Isto seria claramente o ideal.

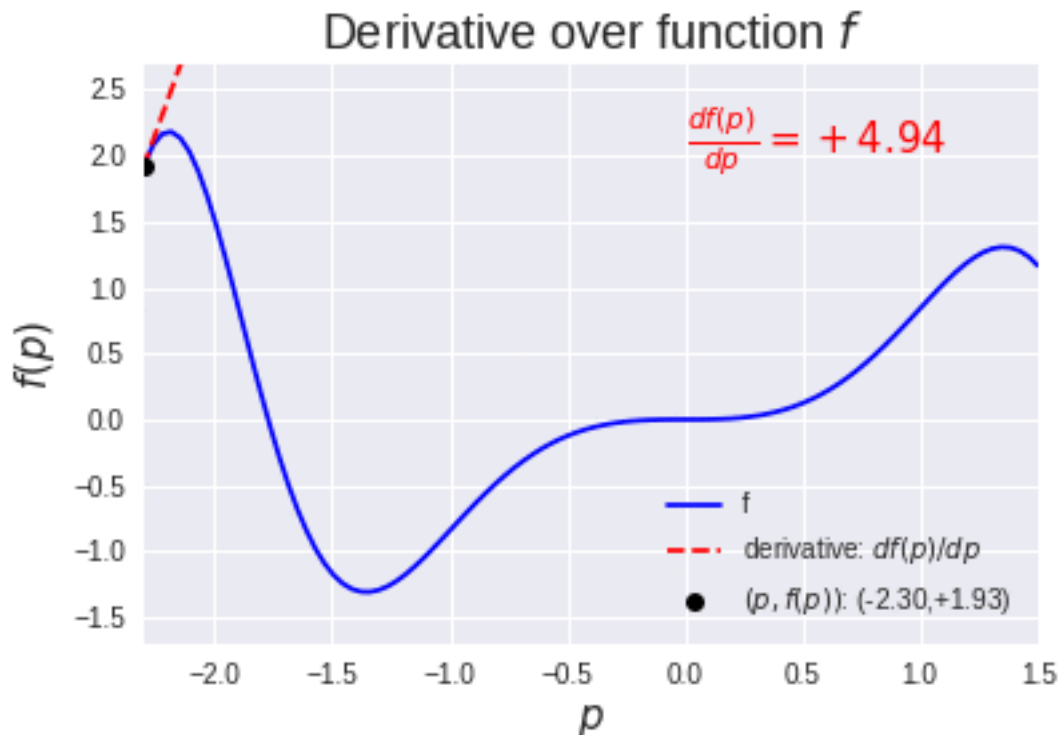
## **Backpropagation**

O algoritmo de Backpropagation que existe há décadas permite teoricamente treinar uma rede neuronal com muitas camadas, mas antes do advento do deep learning não havia grande sucesso no treino de redes com mais do que 2 camadas. Trata-se de um algoritmo que vai provocando uma descida pelo gradiente a partir dos valores dados pela função de custo. A partir do valor de custo (**loss**) calcula-se o gradiente dos erros relativamente a cada um dos pesos das ligações da rede. Assim obtém-se uma direcção na alteração a fazer aos pesos de modo a diminuir o erro.

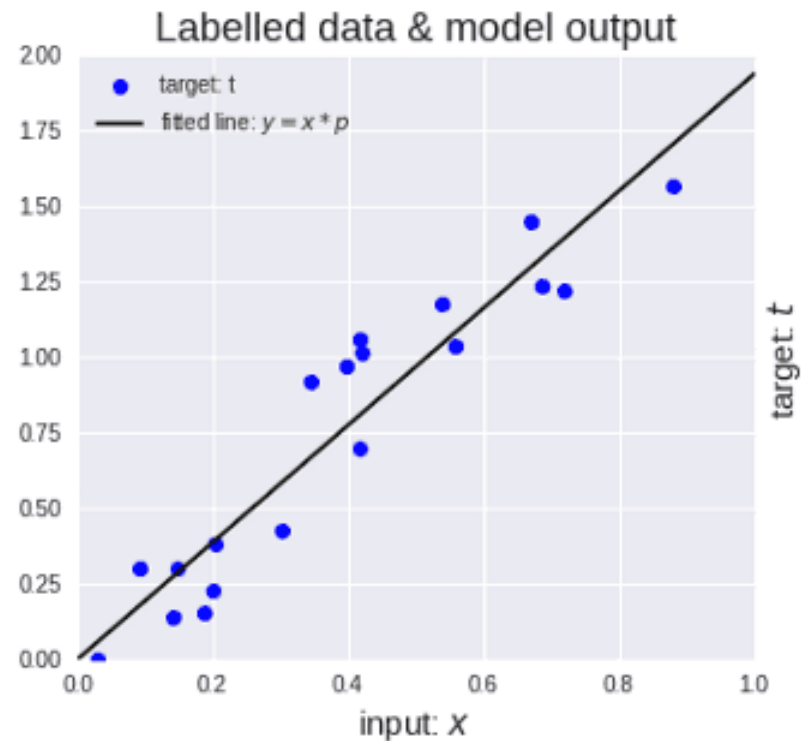
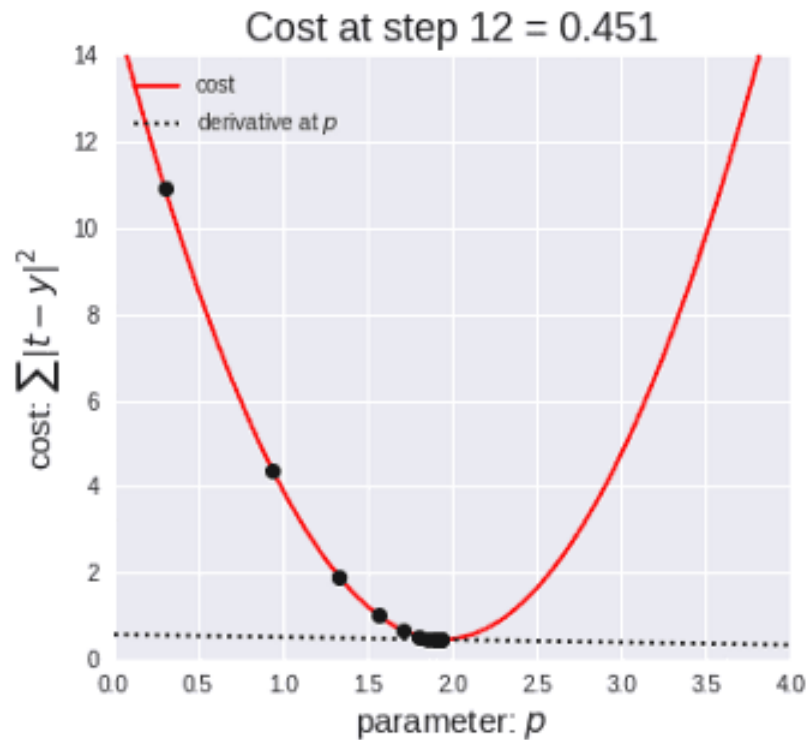
# ARTIFICIAL NEURAL NETWORKS

## Gradient Descent

O Gradient descent é um algoritmo de otimização utilizado para encontrar o mínimo local de uma função. Embora não garanta um mínimo global, gradient descent é especialmente útil para o caso de funções relativamente às quais é difícil de resolver analiticamente para soluções precisas, tal como a definição de derivadas para zero e respetiva resolução.



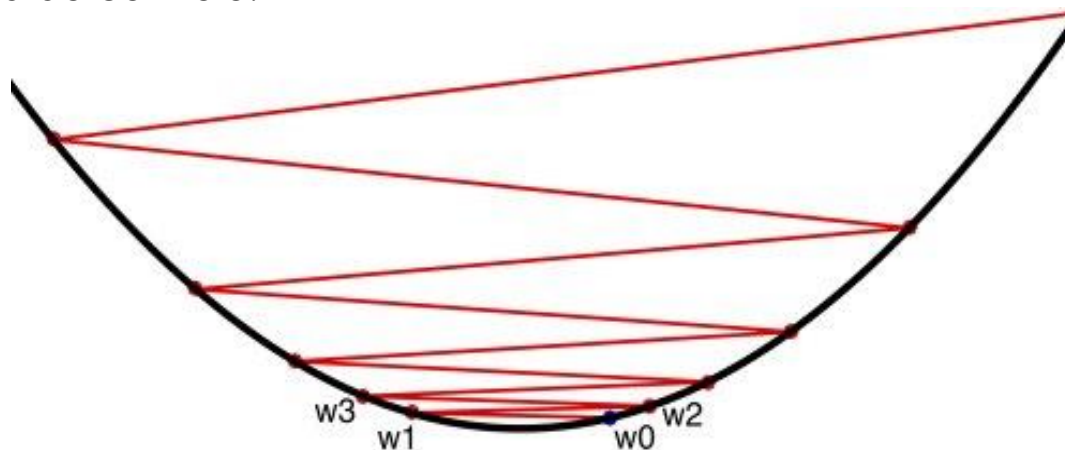
# ARTIFICIAL NEURAL NETWORKS



# BREAKTHROUGH

## Vanishing e/ou Exploding Gradient Problem

Antes do *deep learning* as redes eram tipicamente inicializados usando números aleatórios. Tal como hoje em dia,, as redes utilizavam o gradiente dos parâmetros da rede em relação ao erro para ajustar os parâmetros para valores melhores em cada iteração de treino (Backpropagation). Em backpropagation, o calculo deste gradiente envolve uma regra em cadeia e é necessário multiplicar o gradiente com cada parâmetro de cada camada ao longo de todas as camadas. São imensas multiplicações especialmente em redes com mais do que 2 camadas. Se muitos pesos ao longo das diversas camadas forem menores que 1 e forem multiplicadas várias vezes então eventualmente o gradiente acaba por tender exponencialmente para 0, desaparecendo, parando o processo de treino. Se muitos pesos ao longo das diversas camadas forem maiores que 1 e forem multiplicadas várias vezes então eventualmente o gradiente acaba por tender para um valor muito grande, e o processo de treino fica fora de controlo.



# BREAKTHROUGH

## Vanishing e/ou Exploding Gradient Problem

O ponto de viragem deu-se em 2006 quando **Geoffrey Hinton** e colegas apresentaram um algoritmo que conseguia ir afinando o processo de aprendizagem usado para treinar as redes com multiplas camadas escondidas. O truque estava na utilização de um algoritmo de gradient descent que conseguia afinar cada camada da rede separada, usar uma serie de redes de uma só camada - que não sofrem de *vanishing/exploding gradients* – para encontrar os parâmetros iniciais de uma deep MLP.

