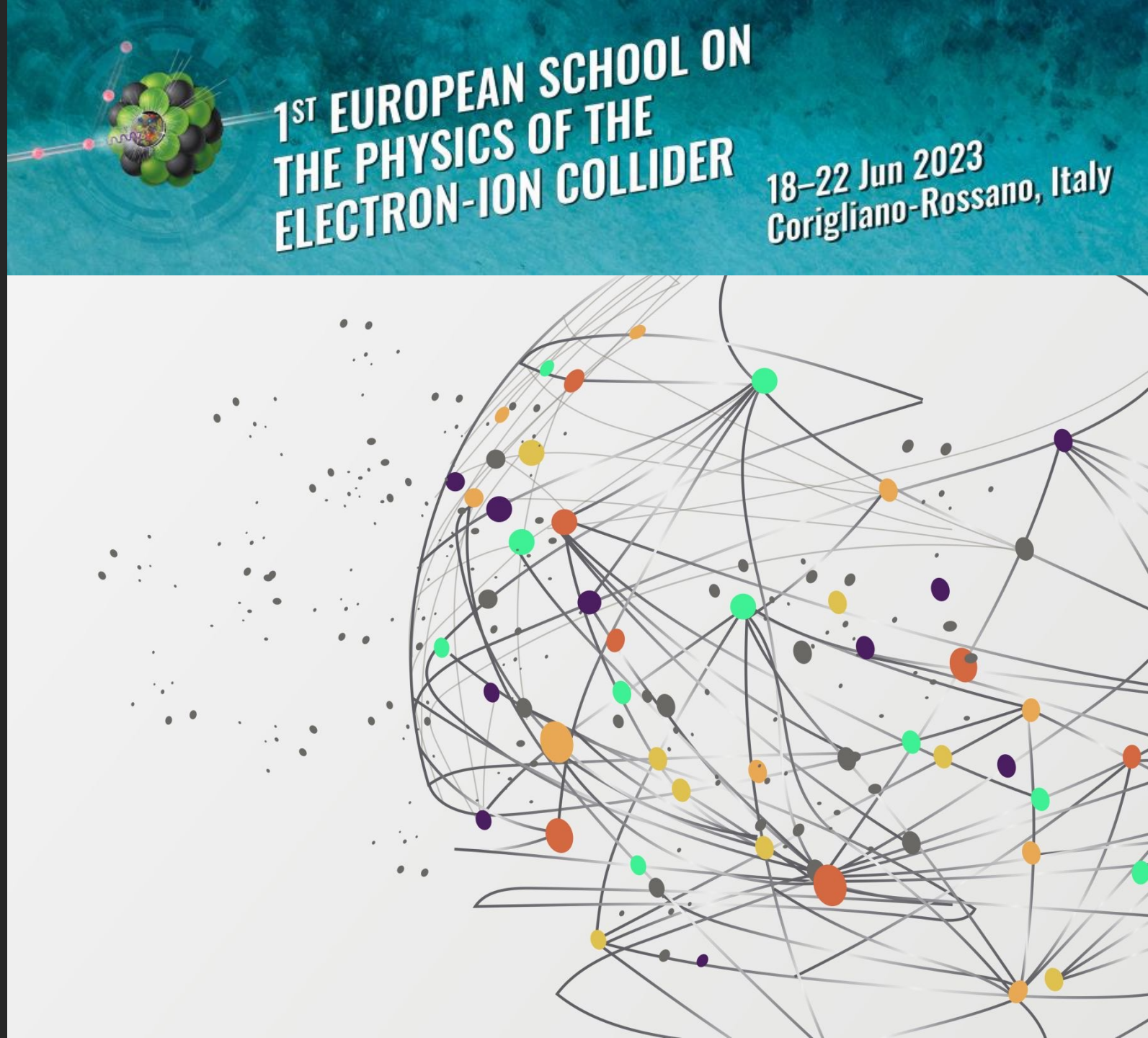
INTRODUCTION TO MACHINE LEARNING TECHNIQUES

G. MINIELLO – PH. D.

INFN - BARI



OVERVIEW

WHAT IS MACHINE LEARNING?

DIFFERENT TYPES OF MACHINE LEARNING

SUPERVISED MACHINE LEARNING AND CLASSIFICATION PROBLEMS

DATA PREPROCESSING, MODEL TRAINING AND EVALUATION

FROM THE MCP PERCEPTRON TO ARTIFICIAL NEURAL NETWORKS

FROM THE DEEP NEURAL NETWORKS TO THE CONVOLUTIONAL NEURAL NETWORKS

AI4EIC

ALL YOU NEED IS…DATA!

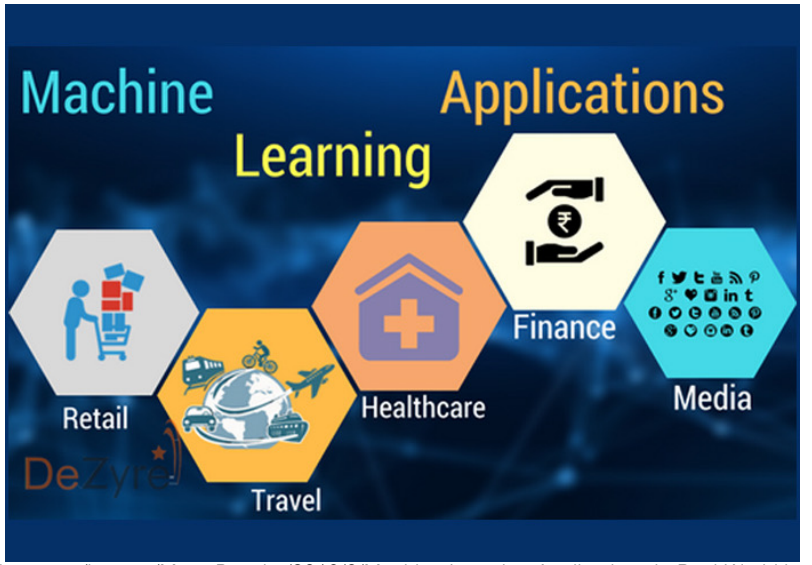
*Nowadays we have an abundance of **structured** and **unstructured data***

Machine Learning (ML) is a branch of **Artificial intelligence** (AI) and Computer Science which focuses on the use of data and algorithms to mimic the way that humans learn, gradually improving its performance.

Many applications in everyday life and in basic and applied science

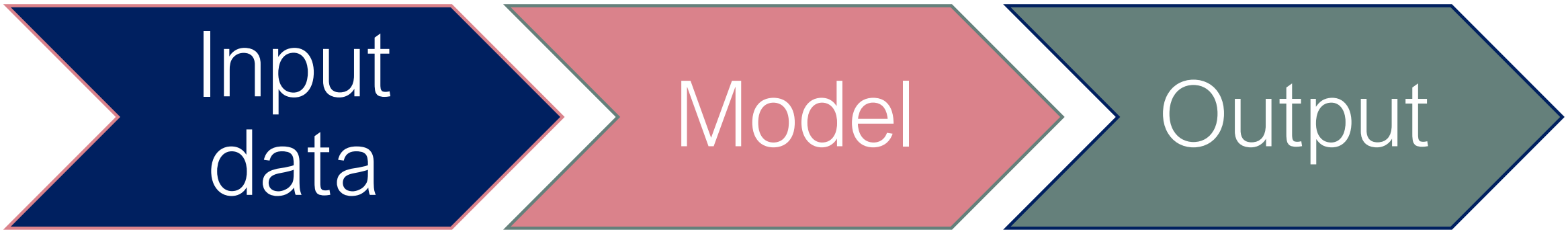
Structured data: categorized as *quantitative data*, highly organized and easily decipherable

Unstructured data: categorized as *qualitative data*, cannot be processed and analyzed via conventional data tools and methods



https://www.thetechnologyheadlines.com/images/Most_Popular/2019/8/Machine-Learning-Applications-in-Real-World.jpg

GENERAL ALGORITHM STRUCTURE IN MACHINE LEARNING

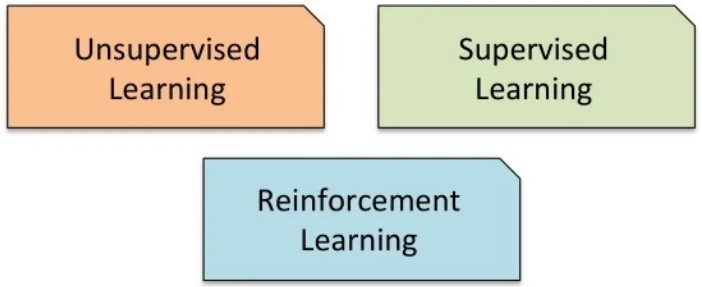


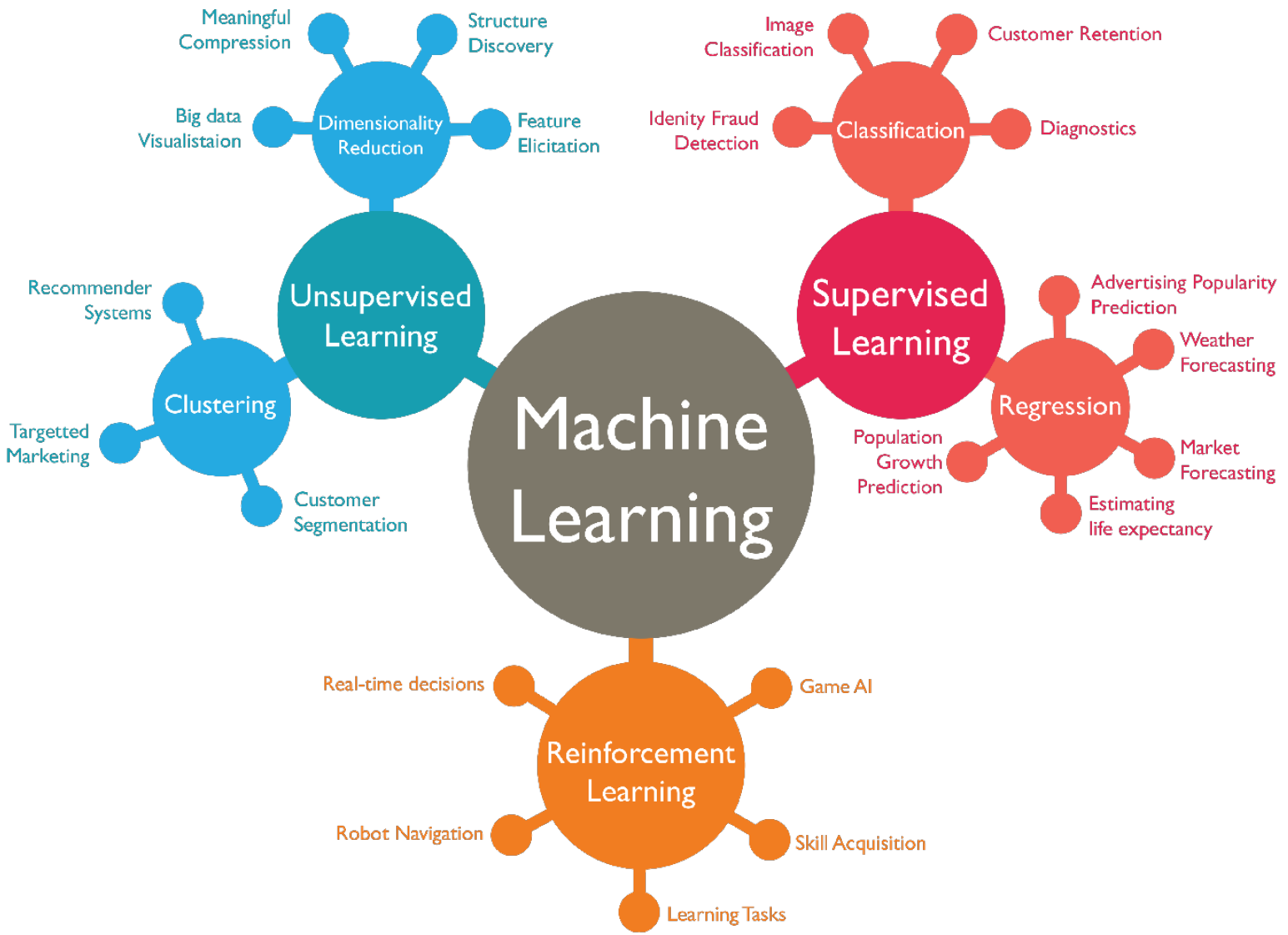
WHAT IS MACHINE LEARNING?

"Machine learning evolved as a subfield of artificial intelligence that involved the development of self-learning algorithms to gain knowledge from data in order to make predictions"

S. Raschka

DIFFERENT TYPES OF MACHINE LEARNING

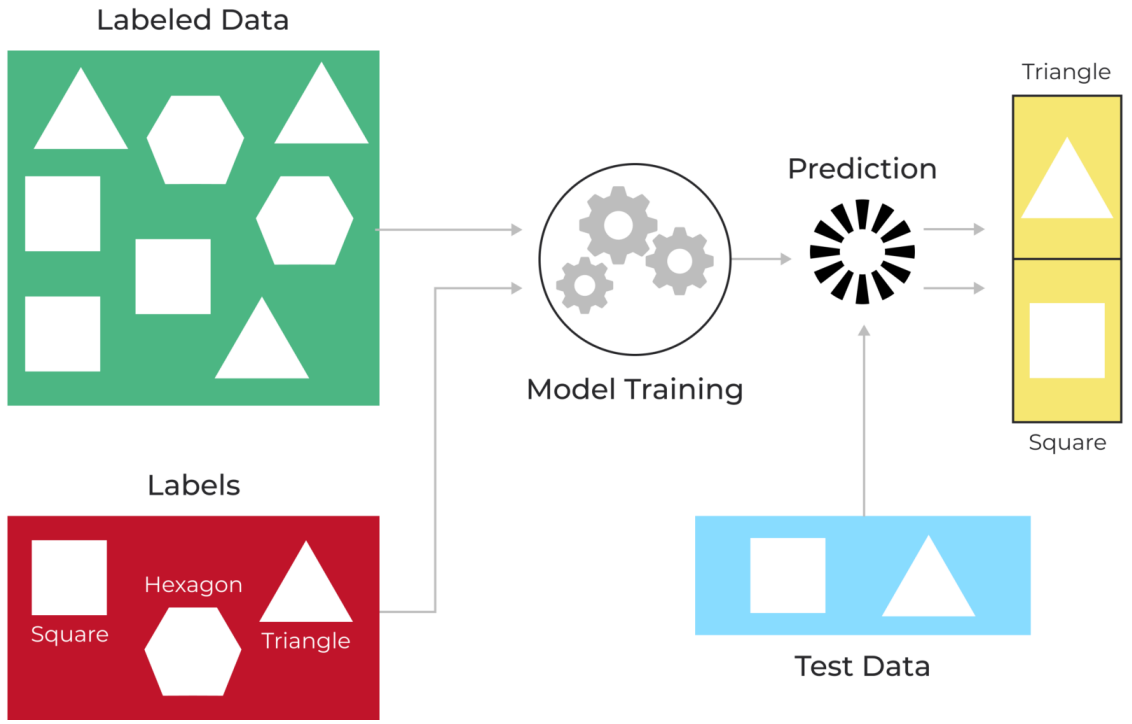




Credits: https://wordstream-files-prod.s3.amazonaws.com/s3fs-public/machine-learning.png

SUPERVISED LEARNING

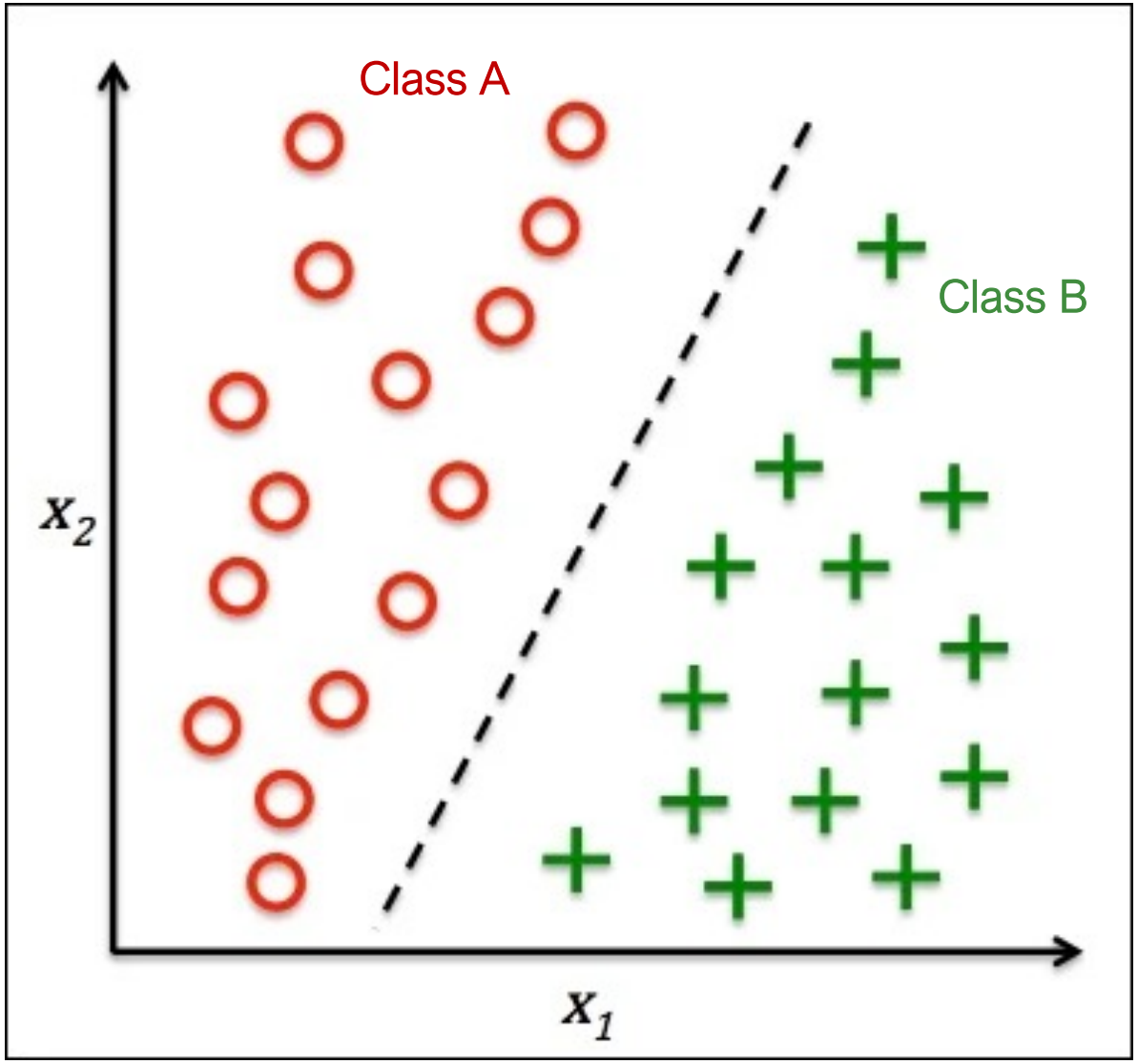
- *Supervised* refers to a set of samples where the desired output signals (labels) are already known.
- *GOAL: learning a model from labeled training data that allows us to make predictions about unseen data.*



Credits: https://postindustria.com/how-to-know-which-machine-learning-algorithms-to-use-techniques-in-machine-learning/

SUPERVISED LEARNING: CLASSIFICATION

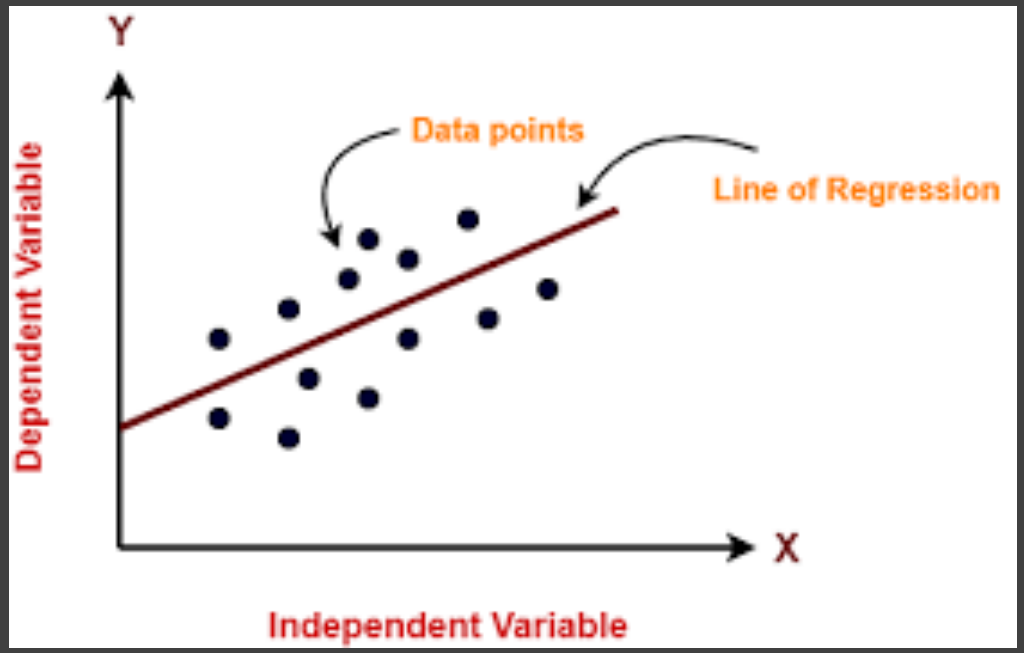
- *The goal of the classification is to predict the categorical class labels of new data based on past observations. Classic examples are:*
  - ***EMAIL SPAM**: binary classification*
  - ***HANDWRITTEN DIGIT RECOGNITION**: multiple class classification*



SUPERVISED LEARNING: REGRESSION

The goal of the regression is the prediction of ***continuous outcomes***

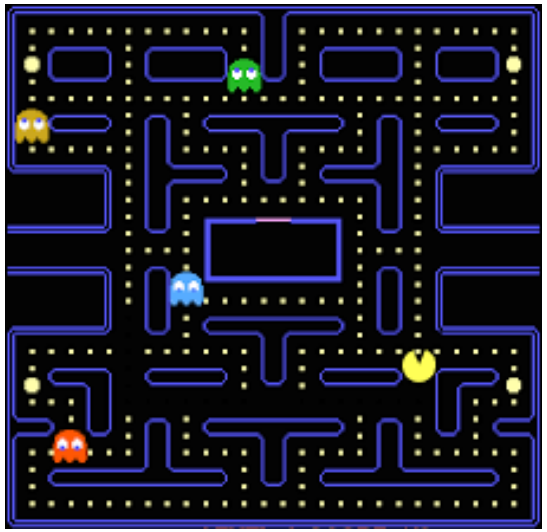
SCORE PREDICTIONS

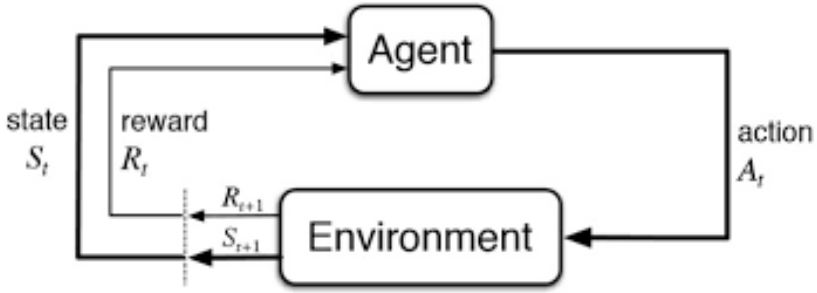


REINFORCEMENT LEARNING

The goal of reinforcement learning is the development of a system which improves by interacting with the environment

PACMAN GAME



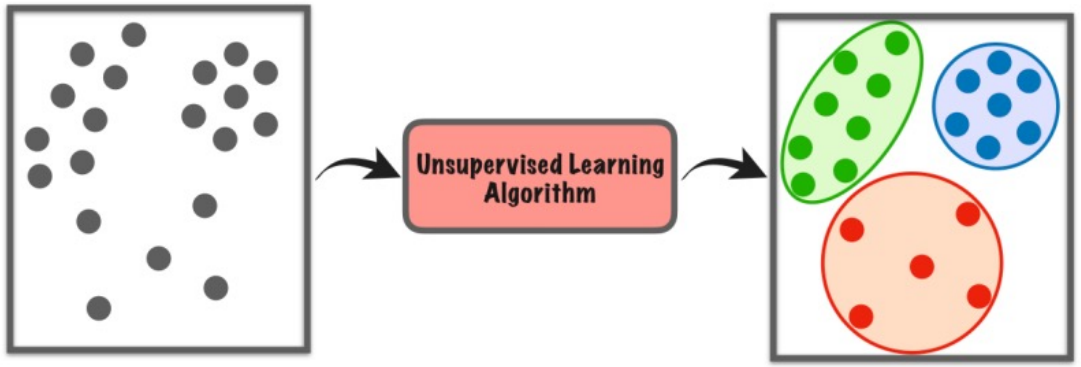


UNSUPERVISED LEARNING

The goal of unsupervised learning is to explore the ***structure of data*** *in order to extract meaningful information without the guidance of a known outcome variable or reward function*

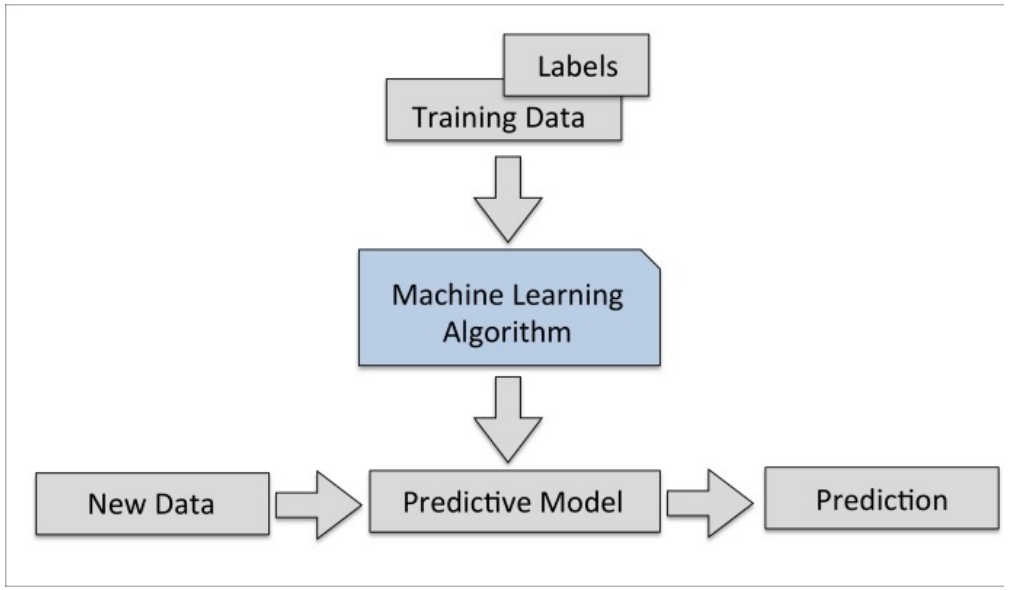
Clustering*: organize information into meaningful subgroups (clusters) without having any prior knowledge of their group memberships*

Dimensionality Reduction



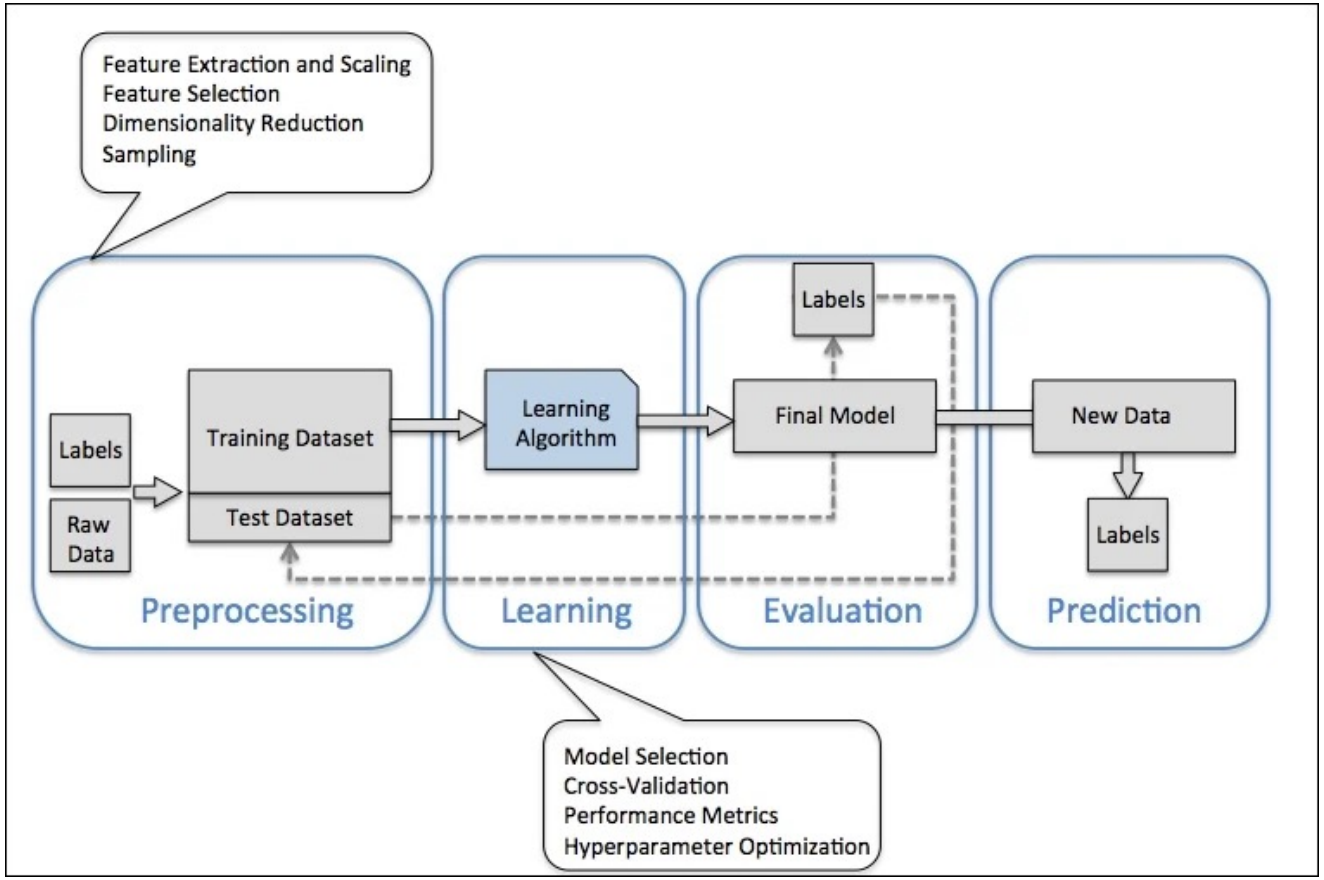
SUPERVISED MACHINE LEARNING

SUPERVISED LEARNING GOALS AND FEATURES



- learning a model from labeled *training* data allowing to make predictions about unseen data
- ***supervised*** → samples where the desired output signals (labels) are already known
- **discrete *output*** (e-mail spam-filtering): ***classification***
- **continuous output** values : ***regression***

ROADMAP FOR BUILDING MACHINE LEARNING SYSTEMS

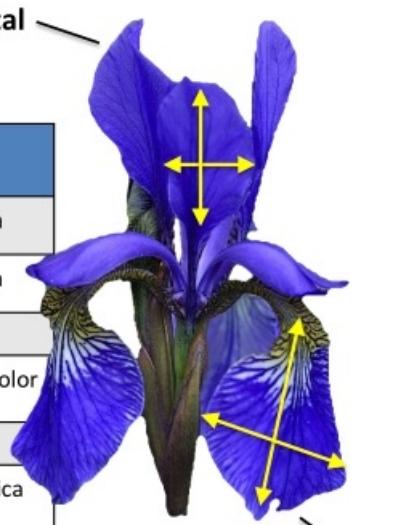


LABELED DATA

IRIS DATASET

Samples
(instances, observations)

	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica



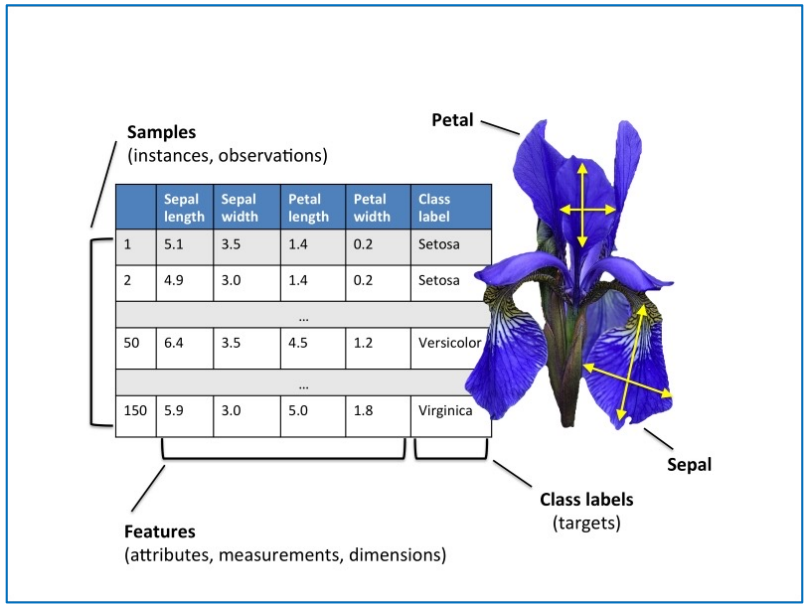
Features
(attributes, measurements, dimensions)

Class labels
(targets)

IRIS MNIST

BASIC TERMINOLOGY AND NOTATION

- The Iris dataset → 150 iris flowers from species:
  - *Setosa*,
  - *Versicolor*,
  - *Virginica*.
  - 1 row of X= 1 flower (4-dim row vector)
  - **1 column of X= 1 feature (150-dim column vector)**
  - **y= class label vector (150-dim column vector)**



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica

$$\boldsymbol{x}_j = \begin{bmatrix} x_j^{(1)} \\ x_j^{(2)} \\ \vdots \\ x_j^{(150)} \end{bmatrix}$$

$$\boldsymbol{x_j} \in \mathbb{R}^{150\times 1}$$

$$X = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & x_4^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & x_4^{(2)} \\ \vdots & \vdots & \vdots & \vdots \\ x_1^{(150)} & x_2^{(150)} & x_3^{(150)} & x_4^{(150)} \end{bmatrix}$$

$$X \in \mathbb{R}^{150\times 4}$$

$$\boldsymbol{x}^{(i)} = \begin{bmatrix} x_1^{(i)} & x_2^{(i)} & x_3^{(i)} & x_4^{(i)} \end{bmatrix}$$

$$\boldsymbol{x^i} \in \mathbb{R}^{1\times 4}$$

$$\boldsymbol{y} = \begin{bmatrix} y^{(1)} \\ \dots \\ y^{(150)} \end{bmatrix} \left(y \in \{\text{Setosa, Versicolor, Virginica}\}\right)$$

$$y \in \mathbb{R}^{150\times 1}$$

PREPROCESSING

WHAT IS PREPROCESSING?

Raw data usually must be suitably shaped to be processed by a learning algorithm→ *preprocessing*

EXAMPLE: in Iris dataset raw data is a series of flower images from which we want to extract meaningful features (color, the hue, the intensity of the flowers, the height, flower lengths and widths,…).

Many machine learning algorithms require that selected features are on the same scale for optimal performance (**normalization**)

Some of the selected features may be highly correlated and therefore redundant to a certain degree →, **dimensionality reduction** techniques (less storage space required → ML algorithm can run much faster!!!)

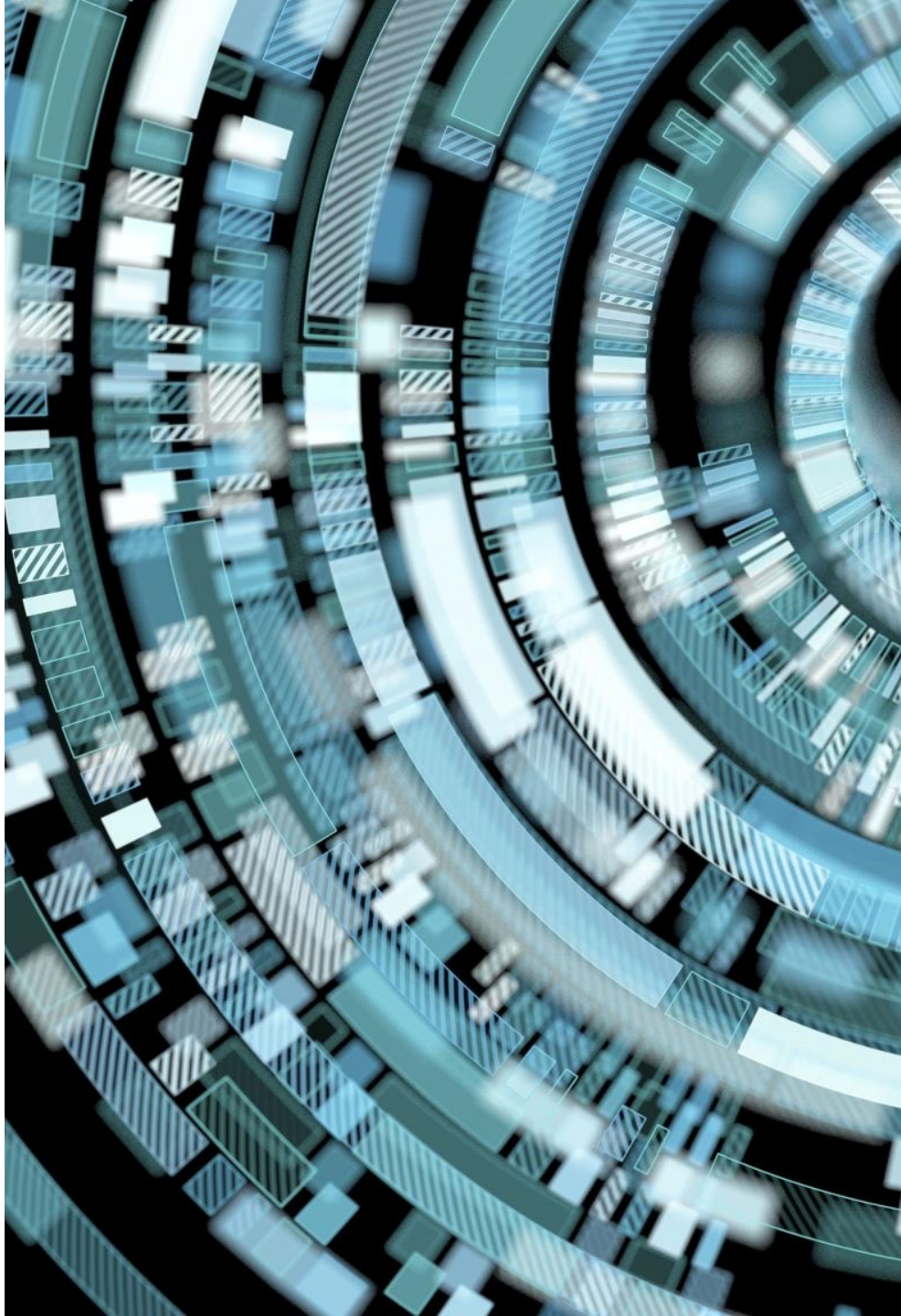
Dataset divided into training dataset and test dataset to determine if the ML algorithm can generalize well on new data:

training dataset to train and optimize our machine learning model

test dataset kept until the very end for **evaluation** of the final model.

