# Applied Machine Learning - Advanced

Prof. Daniele Bonacorsi

## Lecture 1

Data Science and Computation PhD + Master in Bioinformatics

**University of Bologna**

# Applied ML “basic”: course recap

# Recap [1/4]

Definition of ML

Types of ML

Supervised ML (vs Unsupervised)

Univariate linear regression

- cost function, GD

Multivariate Linear regression

- cost function, GD
  - feature scaling, learning rate, polynomial regression

Model representation

- computing parameters analytically with the normal equation

# Recap [2/4]

## Logistic regression

- classification
  - hypothesis representation, decision boundaries
- logistic regression model
  - cost function
  - simplified cost function and GD
  - advanced optimisations
  - multiclass classification

## Regularization

- the problem of overfitting
- cost function
- regularised linear regression
- regularised logistic regression

# Recap [3/4]

## Advices for applying ML

- evaluating a learning model
  - evaluating a hypothesis
  - model selection and TVT sets
- Bias and variance
  - Diagnosis
  - Regularisation and bias/variance
  - Learning curves
- Error metrics for skewed classes
  - trade-off precision vs recall
  - ROC and AUC

# Recap [4/4]

Neural Networks

- NN or feature crosses?
- intuition about the motivation towards NNs

Examples 100% with pure **sklearn**

# Tools for the Basic and Advanced part

**You need to install NOTHING.**

- You just need a **network** and a **browser**

The course is definitely **fluid** over time.. Effort to keep the course up-to-date but at the same not to disrupt it completely in one AY

- part 1 used mostly sklearn, part 2 uses mostly Keras+TensorFlow

You can code in the lecture slots and/or at home

- I am giving you paths and guidance code, you explore..
- I will give you "solutions", where needed

# DISCLAIMER before starting

I am assuming that:

- you attended the BASIC part, and I give that content from granted
- if you did not attend it, it is because you knew everything already!

So:

- we will give for granted we know how to work on Colab
- we will use python with less mercy..
- we will use numpy, pandas, matplotlib, seaborn, etc..
- we will use sklearn with a bit of familiarity..

In case of troubles/difficulties to follow, speak-up or contact me privately! Remember we have tutors, too!

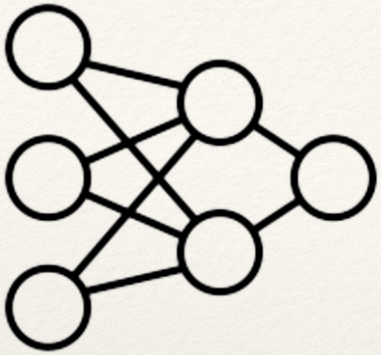
# Be prepared

We will jump.

From not-NN algos, to just NNs deep and forever. And back.

IMO it is the (educationally) right way to go

- If you disagree, TELL ME IN THE FINAL QUESTIONNAIRE!



# NNs - intro

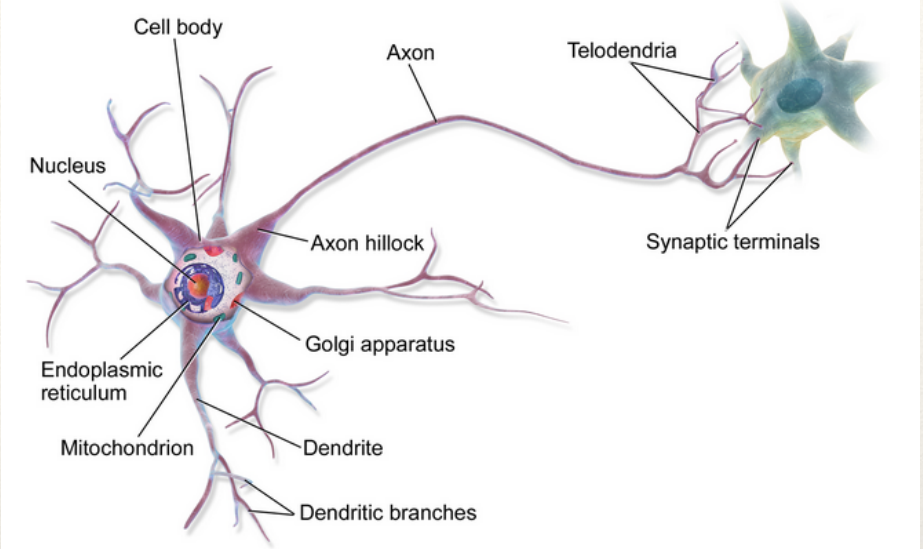
# BNN

The architecture of biological neural networks (BNN) is still the subject of active research, but some parts of the brain have been mapped, and we know that **neurons are relatively simple** and they are **connected in large networks**, that are **often organized in consecutive layers**…

# From BNN to ANN

ANN as simulation of Biological NN

- NN are developed by simulating networks of human neurons




The human neuron has a cell body, a number of input wires (*dendrites*) and an output wire (*axon*)

- we can think of it as a **computational unit** that gets inputs, computes, and spits output to other neurons (which we will call **nodes** or **units**)
- we can think of it simply as a **logistic unit** that computes some h as a sigmoid

Communication in BNNs is done via pulses of electricity

- we can think of **I/O** passing in/out **numerical values**
- a **wide** network..