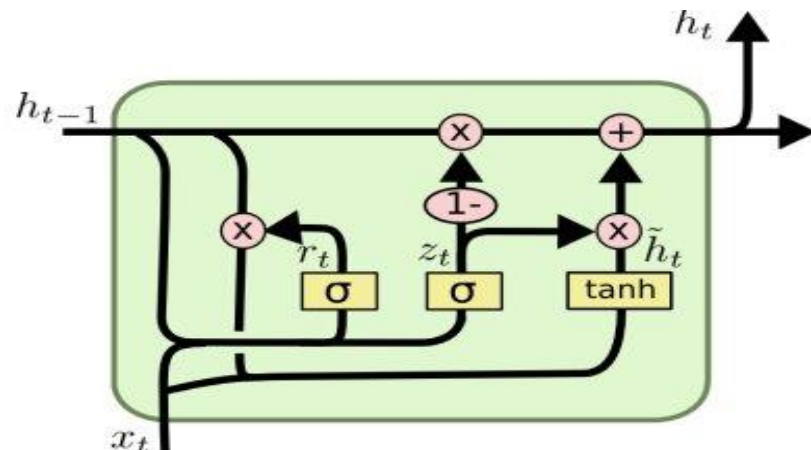
# DEEP LEARNING

## Convolutional Neural Network

Tipicamente associadas a visão por computador e reconhecimento por imagem, as Convolutional Neural Networks (**CNNs**) utilizam o conceito matemático da convolução para mimetizar o funcionamento do córtex visual biológico.

## Long Short Term Memory Network (LSTM)

Trata-se de uma rede neuronal recorrente que é otimizada para aprender a partir de, e actuar em dados temporais, com dimensão temporal indefinida ou desconhecida entre eventos com relevância. A sua particular arquitectura permite a persistência dando à rede uma certa memória. Os recentes avanços no reconhecimento da escrita manual e reconhecimento automático do discurso falado têm beneficiado deste tipo de redes. Em alternativa existem as redes GRU que são computacionalmente mais eficientes com desempenho praticamente semelhante.



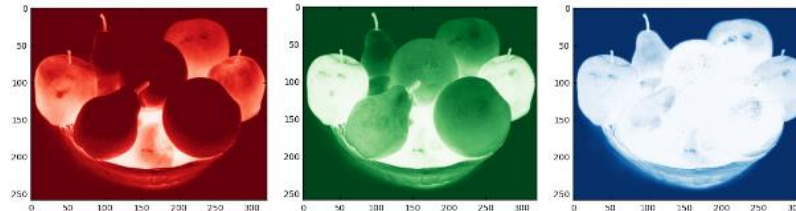
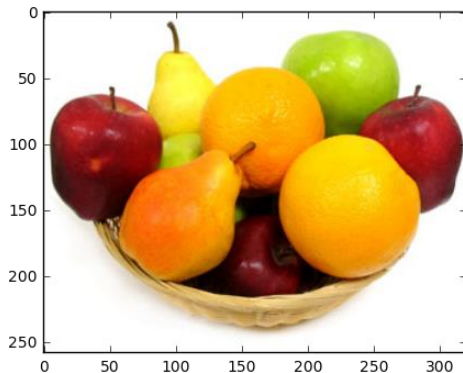
# FRAMEWORKS (CORE COMPONENTS)

Componentes fundamentais de qualquer **DL framework**:

- ❑ **O objecto Tensor**
- ❑ **Operações sobre o objecto Tensor**
- ❑ **Computação sobre grafos** e optimizações
- ❑ Ferramentas de **diferenciação automática**
- ❑ Extensões **BLAS** / **cuBLAS** e **cuDNN**

# ○ OBJECTO TENSOR

No cerne de qualquer **framework** está o objecto **Tensor**. Um tensor é uma generalização de uma matriz de n-dimensões (do género dos **ndarrays** do **numpy**). Por exemplo consideremos uma imagem Bitmap a cores (**RGB**), de dimensão 258 x 320 (altura x largura). Trata-se de um tensor 3-dimensional (altura, largura, canais de cores).



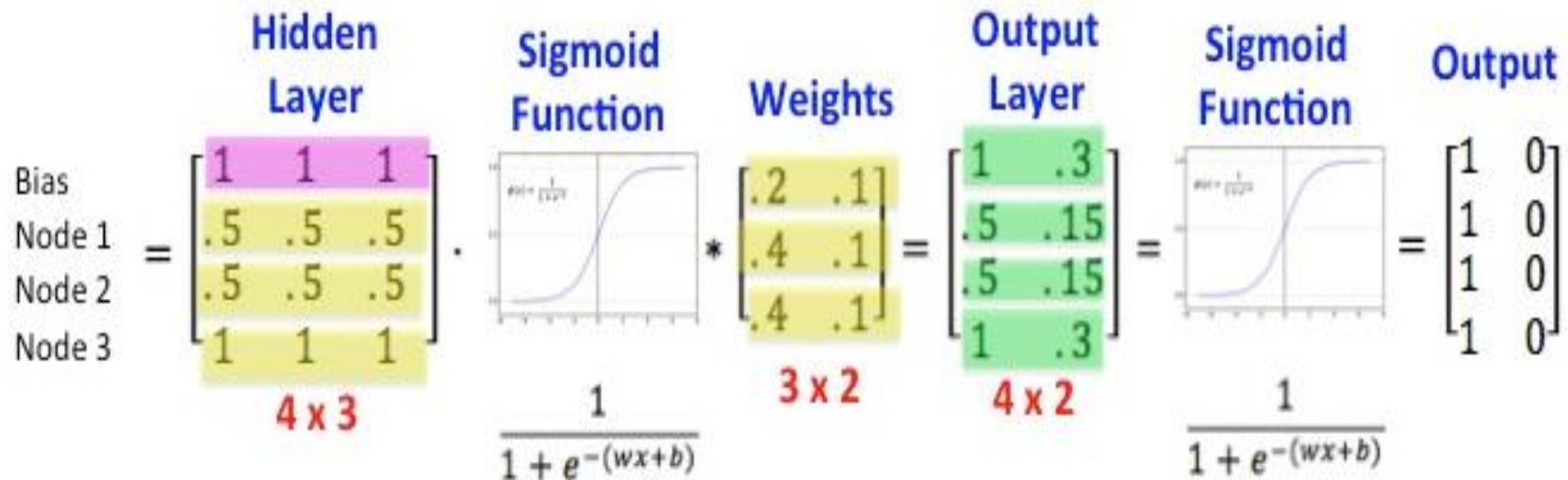
	0	1	2	3	4	5	6	7	8	9	...	310	311	312	313	314	315	316	317	318	319
0	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	...	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0, 1.0]	[1.0, 1.0,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

Analogamente um conjunto de 100 imagens pode ser representado por um tensor 4-dimensional (id da imagem, altura, largura, canais de cores).



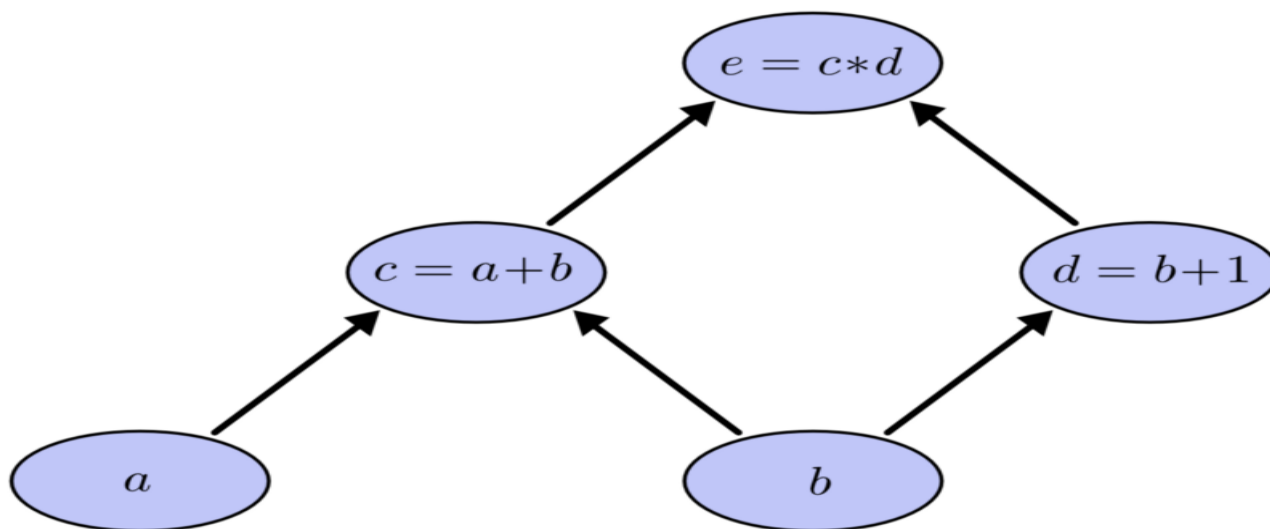
# OPERAÇÕES SOBRE O OBJECTO TENSOR

Podemos olhar para uma rede neuronal como sendo uma serie de operações efectuadas no tensor de entrada para produzir um resultado. A aprendizagem é conseguida pela correção dos erros entre o resultado obtido e o esperado. Estas operações vão desde simples multiplicação de matrizes até operações mais complicadas como **convolutions**, **pooling** ou **LSTMs**.



# COMPUTAÇÃO SOBRE GRAFOS

Até agora teríamos de considerar classes para representar tensores e operações sobre os mesmos. O poder das redes neurais depende da sua capacidade de encadear múltiplas operações de modo evidenciar as suas propriedades de aprendizagem. Torna-se assim necessário dotar o *framework* da capacidade de computação sobre grafos e sua otimização.



# DIFERENCIAÇÃO AUTOMÁTICA

Outra das vantagens de ter a capacidade de computação sobre grafos reside no facto de tornar o cálculo dos gradientes usados na fase de aprendizagem num processo de computação modular e simples. Isto graças à regra da cadeia no calculo diferencial que permite fazer os cálculos de uma composição de funções de um modo sistemático. Considerando as redes neuronais como uma composição de não linearidades que dão origem a funções mais complexas, a diferenciação destas funções pode ser vista como a travessia de um grafo da saída até à entrada.

A **Symbolic Differentiation** ou **Automatic Differentiation** é um modo programável em que os gradientes podem ser calculados usando computação sobre grafos.