SUPERVISED LEARNING ALGORITHMS

- *Perceptron*
- *ADALINE*
- *Logistic Regression*
- *Artificial neural network*
- *Support Vector Machine*
- *Decision Trees*
- *K-nearest neighbors*



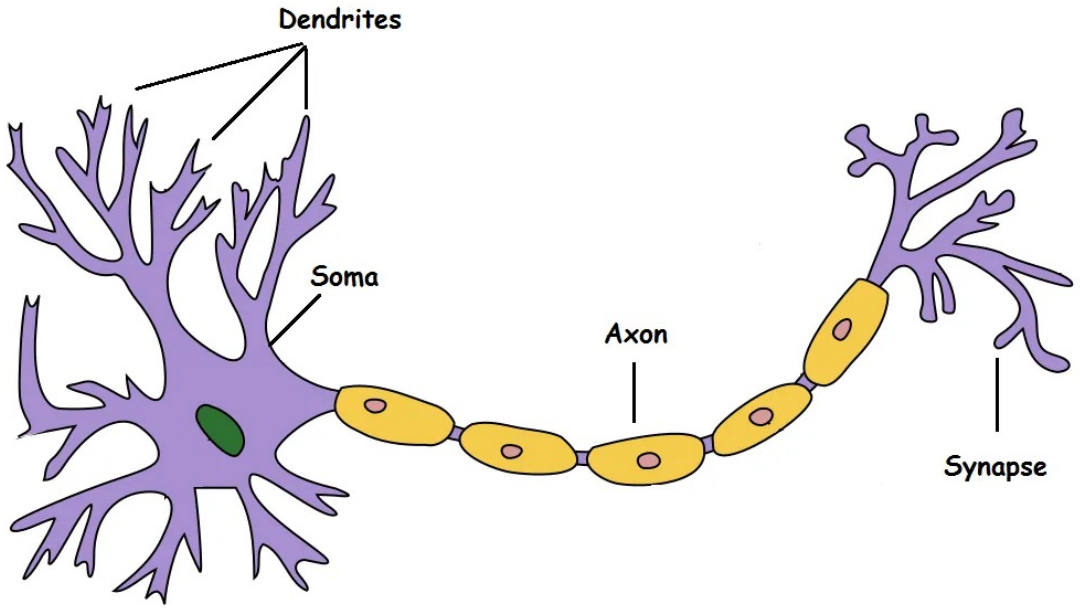
SUPERVISED LEARNING ALGORITHMS

- *Perceptron*
- *ADALINE*
- *Logistic Regression*
- *Artificial neural network*
- *Support Vector Machine*
- *Decision Trees*
- *K-nearest neighbors*

A BRIEF HISTORY OF EARLY ML MODELS

- Biological Neuron
- Mc Culloch-Pitts Neuron
- Perceptron
- Multi-layer perceptron

WHAT ARE NEURONS AND HOW DO THEY LOOK LIKE?



Neurons: interconnected nerve cells in the brain involved in the processing and transmitting of chemical and electrical signals

Dendrite: Receives signals from other neurons
Soma: Processes the information
Axon: Transmits the output of this neuron
Synapse: Point of connection to other neurons

MC CULLOCH-PITTS NEURON: THE SIMPLEST ARTIFICIAL NEURON

- **1943**: Warren McCullock and Walter Pitts published the first model of a *simplified brain cell*, the so-called *McCullock-Pitts (MCP) neuron*, in 1943 (W. S. McCulloch and W. Pitts. *A Logical Calculus of the Ideas Immanent in Nervous Activity*. The bulletin of mathematical biophysics, 5(4):115–133, 1943).

Bulletin of Mathematical Biology Vol. 52, No. 1/2, pp. 99–115, 1990.
Printed in Great Britain.

0092–8240/90$3.00 + 0.00
Pergamon Press plc
Society for Mathematical Biology

A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY*

WARREN S. MCCULLOCH AND WALTER PITTS
University of Illinois, College of Medicine,
Department of Psychiatry at the Illinois Neuropsychiatric Institute,
University of Chicago, Chicago, U.S.A.

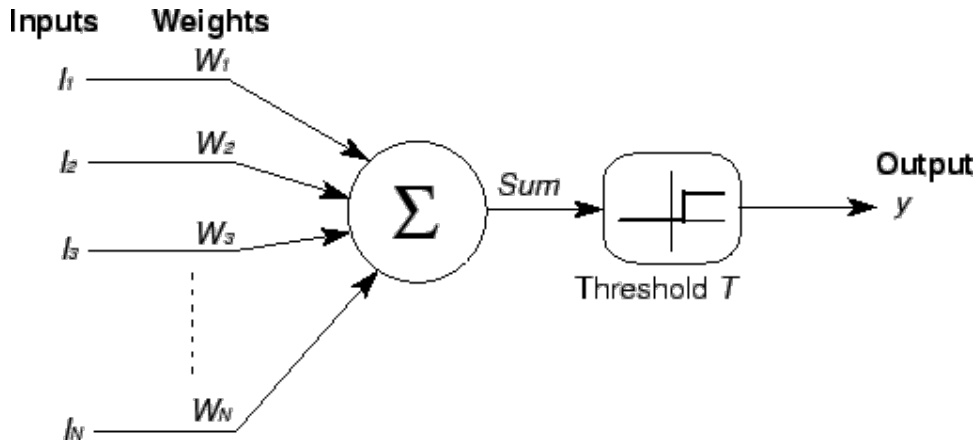
Because of the "all-or-none" character of nervous activity, neural events and the relations among them can be treated by means of propositional logic. It is found that the behavior of every net can be described in these terms, with the addition of more complicated logical means for nets containing circles; and that for any logical expression satisfying certain conditions, one can find a net behaving in the fashion it describes. It is shown that many particular choices among possible neurophysiological assumptions are equivalent, in the sense that for every net behaving under one assumption, there exists another net which behaves under the other and gives the same results, although perhaps not in the same time. Various applications of the calculus are discussed.

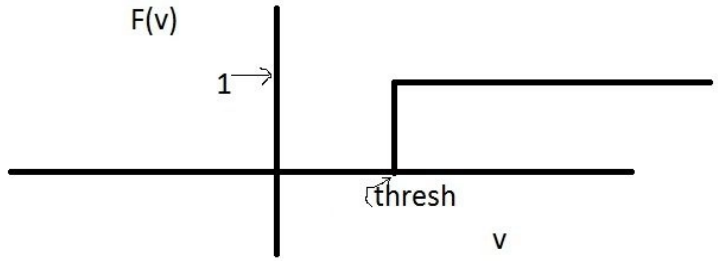
MC CULLOCH-PITTS MODEL

Mc Culloch and Pitts describe a nerve cell as a logic gate with binary outputs

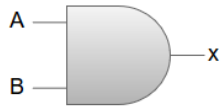
multiple signals arrive at the dendrites and integrated into the cell body

if the total signal exceeds a certain threshold ν an output is generated and pass on by the axon





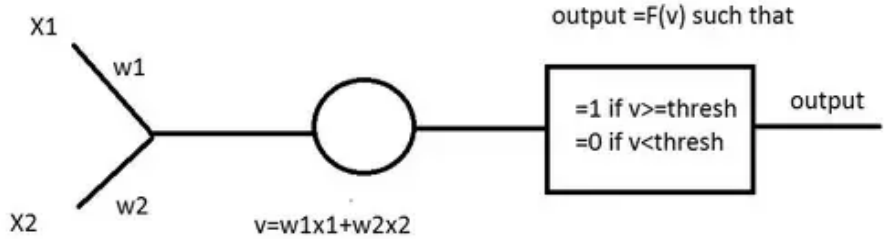
AND Gate:



Algebraic Function: x = AB

Truth Table:

A	B	x
0	0	0
0	1	0
1	0	0
1	1	1

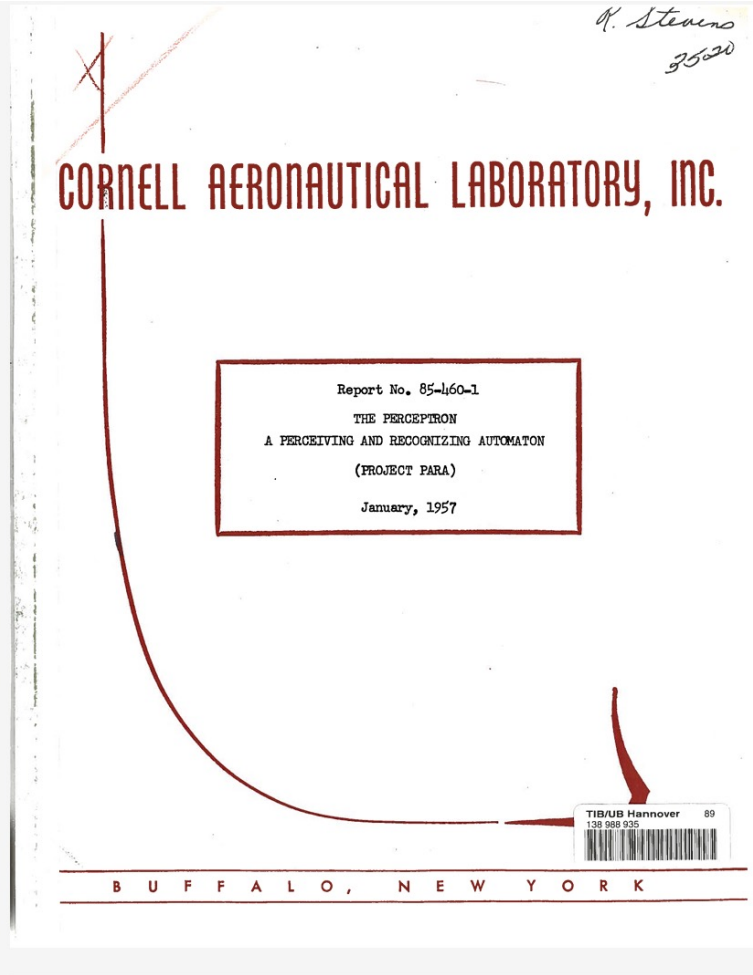


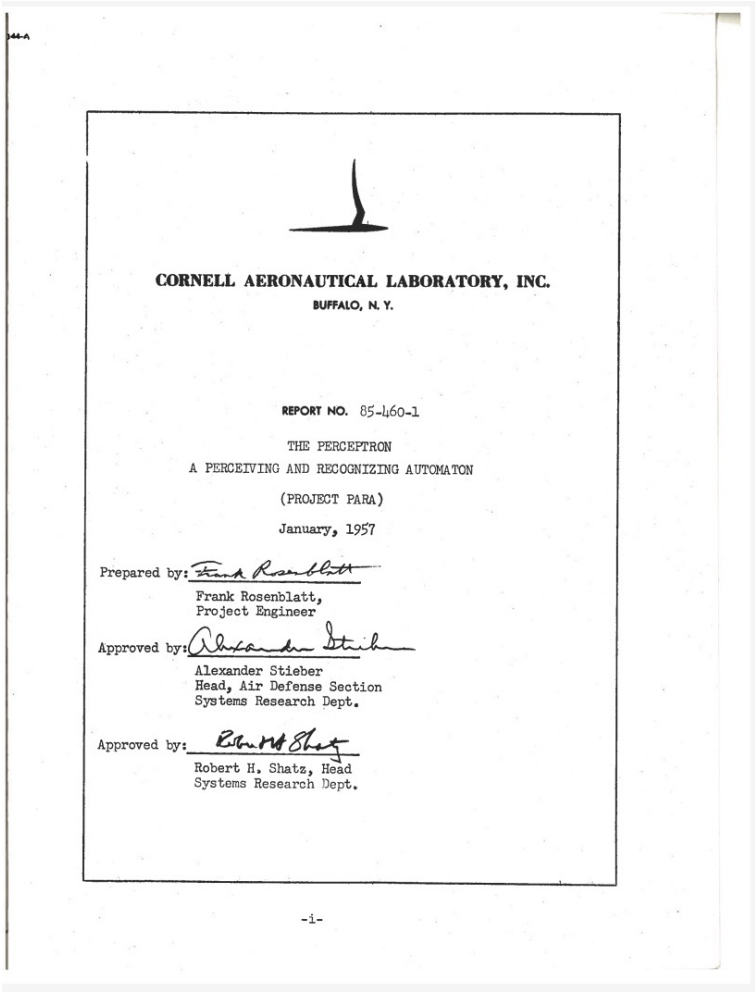
LIMITATIONS OF MC CULLOCH-PITTS MODEL

Non boolean inputs

What if we want to assign more importance to some inputs?

How can we set a threshold?





ROSENBLATT'S PERCEPTRON

- Few years later, Frank Rosenblatt published the first concept of the **perceptron learning rule based on the MCP neuron model** (F. Rosenblatt, *The Perceptron, a Perceiving and Recognizing Automaton*. Cornell Aeronautical Laboratory, **1957**)
- Rosenblatt proposed **an algorithm that automatically learns the optimal weight coefficients** that are then multiplied with the input features in order to make the decision of whether a neuron fires or not.
- In the context of supervised learning and classification, such an algorithm could then be used to predict if a sample belonged to one class or the other.

ROSENBLATT'S PERCEPTRON

- Let's think about a binary classification task where we refer to our two classes as 1 (positive class) and -1 (negative class) for simplicity.
- We can define an ***activation function*** $\boldsymbol{\phi(z)}$ as a linear combination of input values *x (feature vector) and corresponding weights*
- *z is the net input* $z = w_1 x_1 + \ldots + w_m x_m$
- If the activation function ϕ of z is greater than a defined threshold θ we predict 1, otherwise we predict -1: **step function**
- We bring θ on the left side: $z - \theta \geq 0$
- We define a weight-zero as $w_0 = -\theta$ *(bias) and* $x_0 = 1$, rewriting *z* in a more compact form:

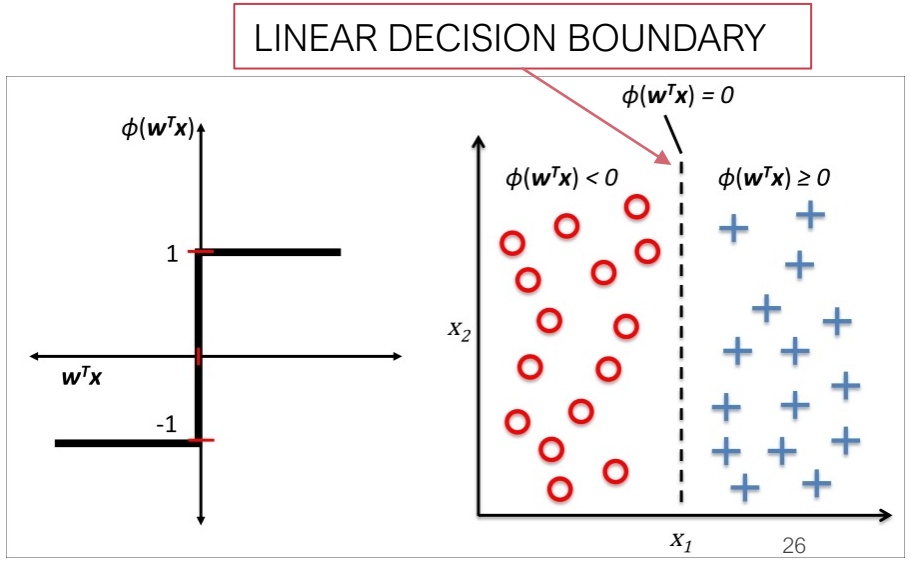
$$z = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m = \boldsymbol{w}^T \boldsymbol{x}$$

- The net input z is squashed in a binary output -1 or 1 by $\phi(z)$ and it can be used to discriminate between two linearly separable classes (right subfigure)

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

Heaviside step function

$$\phi(z) = \begin{cases} 1 \ \ if \ z \geq \theta \\ -1 \ \ otherwise \end{cases}$$



ROSENBLATT'S PERCEPTRON RULES

The idea behind the MCP neuron and Rosenblatt's *thresholded* :

- using a reductionist approach to mimic how a single works: it either *fires* or it doesn't.

Rosenblatt's initial perceptron rule is fairly simple and can be summarized by the following steps:

1. Initialize the weights to 0 or small random numbers.
2. For each training sample x_i perform the following steps:
   1. **Compute the output value (predicted)** $\hat{y}_i$
   2. **Update the weights** $w_j := w_j + \Delta w_j$
   3. **The update can be calculate using this formula** $\Delta w_j = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_j^{(i)}$ where η is the **learning rate** which a constant value ranges from 0.0 to 1.0, y_i is the true class label of the i-th training sample and $\hat{y}_i$ is the value of the i-th training sample predicted by the algorithm

!!! It is important to note that all weights in the weight vector are updated simultaneously, which means that we do not recompute the $\hat{y}_i$ until all the weights are updated!!!

ROSENBLATT'S PERCEPTRON EXAMPLE

Example for a two-dimensional dataset:

$$\Delta w_0 = \eta(y^{(i)} - output^{(i)})$$
$$\Delta w_1 = \eta(y^{(i)} - output^{(i)})x_1^{(i)}$$
$$\Delta w_2 = \eta(y^{(i)} - output^{(i)})x_2^{(i)}$$

In the case that the perceptron predicts the class label correctly:

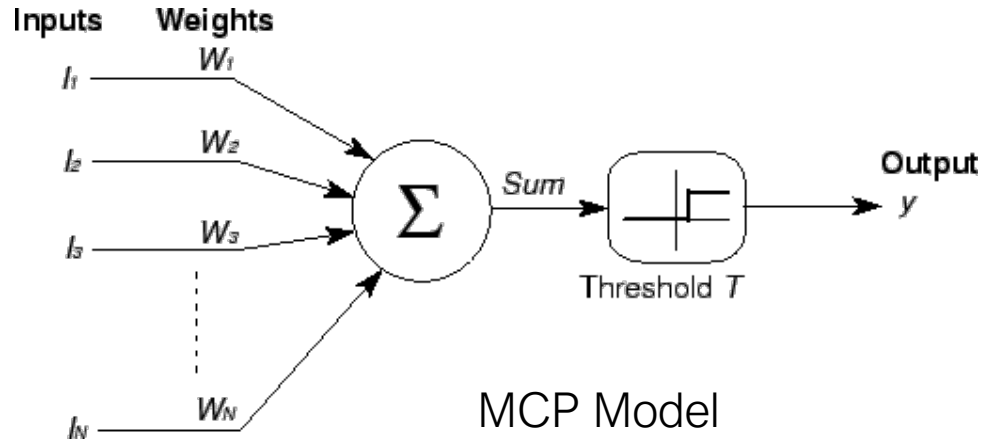
$$\Delta w_j = \eta(-1 - (-1))x_j^{(i)} = 0$$
$$\Delta w_j = \eta(1 - 1)x_j^{(i)} = 0$$

In the case of a wrong prediction:

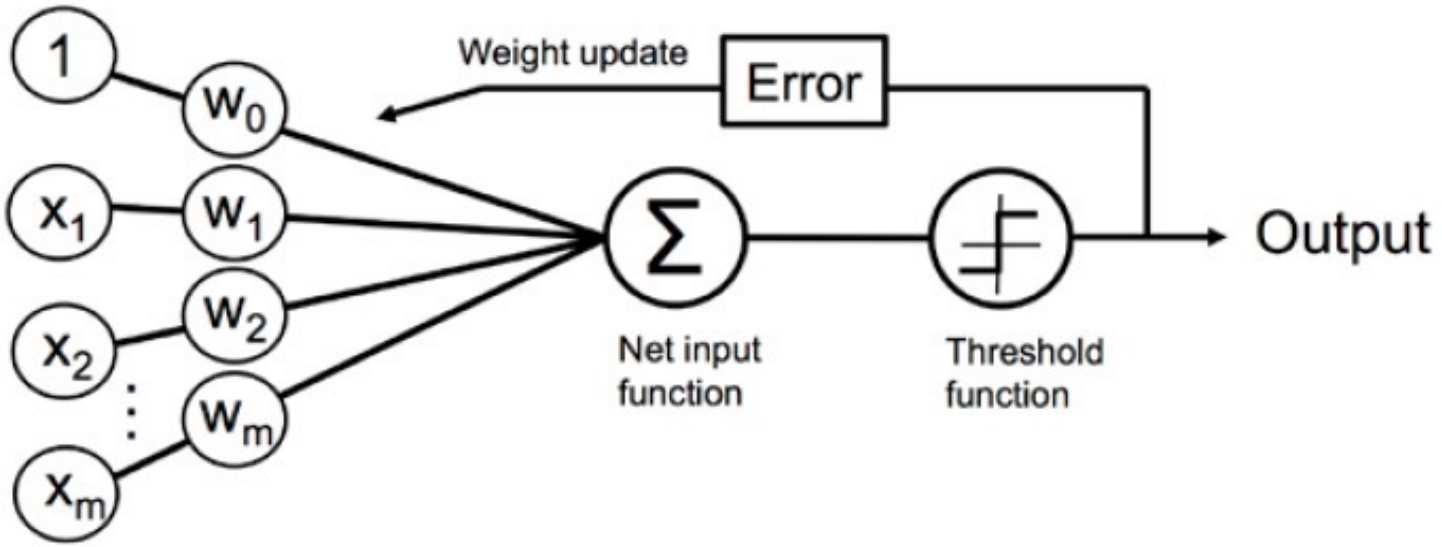
$$\Delta w_j = \eta(1 - (-1))x_j^{(i)} = 2\eta x_j^{(i)}$$
$$\Delta w_j = \eta(-1 - 1)x_j^{(i)} = -2\eta x_j^{(i)}$$

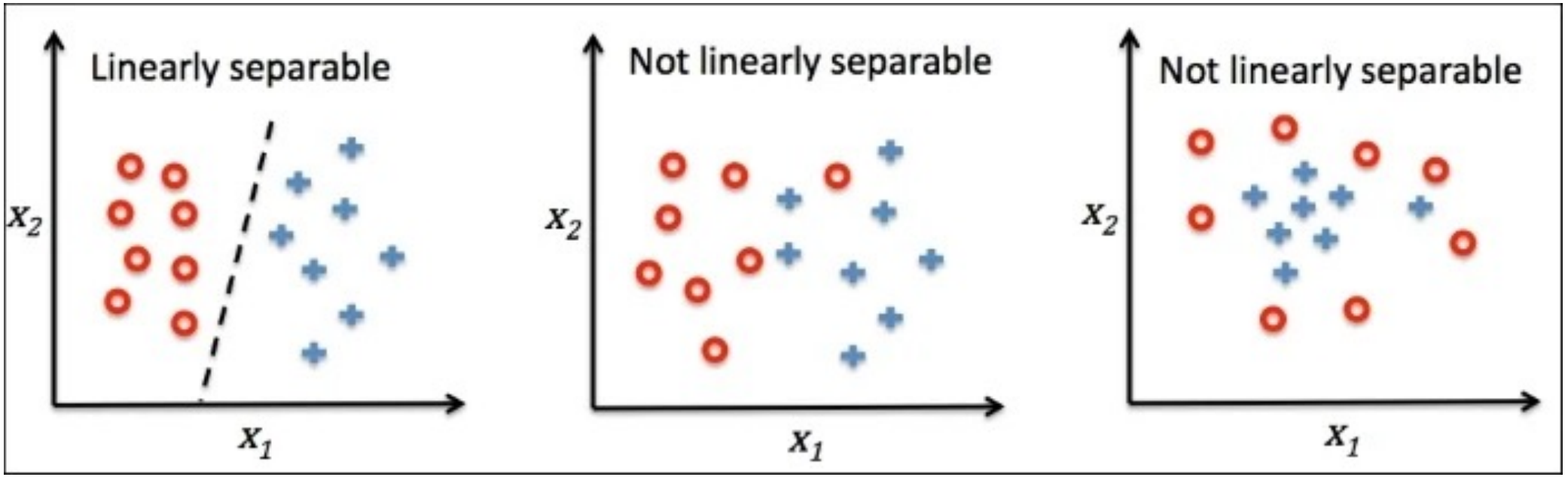
ROSENBLATT'S PERCEPTRON: FINAL SCHEME

The learning algorithm passes over the training dataset until all the input vectors are classified correctly (until it achieves **convergence**)



MCP Model

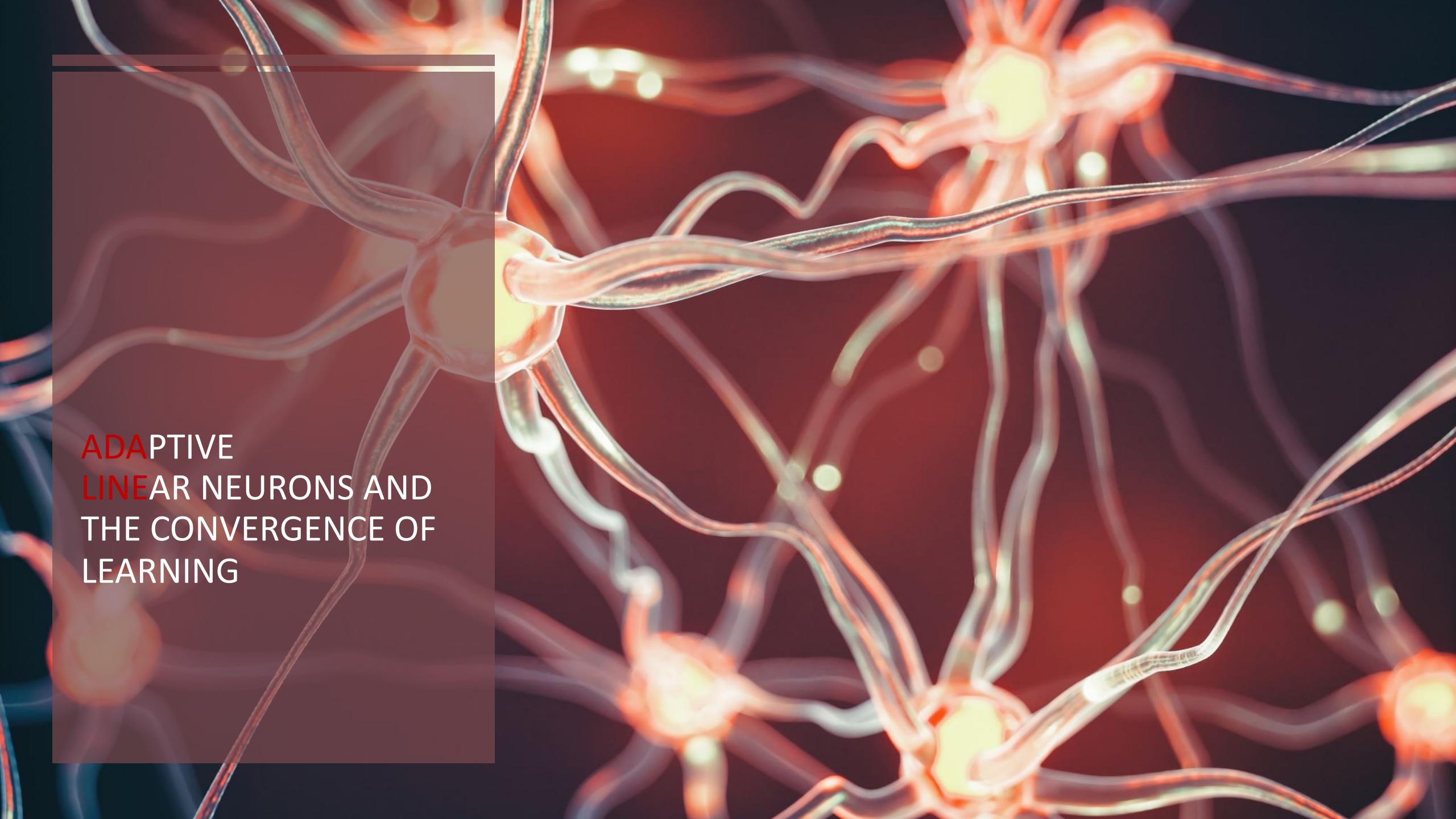




!!! IMPORTANT !!!

- The convergence of the perceptron is only guaranteed if the two classes are linearly separable and the learning rate is sufficiently small.
- If the two classes can't be separated by a linear decision boundary, we can set a maximum number of passes over the training dataset (*epochs*) and/or a threshold for the number of tolerated misclassifications—the perceptron would never stop updating the weights otherwise!!!!

DEEP NEURAL NETWORK OR KERNEL FUNCTION


ADAPTIVE
LINEAR NEURONS AND
THE CONVERGENCE OF
LEARNING

ADAPTIVE LINEAR NEURON (ADALINE)

Adaline was published only a few years after Frank Rosenblatt's perceptron algorithm, by Bernard Widrow and his doctoral student Tedd Hoff

It can be considered as an improvement on the latter (B. Widrow et al. Adaptive *"Adaline" neuron using chemical "memistors"*. Number Technical Report 1553-2. Stanford Electron. Labs. Stanford, CA, October 1960.

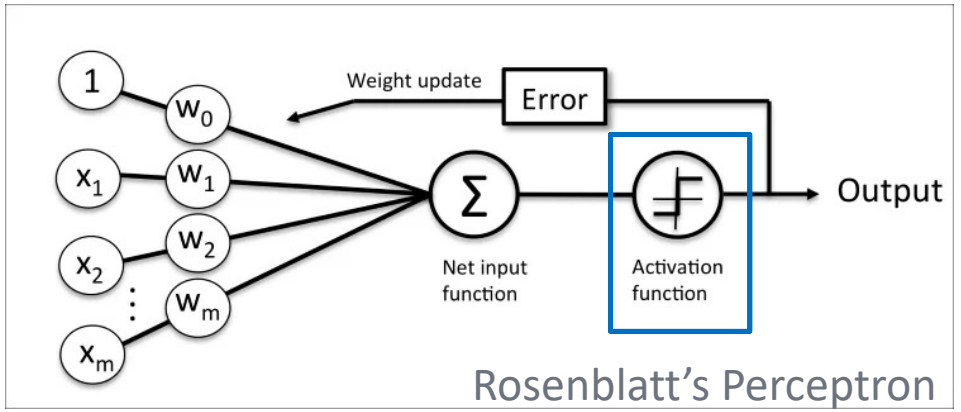
The Adaline algorithm illustrates the **key concept of defining and minimizing cost functions**, which will lay the groundwork for understanding more advanced machine learning algorithms for classification (e.g. logistic regression)

ADALINE

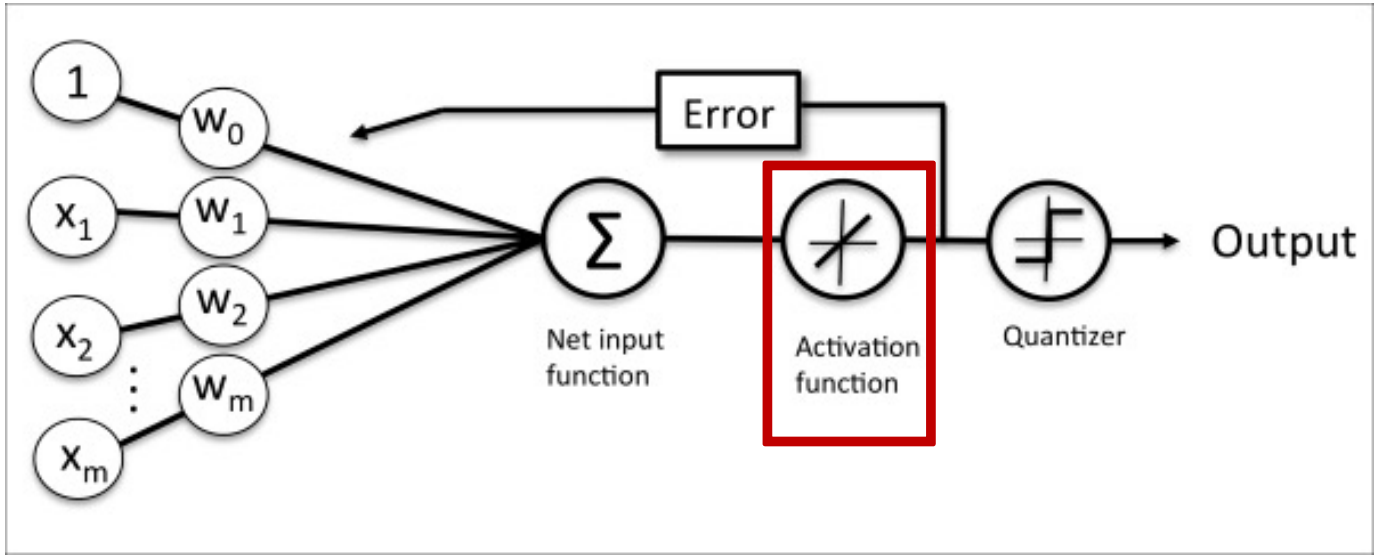
- The **key** difference between the Adaline rule (a.k.a. the *Widrow-Hoff rule*) and Rosenblatt's perceptron is that **the weights are updated based on a linear activation function rather than a unit step function** like in the perceptron.
- In Adaline, this linear activation function ϕ(z) is simply the **identity function** of the net input so that

$$\phi\left(w^T x\right) = w^T x$$

- While the linear activation function is used for learning the weights, a ***quantizer***, which is similar to the unit step function that we have seen before, can then be used to **predict the class labels**
- If we compare the preceding figure to the illustration of the perceptron algorithm that we saw earlier, **the difference** is that we **use the continuous valued output from the linear activation function to compute the model error and update the weights**, rather than the binary class labels, which is more "powerful" since it tells us by "how much" we are right or wrong

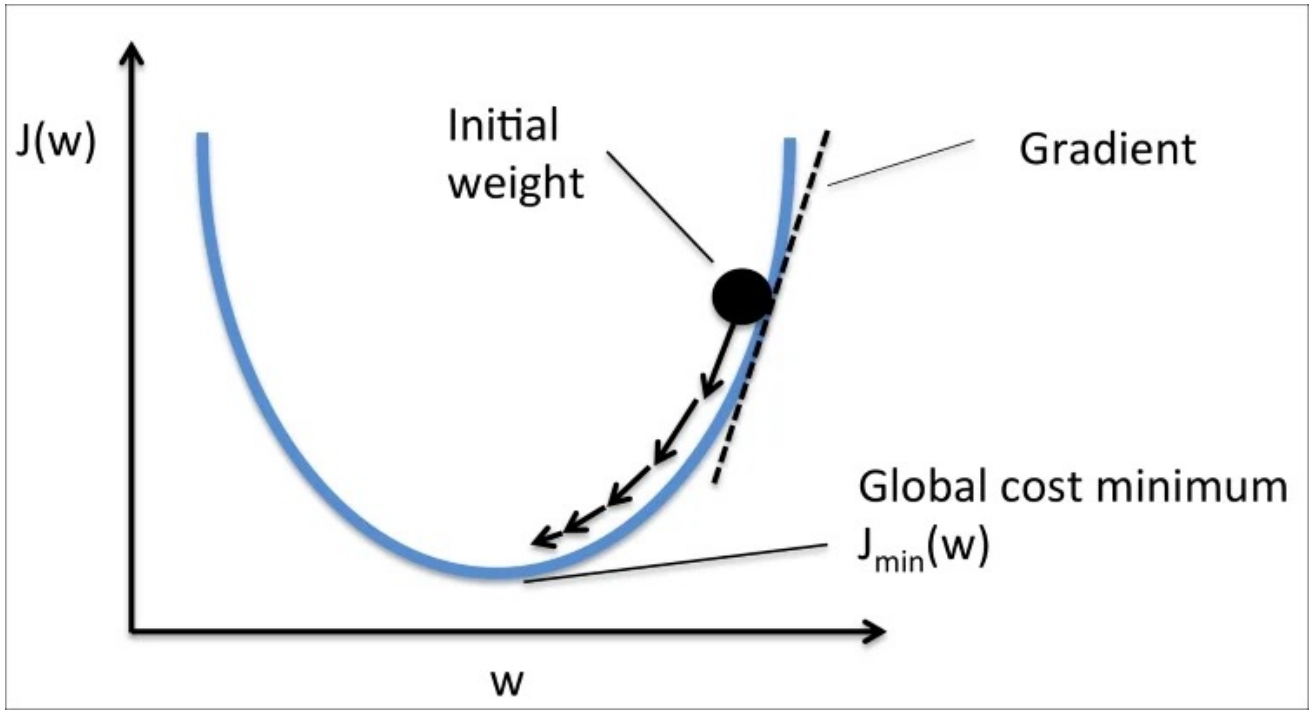


Rosenblatt's Perceptron



MINIMIZING COST FUNCTIONS WITH GRADIENT DESCENT

- In supervised machine learning algorithms we define an ***objective function*** that must be optimized during the learning process.
- This **objective function** is often a ***cost function*** that we want to **minimize**.
- In the case of Adaline, we can define the cost function J to learn the weights as the **Sum of Squared Errors** (**SSE**) between the calculated outcomes and the true class labels $J(\boldsymbol{w}) = \frac{1}{2}\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2$
- The term $\frac{1}{2}$ is to make it easier to derive the gradient
- **The main advantage of this continuous linear activation function —in contrast to the unit step function— is that the cost function becomes differentiable**.
- Another property of this cost function is that it is convex; thus, we can use a simple, yet powerful, **optimization algorithm called *gradient descent*** to find **the weights that minimize our cost function** to classify the samples in the Iris dataset.



$$J(\boldsymbol{w}) = \frac{1}{2}\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^2$$

Principles behind gradient descent:

- *climbing down a hill* until a local or global cost minimum is reached.
- **In each iteration** we take a step away from the gradient where the **step size is determined by the value of the learning rate** as well as the slope of the gradient

$$\Delta \boldsymbol{w} = -\eta \nabla J(\boldsymbol{w})$$

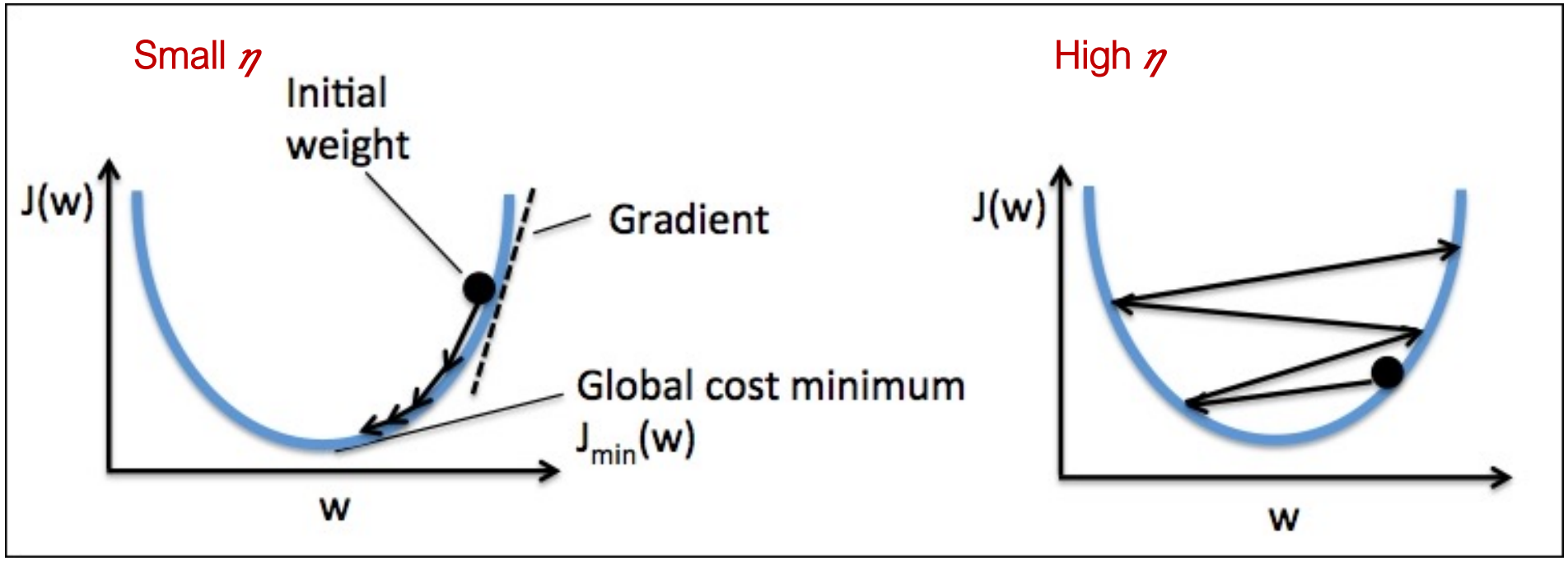
ADALINE'S COST FUNCTION

ADALINE VS PERCEPTRON

- Although the Adaline learning rule looks identical to the perceptron rule $\phi\left(\boldsymbol{w}^T \boldsymbol{x}\right) = \boldsymbol{w}^T \boldsymbol{x}$ is a real number and not an integer class label
- The **weight update** is calculated based on **all samples in the training set** (instead of updating the weights incrementally after each sample), which is why this approach is also referred to as "**batch**" gradient descent.