**Vastly oversimplified** model..

# Why NNs as ML algorithms?

Important to model **non-linear relationships among attributes**.

NNs are at the very core of **Deep Learning**

- versatile, powerful, scalable

Ideal to tackle large and highly complex ML tasks, such as:

- classifying billions of **images** (e.g. Google Images)
- powering **speech recognition** services (e.g. Siri, Alexa, ..)
- **recommending** the best videos to watch to *O*(100M) users every day (e.g. YouTube)
- **learning to beat humans** in very specific tasks (e.g. from medical applications - e.g. radiology - to gaming - e.g. Go and DeepMind's AlphaZero)

# Brief history of NNs

**1943** (!): first introduction by McCulloch and Pitts

- a neurophysiologist and a mathematician
- "*A Logical Calculus of Ideas Immanent in Nervous Activity*"
- the first ANN architecture in human history
  - a simplified computational model of how biological neurons might work together in animal brains to perform complex computations using propositional logic

**1960s**: the successes of ANNs stopped - the "**first AI winter**"

**Early 1980s**: revival of interest in connectionism, new architectures, better training techniques. Slow progress, though.

**1990s**: other (not NN) powerful ML techniques were invented, e.g. SVMs. Seemed to offer better results and stronger theoretical foundations than ANNs - the "**second AI winter**"

**2000s-2010s**: a new "AI spring": why now? will it last?

# Acceleration in 2010s on ANNs

*[ DISCLAIMER: also personal opinions here ]*

Opinions may vary, but IMO main actors have been:

- the raise of **Big Data** to train ANNs
  - often outperform other ML techniques on very large and complex problems
- the **technology progresses** (e.g. GPUs, TPUs, FPGAs, ..)
  - tremendous increase in computing technology since the 1990s, training time down by large factors, in part due to Moore's Law, in part to gaming industry (powerful GPU cards)
- "Democratisation" [*] of massive computing resources via **Cloud** approaches
  - [*] not free-of-charge, based on digital companies' business models, debatably "democratic" in a social sense, but offer options unavailable with on-premise facilities
- training algos improvements from the 1900s
  - fairly? minor.. but relatively small tweaks had a huge positive impact, still

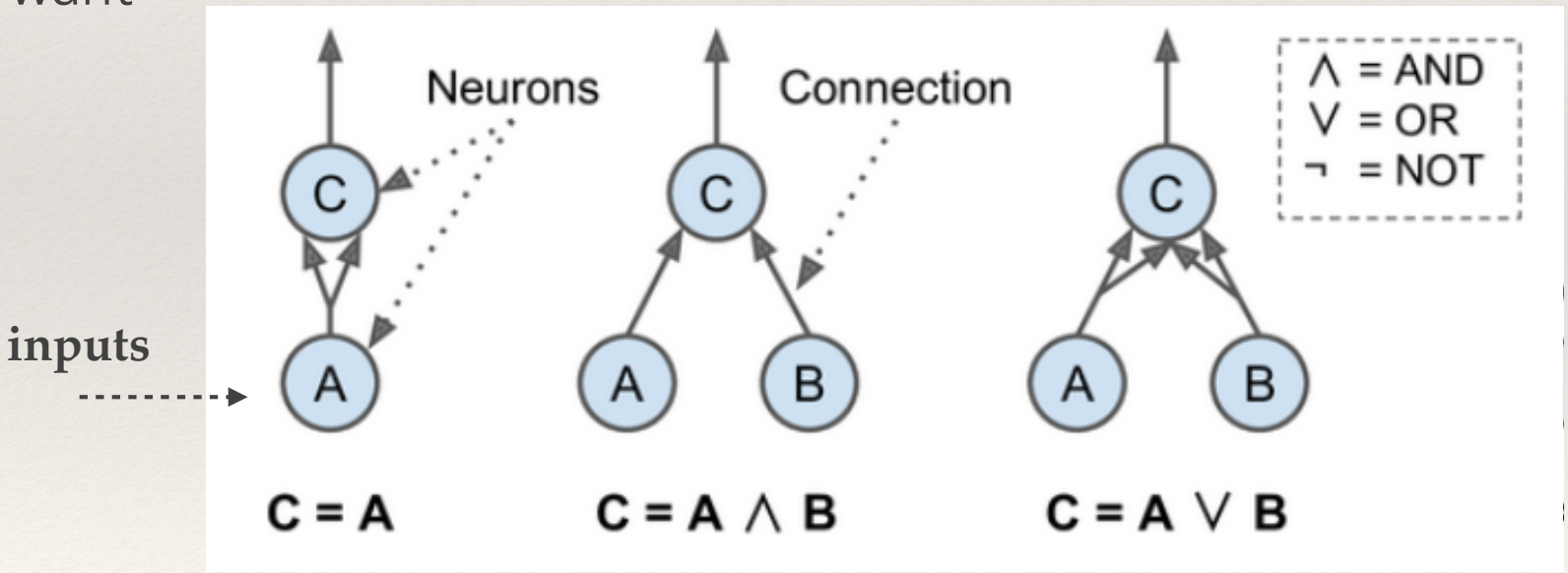
Today, it is a fact that **ML/DL is among the core transformative technologies at the basis of most world-wide activities aiming at extracting actionable insight from (big) data**