# EXTENSÕES BLAS / CUBLAS E CUDNN

Embora estas extensões não sejam obrigatórias para termos um *framework* completamente funcional, o problema surge pelo facto de normalmente a implementação ser feita numa linguagem de programação de alto nível (**Java** / **Python** / **Lua**), levando inerentemente a um limite na velocidade de processamento que se consegue. Isto acontece porque até na mais simples das operações uma linguagem de alto nível demora sempre mais (ciclos de CPU) do que quando executada numa linguagem de baixo nível.

**BLAS** ou **B**asic **L**inear **A**lgebra **S**ubprograms são uma colecção de funções optimizadas para processamento de matrizes, inicialmente escritas em Fortran. Podem ser utilizadas para efectuar operações sobre os tensores conseguindo-se velocidades de processamento substancialmente superiores. Existem muitas outras alternativas tais como o **Intel MKL** ou o **ATLAS** que executam funções similares.

O pacote **BLAS** optimiza assumindo que as instruções serão executadas num **CPU**. No caso do deep learning este não é geralmente o caso e o **BLAS** poderá não aproveitar completamente o paralelismo disponibilizado pelos **GPUs**. Para resolver esta questão a **NVIDIA** desenvolveu o **cuBLAS** o qual está optimizado para **GPUs**. Vem agora incluído no **CUDA toolkit**. O **cuDNN** é uma biblioteca construída utilizando o **cuBLAS** que disponibiliza operações optimizadas especificamente para redes neuronais.

Para **AMD** existem bibliotecas semelhantes (optou por open source).

# FRAMEWORKS

**Theano** – é uma biblioteca Python que permite definir, otimizar e calcular expressões matemáticas com matrizes multidimensionais eficientemente. Trabalha com GPUs e executa eficientemente cálculos diferenciais. (University of Montreal's lab, MILA )

**Lasagne** – biblioteca light para construção e treino de redes neurais usando **Theano**.

**Blocks** – um framework baseado em theano para construção e treino de redes neurais.

**TensorFlow** – uma biblioteca *open source* para computação numérica utilizando grafos. (Google Brain team)

**Keras** – biblioteca Deep learning para Python. Convnets, corre em Theano ou TensorFlow.

**MXNet** – framework deep learning desenhado para eficiência e flexibilidade. (Amazon)

**PyTorch** - Tensors e redes neurais flexíveis com forte suporte de GPUs. (Facebook Artificial Intelligence Research team (FAIR))

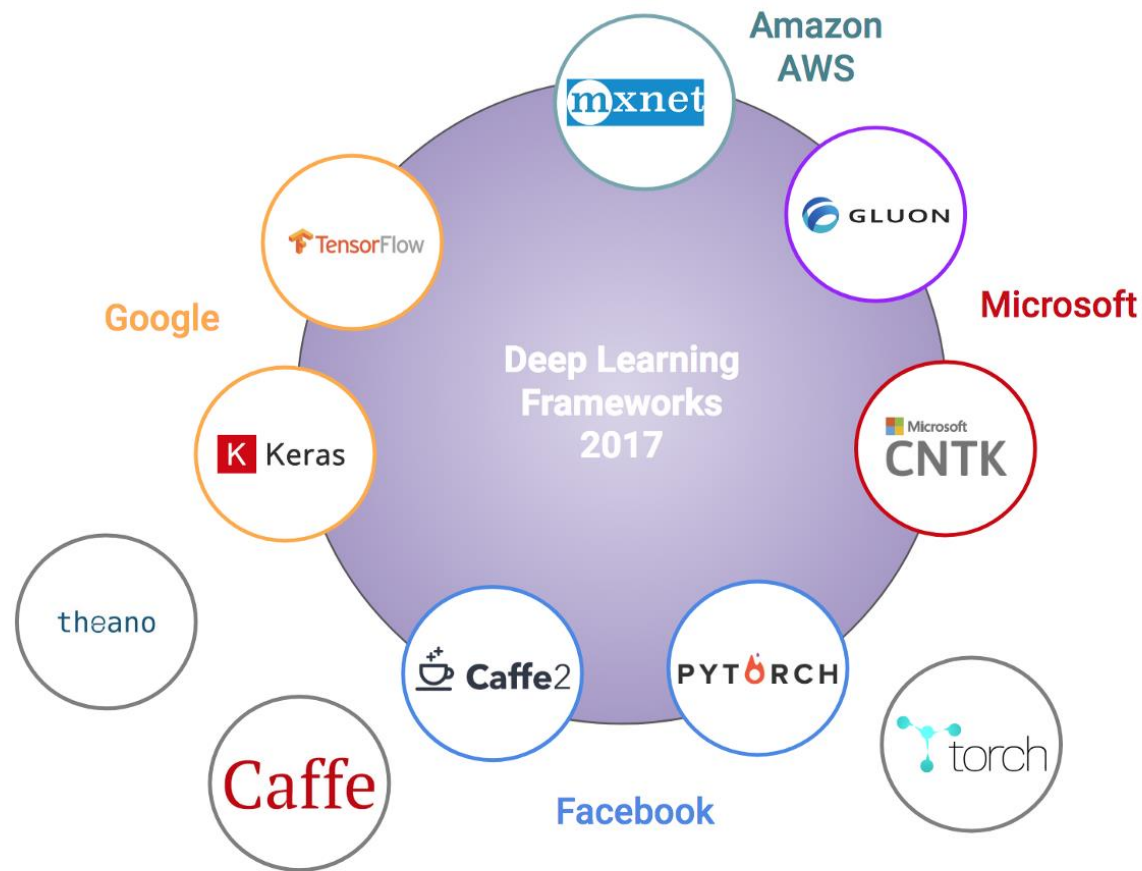
**Torch** - (Ronan Collobert)

**Caffe** - (Berkeley Vision and Learning Center)

**CNTK** - (Microsoft)

**Deeplearning4j** - (SkyMind)

# FRAMEWORKS



# CONVOLUTIONAL NEURAL NETS

## Convolution

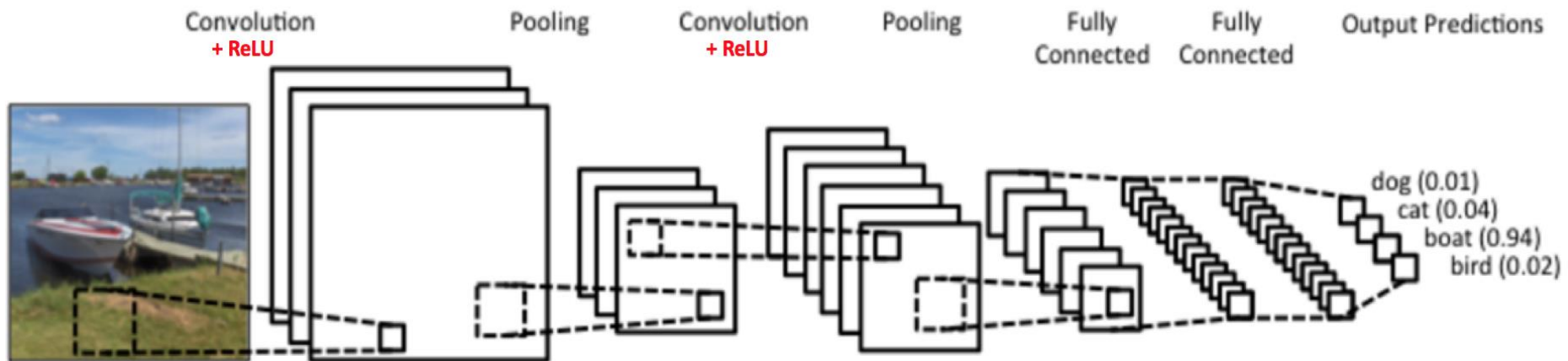
A convolução pode ser vista como uma janela deslizando em cima de uma matriz que representa uma imagem. Consegue-se assim mimetizar o funcionamento do córtex visual biológico.

**Convolutional Neural Networks (ConvNets or CNNs)** são uma categoria de redes neuronais que se têm afirmado muito eficientes em áreas como a visão por computador. As **ConvNets** têm tido sucesso em reconhecimento facial, de objectos e sinais de trânsito além de suportar visão em robôs e carros autónomos.

# CNN ARCHITECTURE

Numa CNN temos geralmente as seguintes entidades: **Input** , **Filters (or Kernels)**, **Convolutional Layer**, **Activation Layer**, **Pooling Layer** e **Batch Normalization layer**. A combinação destas camadas em diferentes permutações e alguma parametrização dá-nos diferentes arquitecturas deep learning.

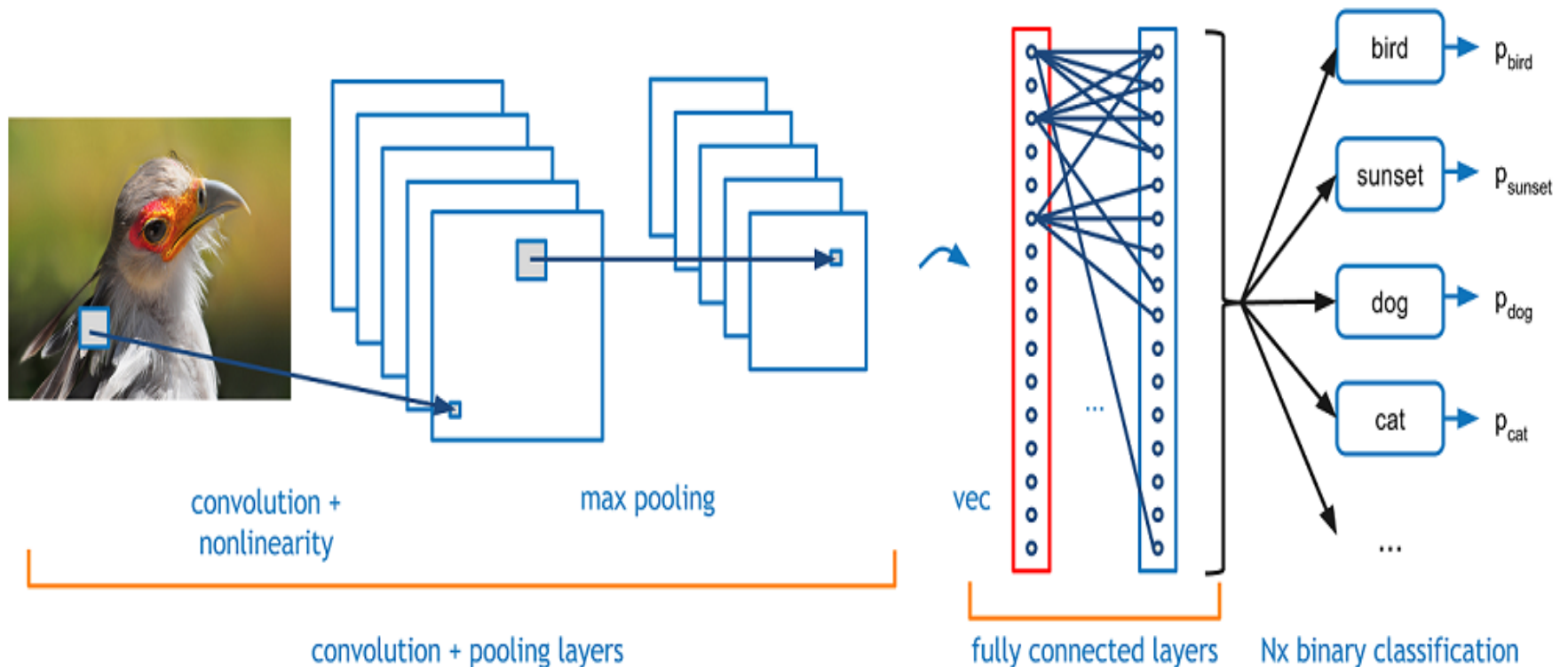
Na figura, a imagem é filtrada por 4  $5 \times 5$  **convolutional kernels** que dão origem a 4 **feature maps**, estes **feature maps** são reduzidos por **max pooling**. A próxima camada aplica 10  $5 \times 5$  **convolutional kernels** a estas imagens e torna-se a fazer uma nova redução. A camada final é completamente ligada onde todas as características geradas são combinadas e utilizadas na classificação (**logistic regression**).