GRADIENT DESCENT: LEARNING RATE

- *η is the learning rate constant that determines the size of the steps*



*With a **very low learning rate**, we can move in the direction of the negative gradient since we are **recalculating it frequently**, but **calculating the gradient is time-consuming**, so it will take us a very long time to get to the bottom.*

*With a **high learning rate** we can cover more ground each step (so it learns faster), but we **risk overshooting the lowest point** since the slope of the hill is constantly changing*

SCALING AND LOGISTIC REGRESSION

SCALING DATA

- Feature data can have different scales and ranges.
- This can be a problem for gradient descent:
  - The **weights updated is proportional to feature value**, so, with features being on different scales
  - certain weights may update faster than others
  - It is difficult to select the most suitable learning rate value
    - If we choose the value based on the input value having the smallest range, small learning rate it takes ages for the large range to converge.
    - if we choose high value for learning rate, the gradient descent might not converge for small ranges.
- Feature scaling is a method used to normalize the range of features of data.

!!! If we have features on a similar scale we help the gradient descent to converge faster to the minimum !!!

SCALING DATA: NORMALIZATION

$$x' = \frac{x - \mu}{\max(x) - \min(x)}$$

$$x' = \frac{x - min(x)}{max(x) - min(x)}$$

Normalization is a scaling technique in which values are shifted and rescaled so that they end up ranging between 0 and 1.

- *The general formula for a **min-max** of [0, 1] is given as the top-left equation*
- *Another form of normalization is called **mean-normalization:***
  - it calculates and subtracts the mean for every feature. *The formula is the equation in the top-right*

SCALING DATA: FEATURE STANDARDIZATION

- Feature standardization makes the values of each feature in the data have zero-mean (when subtracting the mean in the numerator) and unit-variance.
- The general method of calculation is to determine the distribution mean and standard deviation for each feature. Next, we subtract the mean from each feature. Then we divide the values (mean is already subtracted) of each feature by its standard deviation

$$x' = \frac{x - \mu}{\sigma}$$

- Logistic regression is a binary classifier
  - The output variable *Y* has two possible values 0 and 1
- Logistic regression is a probabilistic model
  - its goal is to model the probability of the positive class (i.e., the class that we want to predict), typically class 1.
- Classification
  - To compute the conditional probability of the response *Y*, given the input variables *X*, $Pr(Y|X)$
  - Consider a single input observation $\boldsymbol{x}$, which we will represent by a vector of features $[x_1, x_2, \ldots, x_n]$, we want to know the probability that this observation $\boldsymbol{x}$ belongs to the positive class 1, $P(Y=1|\boldsymbol{x})$.

LOGISTIC REGRESSION

LOGISTIC REGRESSION

- To explain the idea behind logistic regression as probabilistic model for binary classification we introduce the **odds** in favor of a particular event.

$$\frac{p}{1-p}$$

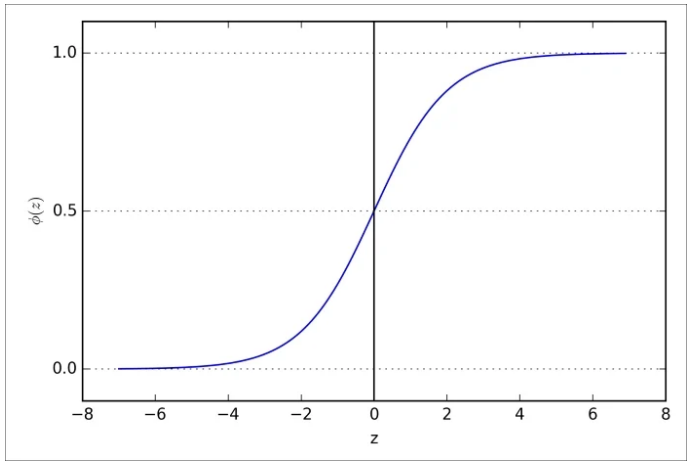
- The logit function is the logarithm of the odds:

$$logit(p) = log\frac{p}{(1-p)}$$

- Logit function takes input values in range 0 and 1 and transforms them to values over the entire real-number range, which we can use to express a linear relationship between feature values and the log-odds

$$logit\big(p(y=1\,|\,\boldsymbol{x})\big) = w_0x_0 + w_1x_1 + \cdots + w_mx_m = \sum_{i=0}^{m} w_ix_i = \boldsymbol{w}^T\boldsymbol{x}$$

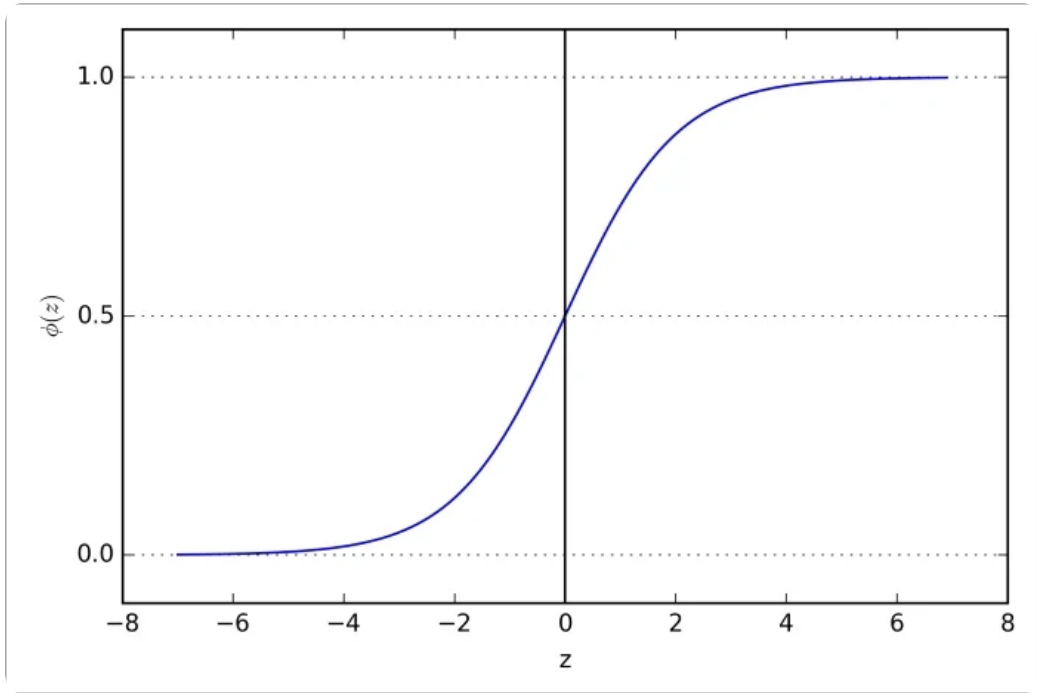
LOGISTIC REGRESSION: SIGMOID FUNCTION



$$\phi(z) = \frac{1}{1-e^{-z}}$$

- We are actually interested in predicting the probability that a certain example belongs to a particular class, which is the inverse form of the logit function.
- It is also called the logistic sigmoid function, which is sometimes simply abbreviated to sigmoid function due to its characteristic S-shape→ **a new NOT LINEAR activation function with an intrinsic probability meaning!**

LOGISTIC REGRESSION: HOW TO MAKE PREDICTION - DECISION BOUNDARY



$$\hat{y} = \begin{cases} 1 & \text{if } \phi(z) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

- *The predicted probability can then simply be converted into a binary outcome via a quantizer (unit step function)*

LOGISTIC REGRESSION: SCHEME

- In Adaline, we used the identity function

$\phi(z) = z$ as activation function.

- In logistic regression, this activation function simply becomes the sigmoid function

- There are many applications where we are not only interested in the predicted class labels, but where the estimation of the class-membership probability is particularly useful

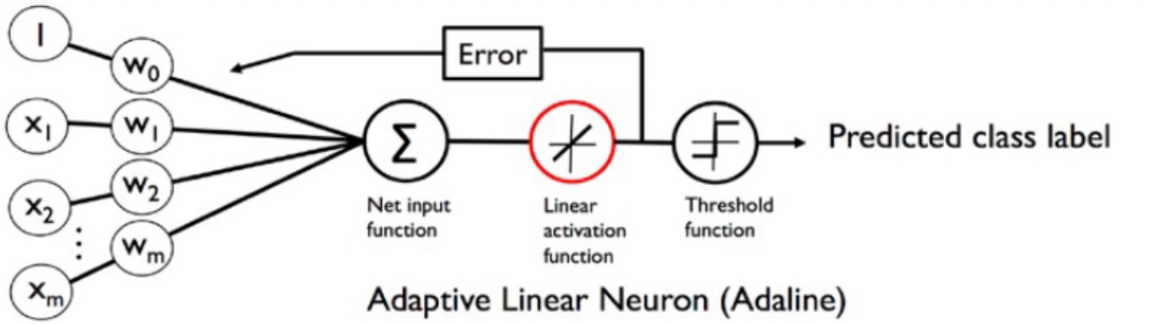
- Using LR we should maximize the likelihood function L

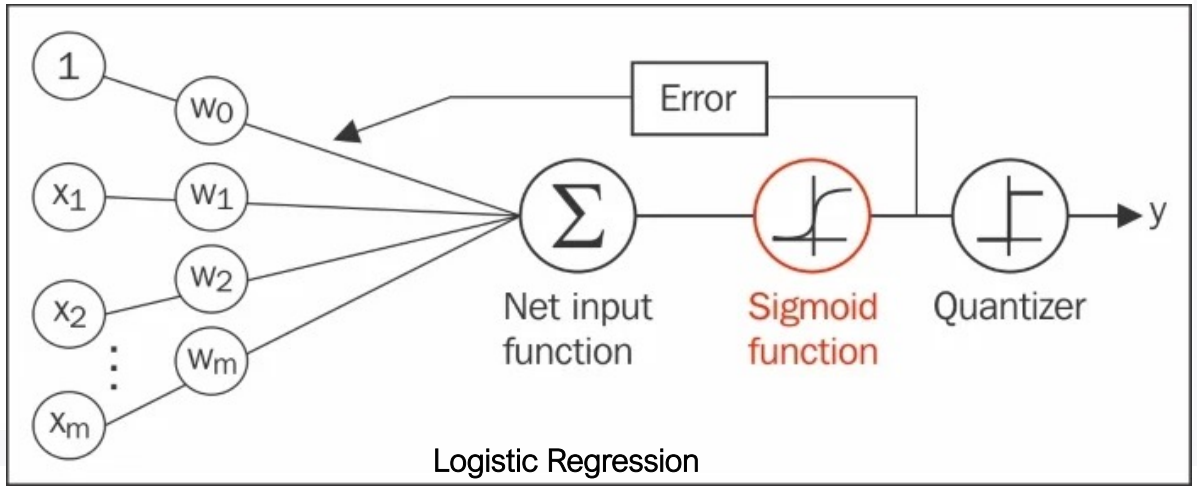
$$L(\boldsymbol{w}) = P(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{w}) = \prod_{i=1}^{n} P\left(y^{(i)} \mid x^{(i)}; \boldsymbol{w}\right) = \prod_{i=1}^{n} \left(\phi\left(z^{(i)}\right)\right)^{y^{(i)}} \left(1 - \phi\left(z^{(i)}\right)\right)^{1-y^{(i)}}$$

- LR uses gradient descent after converting the log-likelihood function in the cost function *J*

$$l(\boldsymbol{w}) = \log L(\boldsymbol{w}) = \sum_{i=1}^{n} \left[y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right) \right]$$

LOG-LIKELIHOOD

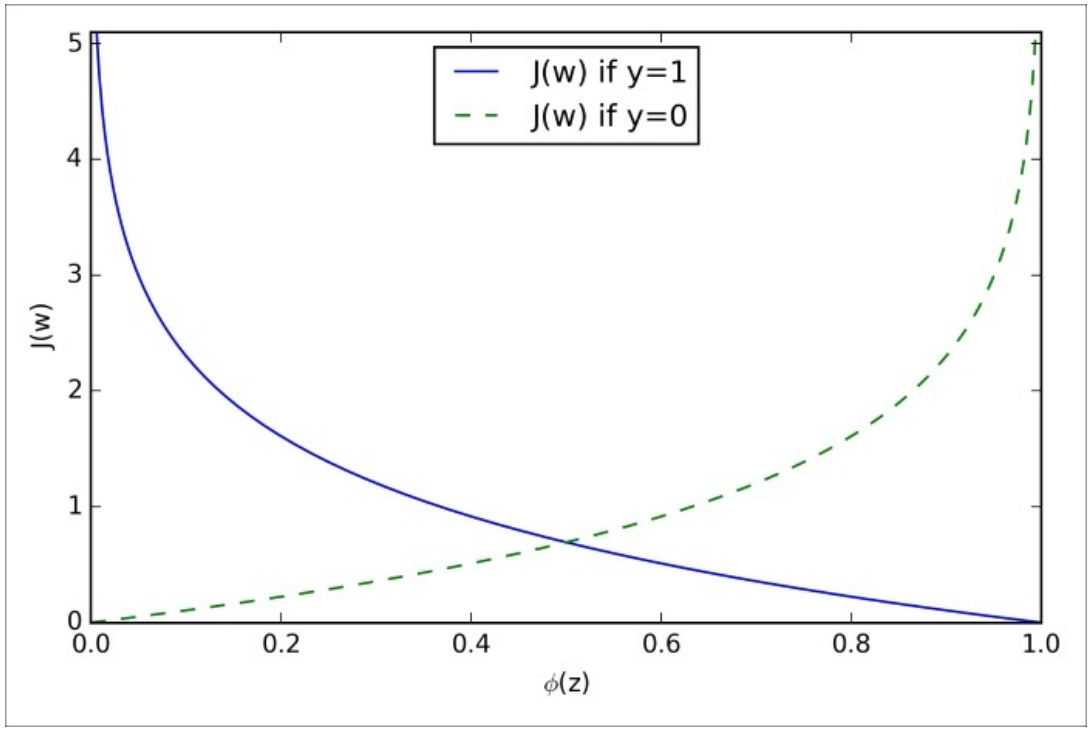




Adaline VS LR

$$J(\boldsymbol{w}) = \sum_{i=1}^{n} \left[-y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) - \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right) \right]$$

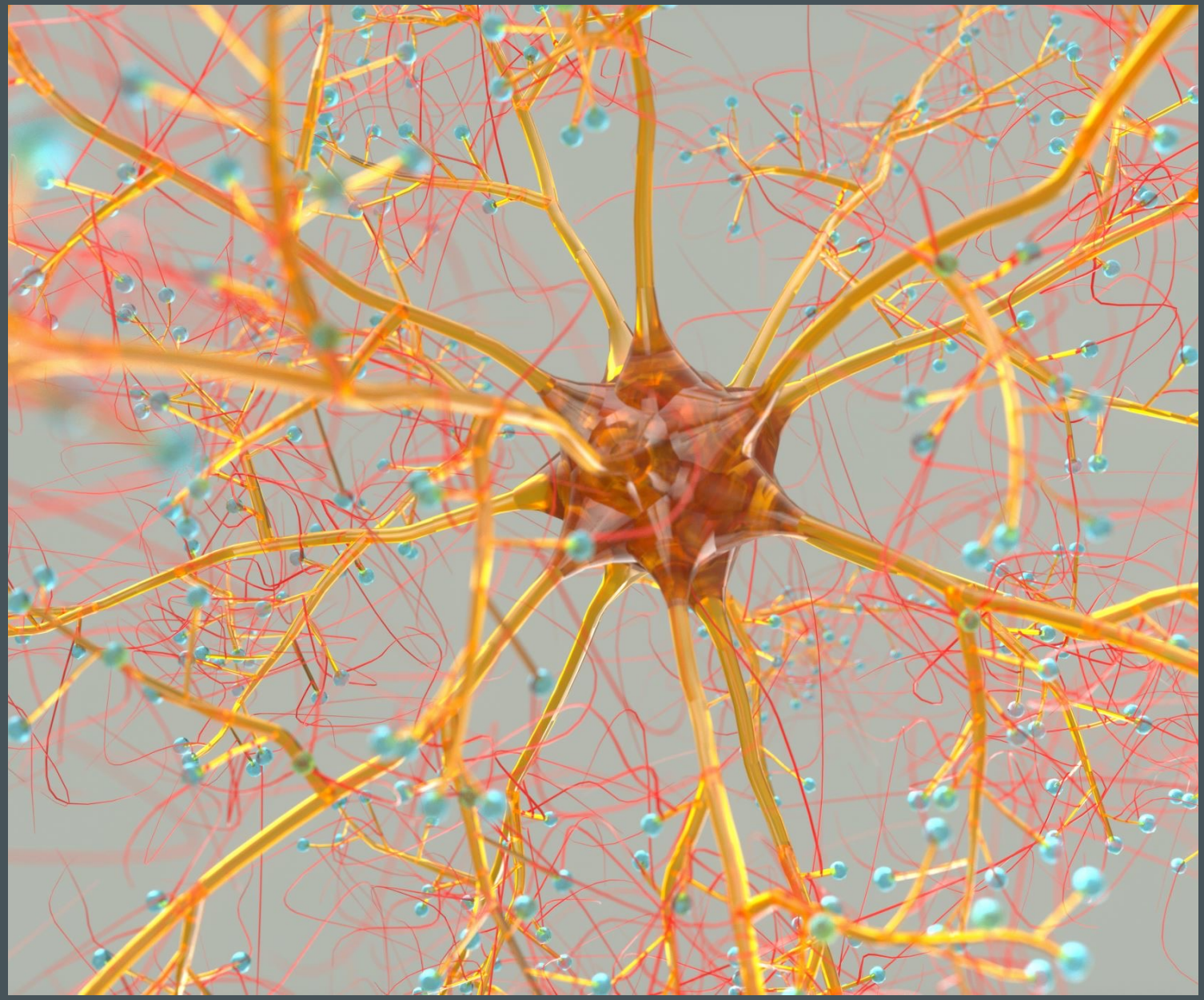
COST FUNCTION



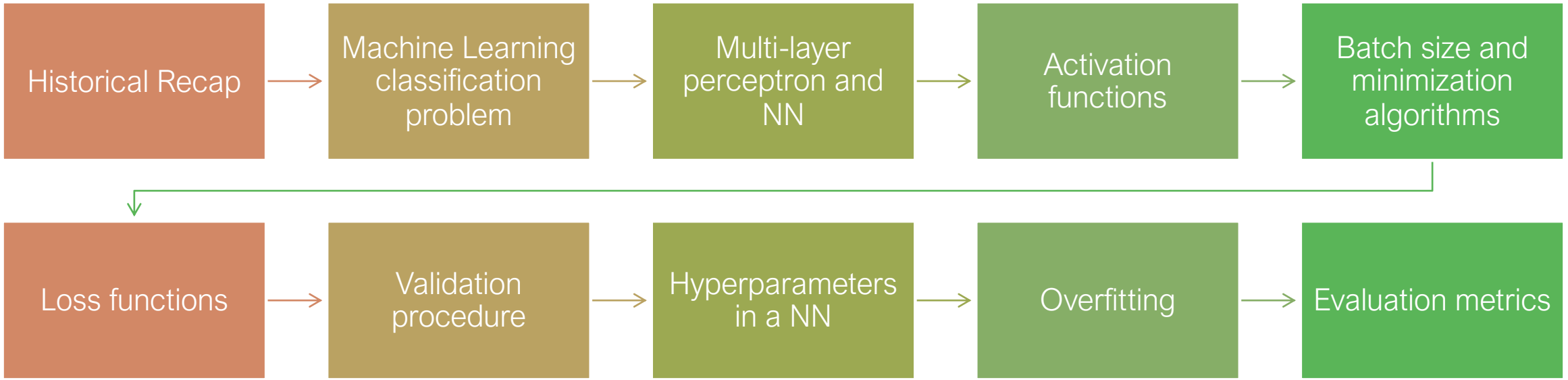
$$J(\boldsymbol{w}) = \sum_{i=1}^{n}\left[-y^{(i)}\log\left(\phi\left(z^{(i)}\right)\right) - \left(1-y^{(i)}\right)\log\left(1-\phi\left(z^{(i)}\right)\right)\right]$$

GOING DEEPER INTO THE LR COST FUNCTION…

ARTIFICIAL NEURAL NETWORKS



CONTENTS


Historical Recap
Machine Learning classification problem
Multi-layer perceptron and NN
Activation functions
Batch size and minimization algorithms
Loss functions
Validation procedure
Hyperparameters in a NN
Overfitting
Evaluation metrics

HISTORICAL RECAP

Artificial neurons represent the building blocks of the multi-layer ANNs

→

The basic concept behind artificial neural based on hypotheses and models of how the human brain works to solve complex problem tasks.

Early studies of NNs in 1940s with Warren McCulloch and Walter Pitt

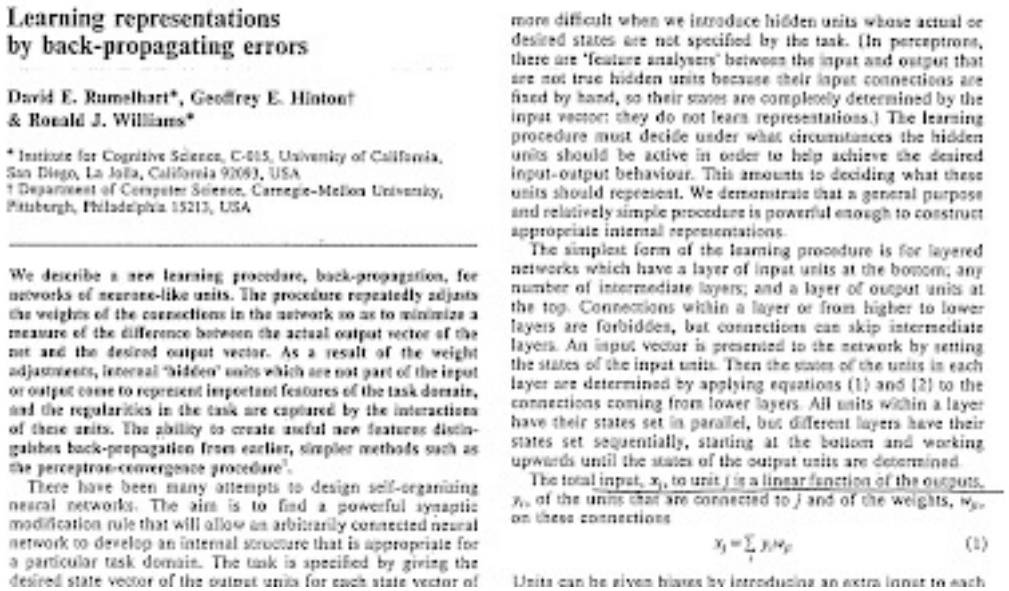
→

In 1950s the first implementation of the McCulloch-Pitt neuron model and then Rosenblatt's perceptron in the 1950s

After 1950s many researchers and machine learning practitioners lost interest in NNs since no one had a good solution for training a neural network with multiple layers.

→

were involved in the (re)discovery and popularization of the backpropagation algorithm t o train neural networks more efficiently (Rumelhart, David E.; Hinton, Geoffrey E.; Williams, Ronald J. (1986). *Learning*



Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

There have been many attempts to design self-organizing neural networks. The aim is to find a powerful synaptic modification rule that will allow an arbitrarily connected neural network to develop an internal structure that is appropriate for a particular task domain. The task is specified by giving the desired state vector of the output units for each state vector of

more difficult when we introduce hidden units whose actual or desired states are not specified by the task. (In perceptrons, there are 'feature analysers' between the input and output that are not true hidden units because their input connections are fixed by hand, so their states are completely determined by the input vector: they do not learn representations.) The learning procedure must decide under what circumstances the hidden units should be active in order to help achieve the desired input-output behaviour. This amounts to deciding what these units should represent. We demonstrate that a general purpose and relatively simple procedure is powerful enough to construct appropriate internal representations.

The simplest form of the learning procedure is for layered networks which have a layer of input units at the bottom; any number of intermediate layers; and a layer of output units at the top. Connections within a layer or from higher to lower layers are forbidden, but connections can skip intermediate layers. An input vector is presented to the network by setting the states of the input units. Then the states of the units in each layer are determined by applying equations (1) and (2) to the connections coming from lower layers. All units within a layer have their states set in parallel, but different layers have their states set sequentially, starting at the bottom and working upwards until the states of the output units are determined.

The total input, x_j, to unit j is a linear function of the outputs, y_i, of the units that are connected to j and of the weights, w_{ji}, on these connections

$$x_j = \sum_i y_i w_{ji} \qquad (1)$$

Units can be given biases by introducing an extra input to each

ML CLASSIFICATION PROBLEM

A **Neural Network** is a **very powerful classification algorithm**

The **simplest version of a Neural Network is the Perceptron**

Perceptron can be considered the building block of a NN

GOAL: the discrimination between 2 classes or among more classes through a training done on a specific dataset.

The algorithm, during the train, adapts several number of parameters in order to discriminate the defined classes

It is quite similar to perceptron algorithm, but we have a more complicated parameter structure

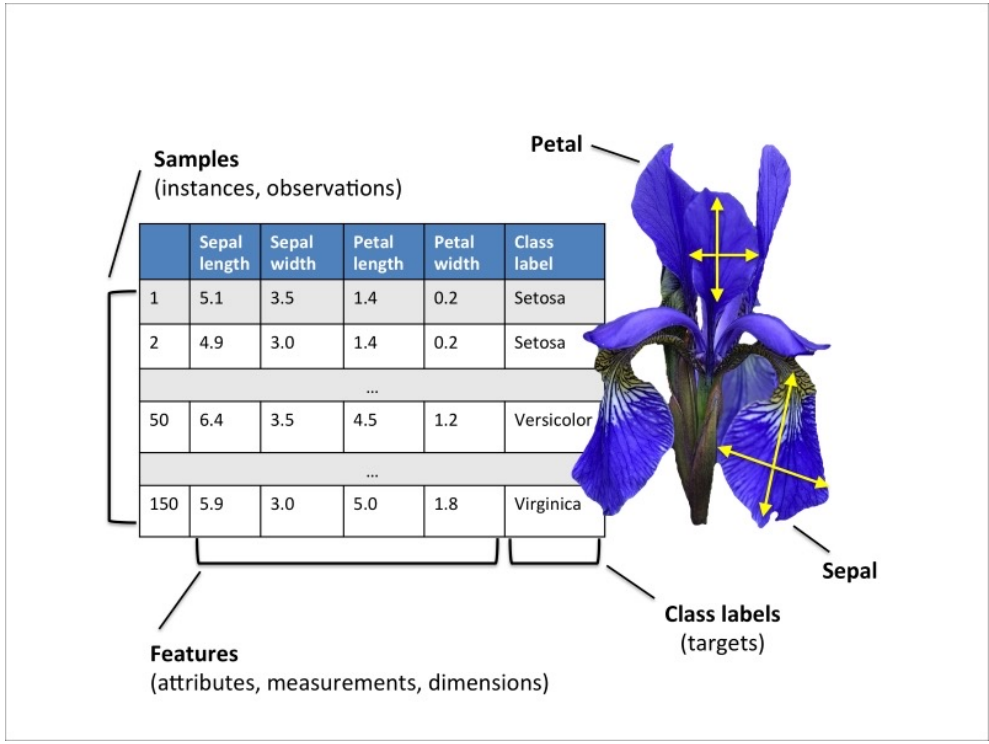
CLASSIFICATION PROBLEM

FIRST STEP: selection of our dataset in which there are a set of features that describes what we want to discriminate (for example the discrimination of different types or irises)

$$\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & x_4^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & x_4^{(2)} \\ \vdots & \vdots & \vdots & \vdots \\ x_1^{(150)} & x_2^{(150)} & x_3^{(150)} & x_4^{(150)} \end{bmatrix}$$

SECOND STEP: defining classes and targets (in the previous example: Iris-setosa, Iris-versicolor and Iris-virginica)

$$\boldsymbol{y} = \begin{bmatrix} y^{(1)} \\ \dots \\ y^{(150)} \end{bmatrix} \left(y \in \{\text{Setosa, Versicolor, Virginica}\} \right)$$



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica

CLASSIFICATION PROBLEM

- We often start from the X (feature array) and Y (label array) of information.
- For our algorithm we need to transform the Y information in a numeric information in order to insert that in the loss function.
- The simplest way to do that is to order our classes from 0 to N where N is the number of classes but we must avoid to use it, because it adds an intrinsic order between our classes: the algorithm could prefer the class with the highest number due to the definition of the loss function.
- The most used method is to create dummy variables

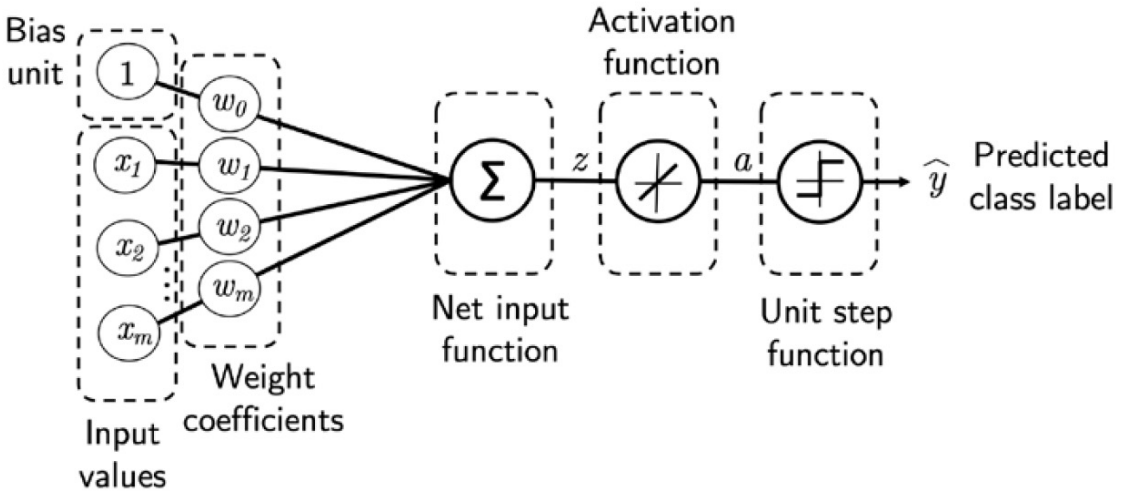
$$class\ 1 = \{1,0,0,\dots,0\}$$
$$class2 = \{0,1,0,\dots,0\}$$
$$\dots$$
$$classN = \{0,0,0,\dots,1\}$$

FOCUS ON:
A dummy variable is a variable that takes values of 0 and 1, where the values indicate the presence or absence of something.

CLASSIFICATION PROBLEM

- The **simplest algorithm** able to discriminate **between two classes** (**binary classificator**) is the **perceptron.**
- The **perceptron is the fundamental building-block of a Neural Network** but…we will need a lot of them!
- In a single perceptron we have:
  - **m+1 weights** (where m is the number of input variables)
  - an **activation function**
  - an **output that is a linear function of inputs and weights**
  - In every **epoch** (pass over the training set), we updated the weight vector :

ADALINE



$$z = \sum_j w_j x_j = \boldsymbol{w}^T \boldsymbol{x}$$

$$\phi(z) = z = a$$

$$\hat{y} = \begin{cases} 1 & \text{if } g(z) \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

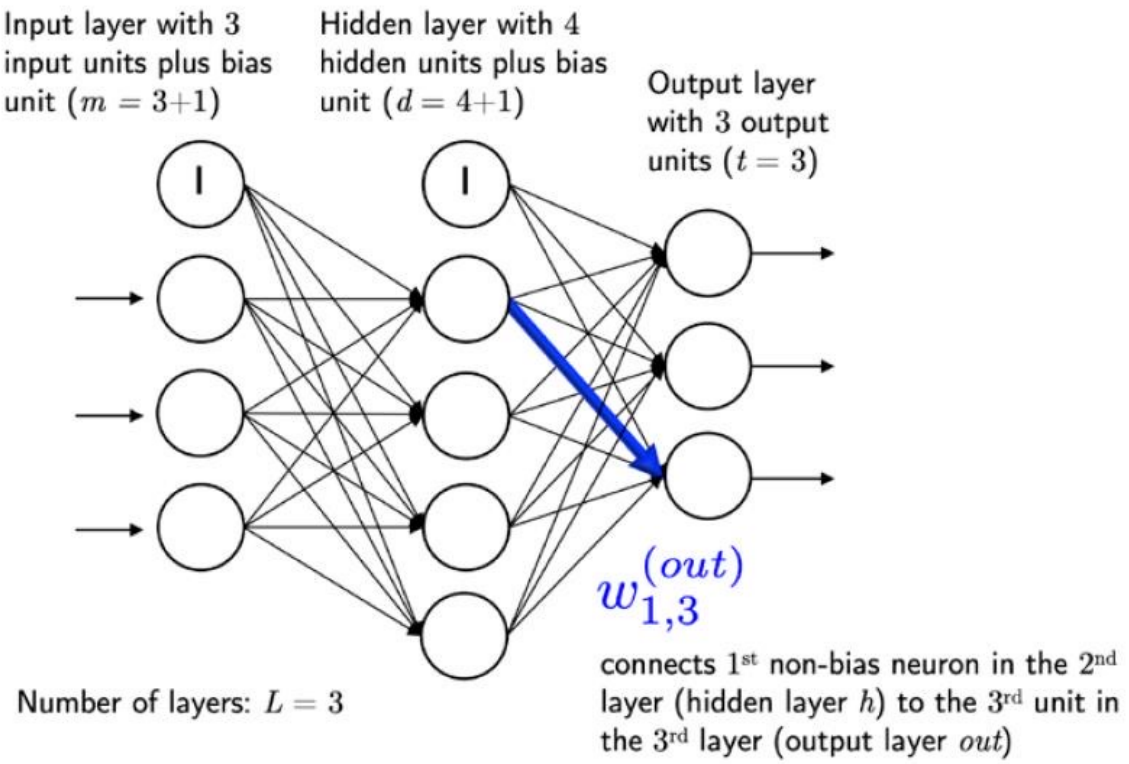
$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w}, \text{ where } \Delta \boldsymbol{w} = -\eta \nabla J(\boldsymbol{w})$$

LINEAR OR NOT LINEAR…THIS IS THE QUESTION!

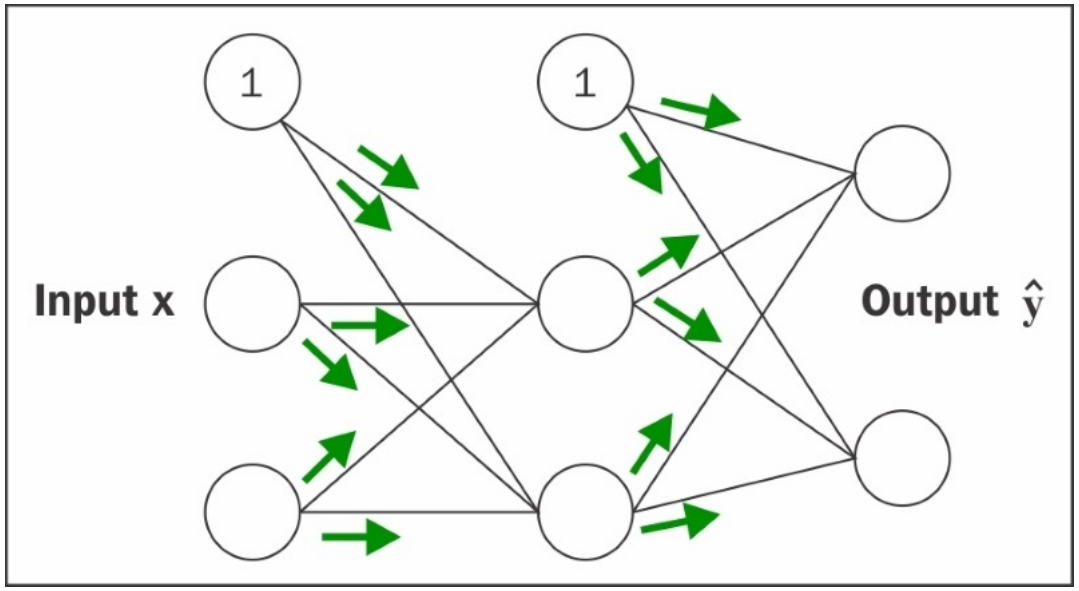
- Linear approach in the perceptron implies the weaker assumption of monotonicity: any increase in our feature must always cause an increase in our model's output (if the corresponding weight is positive), or always causes a decrease in our model's output (if the corresponding weight is negative).
  - LIMIT: the probability related to something is not always proportional to features
  - Use Case: we want to predict probability of death based on body temperature. For individuals with a body temperature above 37°C , higher temperatures indicate greater risk. However, for individuals with body temperatures below 37°C, higher temperatures indicate lower risk! In this case, we might resolve the problem with some clever pre-processing. Namely, we might use the distance from 37°C as our feature.
- In order to overcome these limitations of linear models and handling a more general class of functions by incorporating one or more hidden layers between input and output.
- We can introduce non linearity also in the activation function:
  - the output became a complicated function of the input variables → losing the linearity of our model.