# Logical computation with neurons

The McCulloch/Pitts model for a biological neuron was later called "**artificial neuron**"

- it has 1 (or more) **binary (on/off) inputs** and 1 **binary** output (*NOTE: all binaries!*)
- it activates its output when more than a certain number of its inputs are active (e.g. 2 in the examples below)

They showed that even with such a simplified model it is possible to build a network of artificial neurons that computes **any basic logical proposition** you want

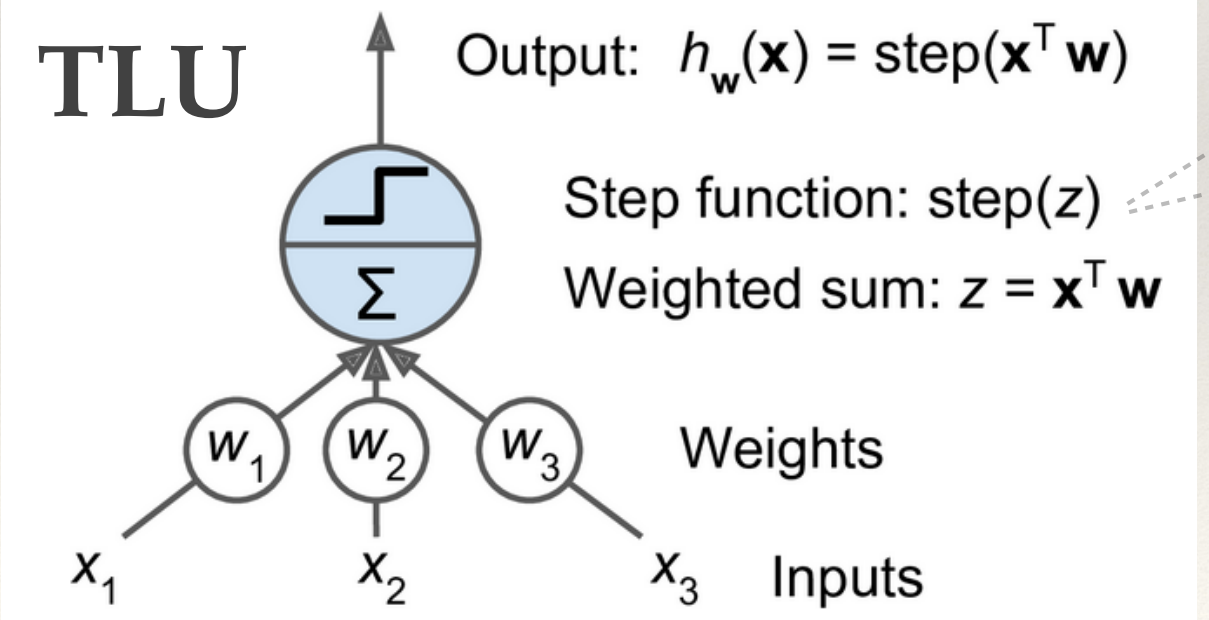



Imagine yourself how to combine them to form more complex logical functions

# The **TLU** and the Perceptron

**1957**, Rosenblatt: "Perceptron" as one of the simplest possible ANN architectures, based on a slightly modified artificial neuron, called **threshold logic unit** (**TLU**) or linear threshold unit (LTU)