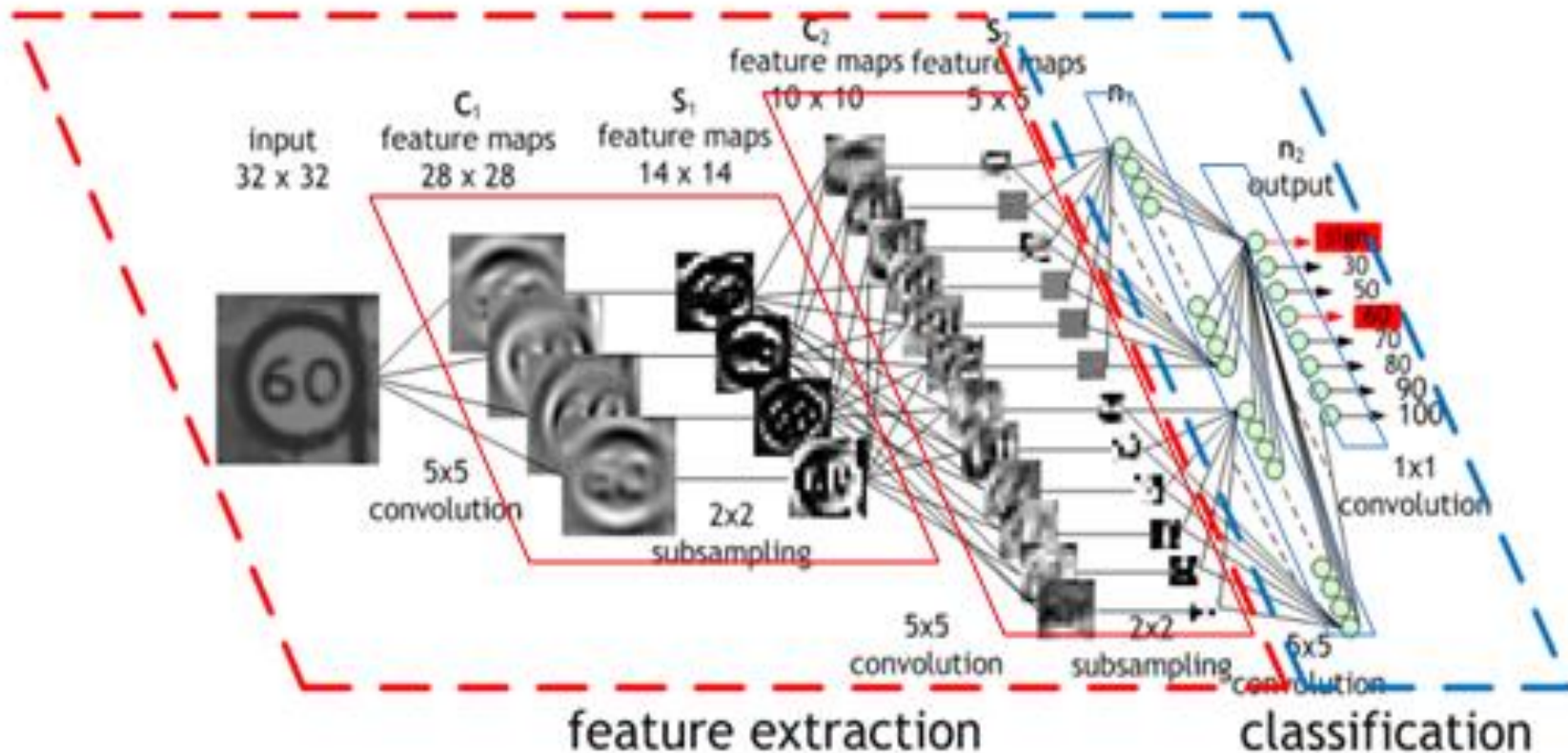
# CNN ARCHITECTURE

Numa CNN temos geralmente as seguintes entidades: **Input**, **Filters (or Kernels)**, **Convolutional Layer**, **Activation Layer**, **Pooling Layer** e **Batch Normalization layer**. A combinação destas camadas em diferentes permutações e alguma parametrização dá-nos diferentes arquitecturas deep learning.



# CNN ARCHITECTURE

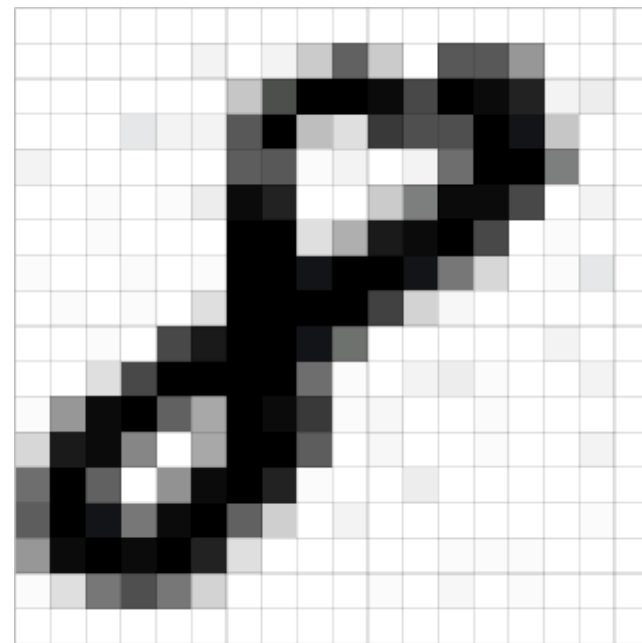
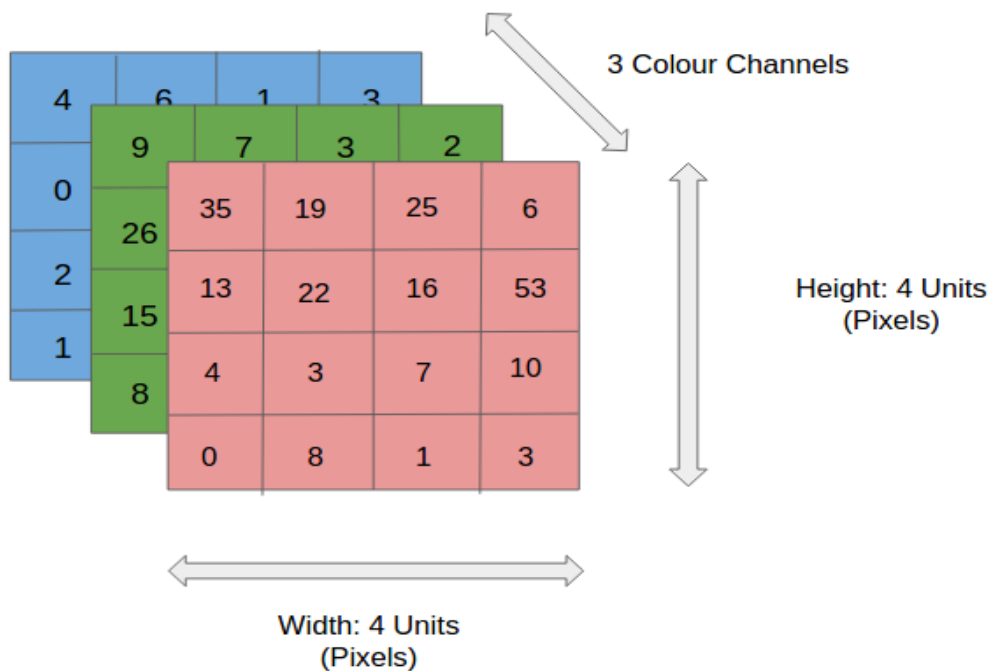
Numa CNN temos geralmente as seguintes entidades: **Input**, **Filters (or Kernels)**, **Convolutional Layer**, **Activation Layer**, **Pooling Layer** e **Batch Normalization layer**. A combinação destas camadas em diferentes permutações e alguma parametrização dá-nos diferentes arquitecturas deep learning.