MULTI-LAYER PERCEPTRON

- First example of NN: **Multilayer perceptron**
  - a net of fully connected perceptron
- In the schematical view on the right every circle is a perceptron with a fixed number of inputs and outputs
- In the example (on the right) we have **an input layer,** only one **hidden layer** and an **output layer**



MULTI-LAYER PERCEPTRON



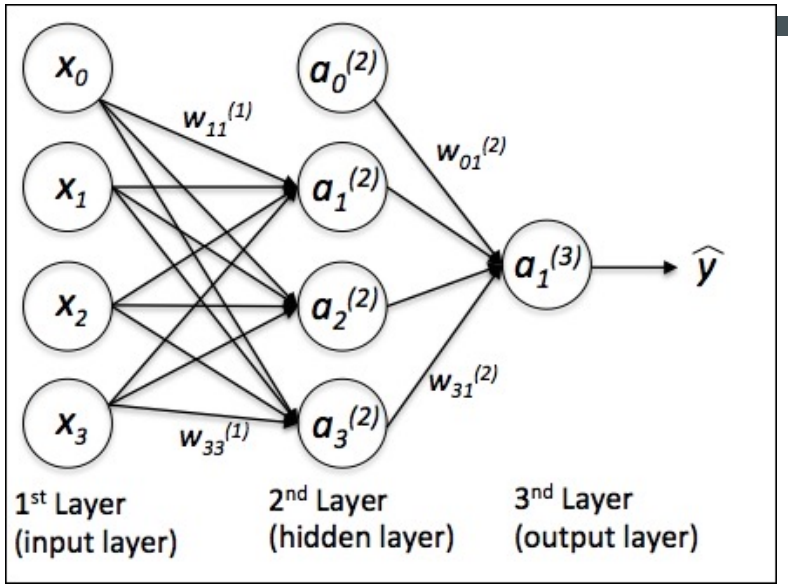
- For every line (which represents a connection from a neuron belonging to a layer to a neuron belonging to the next one) we have an associated weight.
- In this case we just have:
  - 2 input variables
  - an hidden layer with size 2 (i.e. with 2 neurons)
  - and 2 outputs,
- $(\#neurons\ \ in\ the\ next\ layer \cdot \#variables) + \#neurons\ in\ the\ layer = 6\ weigths$

  (between input layer and hidden layer)
- $(\#neurons\ in\ the\ next\ layer \cdot \#hidden\ layers\ size) + neurons\ in\ the\ layer = 6\ weigths$

  (between hidden layer and output layer)

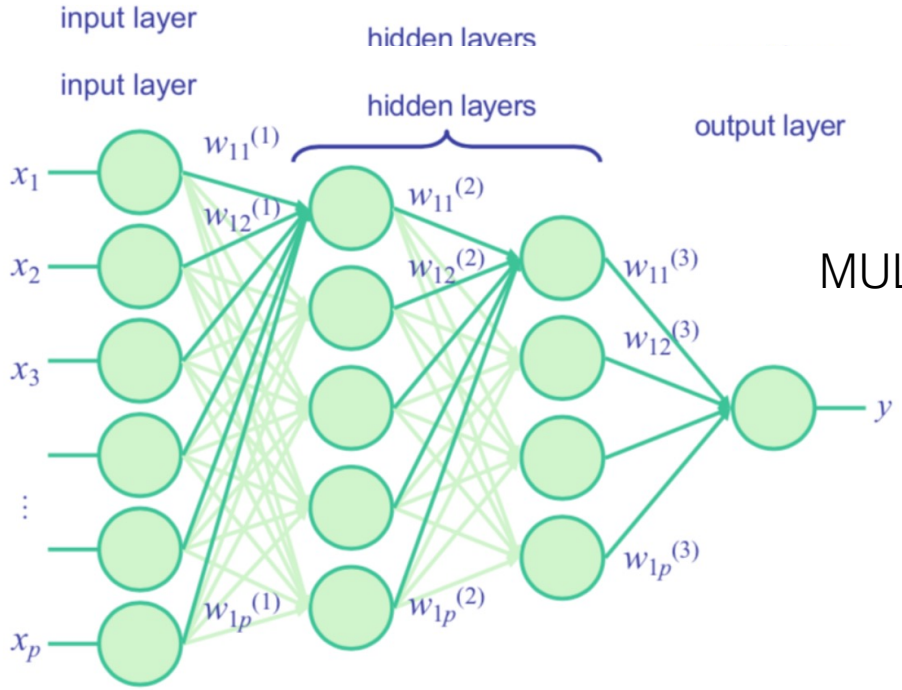
TOTAL=12 WEIGHTS

INTRODUCING
THE MULTI-LAYER NEURAL NETWORK ARCHITECTURE



3 LAYERS

- Problem: how can we connect multiple single neurons to a **multi-layer feedforward neural network**?
  - This special type of network is also called a **multi-layer perceptron** (**MLP**).
  - The following figure explains the concept of an MLP consisting of three layers: one input layer, one **hidden layer**, and one output layer.
  - The **units in the hidden layer are fully connected to the input layer,** and **the output layer is fully connected to the hidden layer**.
  - If such a network has **more than one hidden layer**, we also call it a ***deep*** **artificial neural network**.



MULTIPLE LAYERS

FOCUS ON TERMINOLOGY

Hidden layer: it is a layer between the input layer and the output layer. It takes in a set of weighted inputs and produces output through an activation function. This layer is named hidden because it does not constitute the input or the output layer.

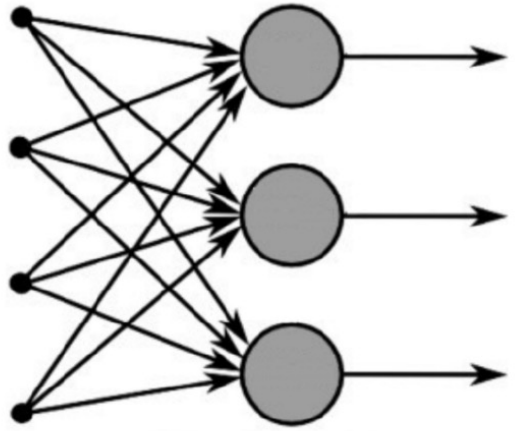
Fully connected: A fully connected layer refers to a neural network in which each neuron applies a linear transformation to the input vector through a weights matrix. As a result, all possible connections layer-to-layer are present, meaning every input of the input vector influences every output of the output vector.

Dense layer: a NN layer is called a dense layer to indicate that it's fully connected.

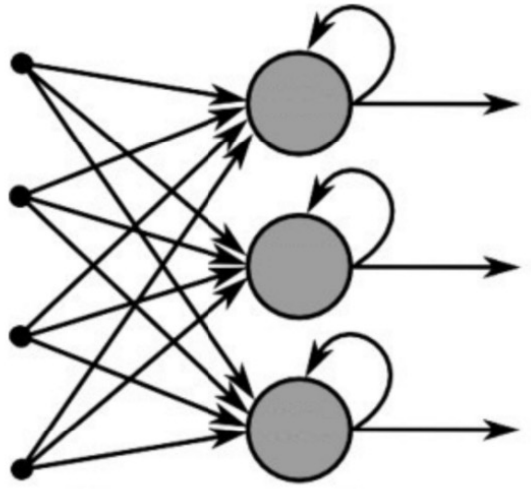
Feed Forward Networks: a neural network where connections between the nodes do not form a cycle. In a feed-forward network information always moves one direction, from input to output, and it never goes backward. Feedforward NN can be viewed as mathematical models of a function f: $R^N \rightarrow R^M$

FOCUS NN TOPOLOGIES

- A Neural Networks (NN) can be classified according to the type of neuron interconnections and the flow of information:
  - **Feed Forward Networks:** a neural network where connections between the nodes do not form a cycle. In a feed-forward network information always moves one direction, from input to output, and it never goes backward. Feedforward NN can be viewed as mathematical models of a function f: $R^N \rightarrow R^M$
  - **Recurrent Neural Network:** allows connections between nodes in the same layer, among each other or with previous layers. Unlike feedforward neural networks, RNNs can use their internal state (memory) to process sequential input data.
- We could add an arbitrary number of hidden layers to the MLP to create deeper network architectures. Practically, we can think of the number of layers and units in a neural network as additional **hyperparameters** that we want to optimize for a given problem task
- The error gradients calculated later **via backpropagation would become increasingly small as more layers are added to a network.**
- This *vanishing gradient* problem makes the model learning more challenging. Therefore, special algorithms have been developed to pretrain such deep neural network structures, which is called *deep learning*.



Feed-Forward Neural Network

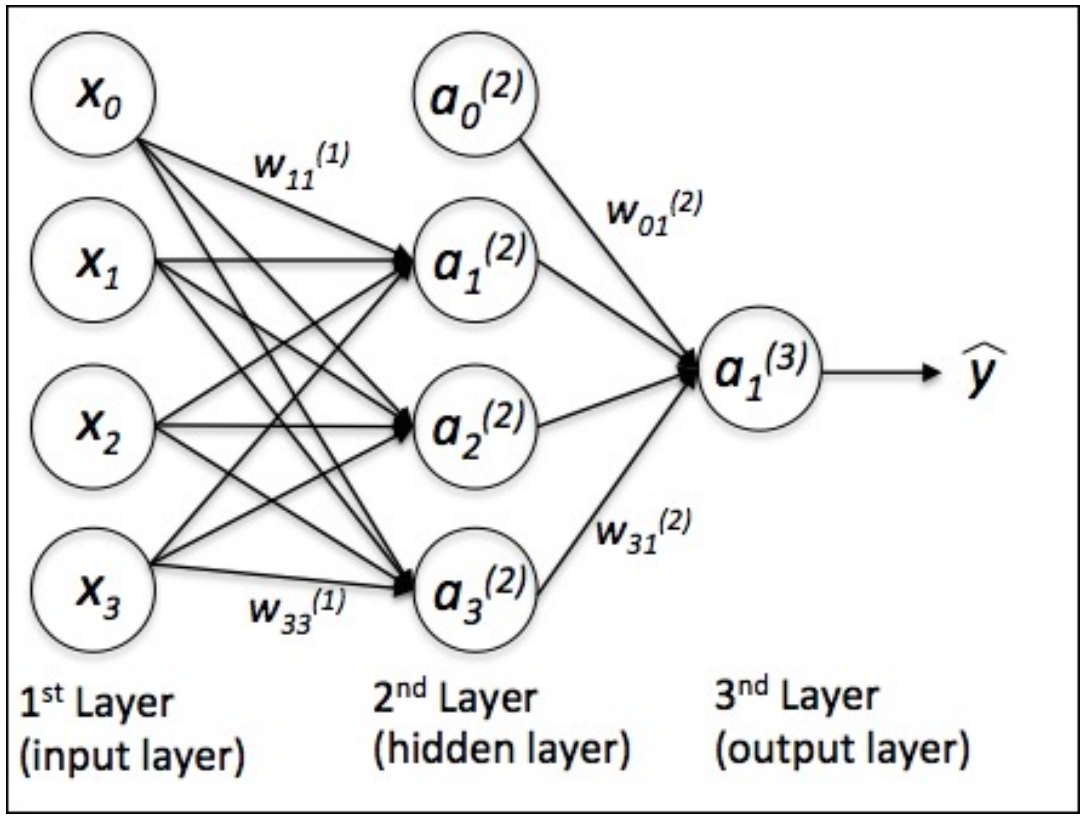


Recurrent Neural Network

INTRODUCING
THE MULTI-LAYER NEURAL NETWORK ARCHITECTURE

- we denote the *ith* activation unit in the *lth* layer as $a_i^{(l)}$
- the activation units $a_0^{(1)}$ e $a_0^{(2)}$ are the **bias units** set equal to 1
- The activation of the units in the input layer is just its input plus the bias unit:

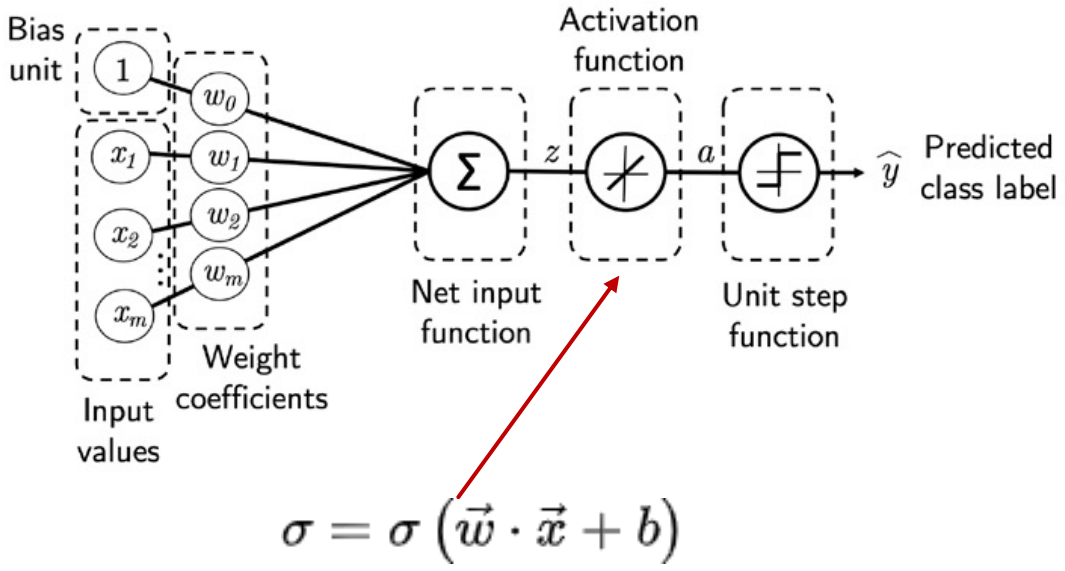
$$a^{(1)} = \begin{bmatrix} a_0^{(1)} \\ a_1^{(1)} \\ \vdots \\ a_m^{(1)} \end{bmatrix} = \begin{bmatrix} 1 \\ x_1^{(i)} \\ \vdots \\ x_m^{(i)} \end{bmatrix}$$



- Each unit in layer *l* is connected to all units in layer *l+1* via a weight coefficient.
- For example, the connection between the *kth* unit in layer l to the *jth* unit in layer *l+1* would be written as $w^l_{j,k}$
- The superscript *i in* $x_m^{(i)}$ stands for th *ith* sample (not layer)
- While one unit in the output layer would suffice for a binary classification task, we saw a more general form of a neural network to perform multi-class classification via a generalization of the **One-vs-All** (**OvA**) technique→ **one-hot** representation of categorical variables.
  - For example, we would encode the three class labels in the familiar Iris dataset (0=Setosa, 1=Versicolor, 2=Virginica) as follows:

$$0 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, 1 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, 2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

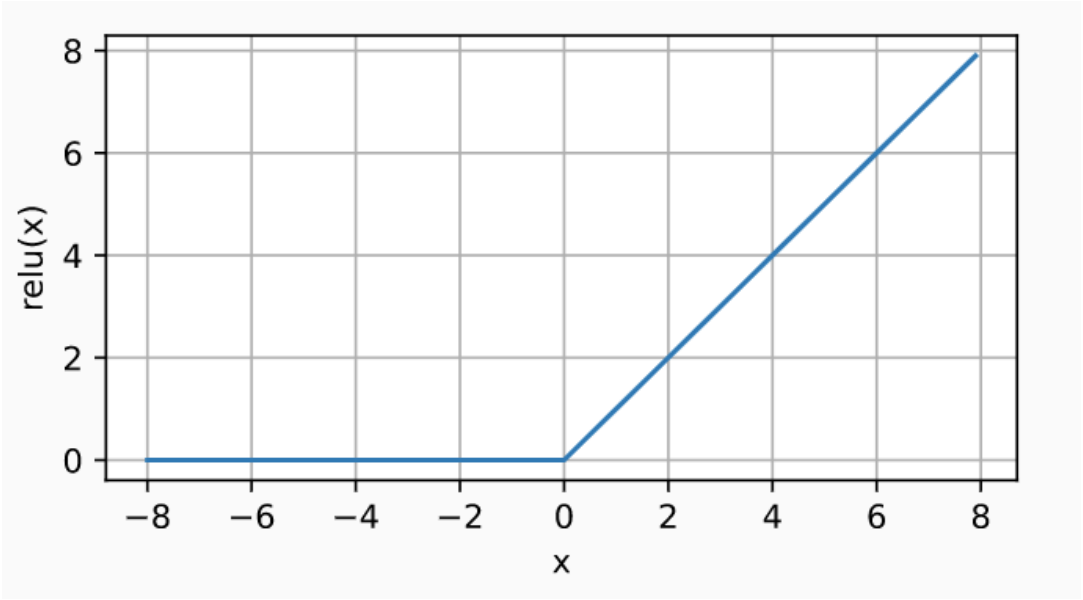
ACTIVATION FUNCTION



- The activation function provides to the activation or not of a node
- The functions are in general differentiable operators in order to transform the inputs to outputs
- Most of them provides to add non-linearity to the model
- The activation function σ has as input the weighted sum of the input variables x, added with the bias b

RECTIFIED LINEAR UNIT (RELU)

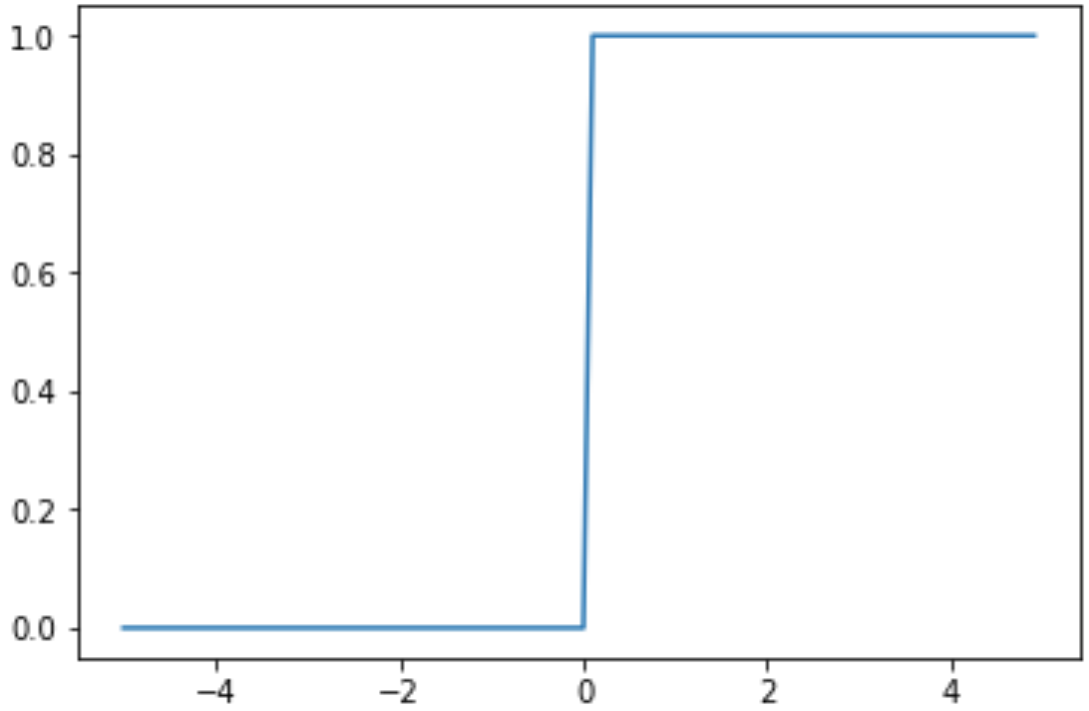
- One the most popular non-linear activation function is the REctified Linear Unit (ReLU).
- It provides a non-linear transformation **and returns the max value between the input x (the argument) and 0.**



$$ReLU(x) = max(0, x)$$

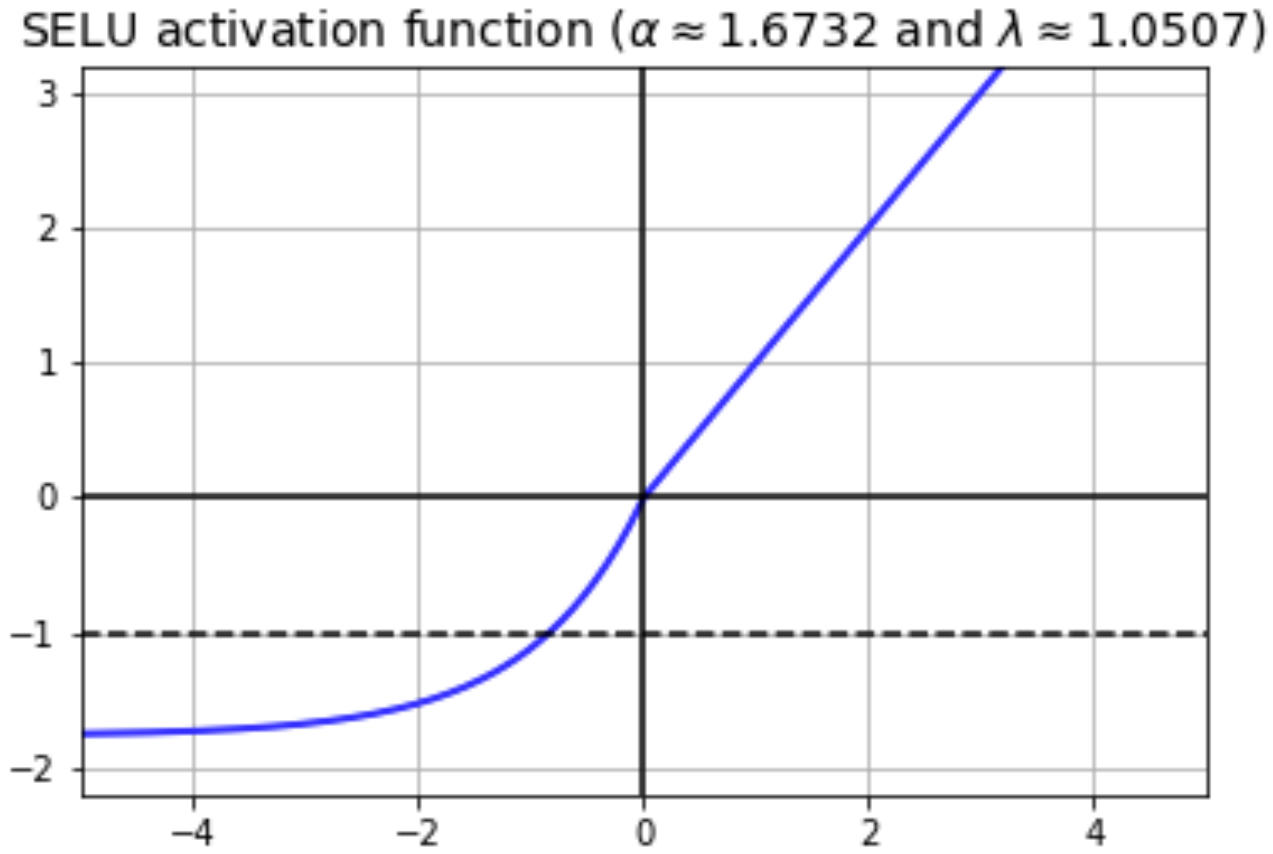
RECTIFIED LINEAR UNIT (RELU)

- The ReLU function is also differentiable in R-{0} and its derivative is the Heaviside function.
- In case the input is equal to zero, it is used the left-side derivative.



$$\frac{dReLU(x)}{dx} = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

SCALED EXPONENTIAL LINEAR UNIT (SELU)

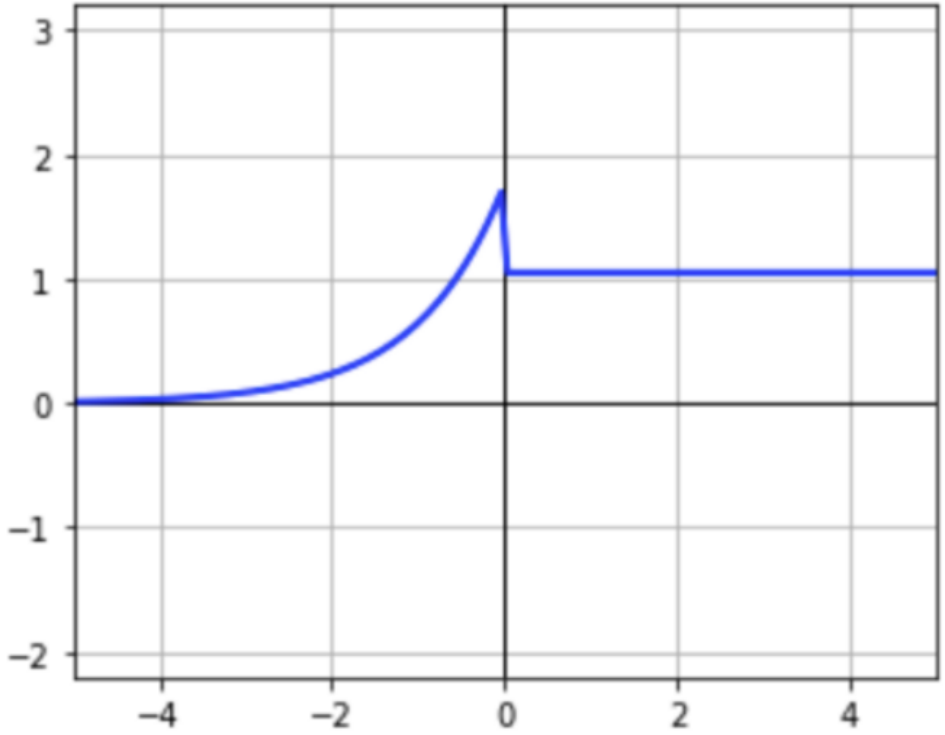


- Another choice is the Scaled Exponential Linear Unit (SELU).
- The functions depends on two parameters and the equation is the following:

$$SELU(x) = \lambda \begin{cases} \alpha(e^x - 1) & x \leq 0 \\ x & x > 0 \end{cases}$$

SCALED EXPONENTIAL LINEAR UNIT (SELU)

- The function is not differentiable in zero.
- Also here is convention to use the left-side value of its derivative.



$$\frac{dSELU(x)}{dx} = \lambda \begin{cases} \alpha e^x & x \leq 0 \\ 1 & x > 0 \end{cases}$$

SIGMOID FUNCTION

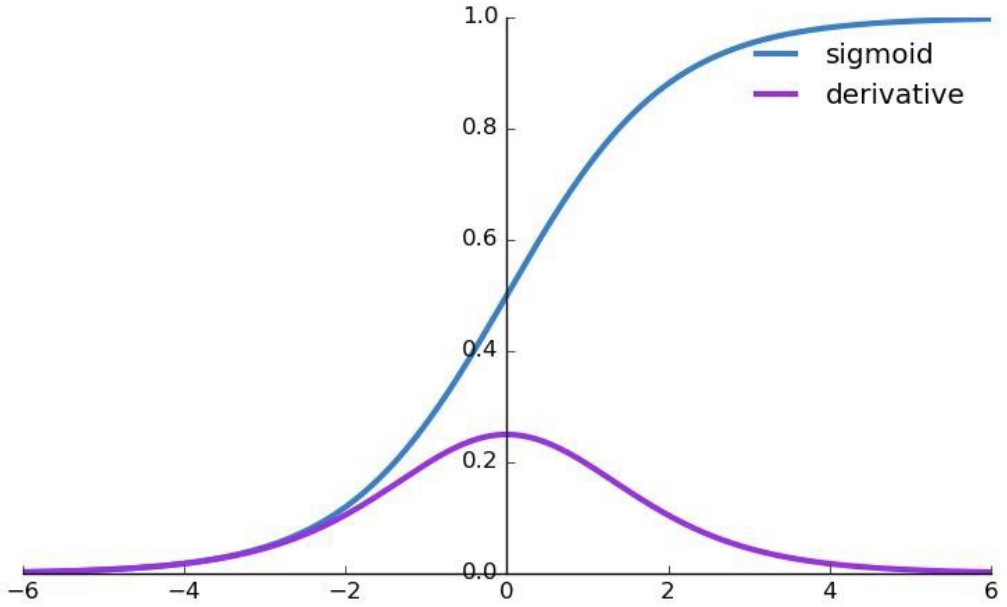
- Sigmoid function:

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

- Derivative Sigmoid function:

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

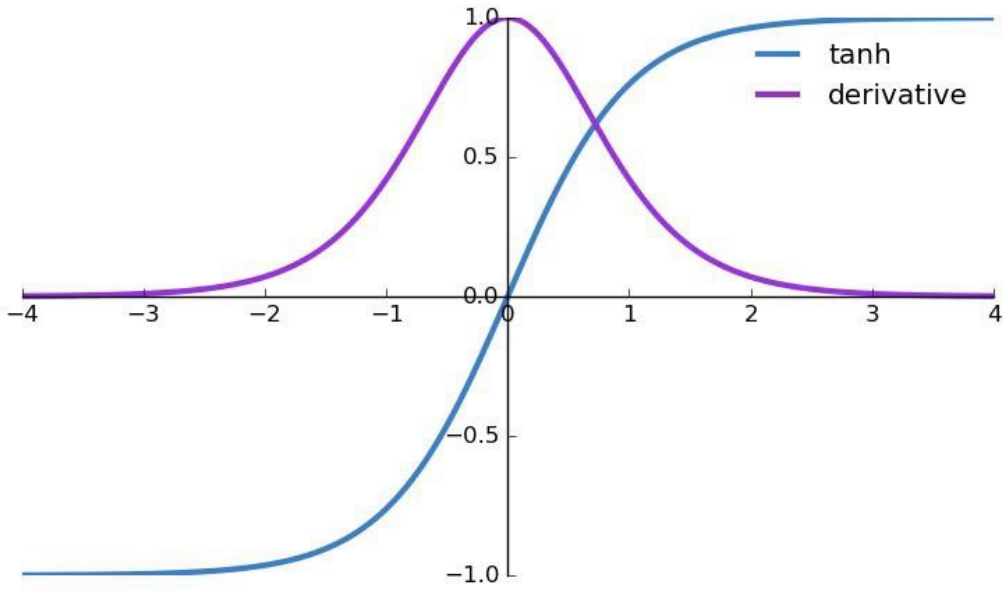
$$= sigmoid(x)(1 - sigmoid(x))$$



TANH FUNCTION

- Tanh function:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}$$



- Derivative Tanh function:

$$\frac{d\, tanh(x)}{dx} = 1 - tanh^2(x)$$

MULTILAYER PERCEPTRON

- Example: in this case there are two outputs.
- The hidden layer output **h** is function of the input **x**:

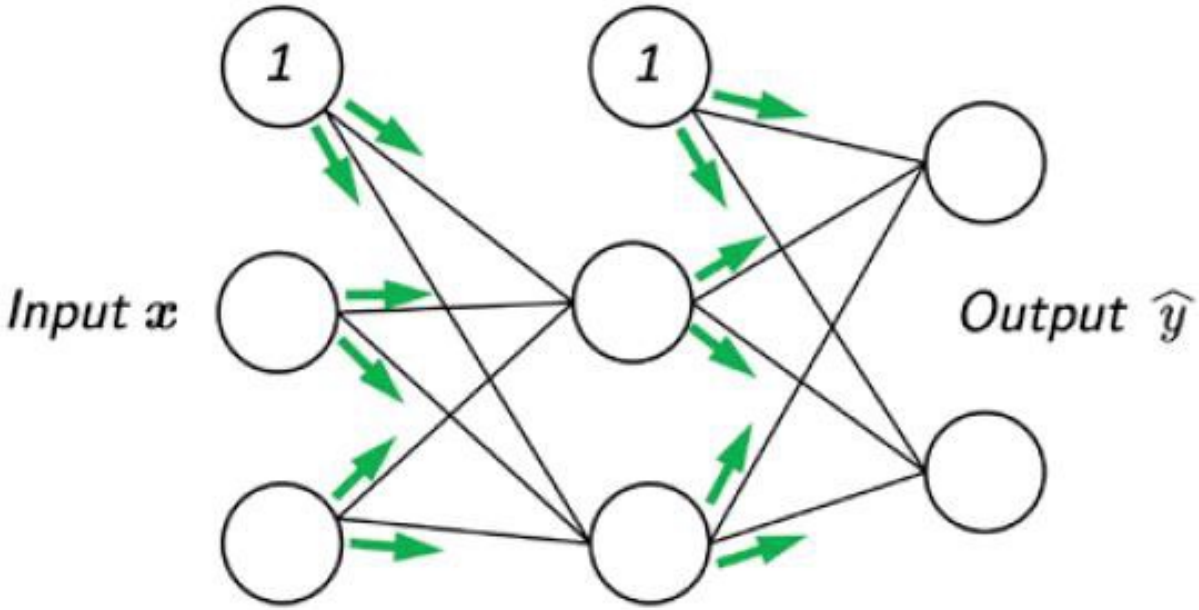
$$h_1 = \sigma^h(w^i_{11}x_1 + w^i_{12}x_2 + b^i_1)$$

$$h_2 = \sigma^h(w^i_{21}x_1 + w^i_{22}x_2 + b^i_2)$$

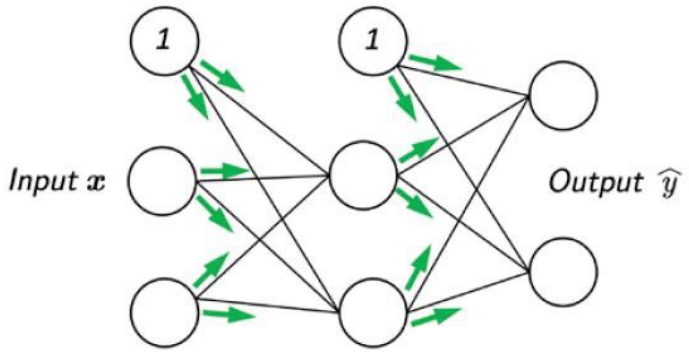
- The output **o** is a different function of its input, i.e. **h**:

$$o_1 = \sigma^o(w^h_{11}h_1 + w^h_{12}h_2 + b^h_1)$$

$$o_2 = \sigma^o(w^h_{21}h_1 + w^h_{22}h_2 + b^h_2)$$



MULTILAYER PERCEPTRON



$$\sum_{i=1}^{\#classes} o_i \neq 1$$

$$o_i \not\geq 0 \;\; \forall \, i \in [1, \#classes]$$

- How can we interpret the two values?
- In classification problem the goal is to understand **how** the input **x** is **related to** a certain **class**
- The output **o** could be seen as the **vector of probabilities of belonging to each class**
- However this is not straightforward

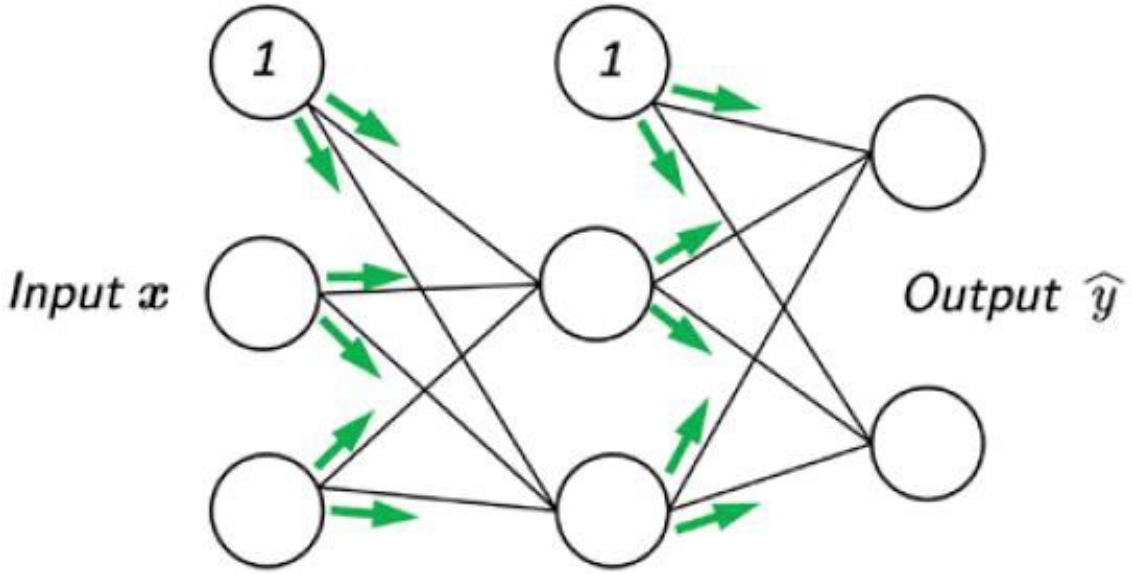
SOFTMAX REGRESSION

- To solve this issue→ we define Softmax activation function which is defined as follows:

$$\mathbf{y} = softmax(\mathbf{o})$$

$$y_i = \frac{e^{o_i}}{\sum_k e^{o_k}}$$

$$y_{pred} = \underset{j}{argmax}\ y_j$$



- So that (for construction):

$$\sum_{i=1}^{\#classes} o_i = 1$$

$$o_i > 0 \quad \forall\, i \in [1, \#classes]$$

PARAMETRIZATION COST

- The layers we are handling are fully-connected
- Adding new neurons (perceptrons) to a network layer or adding a new layer makes our model more complex and capable of facing a wide range of more challenging problems→ we are facing PARAMETRIZATION COST
- The complexity of the model, however, faces directly with the increase of computational time, which could become extremely high.

PARAMETRIZATION COST

- Suppose to have a hidden layer with **d input and q outputs**.

$$q = \#neurons$$

- The parametrization cost is $\propto \mathcal{O}(d \cdot q)$
- It is possible to reduce the parametrization cost introducing an **hyperparameter n** so that: $\mathcal{O}\left(\frac{d \cdot q}{n}\right)$

VECTORIZATION FOR MINIBATCHES → BATCH SIZE N

- Input $X \in \mathcal{R}^{n \times d}$
- Weights $W \in \mathcal{R}^{d \times q}$
- Bias $b \in \mathcal{R}^{1 \times q}$
- Outputs $O = XW + b,\ O \in \mathcal{R}^{n \times q}$

TELLS US

How many samples are feeding our network AT THE SAME TIME!