- the **inputs/output** are now **numbers** (not binary on/off values)
- each input connection is associated with a **weight**
- the TLU computes a **weighted sum of its inputs**, applies a **step function** to that sum, and **outputs** the result




$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

$$\text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

1 single TLU can act as a simple **linear binary classifier**

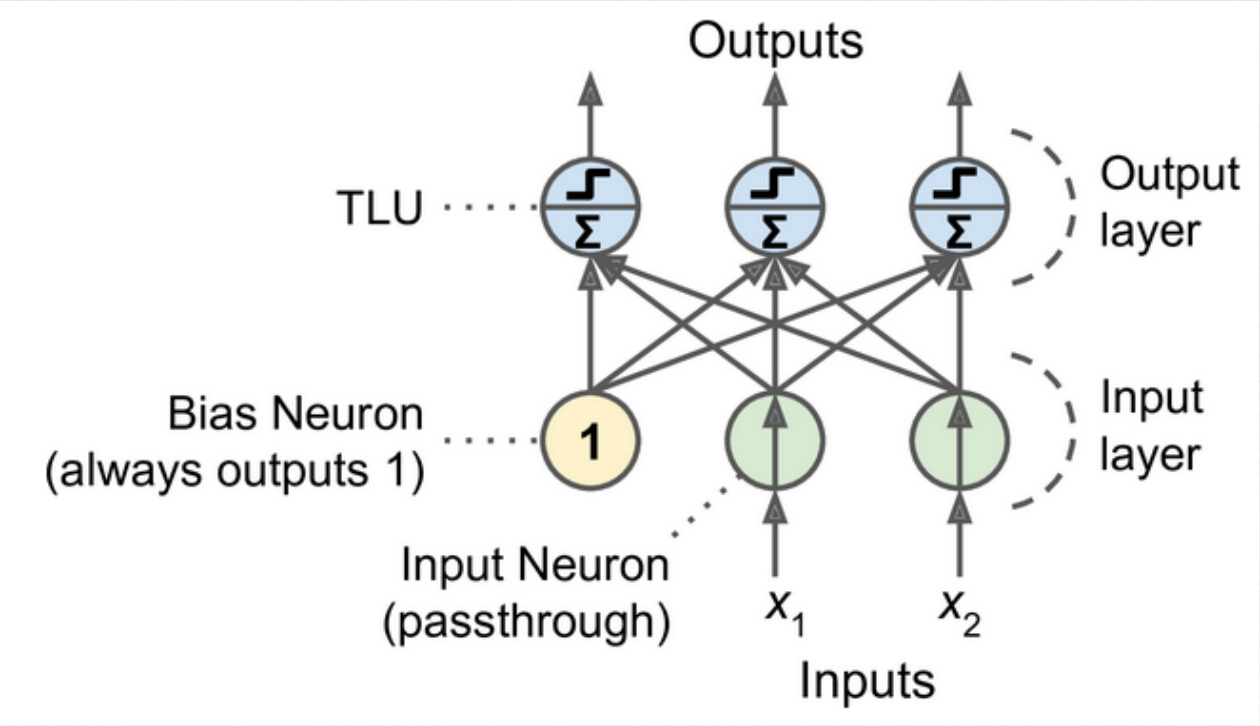
- linear combination of the inputs, and depending of the thresholds → positive/ negative class (just like Logistic Regression classifier or a linear SVM !)

# The TLU and the **Perceptron**

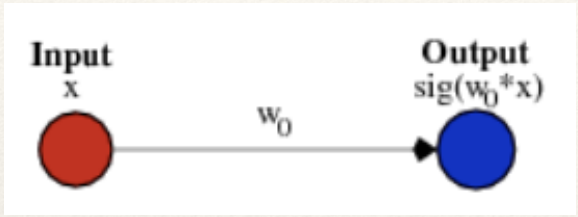
A **Perceptron** is then composed of **a single layer of TLUs** with each TLU connected to all the inputs