# INPUT LAYER - TENSOR

## Input Layer

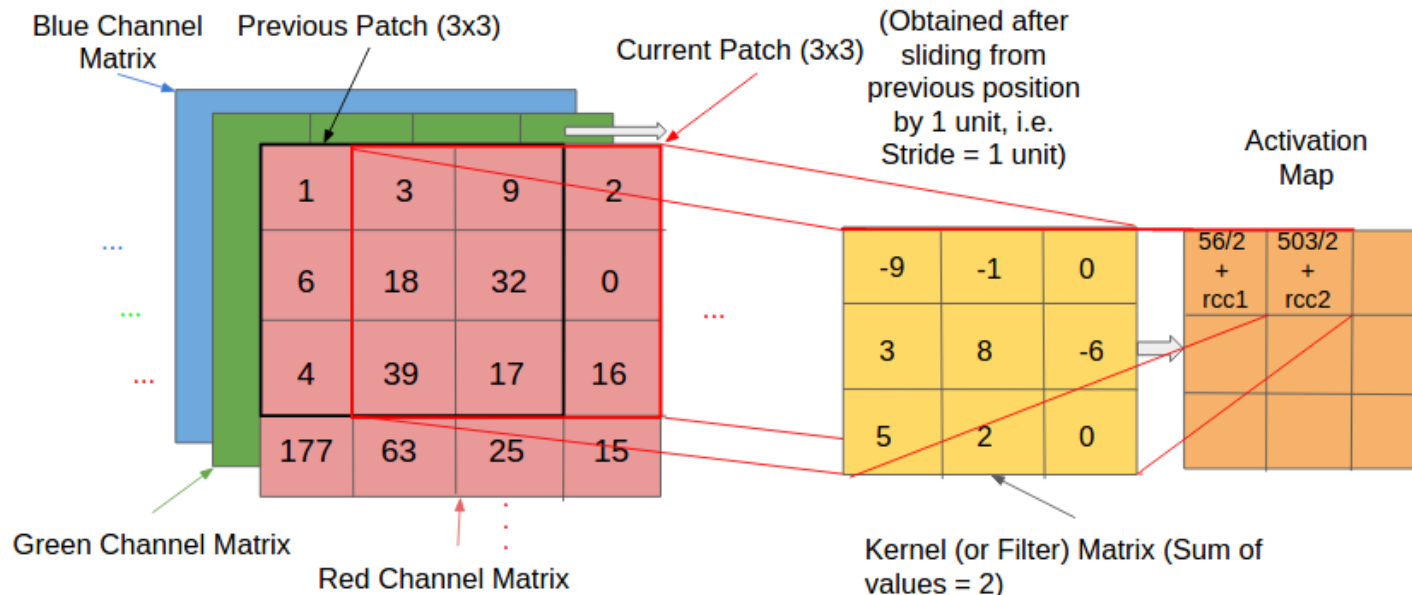
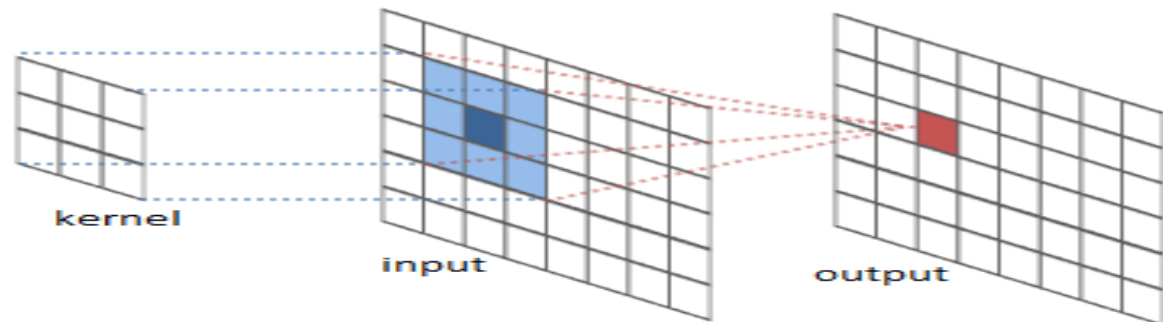
A entrada habitual para uma CNN é uma matriz in-dimensional ou seja um tensor. No caso de uma imagem a cores temos geralmente 3 dimensões - largura, altura e canais de cores.



# FILTERS OU KERNELS

## Filters ou Kernels

Um filtro é aplicado por deslizamento a todas as posições de um tensor dando como resultado um novo valor (soma ponderada) dos valores aos quais se aplica.



# CONVOLUTIONAL LAYER

## Convolutional Layer

Trata-se da camada resultante do produto escalar do tensor de entrada pelo filtro (chamados **kernel**, **Patch Filters** ou **Convolution Kernels**). As matrizes resultantes denomina-se normalmente de **Feature Map**, **Convolved Feature** ou **convolutional matrix**,.

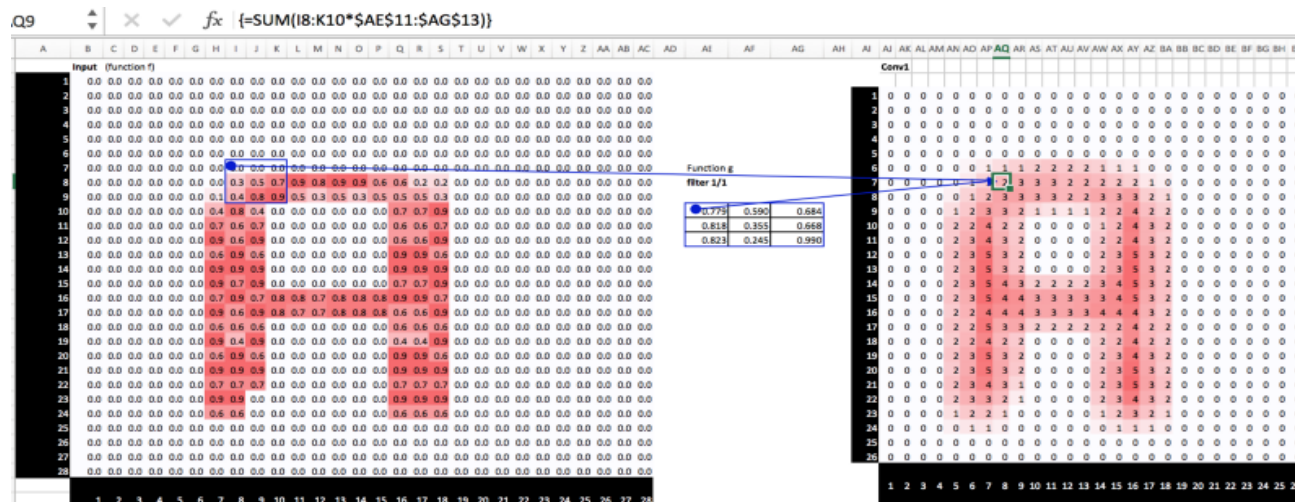
The diagram shows the multiplication of two 5x5 matrices. The first matrix is:

35	40	41	45	50
40	40	42	46	52
42	46	50	55	55
48	52	56	58	60
56	60	65	70	75

The second matrix is:

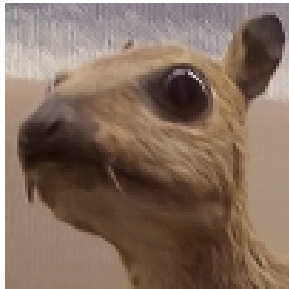
	0	1	0	
	0	0	0	
	0	0	0	

The result is a 5x5 matrix with the value 42 highlighted in a red box, indicating the result of the multiplication of the 3rd row of the first matrix and the 2nd column of the second matrix.



# CONVOLUTIONAL LAYER

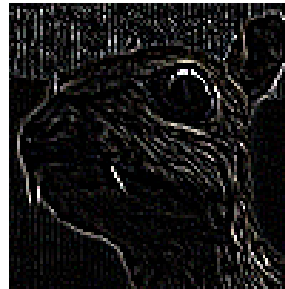
Input image



Convolution  
Kernel

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

Feature map



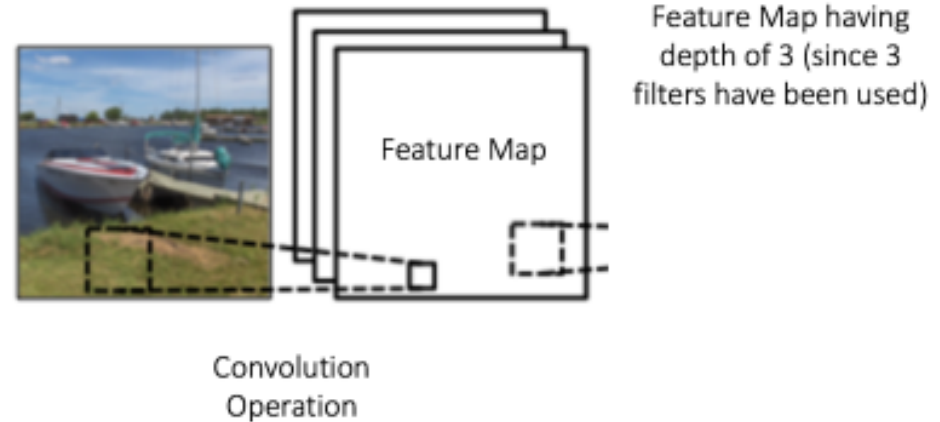
Na prática a **CNN** aprende os valores destes filtros por si só durante o processo de treino. Embora seja necessário especificar parâmetros tais como “quantidade de filtros”, “dimensão do filtro”, “arquitetura da rede”, etc. antes do início do processo de treino. Quanto mais filtros tivermos, mais características da imagem são extraídas e melhor se torna a rede no reconhecimento de padrões na imagem.

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 LAYER

## 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 este 3 **feature maps** como uma stack de matrizes 2D. A **depth** desta camada é portanto de 3.



## Stride (hyperparameter)

É o número de pixels 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 pixels 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.

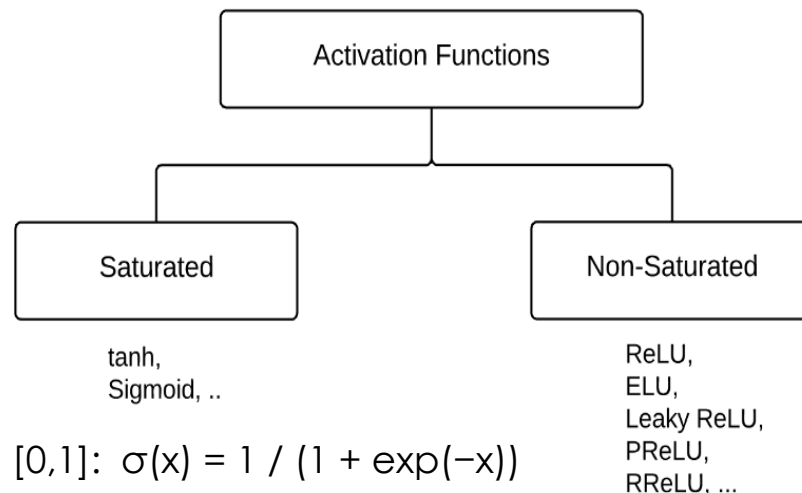
**Sigmoid** - Recebe um valor real e produz um valor entre  $[0,1]$ :  $\sigma(x) = 1 / (1 + \exp(-x))$

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

**ReLU** - ReLU significa **Rectified Linear Unit**. **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** - **Exponential Linear Units** 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.