SAVING COMPUTATIONAL TIME

LOSS FUNCTION

- To measure the quality of our predicted probabilities we need a loss function
- We will suppose that the **entire dataset** (or the batch we are considering) **has n samples** {X,Y}
- The **i-th{X,Y} entry** is made by the feature vector **x-ith** and the **one-hot label vector y-ith**
- The **predicted class** can **be compared with the real class** by checking the probability associated to the actual class according to the model.
- According to the maximum likelihood estimation, we want to maximize P(Y|X), or minimize the negative log-likelihood.

$$\mathcal{P}(\mathbf{Y}|\mathbf{X}) = \prod_{i=1}^{n} \mathcal{P}(\mathbf{y}^{(i)}|\mathbf{x}^{(i)})$$

$$-log\mathcal{P}(\mathbf{Y}|\mathbf{X}) = \sum_{i=1}^{n} -log\mathcal{P}(\mathbf{y}^{(i)}|\mathbf{x}^{(i)})$$

CROSS-ENTROPY LOSS FUNCTION

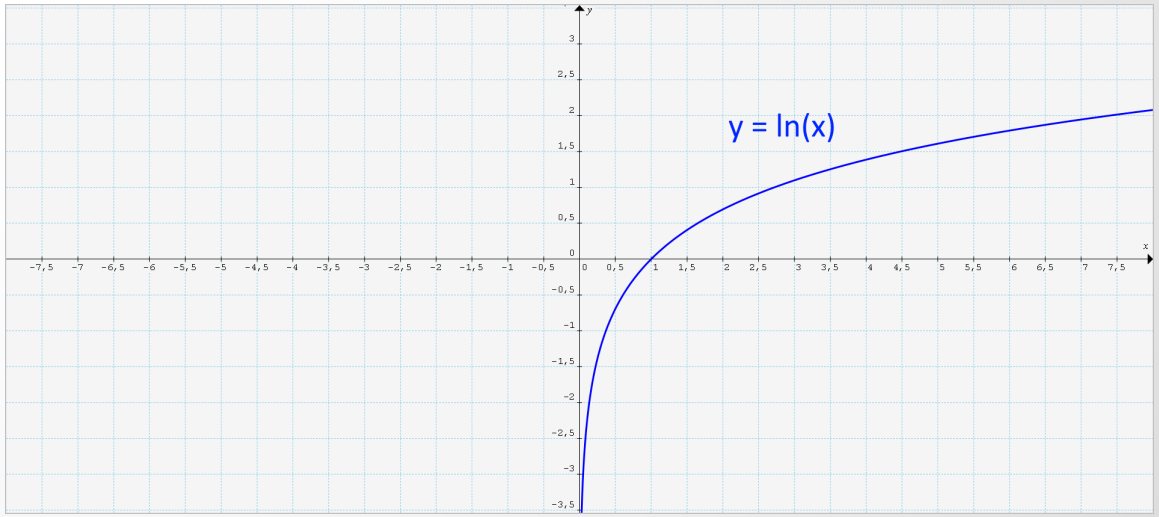
- The negative log-likelihood is equal to:

$$-log\mathcal{P}(\mathbf{Y}|\mathbf{X}) = \sum_{i=1}^{n} l(\mathbf{y}^{(i)}, \mathbf{y}_{pred}^{(i)})$$

- Where **l(y,ypred)** is the loss function, also called **cross-entropy**, defined as:

$$l(\mathbf{y}^{(i)}, \mathbf{y}_{pred}^{(i)}) = -\sum_{j=1}^{\#Classes} y_j^{(i)}\ log\ y_j^{(i)\ pred}$$

$$NB : y_j^{(i)\ pred} \leq 1 \rightarrow log\ y_j^{(i)\ pred} \leq 0$$



CROSS-ENTROPY LOSS FUNCTION

- The cross-entropy loss function is a common choice in classification problems
- Moreover it is generalizable when the vector of label y doesn't contain only binary entries like (1,0,0), but is a generic probability vector

$$l(\mathbf{y}^{(i)}, \mathbf{y}^{(i)}_{pred}) = -\sum_{j=1}^{\#Classes} y_j^{(i)} \, log \, y_j^{(i)\,pred}$$

- This is the case where we observe not just a single outcome but an entire distribution over outcomes

CROSS-ENTROPY LOSS FUNCTION AND SOFTMAX

- The softmax and the corresponding loss function are very common but…what is the corresponding loss function?
- Let's compute it:

REMINDER:

CROSS-ENTROPY

$$l(\mathbf{y}^{(i)}, \mathbf{y}^{(i)}_{pred}) = -\sum_{j=1}^{\#Classes} y_j^{(i)} \, log \, y_j^{(i)\,pred}$$

SOFTMAX

$$y_i = \frac{e^{o_i}}{\sum_k e^{o_k}}$$

$$l(y, y^{pred}) = -\sum_{j=1}^{\#Classes} y_j \, log \frac{e^{o_j}}{\sum_{k=1}^{\#Classes} e^{o_k}}$$

$$= \sum_{j=1}^{\#Classes} y_j \log \sum_{k=1}^{\#Classes} e^{o_k} - \sum_{j=1}^{\#Classes} y_j \, o_j$$

$$= \log \sum_{k=1}^{\#Classes} e^{o_k} - \sum_{j=1}^{\#Classes} y_j \, o_j$$

CROSS-ENTROPY LOSS FUNCTION AND SOFTMAX

- To understand better let's have a look at the derivative w.r.t. any output o i-th:

$$\partial_{o_i} l(y, y^{pred}) = \frac{e^{o_i}}{\sum_{k=1}^{\#Classes} e^{o_k}} - y_i = softmax(o)_i - y_i$$

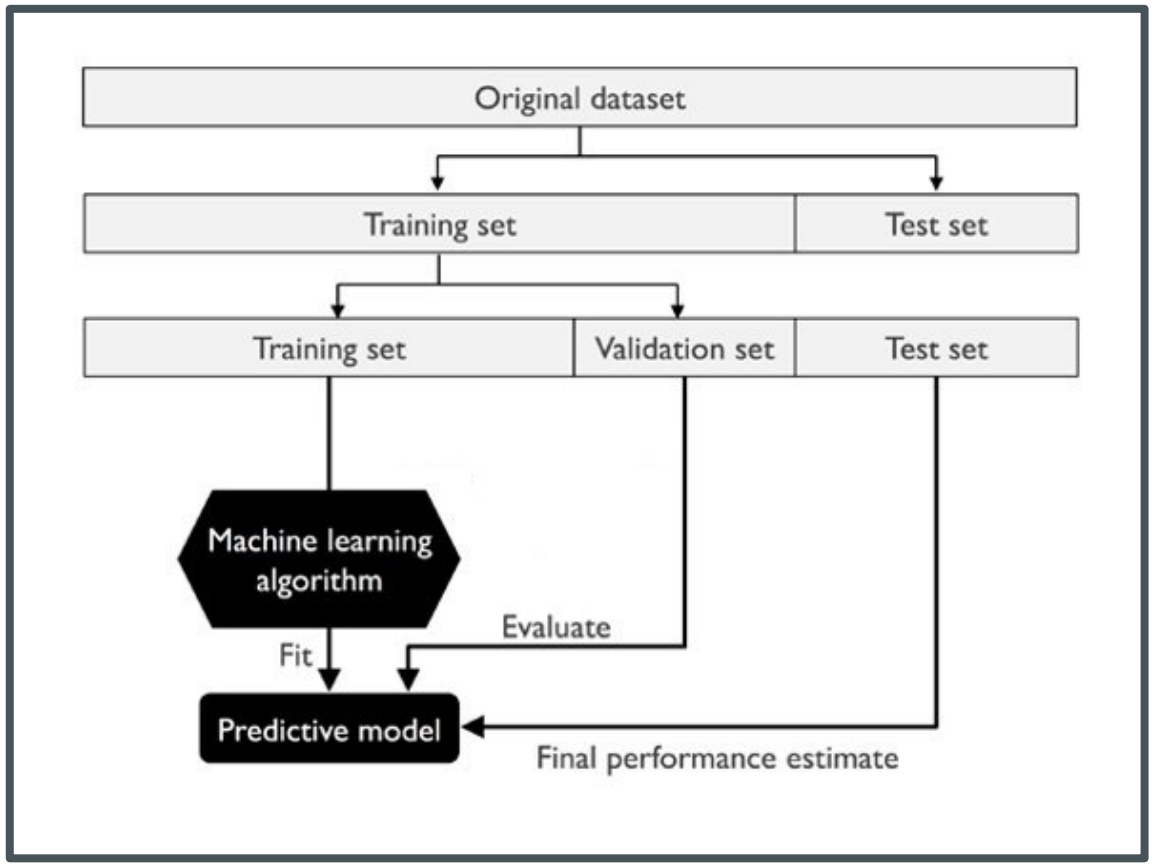
The derivative is the difference between the probability assigned by our model, as expressed by the softmax operation, and the y true vector.

EXAMPLES OF OTHER LOSS FUNCTIONS FOR ANNS

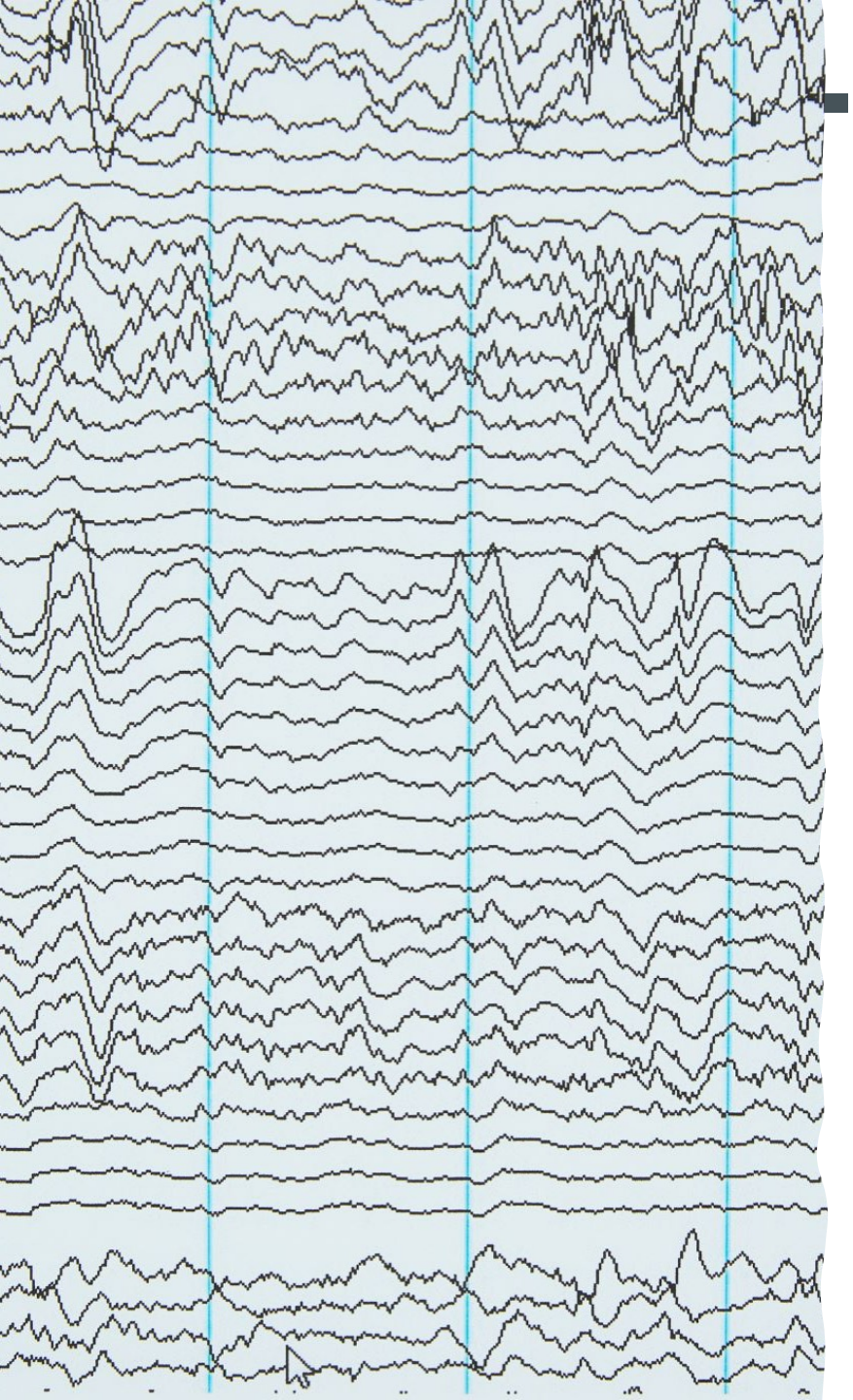
- Mean Squared Error(MSE)/ Quadratic Loss/ L2: $$MSE(y^{(i)}, y^{(i)}_{pred}) = \frac{\left(y^{(i)} - y^{(i)}_{pred}\right)^2}{n}$$
- Mean Absolute Error (MAE)/ L1 Loss: $$MAE(y^{(i)}, y^{(i)}_{pred}) = \frac{\left|y^{(i)} - y^{(i)}_{pred}\right|}{n}$$
- Mean Bias Error (MBE): $$MBE(y^{(i)}, y^{(i)}_{pred}) = \frac{\left(y^{(i)} - y^{(i)}_{pred}\right)}{n}$$

VALIDATION PROCEDURE

VALIDATION



- Task: we want to optimize our model, without touching the test dataset and avoiding the risk of overfitting
- We are going to use our test dataset only after the training is finished in order to assess the very best model
- The best practice to address this problem is to split our dataset in three (instead of two) parts, incorporating a *validation dataset* (or *validation set*) in addition to the training and test datasets.



TRAINING DESCRIPTION

The main idea of the model training is to iterate over the network different times (number of epochs).

In each epoch k stochastic minibatches of n (batchsize) entries (items) are selected from the dataset

We then compute the derivative (gradient) of the average loss on the minibatch regarding the model parameters.

Finally, we multiply the gradient by a predetermined positive value η (the learning rate) and subtract the resulting term from the current parameter values.

The epoch ends after k iterations, i.e. all over the k batches.

MINIBATCH STOCHASTIC GRADIENT DESCENT

- We iteratively sample random minibatches from the data, updating the parameters in the direction of the negative gradient.
- Backward propagation of the training, parameters updating as follows:

$$\mathbf{w} \rightarrow \mathbf{w} - \frac{\eta}{n} \sum_{i=0}^{n} \partial_{\mathbf{w}} \, l^i(\mathbf{w}, \mathbf{b})$$

$$\mathbf{b} \rightarrow \mathbf{b} - \frac{\eta}{n} \sum_{i=0}^{n} \partial_{\mathbf{b}} \, l^i(\mathbf{w}, \mathbf{b})$$

HYPERPARAMETERS

An hyperparameter is an internal parameter of the model that must be fixed before training, such parameter influences the fit procedure in a way not well known a priori.

We cannot know which value is perfect for our model and we need to try different reasonably values to figure out which one is the best.

No Free Lunch theorem: no single classifier works best across all possible scenarios

HYPERPARAMETERS

- With a DNN we can change a lot of parameters, most of which are:
  - The loss function
  - The activation function of every layer
  - The learning rate
  - The number of epochs
  - The number of hidden layers and the number of cells in them
  - …many others
- The hyperparameters can change from an algorithm to another, here we mentioned only the main parameters of a DNN.

LEARNING RATE

- Adjusting the learning rate η is often just as important as the actual algorithm
  - If η is too large the optimization does not converge → we may simply end up bouncing around the minimum and thus not reach optimality
  - if η is too small it takes too long to train or we end up with a suboptimal result
- What we can do: we can decide to start from a reasonable value for the learning rate and then use the method implemented in Keras: "ReduceLROnPlateau".
  - It reduces the value of η when a monitor (set by us) has stopped improving→ we can obtain a large η value at the beginning of the training with a progressive reduction when we are approaching to the optimized model.

NUMBER OF EPOCHS

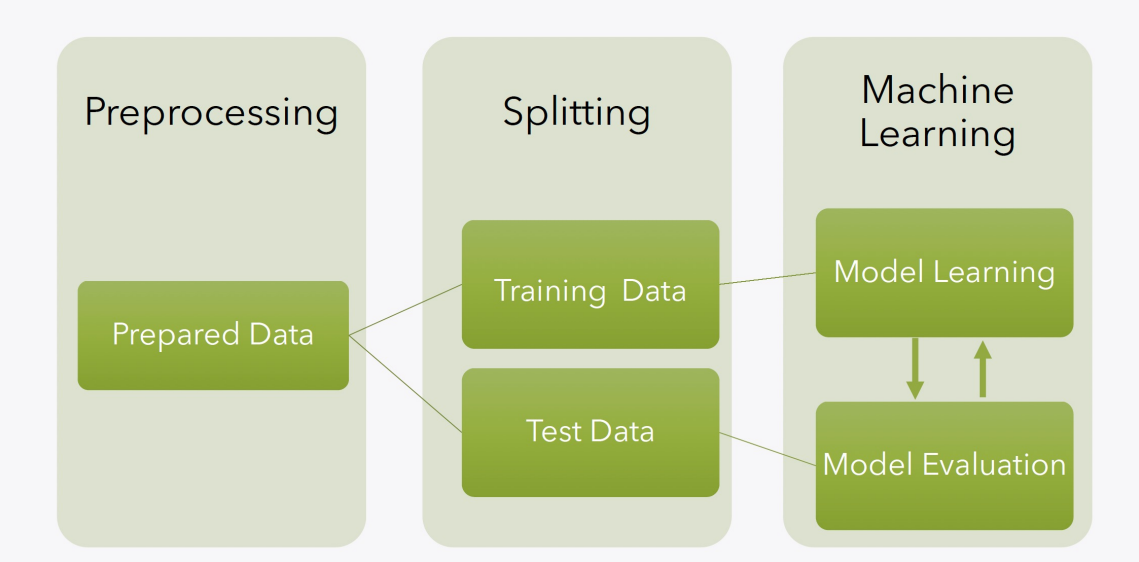
- Epoch: **a complete step of the training**
  - it includes the evaluation and the consecutive updating of the weights
- Number of epochs is a delicate choice:
  - A large number of epochs can induce our model to an overfitting problem
  - a too small number of epochs can lead to an under fitting problem
- **To avoid a wrong choice** we can use the '**EarlyStopping**', also implemented by Keras:
  - it allows to stop the training when a monitor (set by us and tipically the loss function) has stopped improving.

HIDDEN LAYERS

- The number of hidden layers add complexity to our model.
- Adding hidden layers makes our algorithm more performing, but at the same time we lead it to an overfitting problem
- Another crucial factor is the number of cells in the hidden layer, also in this case a lot of cells increase the complexity of the model and increase the risk to an overfitting problem
- This choice has to be done carefully, it is the most difficult one and only comparing the evaluation metrics between different approaches we can know which is the best one.

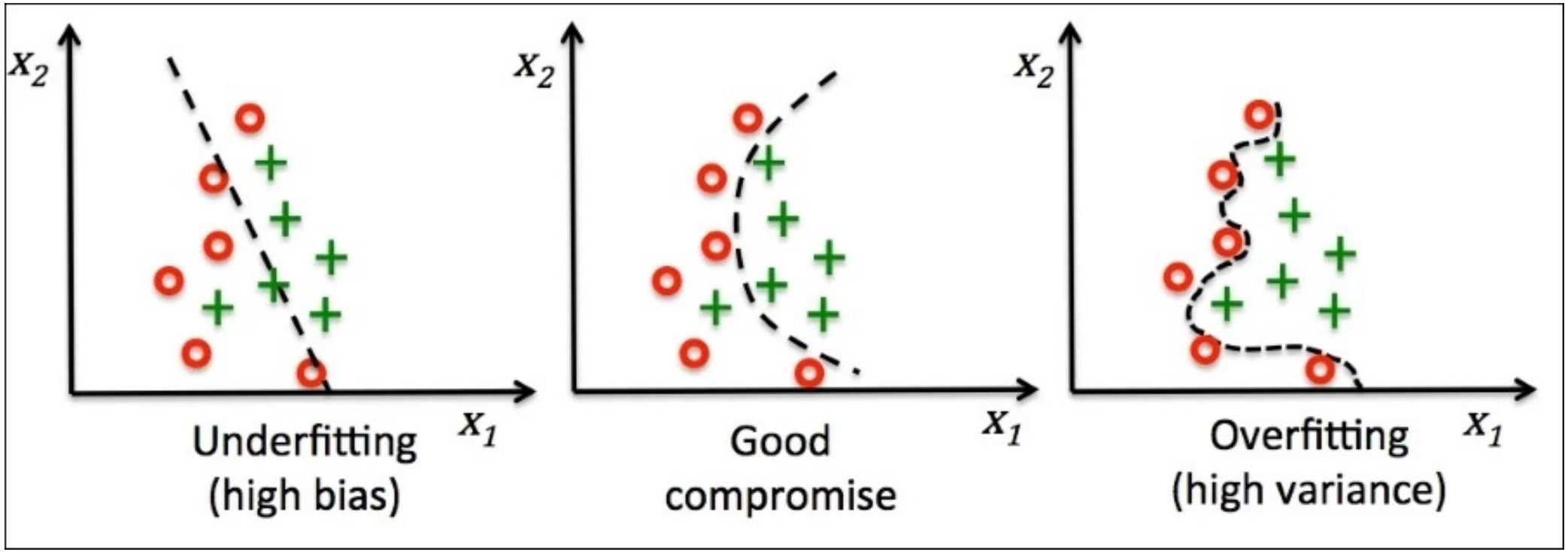
HOW TO ANALYZE DATA IN ML

TRAINING-TEST DATA SPLITTING



OVERFITTING AND UNDERFITTING

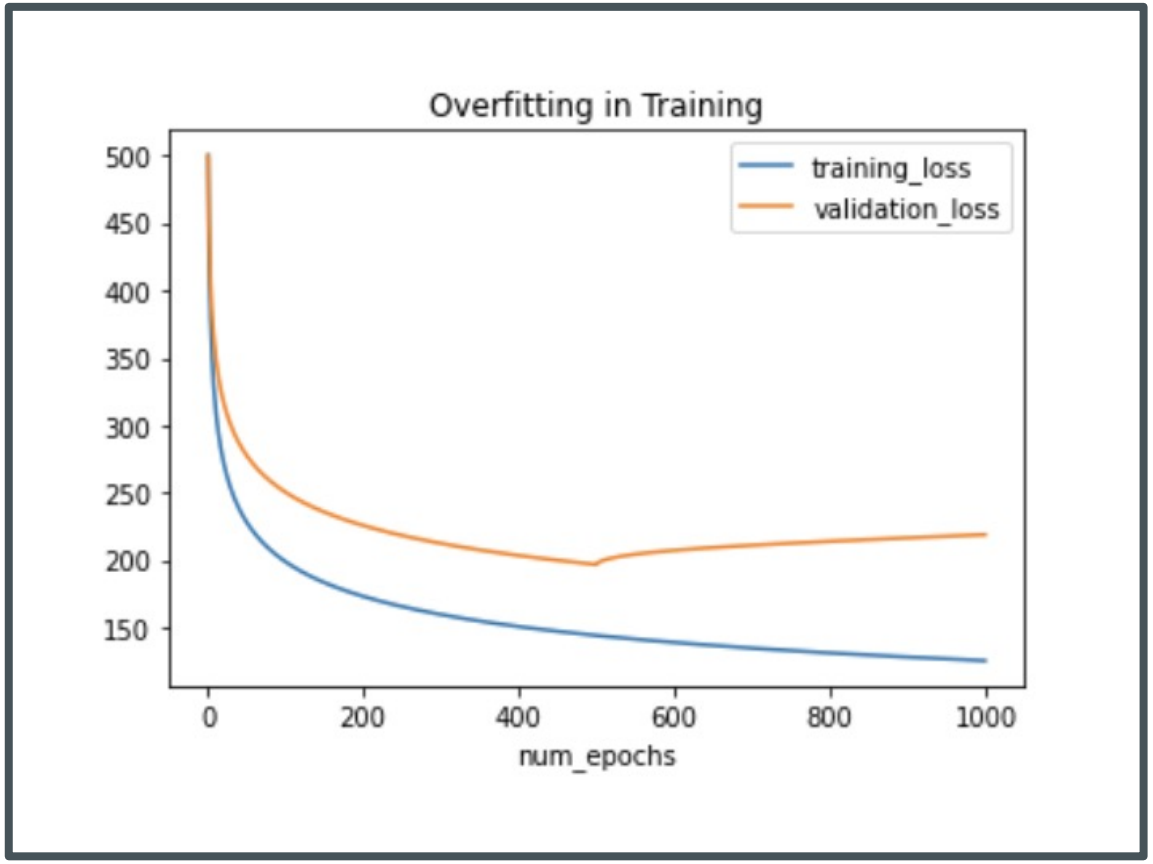
- **Overfitting** is a common problem in machine learning:
  - where a model performs well on training data but does not generalize well to unseen data (test data).
  - If a model suffers from overfitting, we also say that the model has a high variance, which can be caused by having too many parameters → too complex
- Similarly, our model can also suffer from **underfitting** (high bias), which means that our model is not complex enough to capture the pattern in the training data
  - → low performance on unseen data.



ILLUSTRATING OVER/UNDERFITTING

The problem of overfitting and underfitting can be best illustrated by using a more complex, nonlinear decision boundary as shown in the figure above

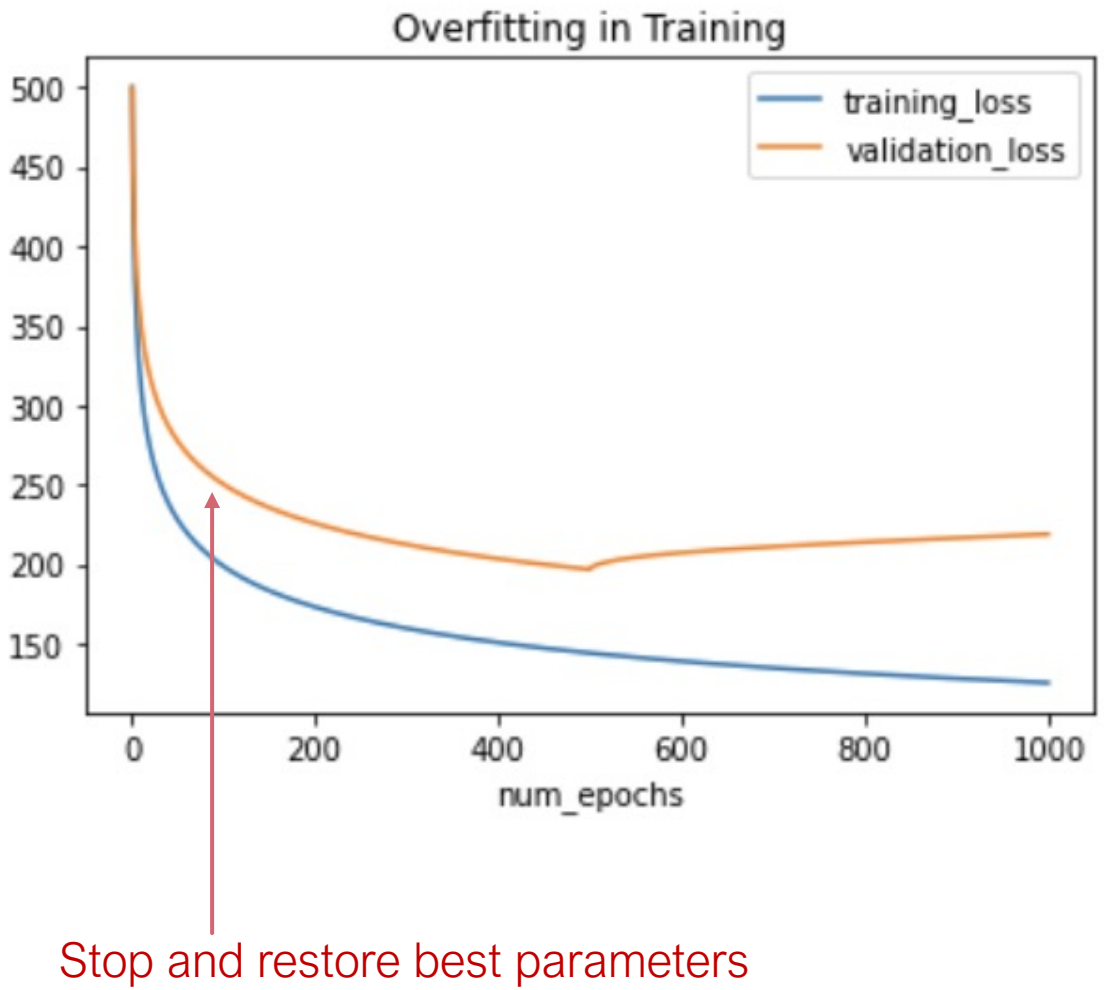
OVERFITTING PROBLEM



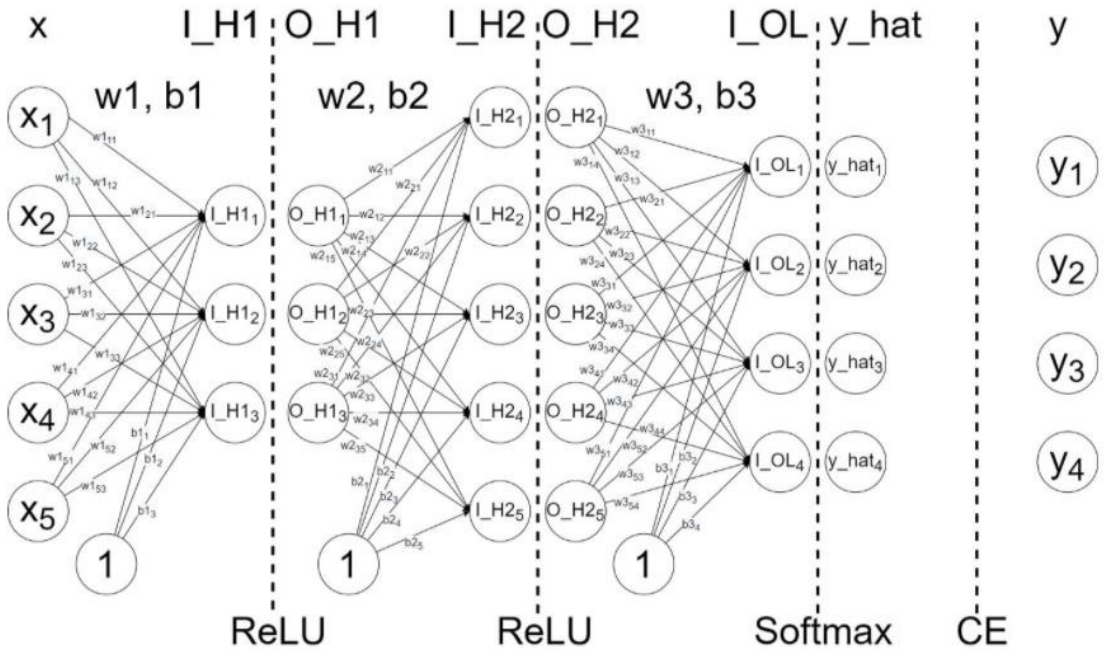
- The more complex the model is, the higher is the risk of overfitting.
- Here a clear example of overfittig, the train loss keeps going down while the validation loss get worse. It is always important to split the training in train and validation set and to have a clear picture of the train history.
- In order to avoid overfitting and make the training stable we have different approach

FACING OVERFITTING PROBLEM – 1

- Introduce a **callback function** that **stops the training if the validation loss get worse and restore the best parameters (Early Stop function).** Reduce overtraining and time needed for the training.



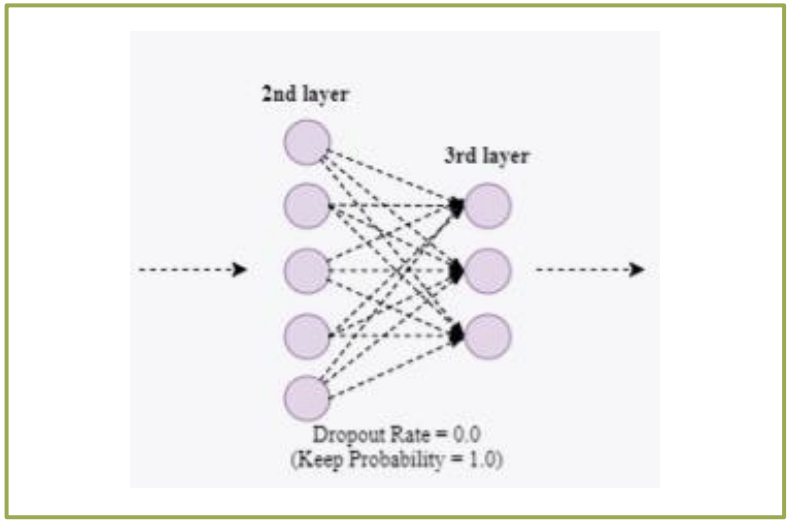
$$loss = CE$$
$$+L1 * (|W3_{11}| + |W3_{12}| + ...) + L1 * (|B3_1| + |B3_2| + ...)$$
$$+L2 * ({W3_{11}}^2 + {W3_{12}}^2 + ...) + L2 * ({B3_1}^2 + {B3_2}^2 + ...)$$
$$+L1 * (|W2_{11}| + |W2_{12}| + ...) + L1 * (|B2_1| + |B2_2| + ...)$$
$$+L2 * ({W1_{11}}^2 + {W1_{12}}^2 + ...) + L2 * ({B1_1}^2 + {B1_2}^2 + ...)$$

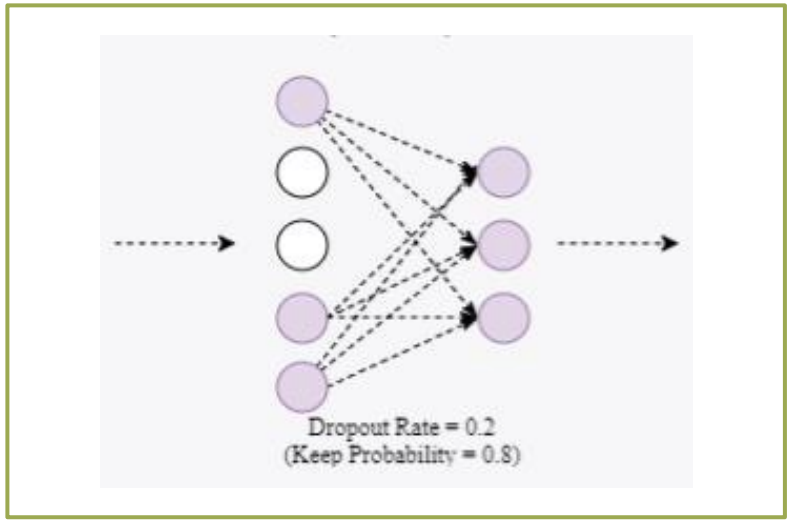


FACING OVERFITTING PROBLEM – 2

- "Weight Decay": introduce penalty terms in the loss function if a weight becomes too large.

FACING OVERFITTING PROBLEM – 3



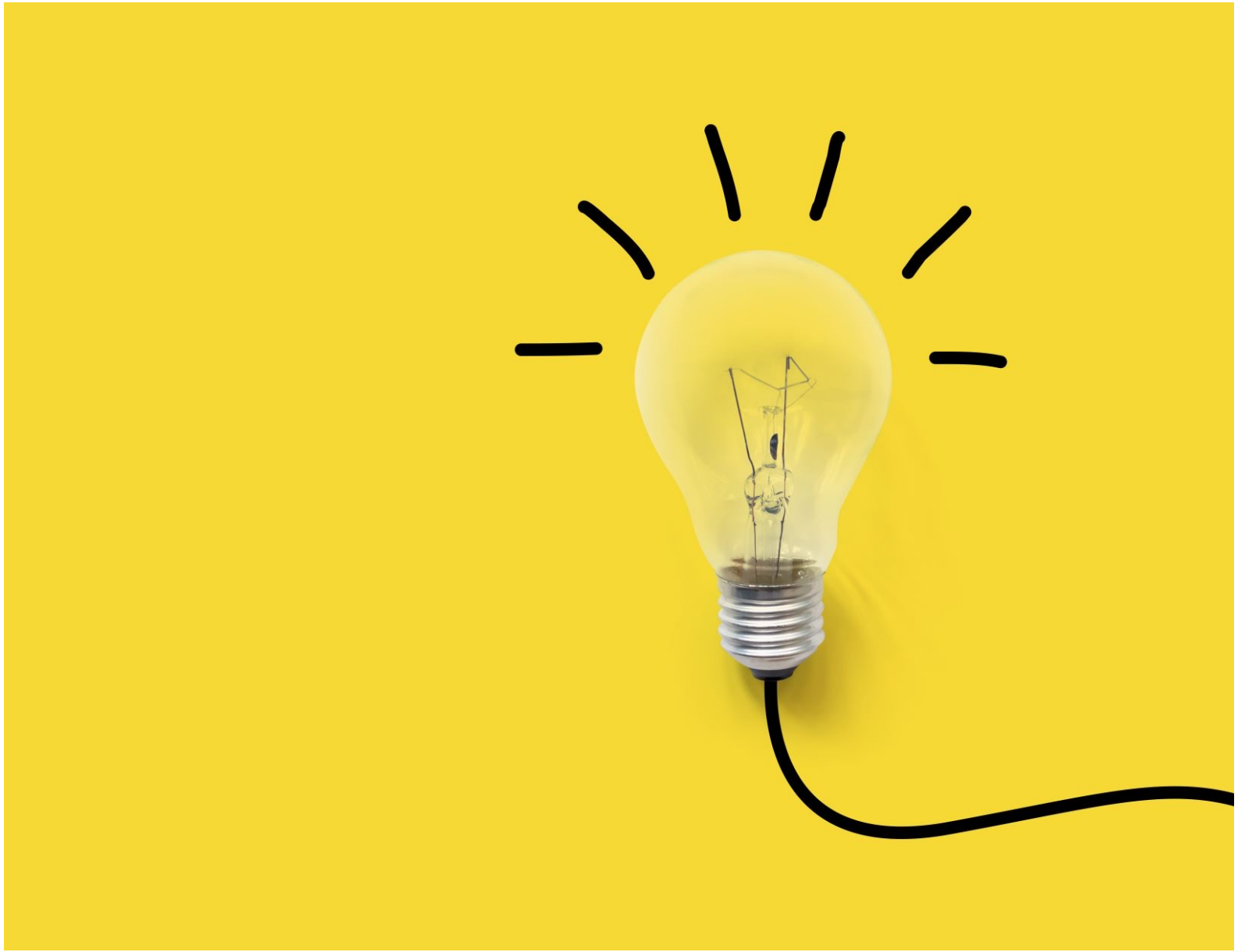


Dropout: it refers to the practice of disregarding certain nodes in a layer at random during training. A dropout is a regularization approach that prevents overfitting by ensuring that no units are co-dependent with one another.

EVALUATION METRICS

EVALUATION METRICS

- The idea of building machine learning models works on a constructive feedback principle:
  - building a model, getting a feedback from metrics, making improvements and continuing until you achieve the desired accuracy
- An important aspect of evaluation metrics is their capability to discriminate among model results
- The real goal is creating and selecting a model which gives high accuracy on sample data:
  - It is crucial to check the accuracy of your model prior to computing predicted values.



DIFFERENT TYPES OF METRICS

- A metric is a function that is used to evaluate the performance of your model.
- Metric functions are similar to loss functions, except that the results from evaluating a metric are not used during the training of the model.

		Predicted Class: Positive	Predicted Class: Negative	
Actual Class	Positive	True Positive (TP)	False Negative (FN) **Type II Error**	**Sensitivity** $\frac{TP}{(TP+FN)}$
	Negative	False Positive (FP) **Type I Error**	True Negative (TN)	**Specificity** $\frac{TN}{(TN+FP)}$
		Precision $\frac{TP}{(TP+FP)}$	**Negative Predictive Value** $\frac{TN}{(TN+FN)}$	**Accuracy** $\frac{TP+TN}{(TP+TN+FP+FN)}$

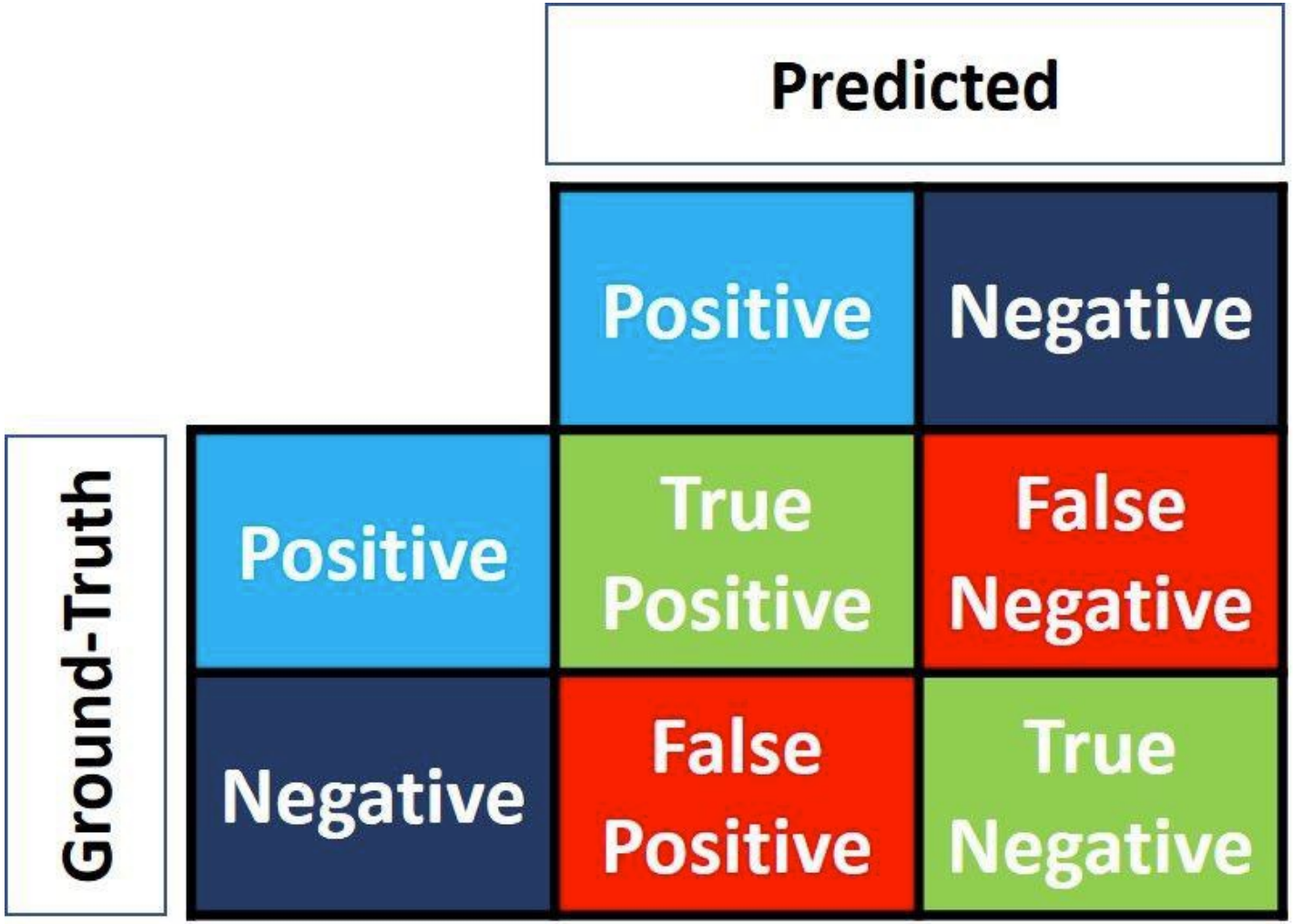
IN H.E.P.:
- POSITIVE: SIGNAL
- NEGATIVE: BACKGROUND

- **TP (true positive)**: the event is POSITIVE, the prediction is POSITIVE
- **FP (false positive)**: the event is NEGATIVE, but the prediction is POSITIVE
- **TN (true negative)**: the event is NEGATIVE, the prediction is NEGATIVE
- **FN (false negative)**: the event is POSITIVE, the prediction is NEGATIVE

NOTE: Precision == purity; recall==sensitivity == TPR == signal efficiency; FPR = FP/(FP+TN)

CONFUSION MATRIX

- The confusion matrix helps us visualizing whether the model is "confused" in discriminating between two or more classes.
- In the figure we have an example of binary model and the corresponding confusion matrix.
- The 4 elements of the matrix represent the 4 metrics that count the number of correct and incorrect predictions the model made.