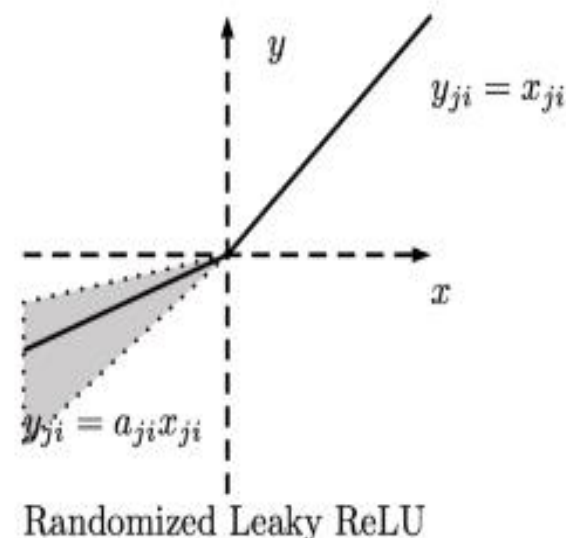
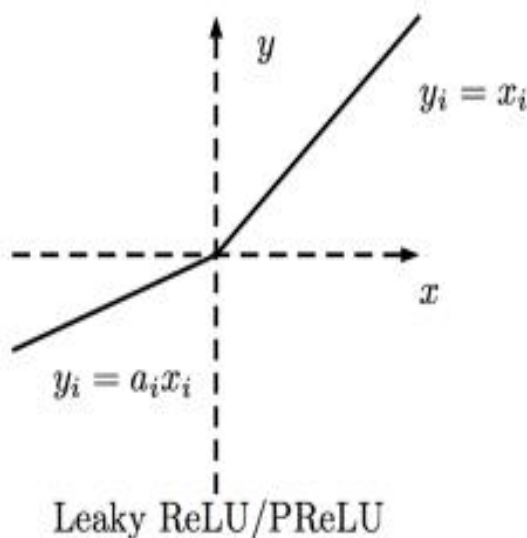
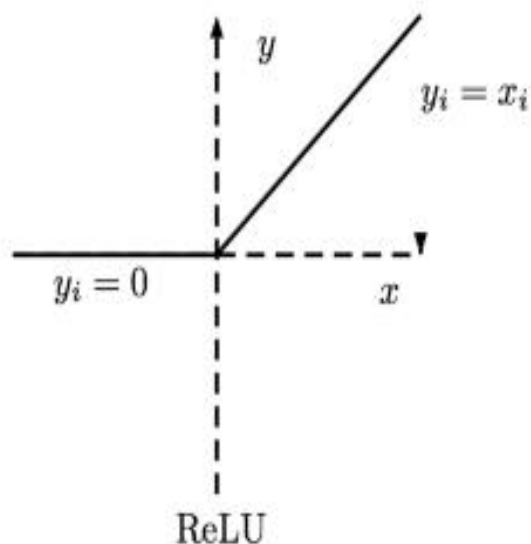
# ACTIVATION LAYER









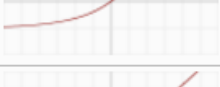
## Parametric Rectified Linear Unit (PReLU)

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 é foi o factor chave na ultrapassagem do desempenho dos humanos na classificação da ImageNet.

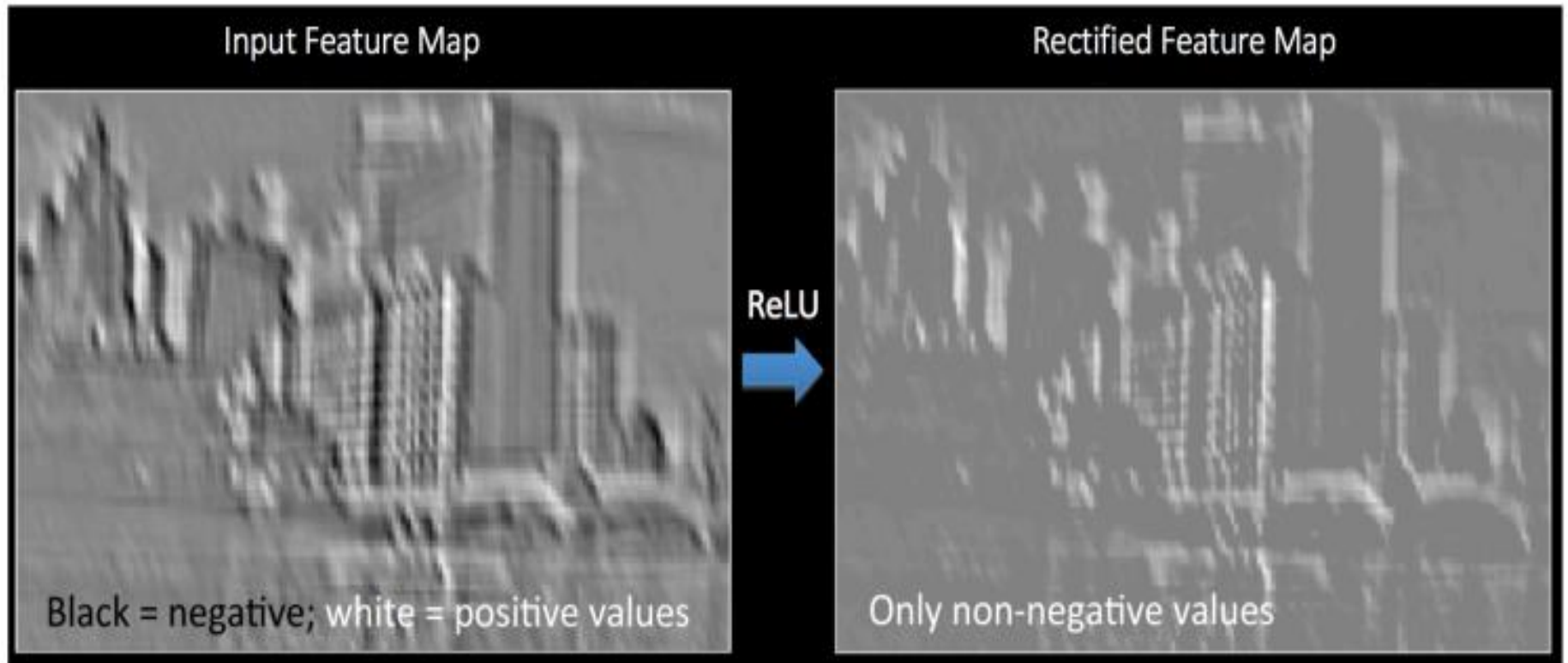
## Randomized Leaky Rectified Linear Unit (RReLU)

É também uma variante da Leaky ReLU. Na **RReLU**, o declive da parte negativa é aleatório numa parte do treino, e depois fixa no teste. A ideia é que no treino o declive  $a_{ji}$  é um número aleatório retirado de uma distribuição uniforme de  $U(l,u)$ .



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)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parametric 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



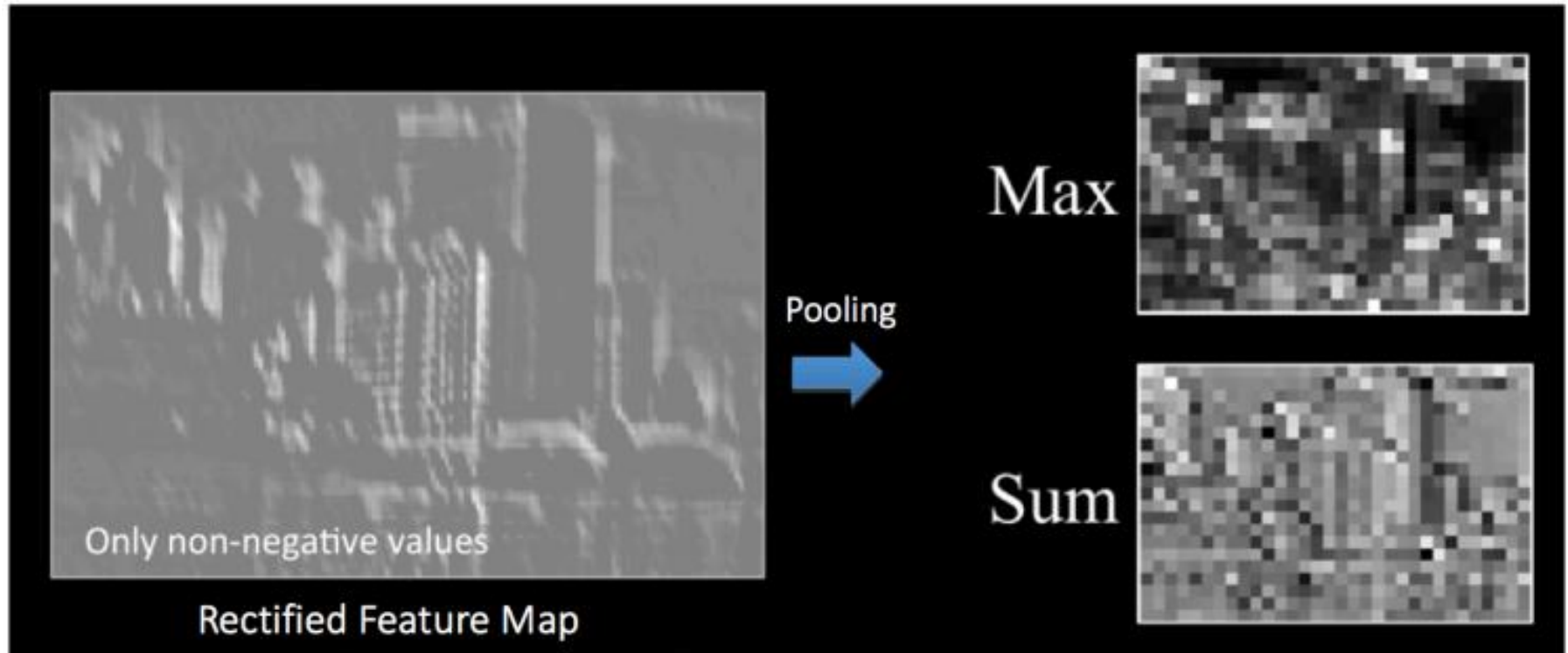
## Noisy Activation functions

Trata-se de funções extendidas de modo a incluir ruído Gaussiano.

# PPOOLING LAYER

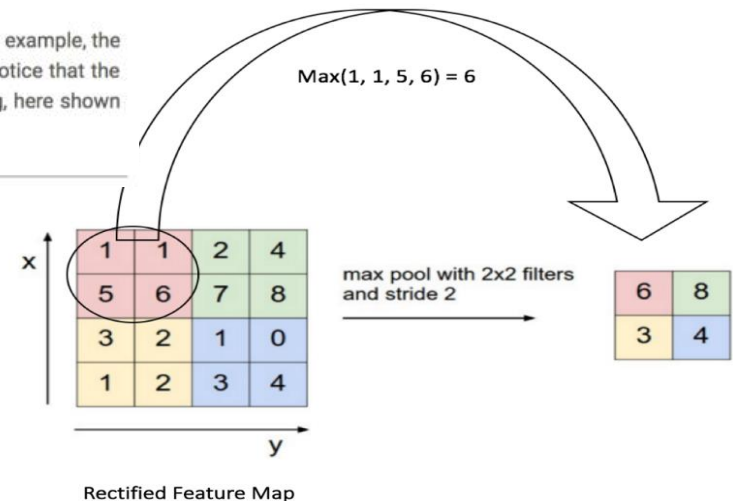
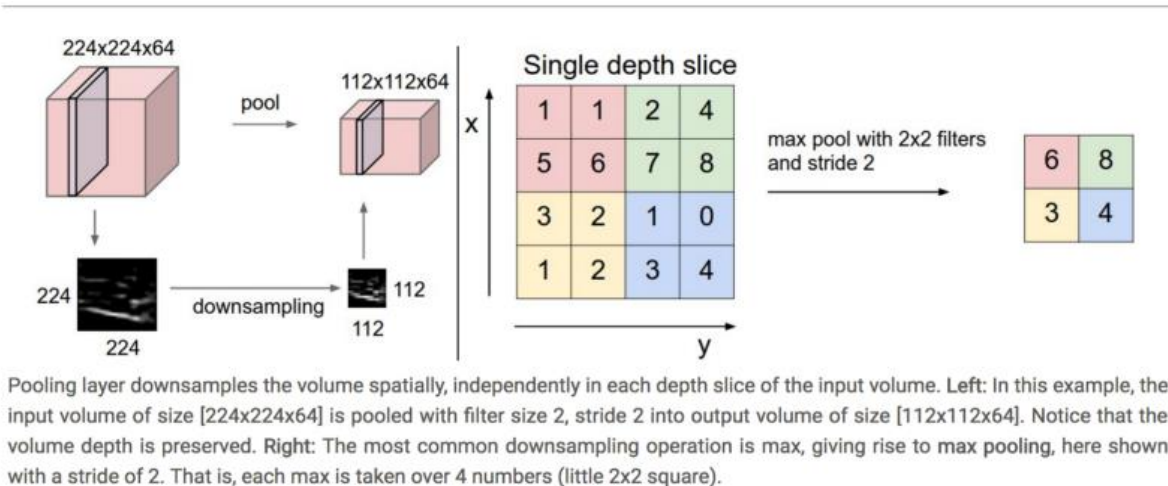
## Pooling Layer

O objectivo da Pooling layer é diminuir progressivamente a dimensão espacial da matriz de modo a reduzir a quantidade de parâmetros e volume de computação na rede, controlando também o overfitting. A Pooling Layer opera independentemente em cada depth slice (feature map) alterando o seu tamanho utilizando a operação MAX ou Average.



# POOLING LAYER

Uma das configurações mais usuais para a **pooling layer** são filtros de tamanho 2x2 aplicados com **stride** de 2 reduzindo cada **feature map** para metade em cada dimensão ignorando assim 75% das activaões. Cada operação **MAX** irá assim dar como resultado o máximo de 4 números. A dimensão **depth** mantém-se inalterada.



# BATCH NORMALIZATION LAYER

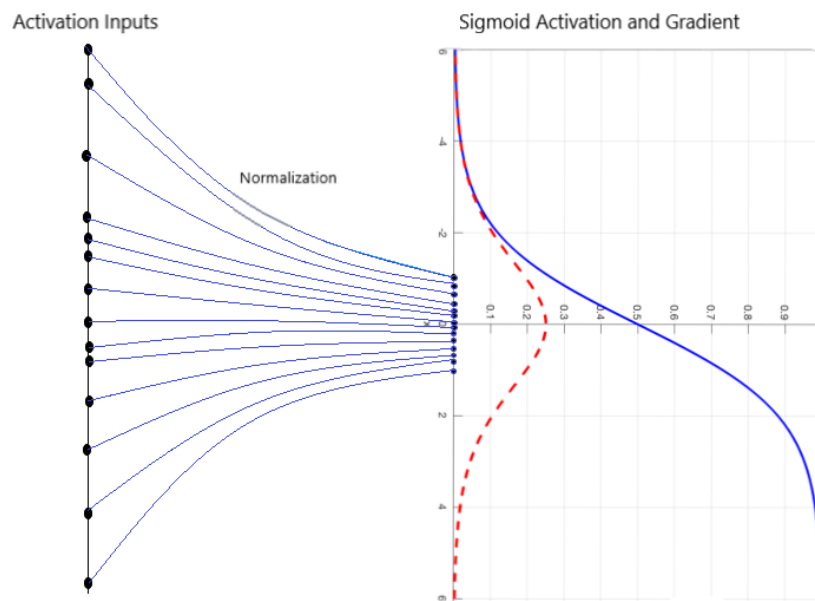
## Batch Normalization layer

Trata-se de um modo efectivo de normalizar cada camada intermédia incluindo os pesos e as funções de activação. Existem 2 vantagens principais na utilização da **batchnorm**:

- ❑ Acrescentando **batchnorm** a um modelo pode resultar em **10x ou mais na melhoria da velocidade de treino**.
- ❑ Como a **normalization** reduz drasticamente a possibilidade de um pequeno número de **outlying inputs** influenciar exageradamente o treino, tende também em reduzir o **overfitting**.

**Normalization** é o processo onde a um conjunto de dados se subtrai a cada elemento o valor médio deste conjunto e divide-se o resultado pelo desvio padrão do conjunto de dados. Ao fazer isto colocamos todos os valores na mesma escala.

Geralmente com as imagens não nos preocupamos com a divisão pelo desvio padrão, subtraindo simplesmente a média.



# FULLY CONNECTED LAYER

## Fully Connected layer

Esta camada final é tradicionalmente uma Multi Layer Perceptron (MLP) que utiliza a função de activação softmax na camada de saída. O termo “Fully Connected” implica que cada neurónio na camada anterior está ligada a todos os neurónios da camada seguinte. A função softmax function é uma generalização da função logística que “esmaga” um vector  $K$ -dimensional, de valores reais arbitrários para valores entre 0 e 1 que totalizam o somatório de 1 (ou seja probabilidades).

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

# TREINO

## Processo de treino de uma CNN:

- 1: inicializar todos os filtros e parametros/pesos com valores aleatorios
- 2: A rede pega numa imagem de treino como entrada, percorre a forward propagation (operações convolution, ReLU e pooling além de forward propagation na camada Fully Connected) e apresenta na saída as probabilidades para cada classe. Obs. Uma vez que no primeiro caso os pesos eram aleatórios as probabilidades também serão aleatórias.
- 3: Calcular o erro total na camada de saída: **Total Error** =  $\sum \frac{1}{2} (\text{target probability} - \text{output probability})$
- 4: Utilizar Backpropagation para calcular os *gradientes* do erro relativamente a todos os pesos na rede e utilizar *gradient descent* para ajustar todos os filtros/pesos e valores de parâmetros de modo a minimizar o erro de saída.

Os pesos são ajustados em proporção à sua contribuição para o erro total. Se a mesma imagem fosse processada as probabilidades iriam estar praticamente certas. Isto significaria que a rede teria aprendido a classificar aquele caso em particular. Parâmetros como número de filtros, dimensão dos filtros, arquitectura da rede etc. foram predefinidos e não são alterados durante o treino. Somente os valores da matriz dos filtros e pesos das ligações é que são ajustados.
- 5: Repetir os passos 2-4 com todas as imagens do conjunto de treino.



# TREINO “A BETTER MODEL”

## Underfitting

Trata-se de um modelo ao qual lhe falta a complexidade necessária para captar correctamente a complexidade inerente ao problema que se pretende resolver. **Consegue-se reconhecer esta situação quando o erro é demasiado grande, tanto nos casos de treino como nos casos de teste (validação).**

## Overfitting

Trata-se de um modelo que está a utilizar demasiados parâmetros e foi treinado demasiado. Concretamente, aprendeu a identificar exactamente cada caso do conjunto de treino, ficando de tal modo específico que não consegue generalizar para imagens semelhantes. **Consegue-se reconhecer esta situação quando o erro nos casos de treino é muito menor que nos casos de teste (validação).**

Genericamente existem várias medidas que se podem tomar para reduzir o **overfitting**:

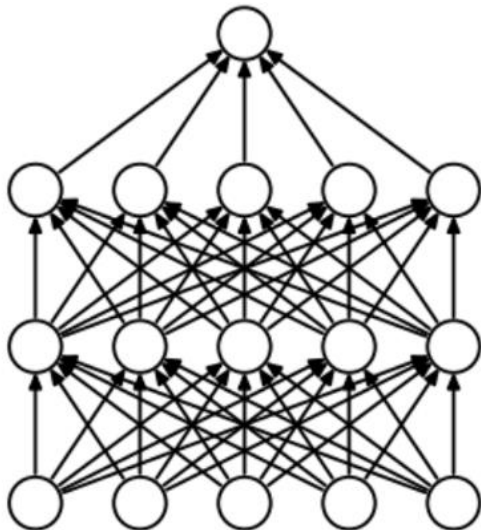
- ☐ Acrescentar mais casos ao conjunto de treino
- ☐ Utilizar arquitecturas que demonstraram generalizar bem
- ☐ Reduzir a complexidade da arquitectura da rede
- ☐ Utilizar **data augmentation**
- ☐ Acrescentar normalização (**Batch Normalization layer**)
- ☐ Acrescentar **Dropout (Dropout layer)**

**ATENÇÃO:** A maior parte do tempo de computação é gasta nas **convolutional layers**, enquanto que a maior parte da memória é gasta nas **dense layers**.

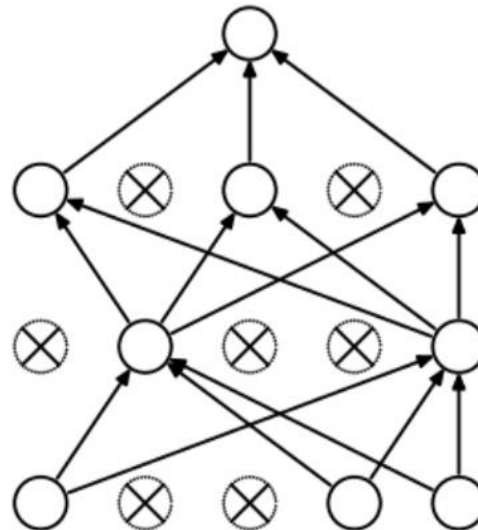
# TREINO “A BETTER MODEL”

## Dropout

O **Dropout**, (só acontece durante o treino) ocorre depois da activação e coloca aleatoriamente activações a zero. Corresponde a apagar partes da rede neuronal. Este apagar da informação que se foi aprendendo vai permitir evitar o **overfitting**, permitindo à rede generalizar. A taxa de **dropout** é especificada no início e depende do problema e dos resultados que se vão conseguindo no treino.