ACCURACY

The most famous metrics is the accuracy defined as the ratio between the number of correct predictions to the total number of predictions

Accuracy values range between 0 and 1. Obviously an accuracy values near to 1 means that our model fits well the datasets

It is important to stress that a good accuracy value on the training dataset does not imply a good discrimination on the test dataset

$$Accuracy = \frac{True_{positive} + True_{negative}}{True_{positive} + True_{negative} + False_{positive} + False_{negative}}$$

PRECISION/ PURITY

- The precision is calculated as the ratio between the number of *Positive* samples correctly classified to the total number of samples classified as *Positive* (either correctly or incorrectly). **The precision is intuitively the ability of the classifier not to label as positive a sample that is negative**
- When the model makes many incorrect *Positive* classifications, or few correct *Positive* classifications, this increases the denominator and makes the precision small. On the other hand, the precision is high when:
  - The model makes many correct *Positive* classifications (maximize *True Positive*).
  - The model makes fewer incorrect *Positive* classifications (minimize *False Positive*).

$$Precision = \frac{True_{positive}}{True_{positive} + False_{positive}}$$

RECALL/ SENSITIVITY/ TPR/ SIGNAL EFFICIENCY

- The recall is calculated as the ratio between the number of *Positive* samples correctly classified as *Positive* to the total number of *Positive* samples. The recall is intuitively the ability of the classifier to find all the positive samples.
- The recall cares only about how the positive samples are classified. This is independent of how the negative samples are classified, e.g. for the precision.
- The decision of whether to use precision or recall depends on the type of problem to be solved:
  - If the goal is to detect all the positive samples (without caring whether negative samples would be misclassified as positive) then we can use recall;
  - if the problem is sensitive to classifying a sample as *Positive* in general, i.e. including *Negative* samples that were falsely classified as *Positive* we can use precision.

$$Recall = \frac{True_{positive}}{True_{positive} + False_{negative}}$$

F1-SCORE

- F1-Score is the harmonic mean of precision and recall values for a classification problem.
- It takes the harmonic mean because punishes extreme values more.
- For example, if we have a model with Precision = 0 and Recall =1, it is clear that this result comes from a dumb classifier which just ignores the input and just predicts one of the classes as output. In this example we will have a F1 score equal to 0

$$F_1 = \left(\frac{\mathrm{recall}^{-1} + \mathrm{precision}^{-1}}{2} \right)^{-1} = 2 \cdot \frac{\mathrm{precision} \cdot \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}}.$$

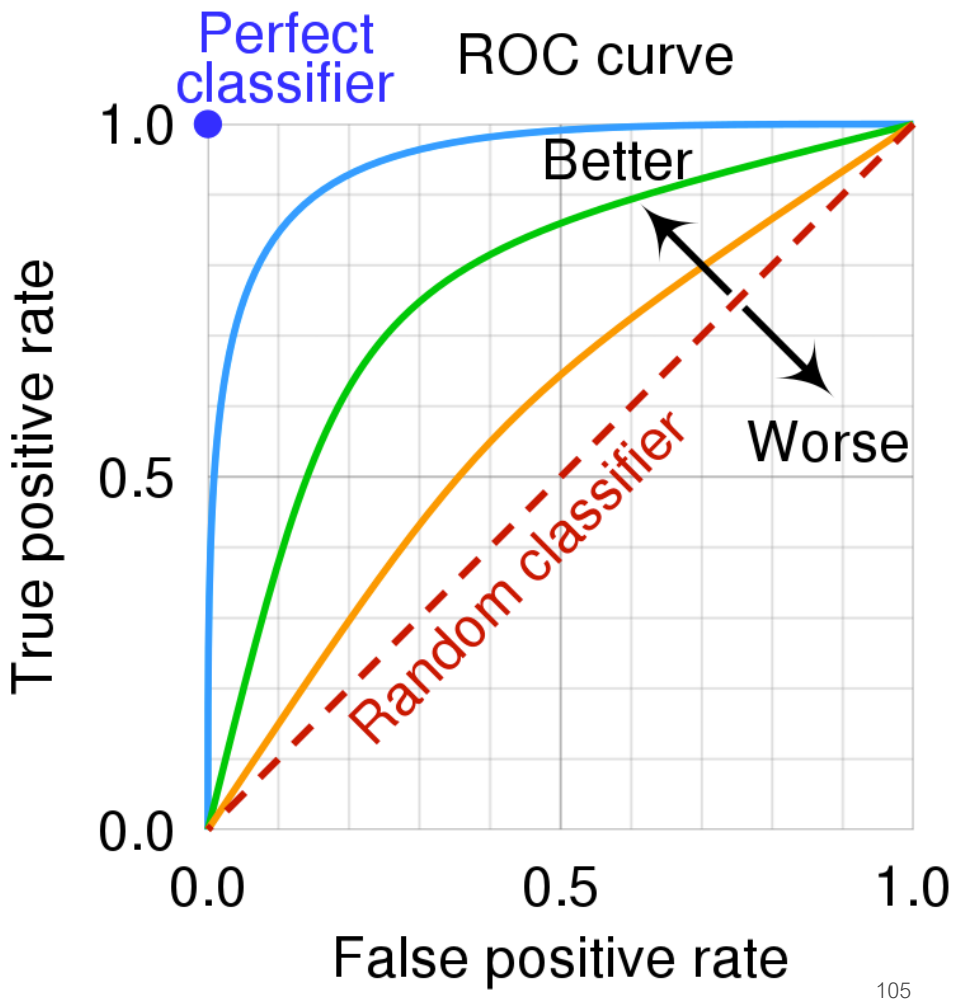
AUC-ROC

- A **Receiver Operating Characteristic curve**, or **ROC curve**, is a plot that illustrates the true positive rate against the false positive rate defined as follows:

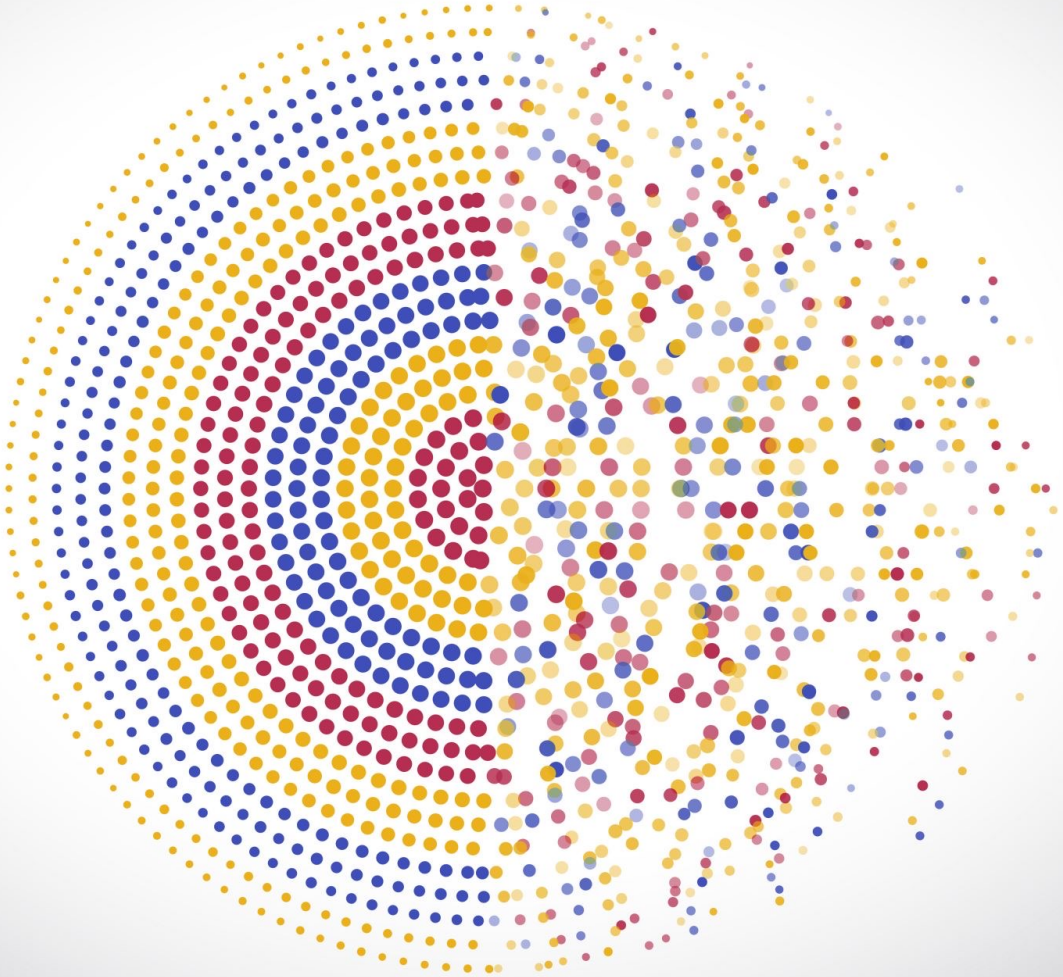
$$TPR = \frac{True_{positive}}{True_{positive} + False_{negative}}$$

$$FPR = \frac{False_{positive}}{False_{positive} + True_{negative}}$$

- The metric connected to the ROC curve is the area under the curve **AUC**.
  - An AUC near to 1 indicates a ROC curve near to the best result, an AUC near to 0 indicates a random classifier.



CONVOLUTIONAL NEURAL NETWORKS



OVERVIEW

1

Digital image classification problem

2

An introduction to convolution in 1 and 2 dimensions

3

An overview on the different layers of a CNN

4

A brief explanation of the Data Augmentation

DIGITAL IMAGES

- Convolutional Neural Networks are a powerful family of DNNs that are specifically designed for the Images Processing Task (but not only!)

WHAT ARE IMAGES DIGITALLY SPEAKING?

- From Wikipedia**: "*A digital image is an image composed of picture elements, also known as pixels, each with finite, discrete quantities of numeric representation for its intensity*."

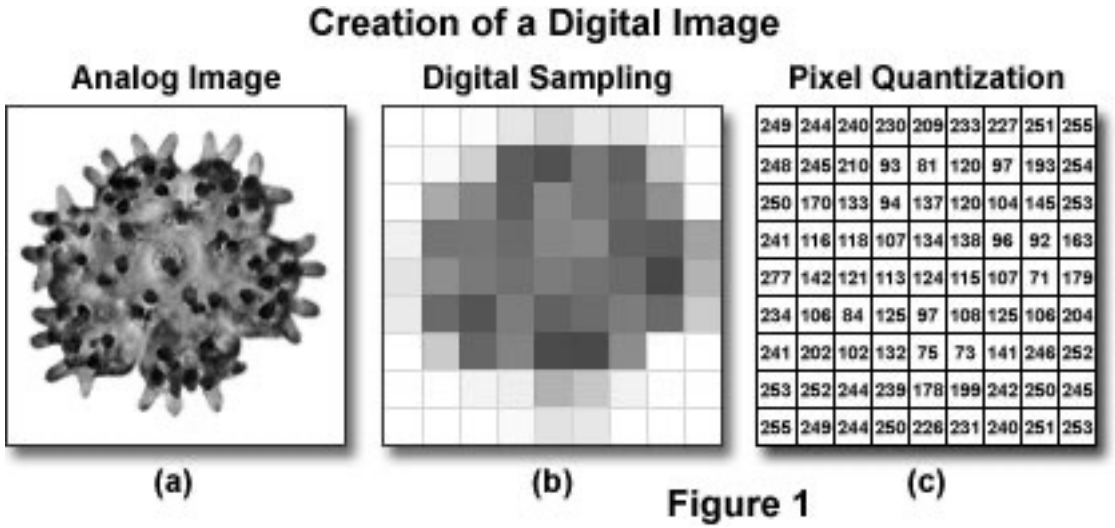
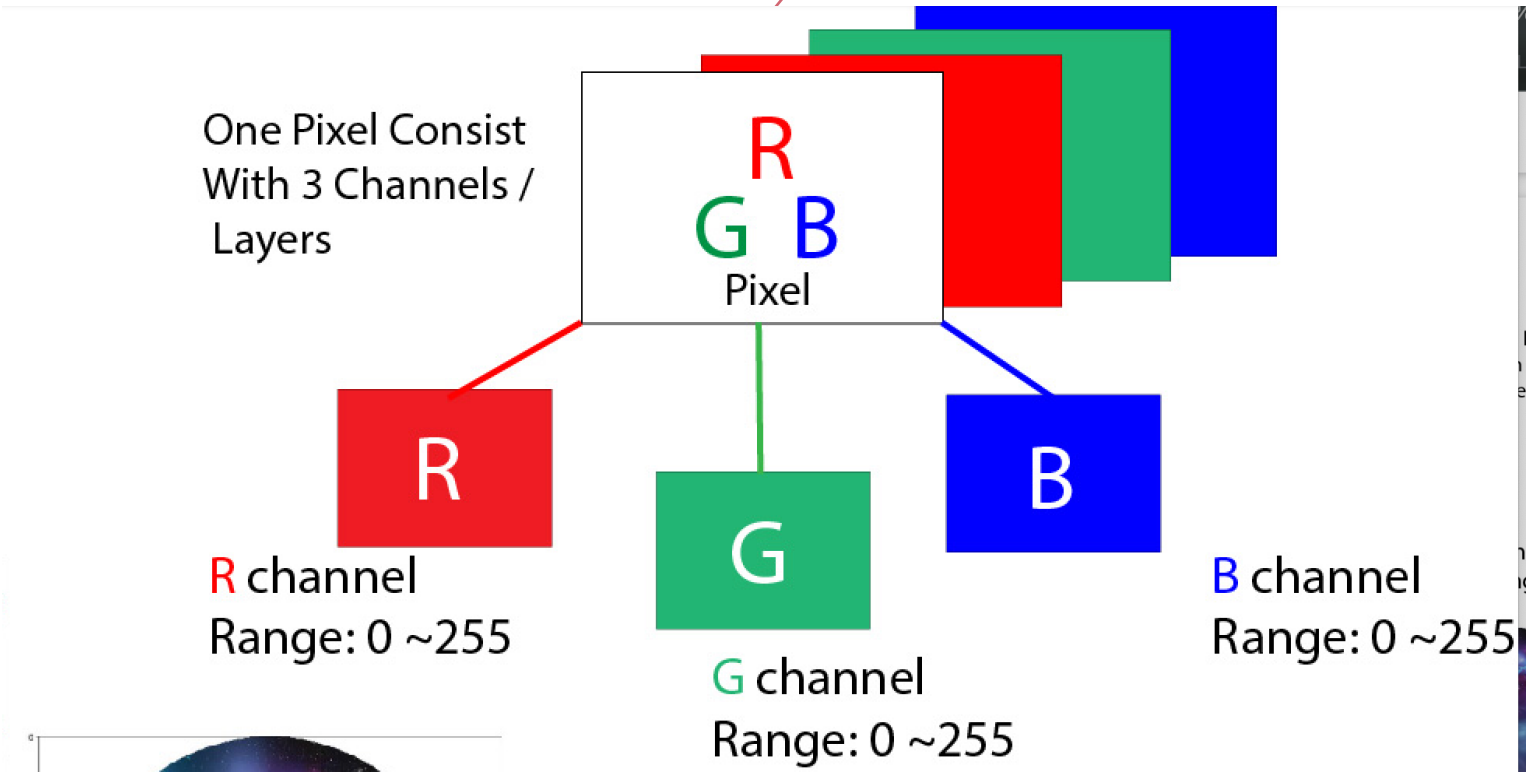


Figure 1

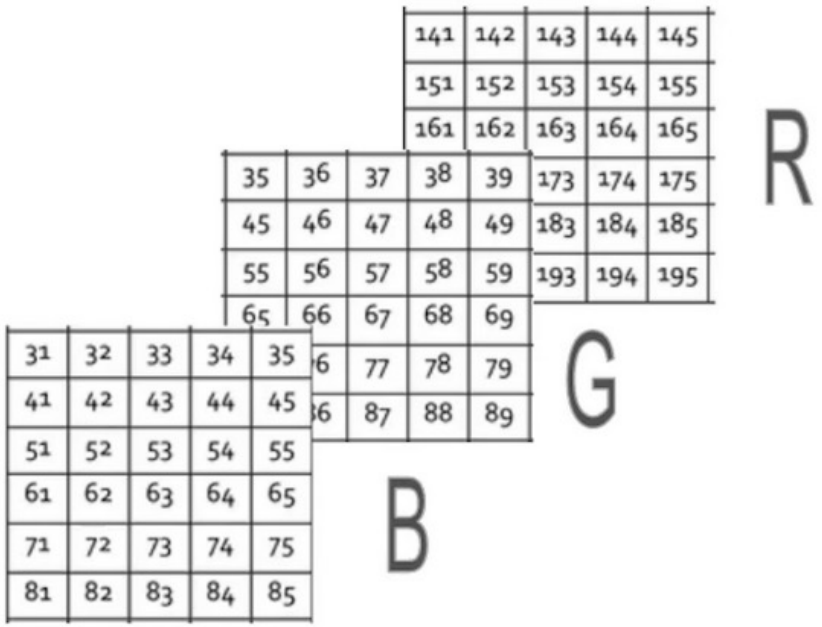
- Images can be seen as matrices, where every cell represents a pixel.
- To every cell (= to every pixel) a number is associated
- In a **greyscale image each pixel value is the grey intensity**

DIGITAL IMAGES

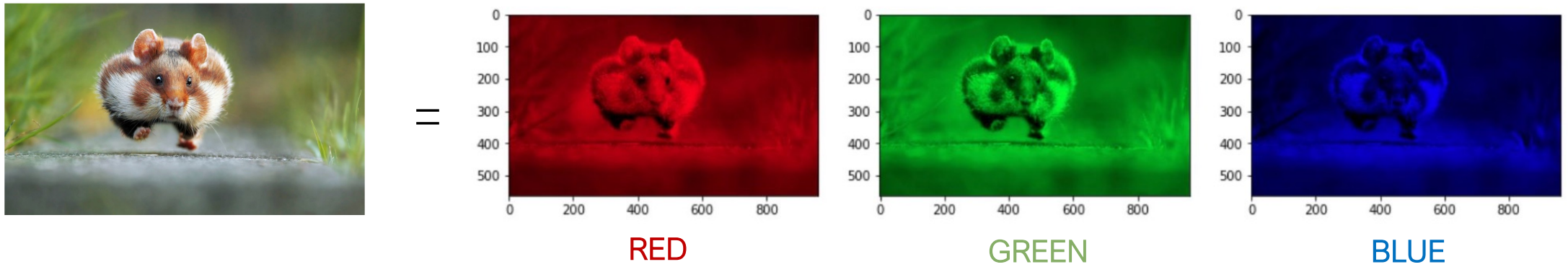
- Colored images are usually coded with the **RGB color model: each pixel is associated to three numbers**, corresponding to the **Red, Green and Blue intensity**
- The image is obtained as the sum of the three components



DIGITAL IMAGES



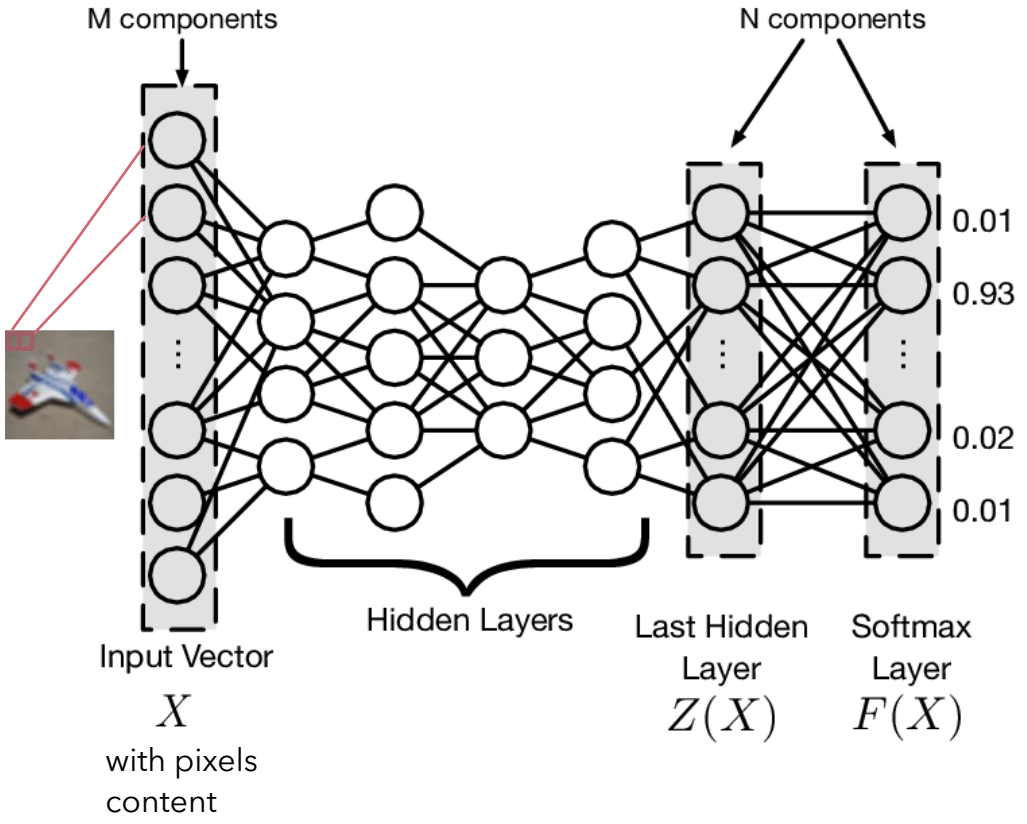
- An RGB image is therefore represented by a matrix (weight)x(height)x3
- A greyscale image is (weight)x(height)x1



HOW CAN WE CLASSIFY IMAGES?

IMAGES CLASSIFICATION WITH DNN

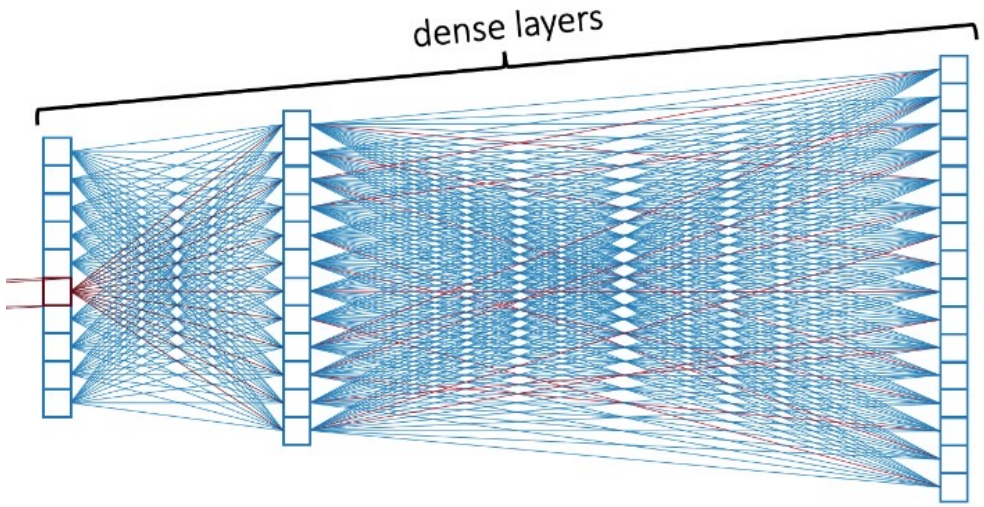
- As we already know, a classification problem can be easily addressed from a NN:
  - The more the problem is complex and not linear the better DNNs perform with respect to single layer networks.
- Images can be given in input to DNN by flattening pixels to form a 1D array.
  - DNNs are also invariant to input features order, therefore they could also be shuffled before being fed the network



IMAGES CLASSIFICATION WITH DNN

Example:

- the input is a **64x64 grayscale image**. Such an image can be represented by 64x64x1 = 4096 values
- the **input layer** of a DNN processing such an image has **4096 nodes**
- if the (fully connected) inner layer has **500 nodes**, we will have **4096x500 = 2048000 weights between the input and the hidden layer**
- If the image were an RGB image the input layer would have 64x64x3 = 12288 nodes
- For complex problems, we usually need multiple hidden layers...

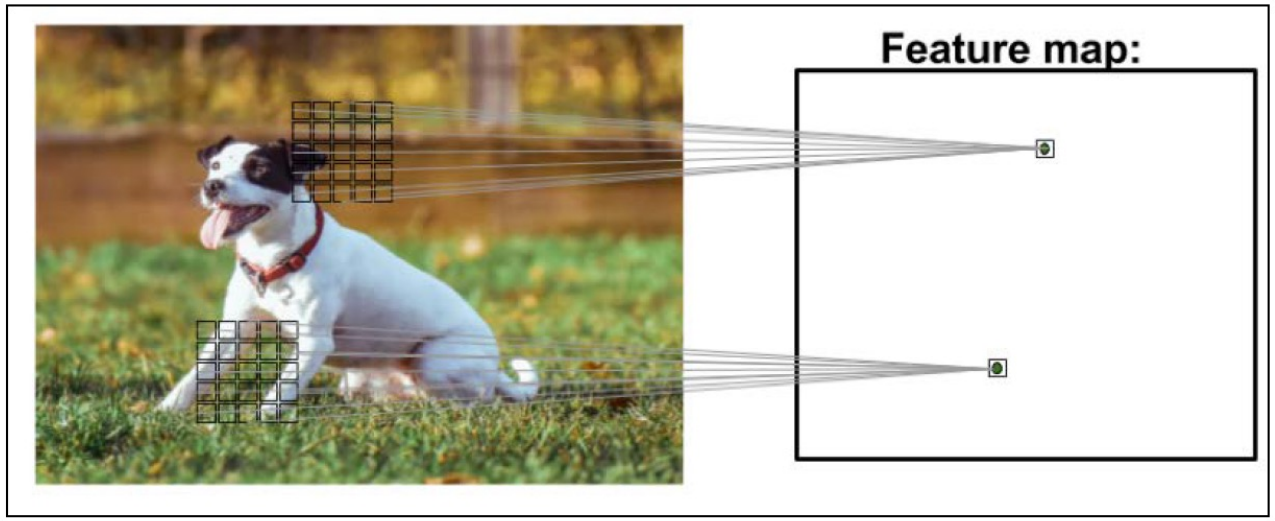


A DNN CANNOT SCALE TO HANDLE LARGE IMAGES. WE NEED A MORE SCALABLE ARCHITECTURE!!!

CONVOLUTIONAL NEURAL NETWORKS

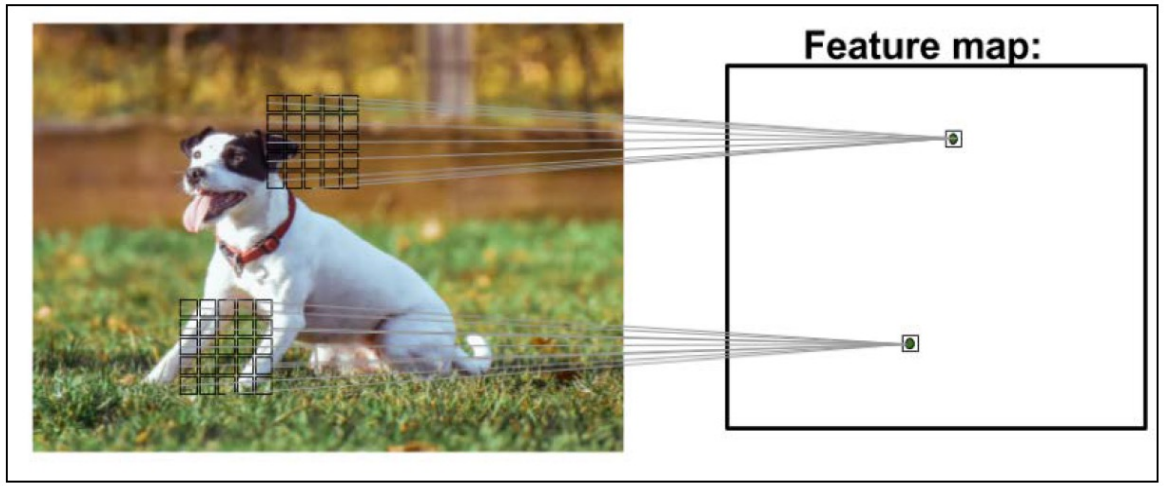
- To flat an image into a 1D array is not the best way to model images
  - any spatial relationship in the data is ignored
- A Convolutional Neural Network (CNN) maintains the spatial structure of the data, and is better suited for finding spatial relationships in the image data

- **The idea behind:** to use filters that automatically learns the most discriminants features in an image, such as edges, filled patterns, specific geometric forms and so on



(Photo by Alexander Dummer on Unsplash)

CONVOLUTIONAL NEURAL NETWORKS



BASIC CONCEPTS:

- **Sparse connectivity**: A single element in the feature map is connected to only a small patch of pixels.
- **Parameter-sharing**: the same weights are used for different patches of the input image

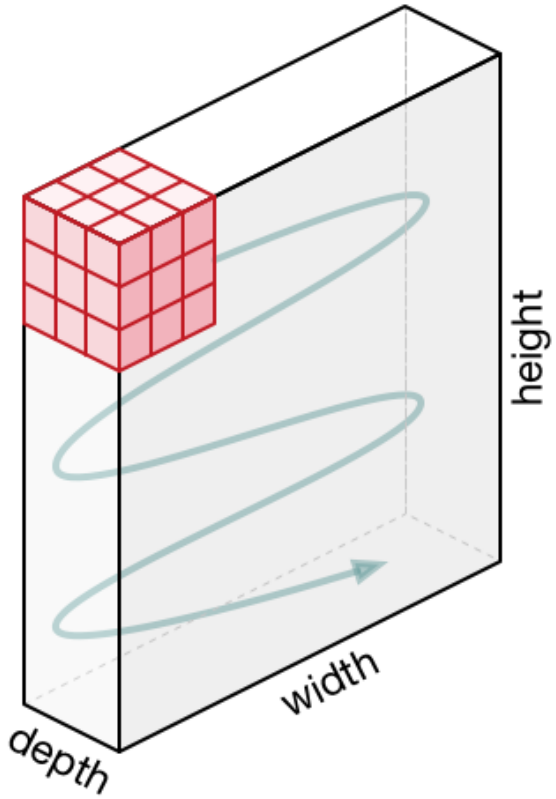
from **Spatial Invariance Principle**: *whatever method is used to recognize objects it should not be concerned with the precise location of the object in the image*

THE CONVOLUTIONAL LAYER

- Let's start from **the Convolutional Layer:**
  - It is the core building block of a Convolutional Network that does most of the computational heavy lifting

INTUITION WITHOUT BRAIN STUFF

- The CONV layer's parameters consist of a set of learnable filters.
- Every filter is small spatially (along width and height) and extends through the full depth of the input volume.
- The output is a single layer with a certain spatial size (width x height x1)



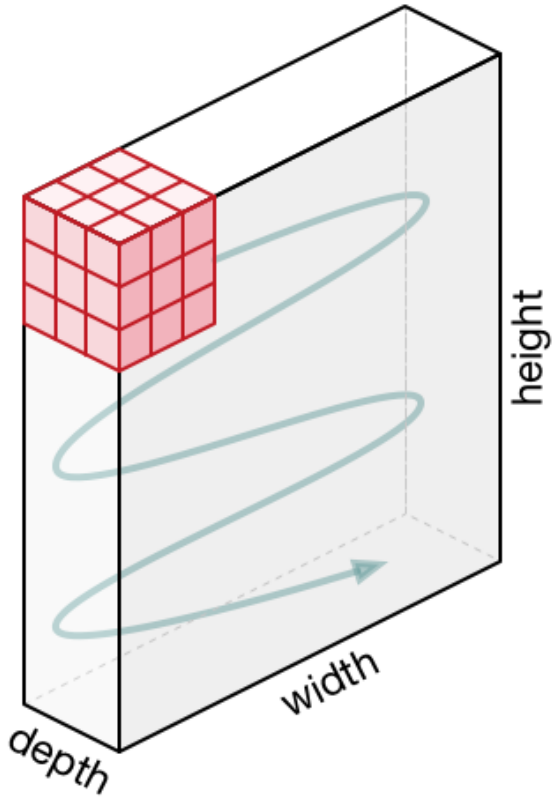
THE CONVOLUTIONAL LAYER

- Let's start from **the Convolutional Layer:**
  - It is the core building block of a Convolutional Network that does most of the computational heavy lifting

INTUITION WITHOUT BRAIN STUFF

Example:

- a typical filter on a first layer of a CNN has size 5x5x3
- Let's suppose we have a 28x28 pixel RGB image
- Convolution is the process of placing the filter 5x5x3 on the top left corner of the image, multiplying filter values by the pixel values and adding the results, moving the filter to the right one pixel at a time and repeating this process



THE CONVOLUTIONAL LAYER

THE MATHEMATICAL VIEW

- Now let's explain the convolution in mathematical details

Discrete convolution in one dimension

- A discrete convolution between two vectors with finite size, x and w, is mathematically defined as:

$$\boldsymbol{y} = \boldsymbol{x} * \boldsymbol{w} \rightarrow y[i] = \sum_{k=-\infty}^{+\infty} x[i-k]\, w[k]$$

- w is typically called *filter* or *kernel*
- The index *i* runs through each element of the output vector y
- In ML applications we always deal with finite feature vectors
- In real word x and w have finite dimensions, let's say n and m respectively, where $m \leq n$.

THE CONVOLUTIONAL LAYER

THE MATHEMATICAL VIEW

- The convolution becomes:

$$\boldsymbol{y} = \boldsymbol{x} * \boldsymbol{w} \rightarrow \quad y[i] = \sum_{k=0}^{k=m-1} \boldsymbol{x}\ [i + m - k]w[k]$$

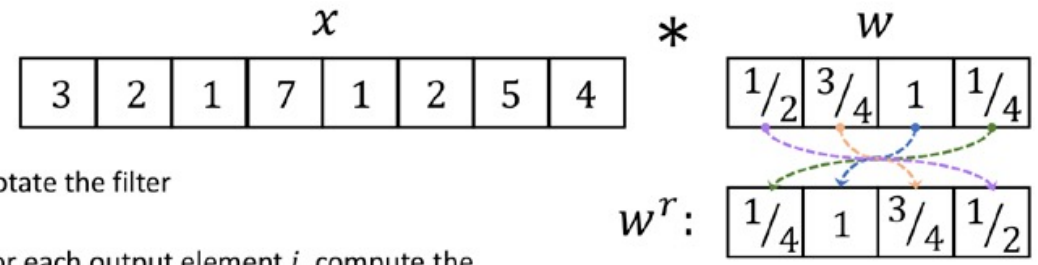
- x and w are indexed in different directions in this summation.
- Computing the sum with one index going in the reverse direction is equivalent to computing the sum with both indices in the forward direction after flipping one of those vectors
- This operation is repeated like in a sliding window approach to get all the output elements.

THE CONVOLUTIONAL LAYER

THE MATHEMATICAL VIEW

Example:

- x = [3 2 1 7 1 2 5 4], w = [½, ¾, 1, ¼]



Step 1: Rotate the filter

Step 2: For each output element i, compute the dot-product $x[i:i+4].w^r$

(move the filter two cells)

$$y[0] = 3\times{}^1/_4 + 2\times1 + 1\times{}^3/_4 + 7\times{}^1/_2$$

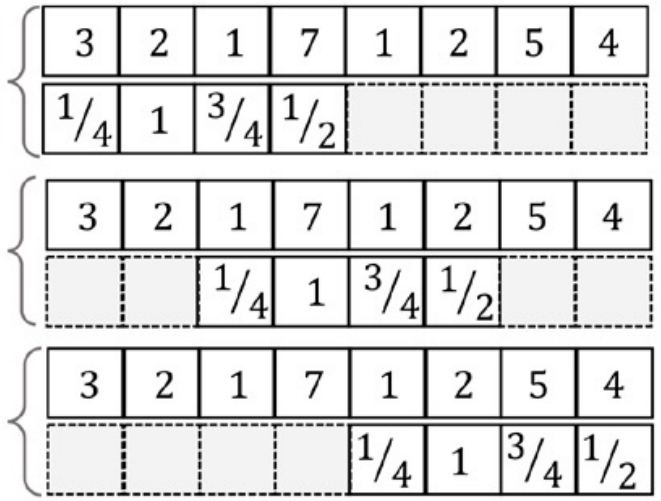
$$\rightarrow y[0] = 7$$

$$y[1] = 1\times{}^1/_4 + 7\times1 + 1\times{}^3/_4 + 2\times{}^1/_2$$

$$\rightarrow y[1] = 9$$

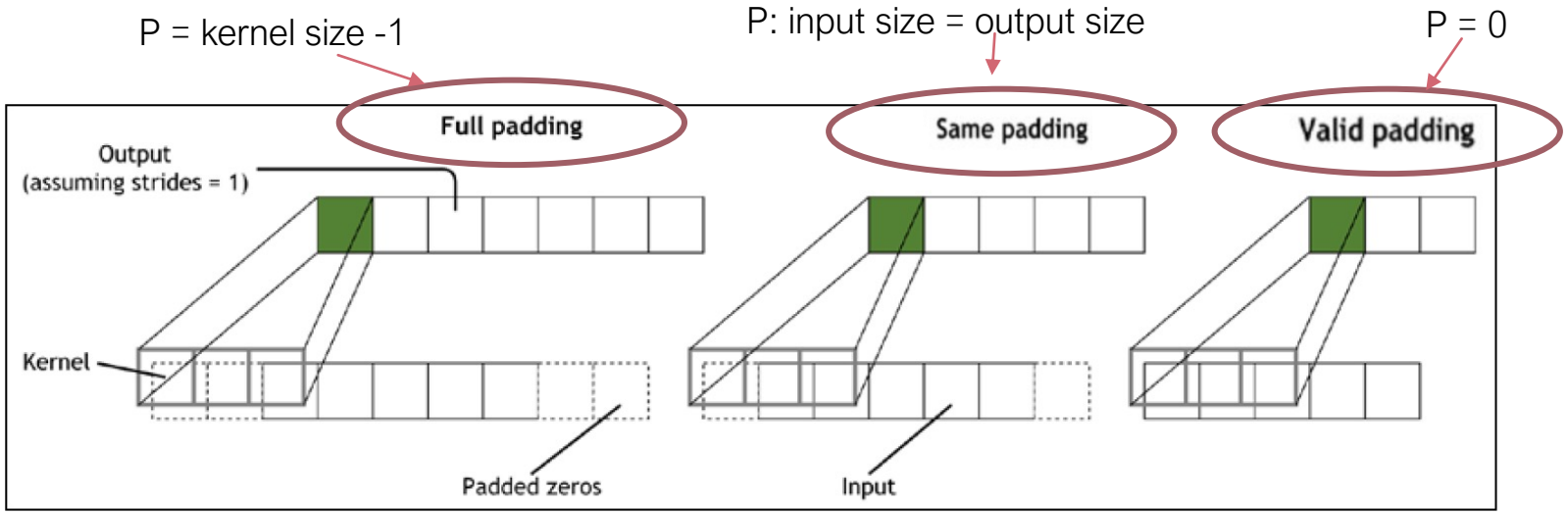
$$y[2] = 1\times{}^1/_4 + 2\times1 + 5\times{}^3/_4 + 4\times{}^1/_2$$

$$\rightarrow y[2] = 8$$



PADDING LAYER

- The result of this convolution is a tensor with a smaller shape than the input one
- To preserve/increase input shape we can use the so called ***padding*** procedure:
  - It consists in **padding zero** pixels to input tensor
  - Usually, a ***same padding*** procedure is used, meaning that the output vector has the same size as the input one.
  - **Valid padding**: we are not adding anything



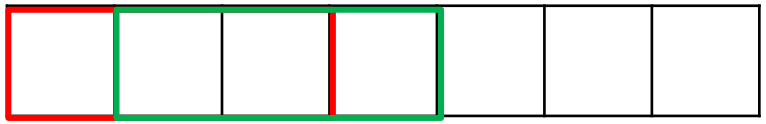
STRIDE PARAMETER

MOVING ALONG INPUTS

- One concept introduced in the previous example is **the number of cells the filter is moved when shifted** across the vector x (to pass from a y index to another)
- It is called **stride**

Example:

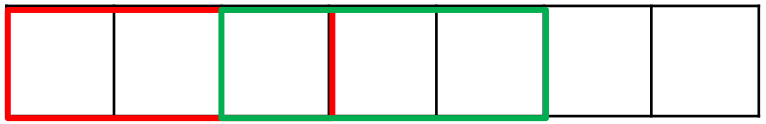
- N=7, filter = 3



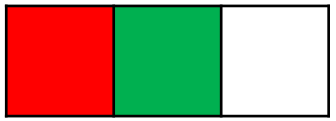
Stride = 1

Output = 5

The bigger is the stride
the smaller is the output dimension!