(a) Standard Neural Net



(b) After applying dropout.

# TREINO “A BETTER MODEL”

## Data Augmentation

Foi referido que o **overfitting** é o resultado d aprendizagem exagerada da especificidade dos casos de treino, não conseguindo generalizar para imagens semelhantes.

**Data augmentation** simplesmente provoca alterações em cada **batch** das nossas imagens. Faz isto através de operações como **flipping**, alterações de **hues**, **stretching**, **shearing** (distorções), **rotation**, etc. fazendo isto de um modo que faça sentido (por exemplo, não faz sentido fazer o flip vertical de uma imagem com um cão). No **Keras** basta criar um **data-augmentation batch generator**. Infelizmente não há maneira de saber à partida a melhor parametrização, só experimentando.



# CONCEITOS

## Epoch

Uma passagem para a frente e uma passagem para trás em todos os casos de treino.

## batch size

Numero de casos de treino utilizados em cada passagem para a frente e para trás . Quanto maior o batch size, mais memória será necessária.

## iterations

Numero de passagens, cada passagem utilizando o número de casos do **batch size**. Ter em atenção que, uma passagem = uma passagem para a frente + uma passagem para trás.

Exemplo: Se tivermos 1000 casos de treino, e o **batch size** for 500, então serão necessárias 2 **iterations** para completar 1 **epoch**.

## learning rate

Parâmetro escolhido pelo programador que vai determinar o tamanho dos passos a dar na actualização dos pesos da rede no processo de treino. Um valor maior de **learning rate** pode significar menos tempo para o modelo convergir para os pesos ideais mas também pode significar que nunca chegam ao valor ideal devido a falta de precisão.

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

$w$  = Weight

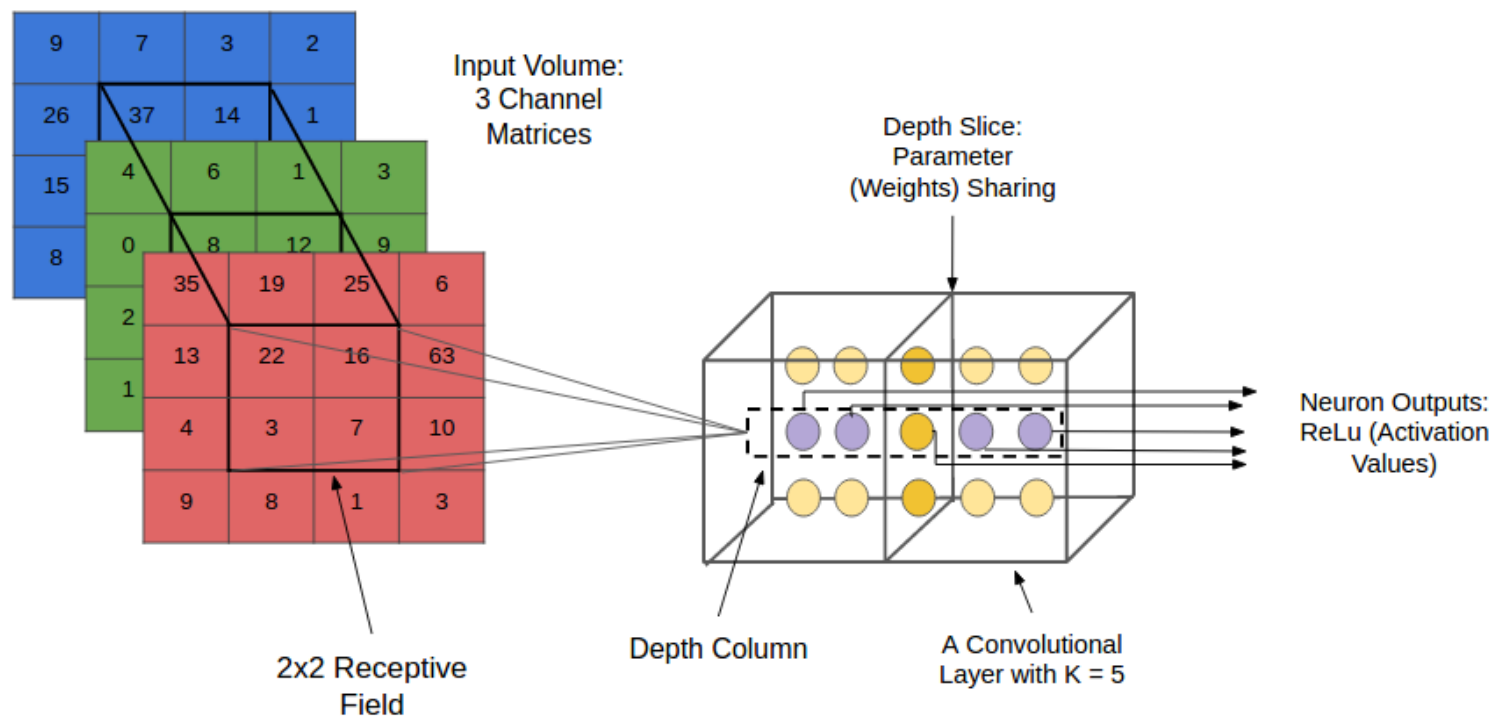
$w_i$  = Initial Weight

$\eta$  = Learning Rate

# CONCEITOS

## Receptive Field (equivalente ao Filter size)

É impraticável ligar todos os neurónios a todas as regiões do volume de entrada. Seriam demasiados pesos a treinar, resultando numa enorme complexidade computacional. Assim, em vez de ligar cada neurónio a todos os pixéis possíveis, especificamos uma região bidimensional chamada '**receptive field**' (digamos de tamanho  $2 \times 2$  unidades) que se estende a toda a **depth** da entrada ( $2 \times 2 \times 3$  no caso de 3 canais de cor). Esse conjunto de pixéis são totalmente ligados à entrada da rede neuronal. É sobre estas pequenas regiões que a camada da rede opera.



# CNN - BOAS PRATICAS

**Input Receptive Field Dimensions** – Por omissão é 2D para imagens, mas poderá ser 1D para por exemplo palavra numa frase ou 3D para vídeo que acrescenta a dimensão temporal.

**Receptive Field Size** - O **patch** deve ser o mais pequeno possível, mas suficientemente grande para detectar características nos dados. É vulgar utilizar 3×3 em imagens pequenas e 5×5 ou 7×7 ou até mais em imagens de grande dimensão. Colocando a 2×2 com o **stride** a 2 consegue-se descartar 75% das activaões vindas da saída da camada anterior.

**Stride Width** – Utilizar o valor por omissão de 1. É mais fácil de perceber e não se precisa de andar a fazer **padding** por causa da saída fora da borda da imagem. Em imagens grandes poderá utilizar-se o 2.

**Number of Filters** – Os filtros é que detectam as características (**feature**). Geralmente usam-se menos filtros na camada de entrada e mais filtros nas camadas mais profundas.

**Padding** – Colocar a zero (zero **padding**).

**Pooling**: É um processo destrutivo ou de generalização para reduzir o **overfitting**.

**Data Preparation**: Considerar sempre a normalização dos dados de entrada, tanto na dimensão das imagens como nos valores de pixéis.

**Pattern Architecture** – É normal padronizar as camadas na arquitectura da rede. Pode ser por exemplo uma ou duas camadas de convolução seguidas de uma camada de **pooling**. Este padrão pode depois ser repetido uma ou mais vezes. As camadas totalmente ligadas só são utilizadas no final e poderão ter mais de que uma camada.

**Dropout** - As **CNNs** têm o habito de **overfitting**, mesmo com **pooling layers**. **Dropout** tanto pode ser utilizado entre camadas completamente ligadas como depois de **pooling layers**.

# REFERÊNCIAS

**Matthew Mayo**, Kdnuggets

[www.datasciencecentral.com/profiles/blogs/matrix-multiplication-in-neural-networks](http://www.datasciencecentral.com/profiles/blogs/matrix-multiplication-in-neural-networks)

<https://colah.github.io/posts/2015-08-Backprop/>

<http://www.kdnuggets.com/2017/02/python-deep-learning-frameworks-overview.html>

<https://docs.gimp.org/en/plugin-convmatrix.html>

(Russakovsky et al., 2015).

<http://cs231n.github.io/convolutional-networks/#pool>

<https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/>

# MÉTRICAS

## Confusion Matrix

$$accuracy = \frac{TP + TN}{TP + FP + FN + TN}$$

		Class	
		Tumor	Non-Tumor
Predicted as	Tumor	TP	FP
	Non-Tumor	FN	TN

Conversely, two measures that separately estimate a classifier's performance on different classes are precision and recall (often employed in biomedical and medical applications)

$$Precision = \frac{TP}{TP + FP} \quad (2); \quad Recall = \frac{TP}{TP + FN} \quad (3)$$

Recall is also called Sensitivity or True Positive Rate (TPR) and it is often used in tumor detection where it gives higher scores to classifiers that not only achieved the high number of true positives but also avoided false negatives, that is, that rarely failed to detect a true cancerous tumor. Recall is obtained by dividing the number of true positives by the sum of true positives and false negatives (equation 3). From equation 3, It is possible to realize that there are two ways to get a larger recall score. First, by either increasing the number of TP or by reducing the number of FN

Precision (or confidence), on the other hand, is often used on ML tasks where it's important to avoid false positives. In other words, the classifier may predict cases where not all true positive instances are detected but when the classifier does predict the positive class, it should provide high confidence that it's correct



# MÉTRICAS

$$F_1 \text{ Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2TP}{2TP + FN + FP}$$

However, when evaluating classifiers, it's often convenient to compute a quantity known as an F1 score, that combines precision and recall into a single number . Mathematically, this is based on the harmonic mean of precision and recall using this formula:

The  $F_1$  Score is also known as the Sørensen–Dice coefficient or Dice similarity coefficient (DSC). DSC also represents a spatial overlap index and reproducibility validation metric. The DSC value ranges from 0, indicating no spatial overlap between two sets of binary segmentation results, to 1, indicating a complete overlap