Stride = 2



Output = 3

THE OUTPUT SIZE

OUTPUT SIZE

- The size of the vector obtained by a convolution can be calculated as follows:

$$o = \left[\frac{n + 2p - m}{s}\right] + 1$$

- **o** = output dimension
- **n** = input dimension
- **p** = padding
- **m** = kernel size
- **s** = stride

Convolution in 2D

THE MATHEMATICAL VIEW

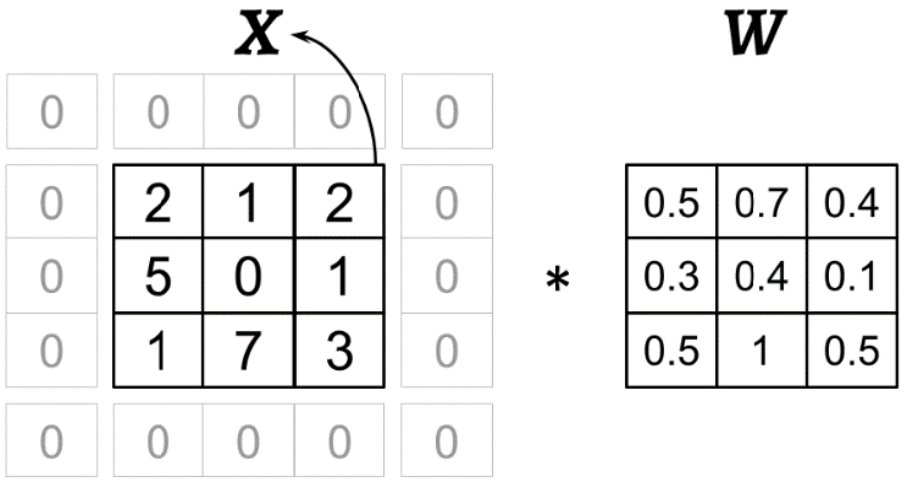
- The concepts we have now discussed are easily extendible to 2D case:

$$\boldsymbol{Y} = \boldsymbol{X} * \boldsymbol{W} \rightarrow \quad Y[i,j] = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} X[i-k_1, j-k_2]\, W[k_1, k_2]$$

  - $\boldsymbol{X}_{n1\times n2}$ and $\boldsymbol{W}_{m1\times m2}$ are now two matrices -> Y is a 2D matrix as well

Example:

- input matrix $\boldsymbol{X}_{3\times3}$
- kernel matrix $\boldsymbol{W}_{3\times3}$
- p=(1, 1)
- stride s=(2, 2)

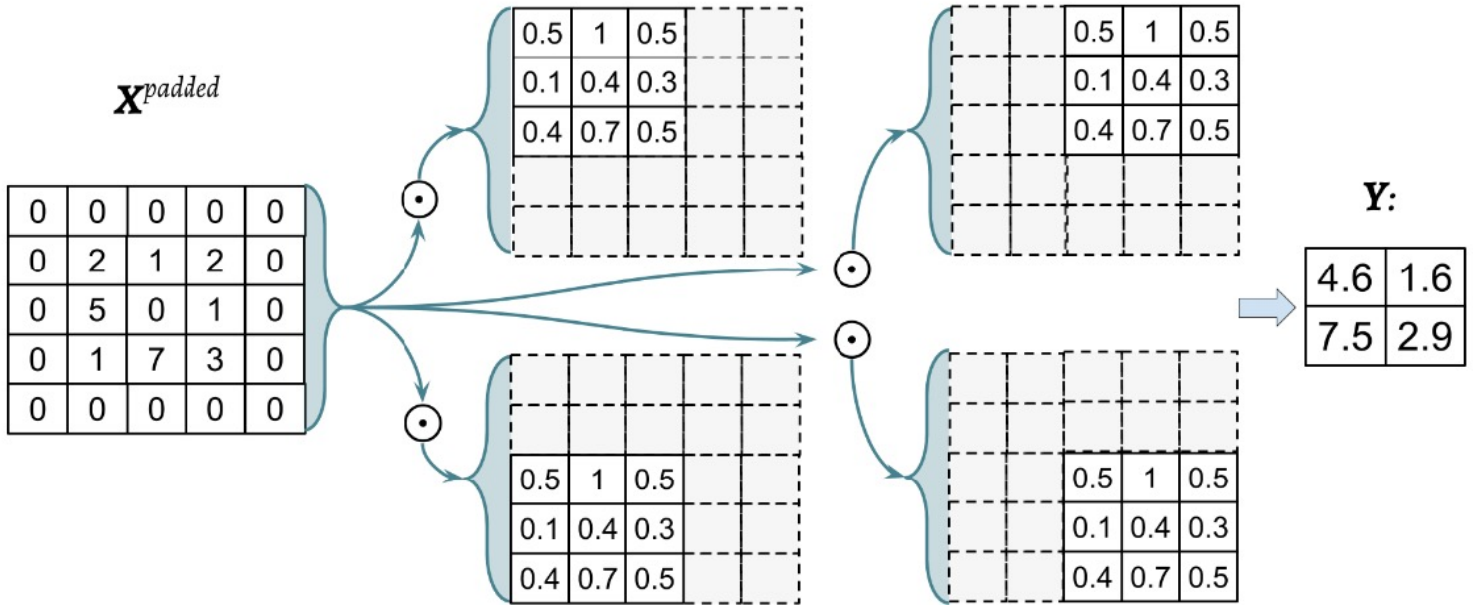


Convolution in 2D

THE MATHEMATICAL VIEW

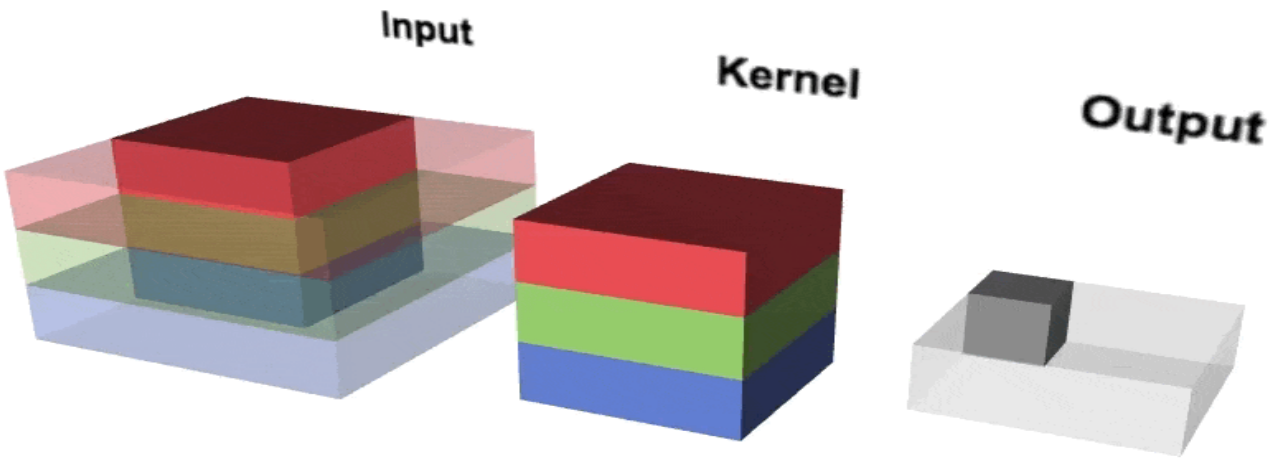
- We can rotate the filter to perform the sum on indices running in the same directions

$$\boldsymbol{W}^r = \begin{bmatrix} 0.5 & 1 & 0.5 \\ 0.1 & 0.4 & 0.3 \\ 0.4 & 0.7 & 0.5 \end{bmatrix}$$

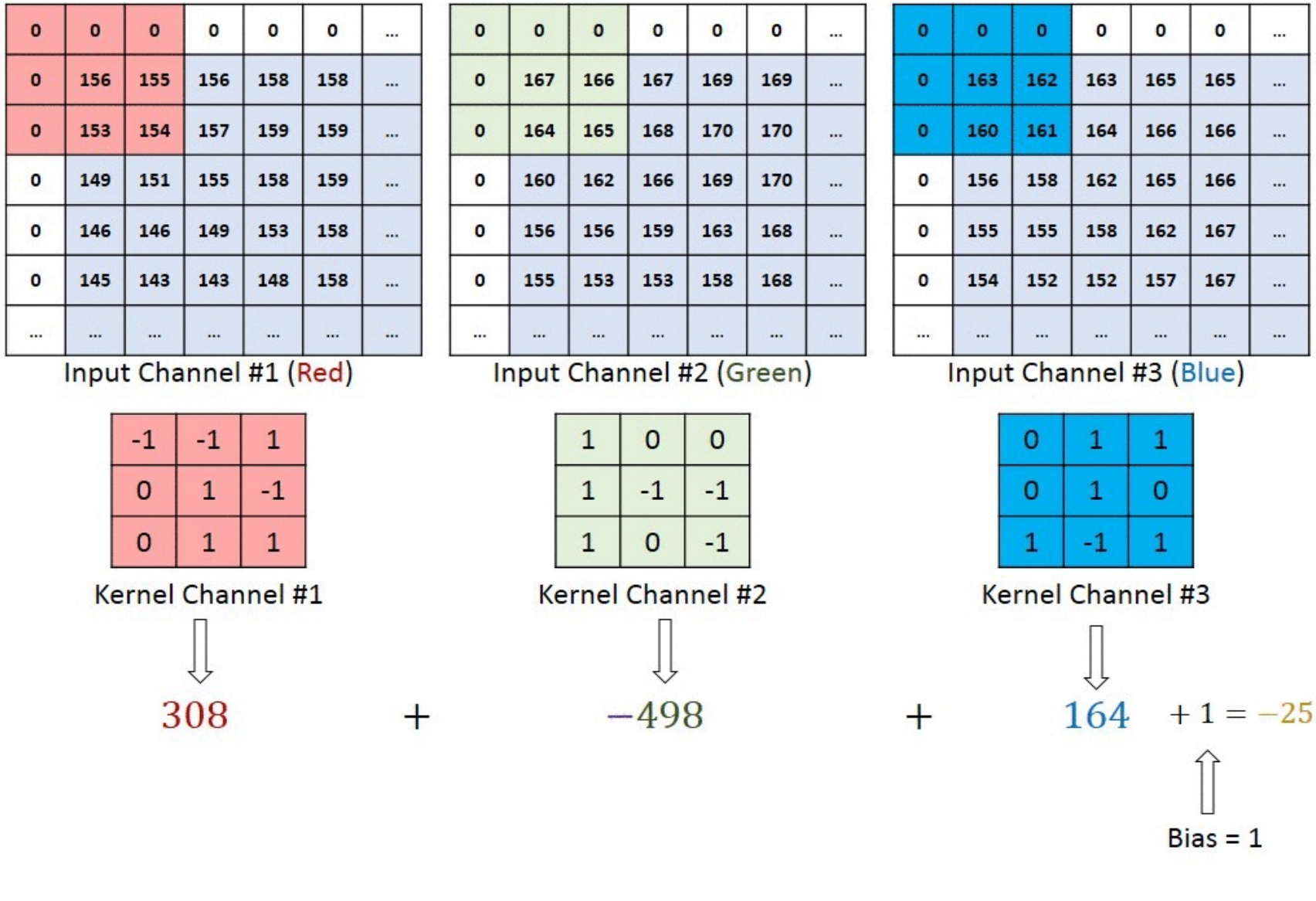


CONVOLUTION IN 2D

- How does a convolutional layer work on a RGB image?
  1. For each channel color there is a different filter
  2. The three outputs are added together
  3. **The output of a convolutional layer with a multi-layer input is a single layer**



CONVOLUTION IN 2D



STRIDES AND PADDING IN 2D

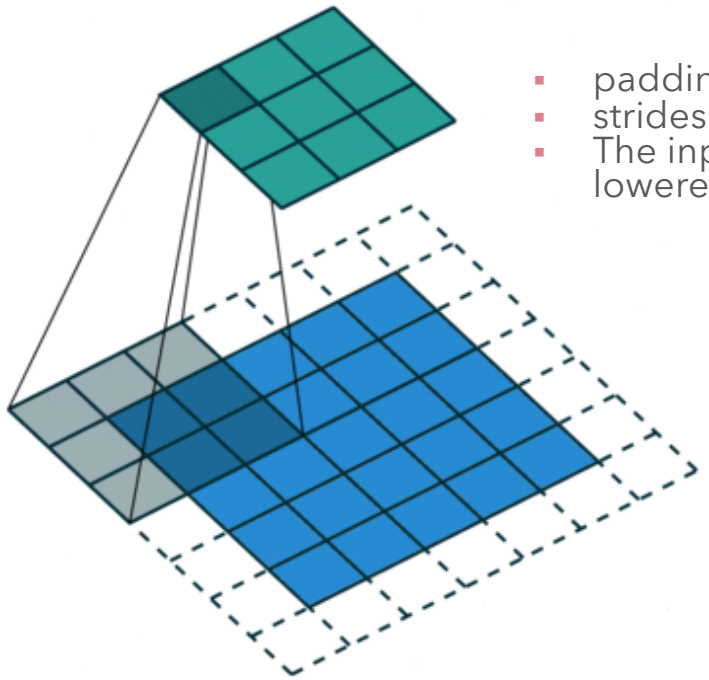
- Zero-padding and strides concepts are the same of 1D case
- The output size of a 2D filter is still calculable with the formula seen before, applied on weight and height separately

Input dimension Zero-padding Kernel dimension

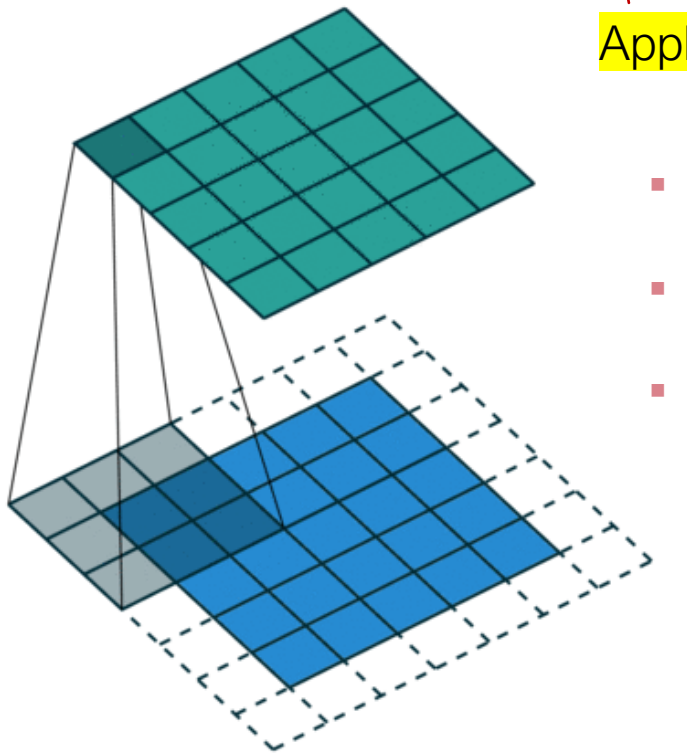
$$o = \left[\frac{n + 2p - m}{s}\right] + 1$$

stride

Applied along x and y axes



- padding (1,1)
- strides (2,2)
- The input shape is lowered



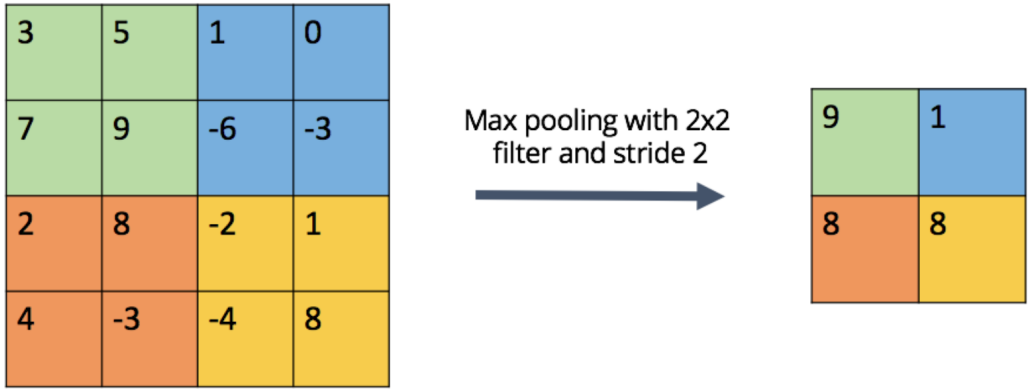
- padding (1,1)
- strides 1
- The input shape is preserved (padding 'same')

THE POOLING LAYER

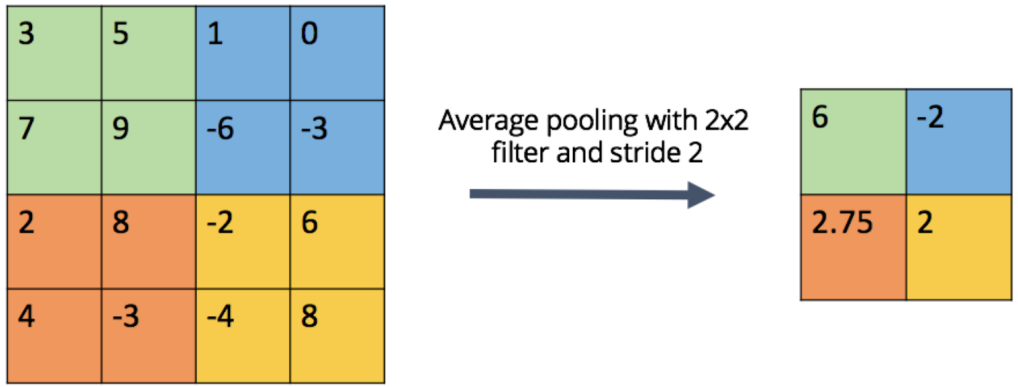
- Tuning zero-padding and strides it is possible to change (usually reduce) the output dimension w.r.t. input dimension
- This task can be also performed with a "Pooling" layer

TWO KINDS OF POOLING:

- **Maximum Pooling (or Max Pooling)**: Calculate the maximum value for each patch of the feature map.

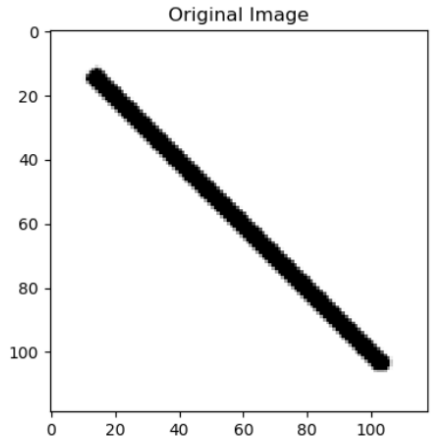


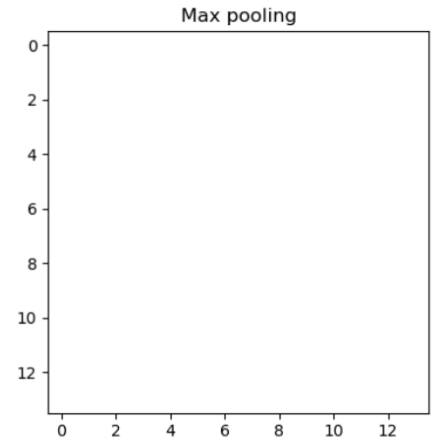
- **Average Pooling**: Calculate the average value for each patch on the feature map.



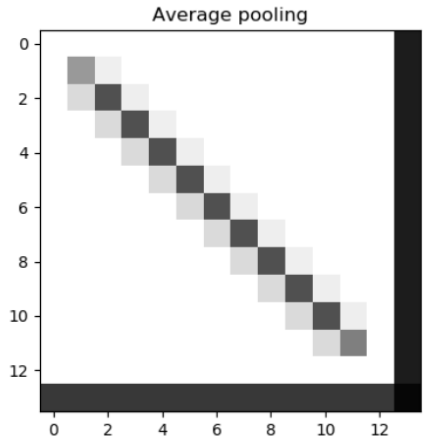
Different Pooling Layers

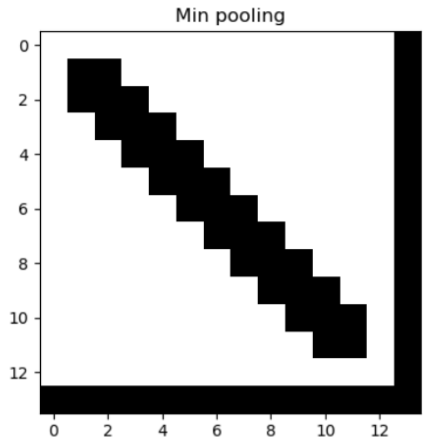
- Which pooling do I have to choose????
  - It depends!





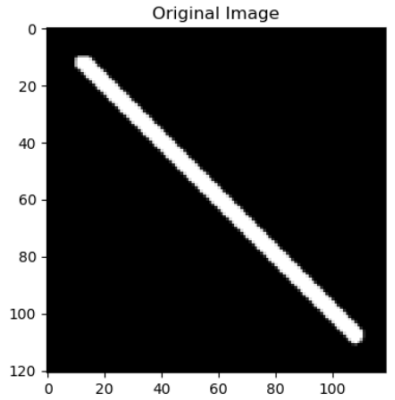
White → 1.0
Black → 0

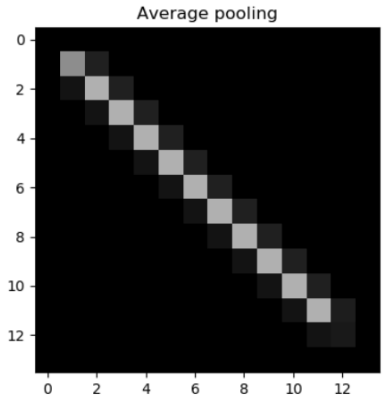


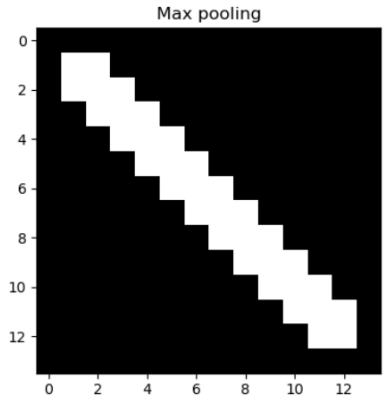


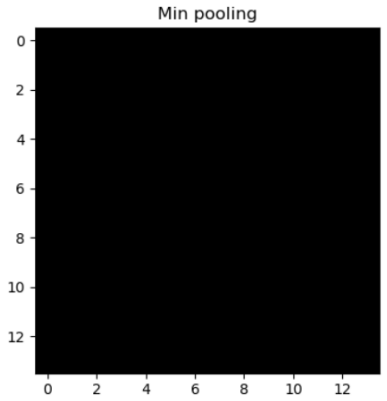
In this case min pooling let us preserving better the original information!

Different Pooling Layers









In this case max pooling let us preserving better the original information!

THE MAX POOLING LAYER

MAIN ASPECTS:

- **Pooling** (max-pooling) **introduces a local invariance**. This means that small changes in a local neighbourhood do not change the result of max-pooling
- Let's take X_1 and X_2 which are

similar but not the same matrix (i.e. 2

Images which are similar

but not the same!!)

$$X_1 = \begin{bmatrix} 10 & 255 & 125 & 0 & 170 & 100 \\ 70 & 255 & 105 & 25 & 25 & 70 \\ 255 & 0 & 150 & 0 & 10 & 10 \\ 0 & 255 & 10 & 10 & 150 & 20 \\ 70 & 15 & 200 & 100 & 95 & 0 \\ 35 & 25 & 100 & 20 & 0 & 60 \end{bmatrix}$$

$$X_2 = \begin{bmatrix} 100 & 100 & 100 & 50 & 100 & 50 \\ 95 & 255 & 100 & 125 & 125 & 170 \\ 80 & 40 & 10 & 10 & 125 & 150 \\ 255 & 30 & 150 & 20 & 120 & 125 \\ 30 & 30 & 150 & 100 & 70 & 70 \\ 70 & 30 & 100 & 200 & 70 & 95 \end{bmatrix}$$

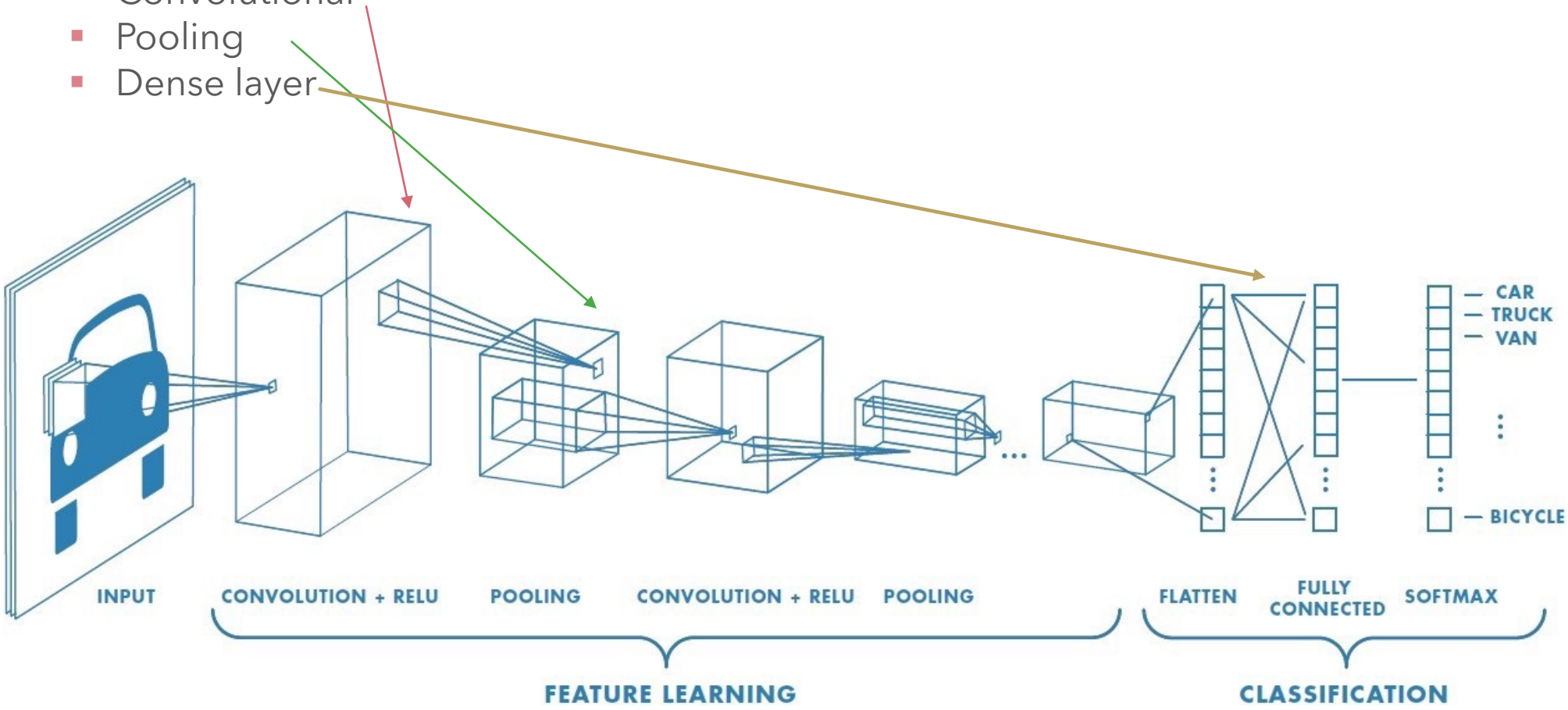
$$\xrightarrow{\text{max pooling } P_{2\times 2}} \begin{bmatrix} 255 & 125 & 170 \\ 255 & 150 & 150 \\ 70 & 200 & 95 \end{bmatrix}$$

ROBUSTNESS W.R.T. NOISE

- Pooling **decreases the size of features**, which results in higher computational efficiency. Furthermore, reducing the number of features may reduce the degree of overfitting as well (and complexity!).
- Traditionally, pooling is assumed to be non-overlapping (**pooling size = stride**)→ so that we have an output in which pixels are independent from one another)

PUTTING EVERYTHING TOGETHER IN A CNN

- A convolutional neural network is a sequence of the following layers ordered in different ways:
  - Convolutional
  - Pooling
  - Dense layer



FILTERS WEIGHTS ARE LEARNT FROM DATA DURING TRAINING

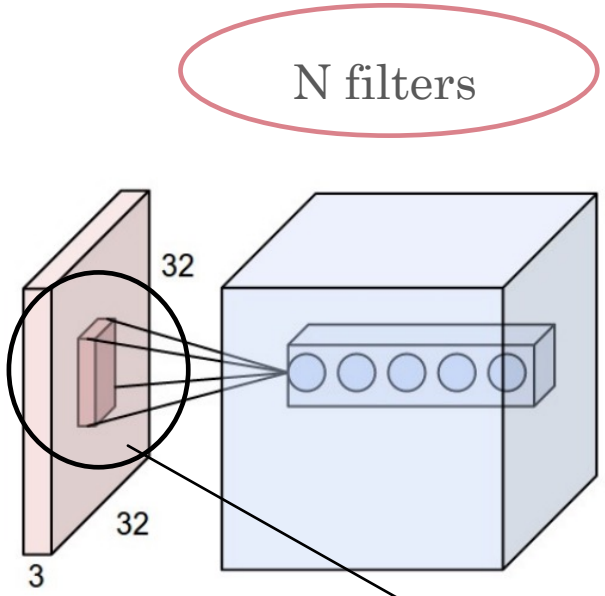
THE NETWORK LEARNS WHICH ARE THE MOST DISCRIMINANT PATTERNS

A CNN PERFORMS THE CLASSIFICATION BY READING THESE EXTRACTED FEATURES

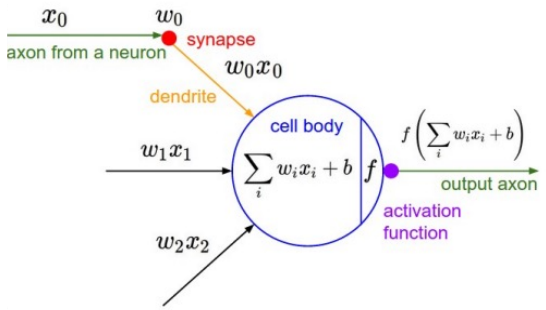
A DNN READS ONLY PIXELS VALUES

PUTTING EVERYTHING TOGETHER IN A CNN

- A convolutional layer is tipically composed of:



Conv operation

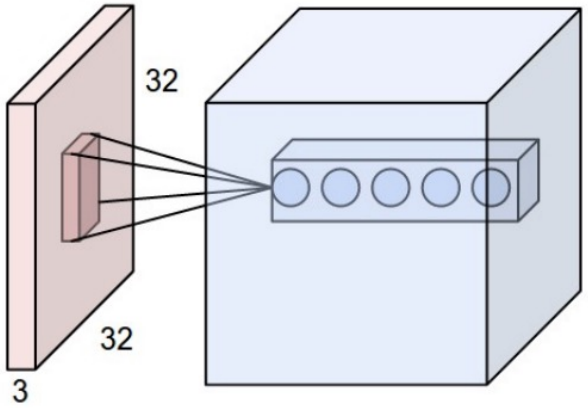


- In a convolution layer, usually **several filters are stacked together**
- Each filter learns some different information from the same image

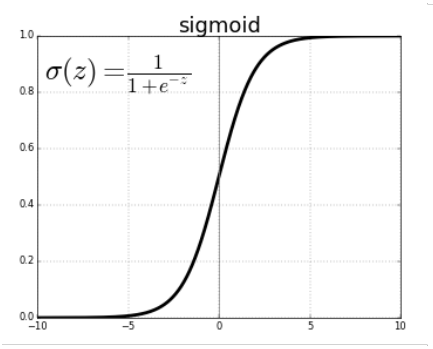
PUTTING EVERYTHING TOGETHER IN A CNN

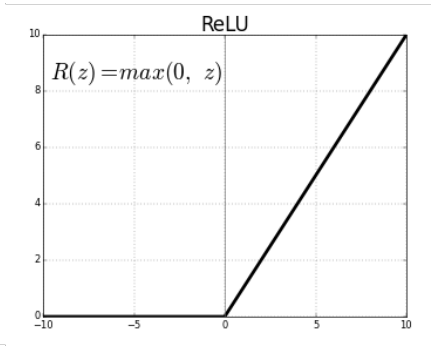
- A convolutional layer is tipically composed of:

N filters + activation function



+

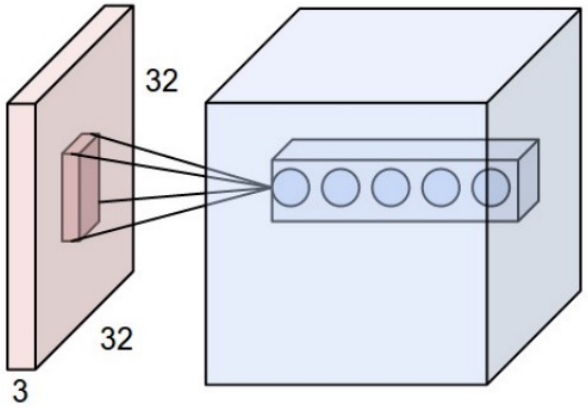




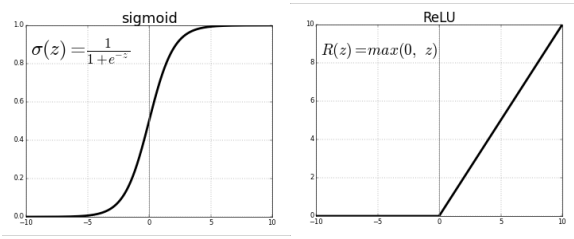
PUTTING EVERYTHING TOGETHER IN A CNN

- A convolutional layer is tipically composed of:

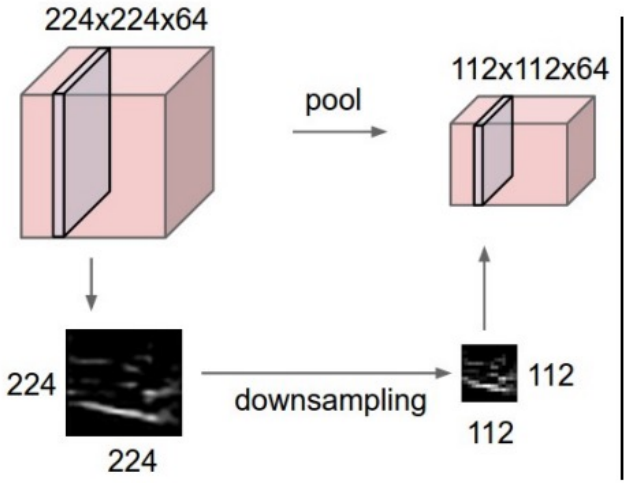
N filters + activation function + pooling layer



+

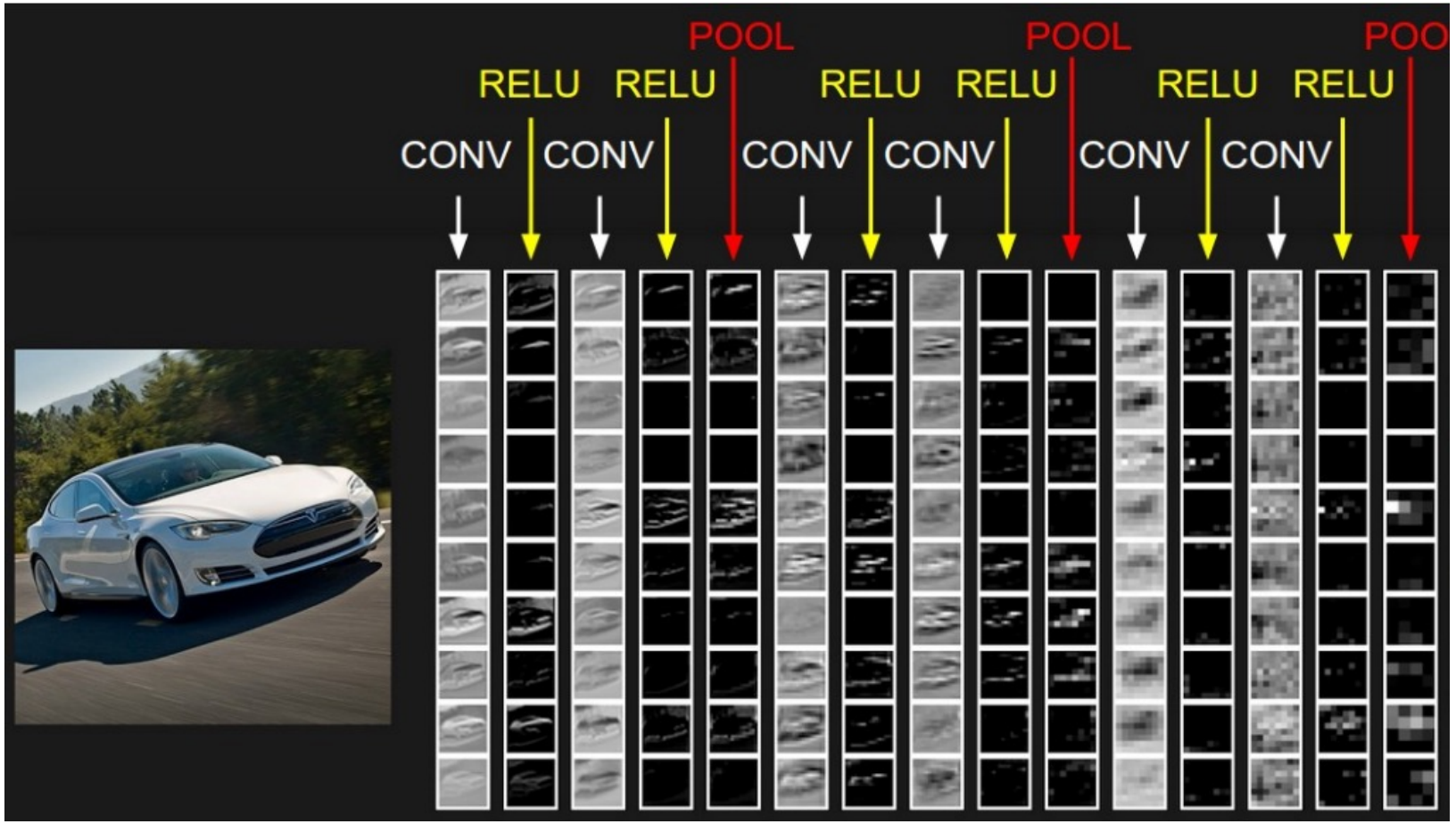


+



PUTTING EVERYTHING TOGETHER IN A CNN

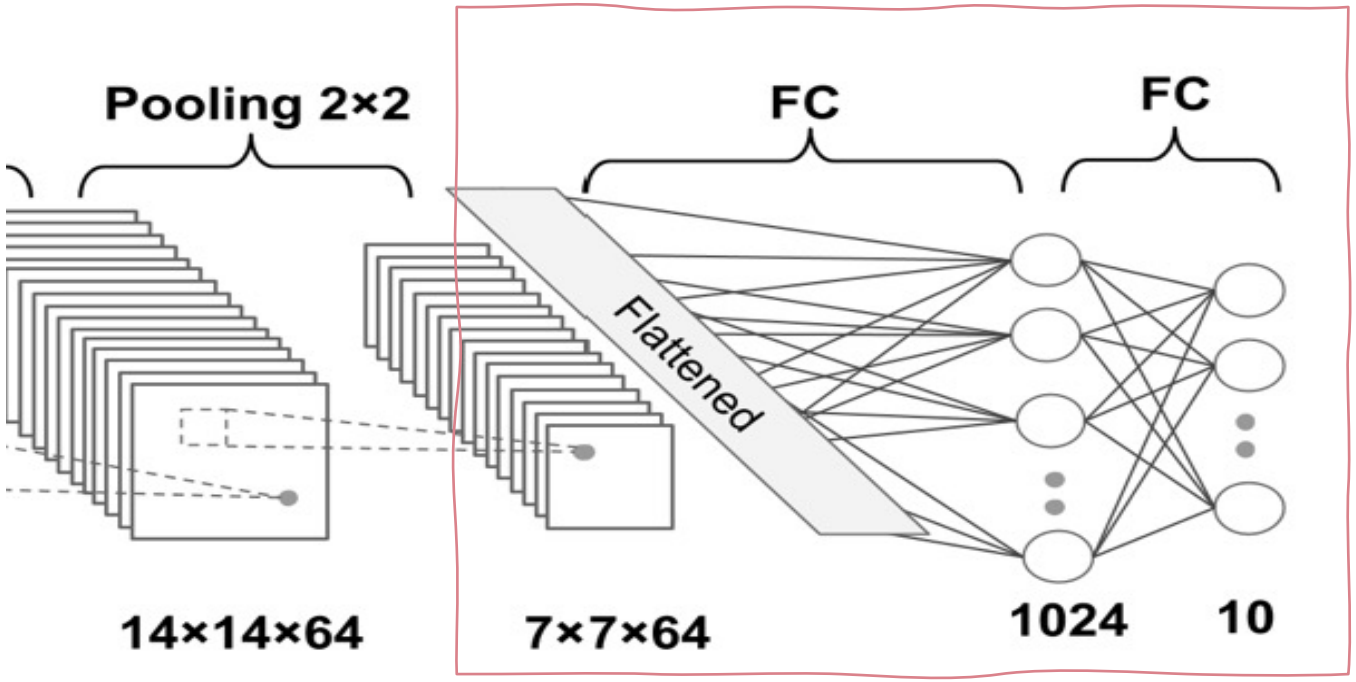
- The sequence convolutional + pooling is not the only possible choice:
  - Modern networks do not use pooling but adjust the output size by tuning the padding and strides of the convolutional layers



DENSE LAYERS FOR CLASSIFICATION

- Now we must **flatten the final output and feed it to a regular Neural Network for classification purposes**
- Adding a Fully-Connected layer is a way of learning non-linear combinations of the high-level features (from filters)

1. The image is **flattened** into a column vector
2. then **fed to a fully-connected neural network**



ACTIVATIONS AND LOSS FUNCTIONS FOR CLASSIFICATION

- Now what our model misses is the final probabilistic interpretation of the output **z** to perform classification:

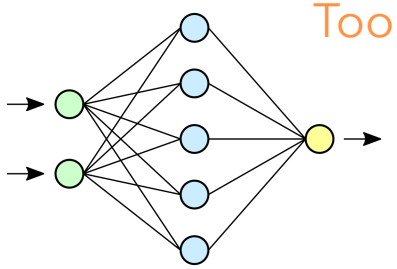
$$z = \boldsymbol{w}^T \boldsymbol{x} = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m$$

- An **activation function** should be applied to the output of the last fully-connected layer:
  - 'Sigmoid' for binary classificatior $\phi(z) = \frac{1}{1 + e^{-z}}$
  - 'Softmax' for multi-classification $p(z) = \phi(z) = \frac{e^{z_i}}{\sum_{j=1}^{M} e^{z_j}}$

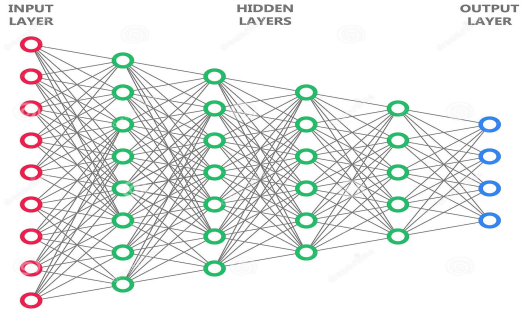
Recall: Sigmoid and Softmax are both probabilities for an event to belong to a given class, therefore they are used only in the outer layer. Other activation functions, like ReLU and tanh are mainly used in the intermediate (hidden) layers to add non-linearities to our model

CNN REGULARIZATION: DROPOUT

- Choosing the size of a network has always been a challenging problem
  - Small networks, or networks with a relatively small number of parameters, are likely to underfit, resulting in poor performance
  - very large networks may result in overfitting, where the network will do extremely well on the training dataset while achieving a poor performance on the test dataset



Too few parameters!



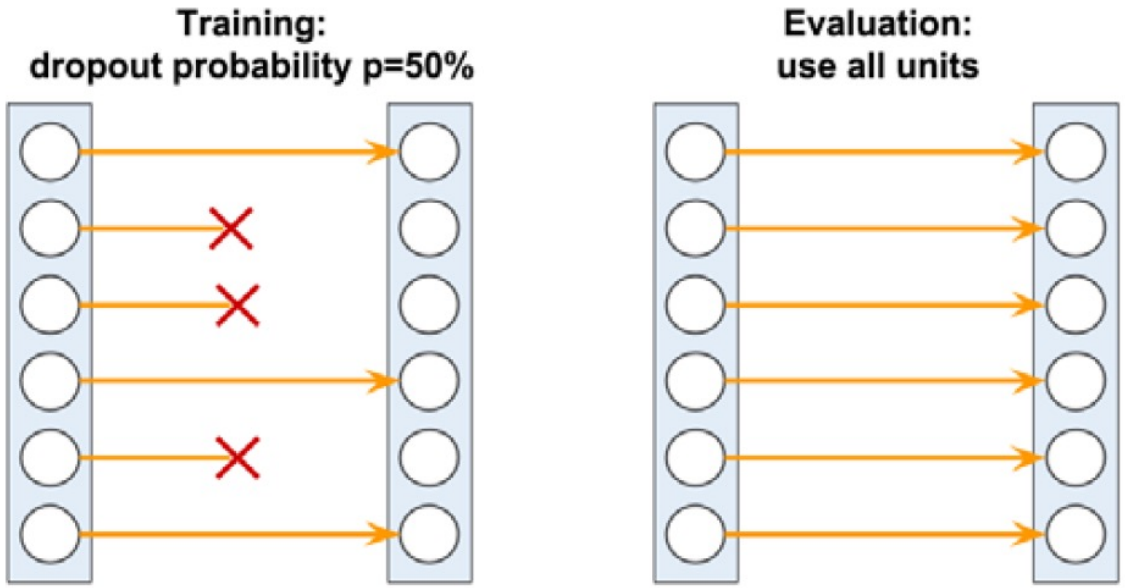
Too many parameters!

- One way to address this problem is **to build a network with a relatively large capacity to do well on the training dataset, then to prevent overfitting we can apply one or multiple regularization schemes** to achieve a good performance on new data

CNN REGULARIZATION: DROPOUT

- **Dropout** has emerged as a popular technique for regularizing: during the training phase **a fraction of the hidden units is randomly dropped** at every iteration with a certain probability (***rate***)

- During prediction, all neurons will contribute to computing the pre-activations of the next layer



ARTIFICIAL INTELLIGENCE FOR THE ELECTRON-ION COLLIDER

ELECTRON-ION COLLIDER

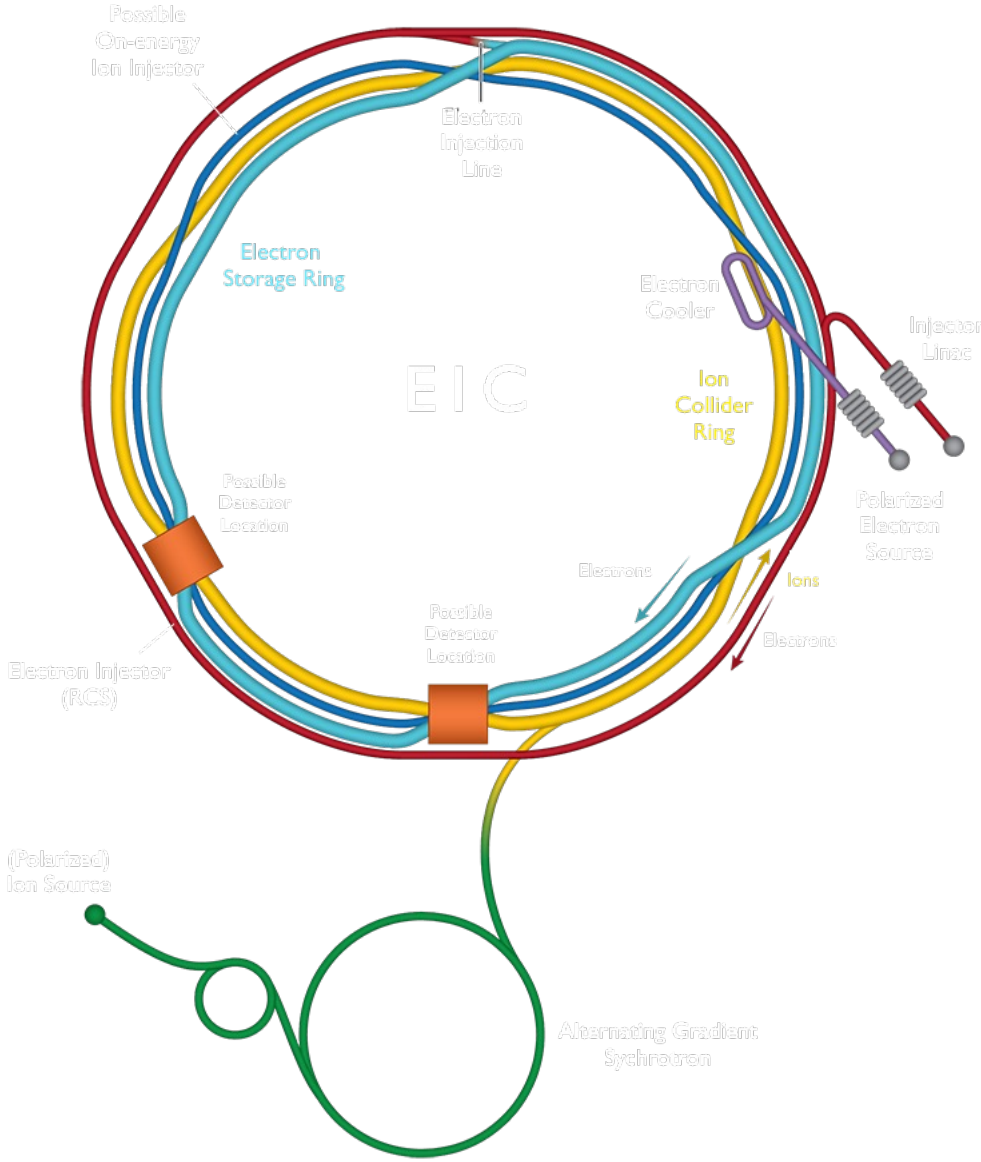
The EIC will be a particle accelerator that collides electrons with protons and nuclei to produce snapshots of those particles' internal structure.

The electron beam will reveal the arrangement of the quarks and gluons that make up the protons and neutrons of nuclei.

The EIC will allow us to study the strong nuclear force (that holds quarks together) carried by the gluons, and the role of gluons in the matter.

EIC will be operating in 2030's: by then AI may also leverage on technologies that are currently **Computing Frontiers**.

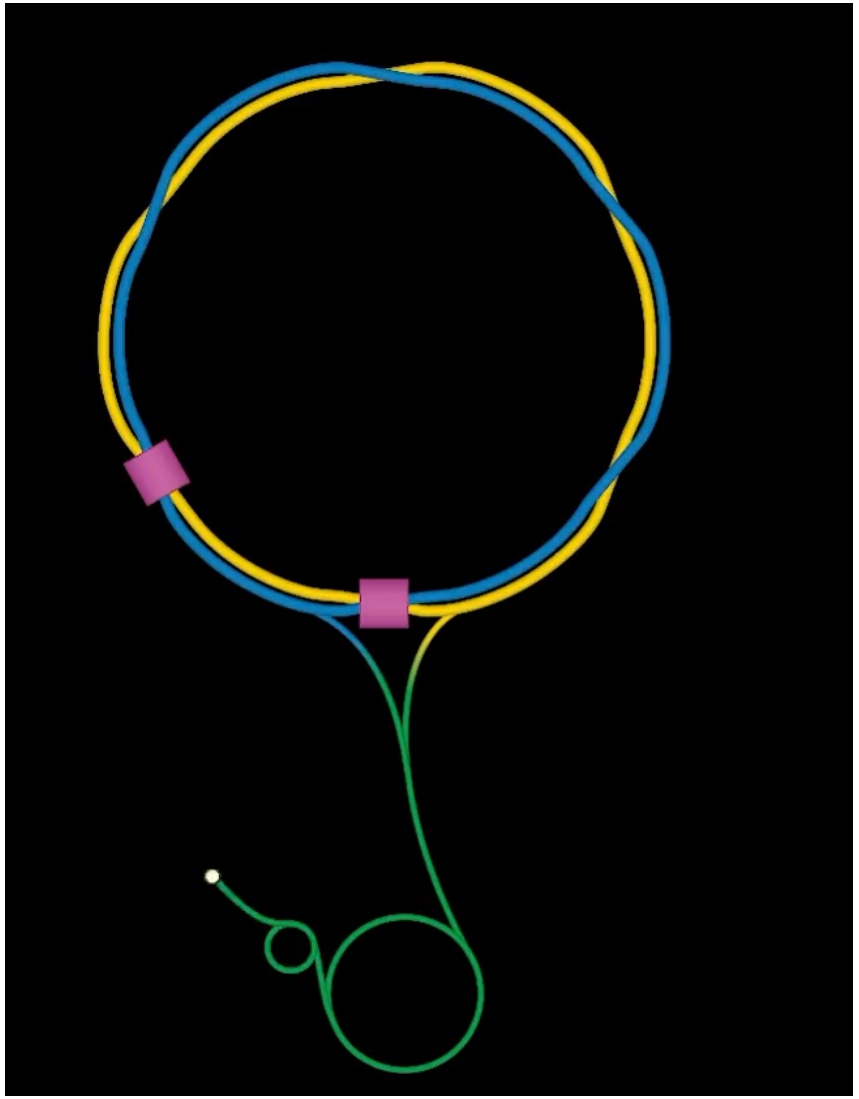
What we learn from the EIC could power the technologies of tomorrow



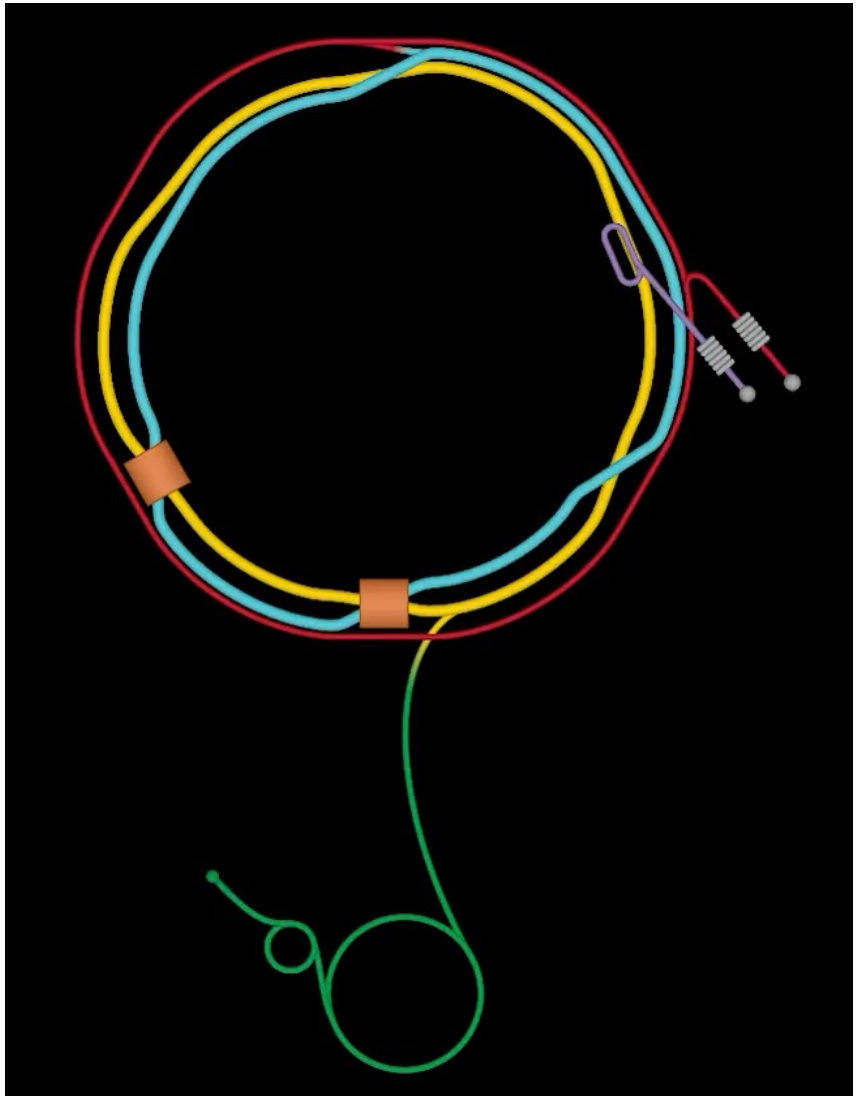
EIC VS RHIC

- The Electron-Ion Collider (EIC) at Brookhaven Lab will reuse the infrastructure from the Relativistic Heavy Ion Collider (RHIC) and build on discoveries at RHIC and the Continuous Electron Beam Accelerator Facility (CEBAF) at Thomas Jefferson National Accelerator Facility (Jefferson Lab)

- EIC will have new features that greatly expand our ability to explore the building blocks of visible matter.



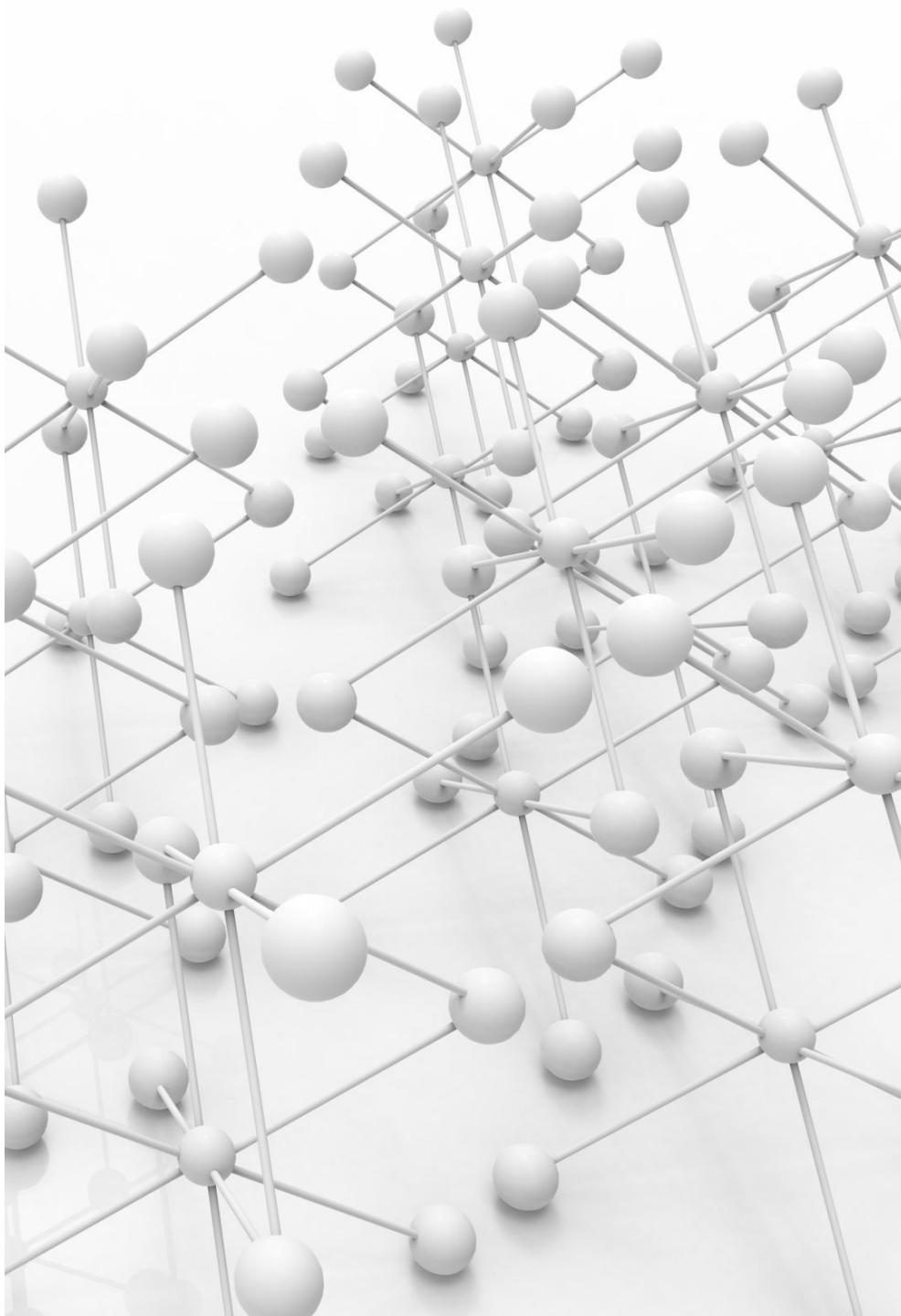
RHIC: Two ion accelerator/storage rings (inside RHIC tunnel).



EIC: One ion accelerator/storage ring plus one electron accelerator ring and one electron storage ring.

AI4EIC: BACKGROUND

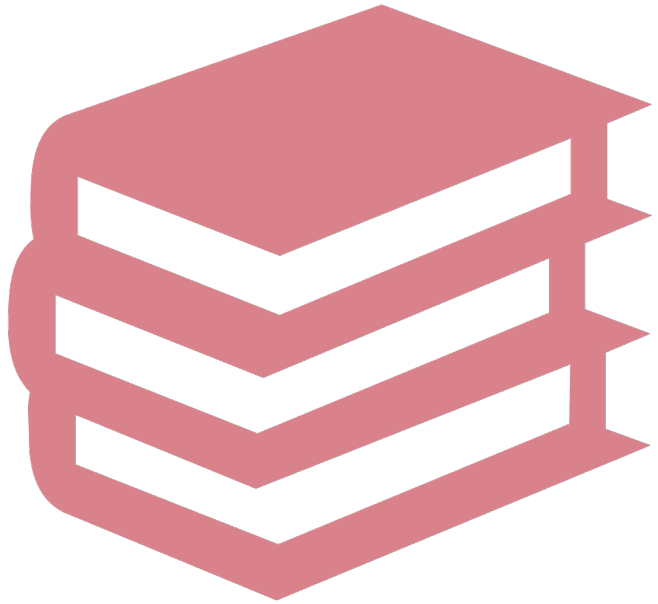
- Artificial Intelligence contributes to all phases of the Electron Ion Collider starting from the Design and R&D.
- **The A.I. goals for EIC**: optimization of this complex problem characterized by multiple parameters and objectives like detector performance and costs.
- AI provides :
  - insight on **hidden correlations among the design parameters**
  - identify **optimal tradeoff solutions**
- The AI-supported Optimization of the **Accelerator and Detector Design** needs reliable **Streaming Readout** and **Simulations** followed by **Reconstruction and Analysis**:



AI4EIC OUTREACH

- AI in our society will be the economic driver of the next decade when EIC will be operating
- The EIC detector can be one for the first large-scale detector to be designed with the assistance of AI in the following areas (in progress):
  - **Accelerator and Detector Design**
  - **Simulations**
  - **Analysis and Reconstruction**
  - **Accelerator and Detector Control**
  - **Streaming Readout**
  - **Computing Frontiers**
  - **Theory and Phenomenology**

REFERENCES



- **MACHINE LEARNING:**
  - Raschka, Sebastian. *Python machine learning*. Packt publishing ltd, 2015.
  - Dive Into Deep Learning: http://d2l.ai/chapter_preface/index.html
  - Keras Guide: Developer guides (keras.io)
  - Scikit: https://towardsdatascience.com/convolution-vs-correlation-af868b6b4fb5
- **AI4EIC:**
  - https://meetings.triumf.ca/event/269/contributions/3407/attachments/2634/3124/WNPPC_2022_ECCE_AI_Ksuresh.pdf
  - https://eic.ai/
  - C. Fanelli «**Design of Detectors at the Electron Ion Collider with Artificial Intelligence**», https://doi.org/10.1088/1748-0221/17/04/C04038
- https://ai4eicdetopt.pythonanywhere.com
- https://cfteach.github.io/nnpss

BACKUP

FOCUS ON ENCODING CLASS LABELS

- Many machine learning libraries require that class labels are encoded as integer values.
- It's good practice to provide class labels as integer arrays to avoid technical glitches.
- To encode the class labels we can map ordinal features
- Thus, we can simply enumerate the class labels starting from 0:

```
>>> import numpy as np
>>> class_mapping = {label:idx for idx,label in
...                  enumerate(np.unique(df['classlabel']))}
>>> class_mapping
{'class1': 0, 'class2': 1}
```

- Next we can use the mapping dictionary to transform the class labels into integers:

```
>>> df['classlabel'] = df['classlabel'].map(class_mapping)
>>> df
   color  size  price  classlabel
0  green     1   10.1           0
1    red     2   13.5           1
2   blue     3   15.3           0
```

- We can reverse the key-value pairs in the mapping dictionary as follows to map the converted class labels back to the original string representation:

```
>>> inv_class_mapping = {v: k for k, v in
class_mapping.items()}
>>> df['classlabel'] = df['classlabel'].map(inv_class_mapping)
>>> df
   color  size  price classlabel
0  green     1   10.1     class1
1    red     2   13.5     class2
2   blue     3   15.3     class1
```

- Alternatively, there is a convenient LabelEncoder class directly implemented in scikit-learn to achieve the same:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> class_le = LabelEncoder()
>>> y = class_le.fit_transform(df['classlabel'].values)
>>> y
array([0, 1, 0])
```

ONE-HOT ENCODING

- Instead of using a simple dictionary-mapping approach to convert the ordinal size feature into integers. Since scikit-learn's estimators treat class labels without any order, we used the convenient LabelEncoder class to encode the string labels into integers.
- It may appear that we could use a similar approach to transform the nominal color column of our dataset, as follows:

```
>>> X = df[['color', 'size', 'price']].values
>>> color_le = LabelEncoder()
>>> X[:, 0] = color_le.fit_transform(X[:, 0])
>>> X
array([[1, 1, 10.1],
       [2, 2, 13.5],
       [0, 3, 15.3]], dtype=object)
```

- After executing the preceding code, the first column of the NumPy array X now holds the new color values, which are encoded as follows:

- If we stop at this point and feed the array to our classifier, we will make one of the most common mistakes in dealing with categorical data:
  - Although the color values don't come in any particular order, a learning algorithm will now assume that *green* is larger than *blue*, and *red* is larger than *green*. Although this assumption is incorrect, the algorithm could still produce useful results.
  - WORKAROUND: **one-hot encoding**
    - the idea behind this approach is to create a new **dummy feature** for each unique value in the nominal feature column
    - We convert the color feature into three new features: blue, green, and red. Binary values can then be used to indicate the particular color of a sample; for example, a blue sample can be encoded as blue=1, green=0, red=0.
    - To perform this transformation, we can use the OneHotEncoder that is implemented

in the scikit-learn.preprocessing module:

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> ohe = OneHotEncoder(categorical_features=[0])
>>> ohe.fit_transform(X).toarray()
array([[ 0. ,  1. ,  0. ,  1. , 10.1],
       [ 0. ,  0. ,  1. ,  2. , 13.5],
       [ 1. ,  0. ,  0. ,  3. , 15.3]])
```

ONE-HOT ENCODING

- An even more convenient way to create those dummy features via one-hot encoding is to use the get_dummies method implemented in pandas. Applied on a DataFrame, the get_dummies method will only convert string columns and leave all other columns unchanged:

```
>>> pd.get_dummies(df[['price', 'color', 'size']])
   price  size  color_blue  color_green  color_red
0   10.1     1           0            1          0
1   13.5     2           0            0          1
2   15.3     3           1            0          0
```

ACTIVATIONS AND LOSS FUNCTIONS FOR CLASSIFICATION WITH CNNS

- Focusing on classification problems, depending on the type of problem and the type of output (logits versus probabilities), we should choose the **appropriate loss function** to train our model

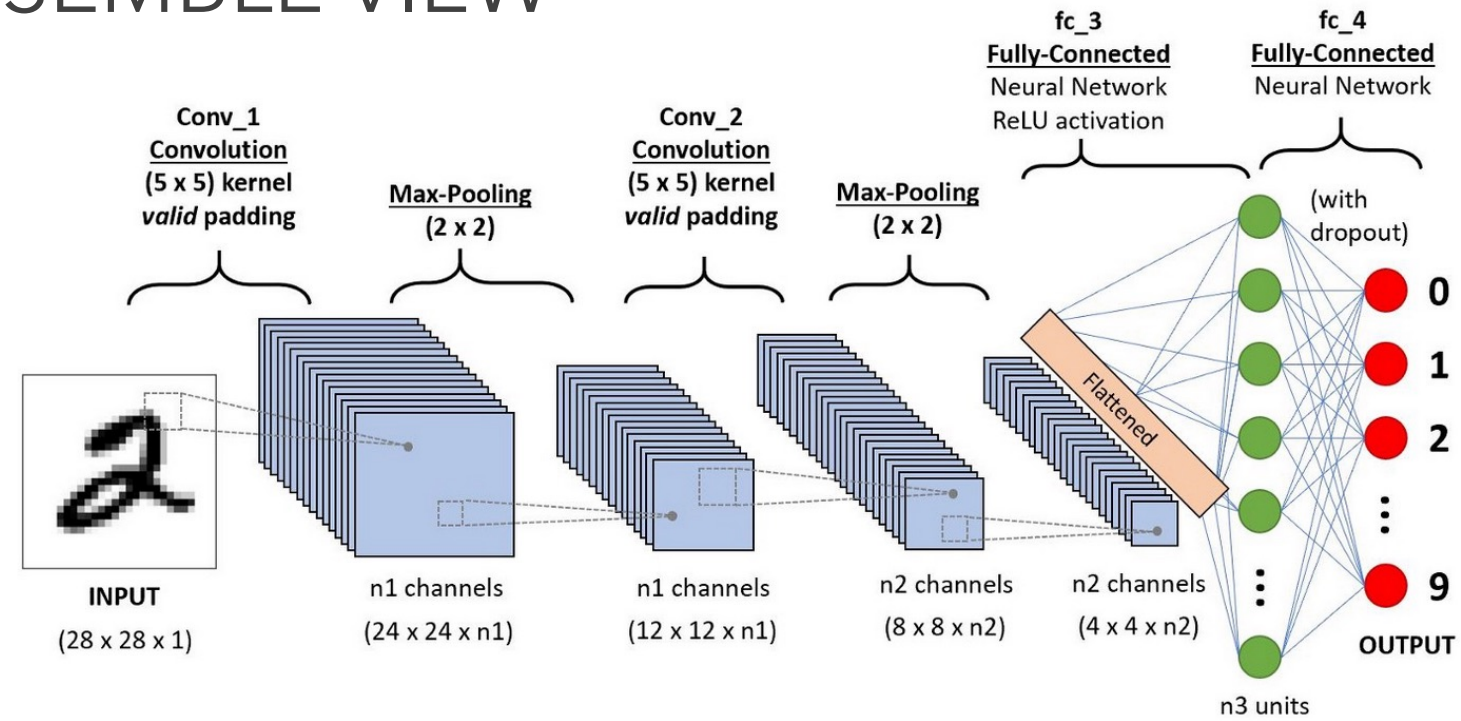
Loss function	Usage	Examples	
		Using probabilities	Using logits
		from_logits=False	*from_logits=True*
`BinaryCrossentropy`	Binary classification	y_true: 1 y_pred: 0.69	y_true: 1 y_pred: 0.8
`CategoricalCrossentropy`	Multiclass classification	y_true: 0 0 1 y_pred: 0.30 0.15 0.55	y_true: 0 0 1 y_pred: 1.5 0.8 2.1
`Sparse CategoricalCrossentropy`	Multiclass classification	y_true: 2 y_pred: 0.30 0.15 0.55	y_true: 2 y_pred: 1.5 0.8 2.1

- With 'from_logits=True' logits are provided as inputs to the loss function (not the activation output), the inverse of the sigmoid function:

$$\text{logit}(p) = \ln\left(\frac{p}{1-p}\right)$$

  - It is preferred due to numerical stability reasons

AN ENSEMBLE VIEW



- Since **the first convolutional filters** learn high level features in the image in input and the input size is larger than in inner layers, the **number of filters is relatively small in order to not insert too many weights**
- A good practice is **to increment this number in the subsequent convolutional steps**
- **Dropout can be inserted not only between dense layers but also between a Conv layer and its input**

DATA AUGMENTATION FOR CNNS

- When the size of training dataset is small, it is a good practice to increase the fit performances applying **Data Augmentation**:
  - It consists in **replicating existent images by applying small changes to it** (**rotation, translation, resizing, flip**..)
- In this way not only the number of training samples increases but each picture is fed to the network with different prospections

THE NETWORK GENERALIZES BETTER!!!

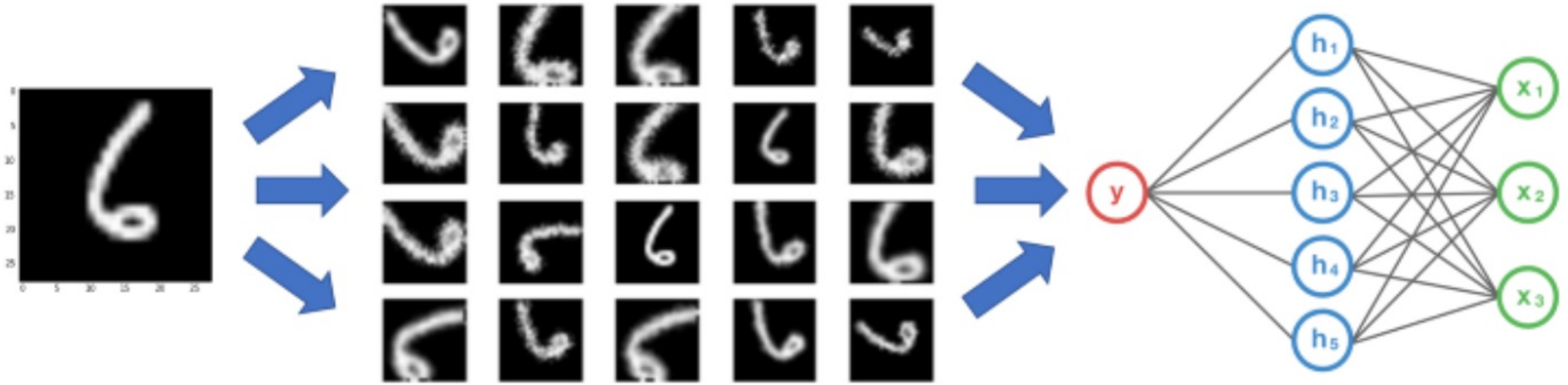
  - EXAMPLE:
    - a poorly trained neural network would think that these three tennis balls shown below are distinct images, instead they are not

DATA AUGMENTATION

- In the real-world scenario, we may have a **dataset** of images taken in a **limited set of conditions, but** our **target application** may exist in a **variety of conditions**, such as different orientation, location, scale, brightness etc.

- A convolutional neural network that can robustly classify objects even if it is placed in different orientations is said to have the property called **invariance** to **translation, viewpoint, size** or **illumination**

- We account for these situations by training our neural network with additional **synthetically modified data**



DATA AUGMENTATION

- Data augmentation can help to increase the amount of **relevant data** in the dataset

Example:

- Let's suppose you have to train a CNN for learning to distinguish between two car brands:

Brand A (Ford)

Brand B (Chevrolet)

- In the dataset all Brand A cars are facing left and all Brand B cars are facing right
- Now, you feed this dataset to your "state-of-the-art" neural network, and you hope to get impressive results once it's trained

DATA AUGMENTATION

- From training you get a 95% accuracy on your dataset
- If you feed a Brand A car to the CNN ….
- …. your neural network output is a Brand B car!

Why did this happen?

- A CNN finds the most obvious features that distinguishes one class from another, here all cars of Brand A were facing left and all cars of Brand B were facing right

Solution

GRID SEARCH

- Grid-search is used to find the optimal hyperparameters of a model which results in the most 'accurate' predictions.
- https://towardsdatascience.com/grid-search-for-model-tuning-3319b259367e
- https://keras.io/api/keras_tuner/tuners/grid/