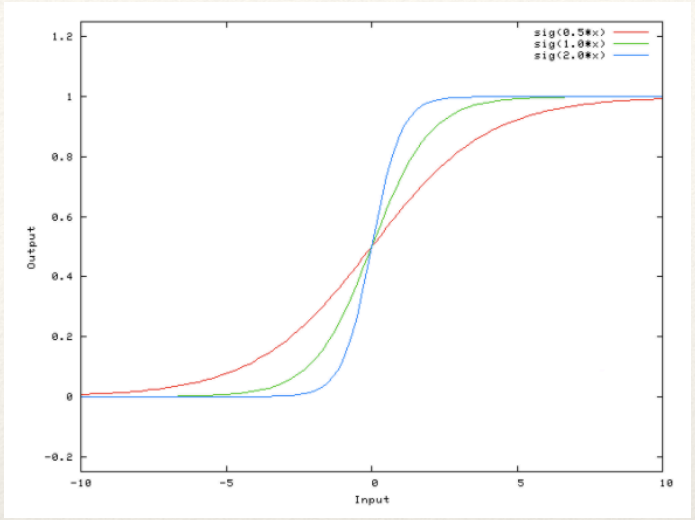
- in the **input layer**, an extra bias feature is generally added ($x_0 = 1$), via a "**bias neuron**" that just outputs 1 all the time.
- When all the neurons in a layer are connected to all the neuron in the previous layer (i.e. here the input neurons), it is called a **fully-connected** or **dense** layer

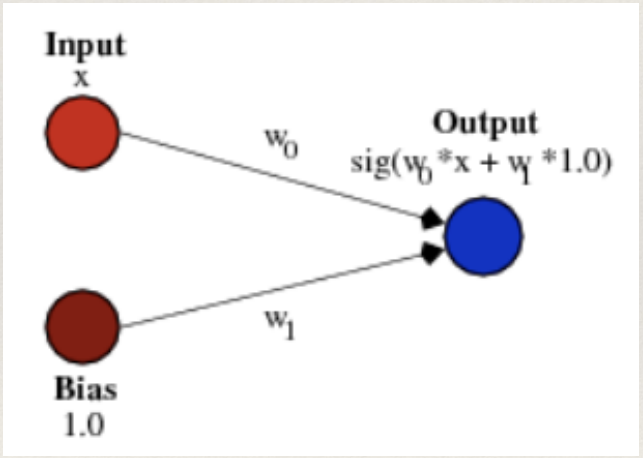
This Perceptron can classify instances simultaneously into 3 different binary classes, which makes it a **linear multi-class classifier**

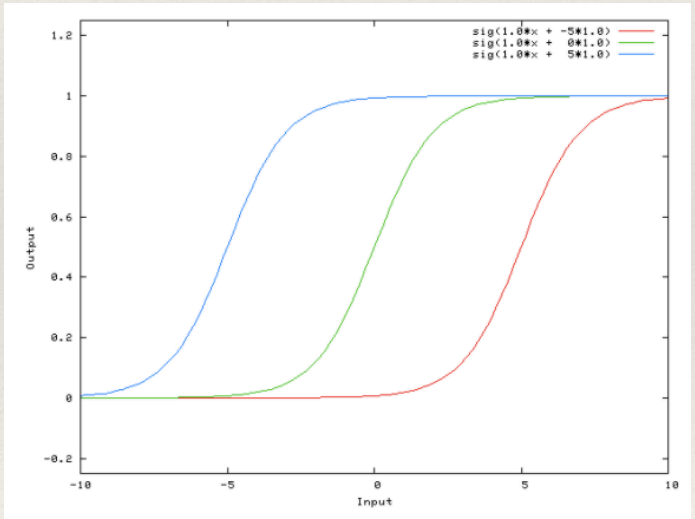


# Why a bias neuron: the long answer




A bias value allows you to shift the activation function to the left or right, which may be critical for successful learning.

# Why a bias neuron: the short answer

Why?

Simple analogy: just like b in **y=ax+b**.

# Training a **Perceptron**

Input from biology and neurosciences

- *1949, Hebb; "Organization of Behavior"* [Ref-Hebb]
- **Hebb's rule** (Hebbian learning): when a biological neuron triggers another neuron, the connection between these two neurons **grows stronger**
  - Lowel: "**Cells that fire together, wire together**"

Largely inspired by Hebb's rule, the Perceptron training algorithm proposed by Rosenblatt was a **weight update**

- the Perceptron is fed one training instance at a time. For each instance it makes its prediction(s). For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction

$$w_{ij}^{(\max)} = w_{ij} + \eta \left( y_j - \hat{y}_j \right) x_i$$

$w_{i,j}$ is the connection weight between the $i^{\text{th}}$ input neuron and the $j^{\text{th}}$ output neuron.
$x_i$ is the $i^{\text{th}}$ input value of the current training instance.
$\hat{y}_j$ is the output of the $j^{\text{th}}$ output neuron for the current training instance.
$y_j$ is the target output of the $j^{\text{th}}$ ourput neuron for the current training instance.
$\eta$ is the learning rate.

$$w_{ij}^{(\max)} = w_{ij} + \eta \left( y_j - \hat{y}_j \right) x_i$$

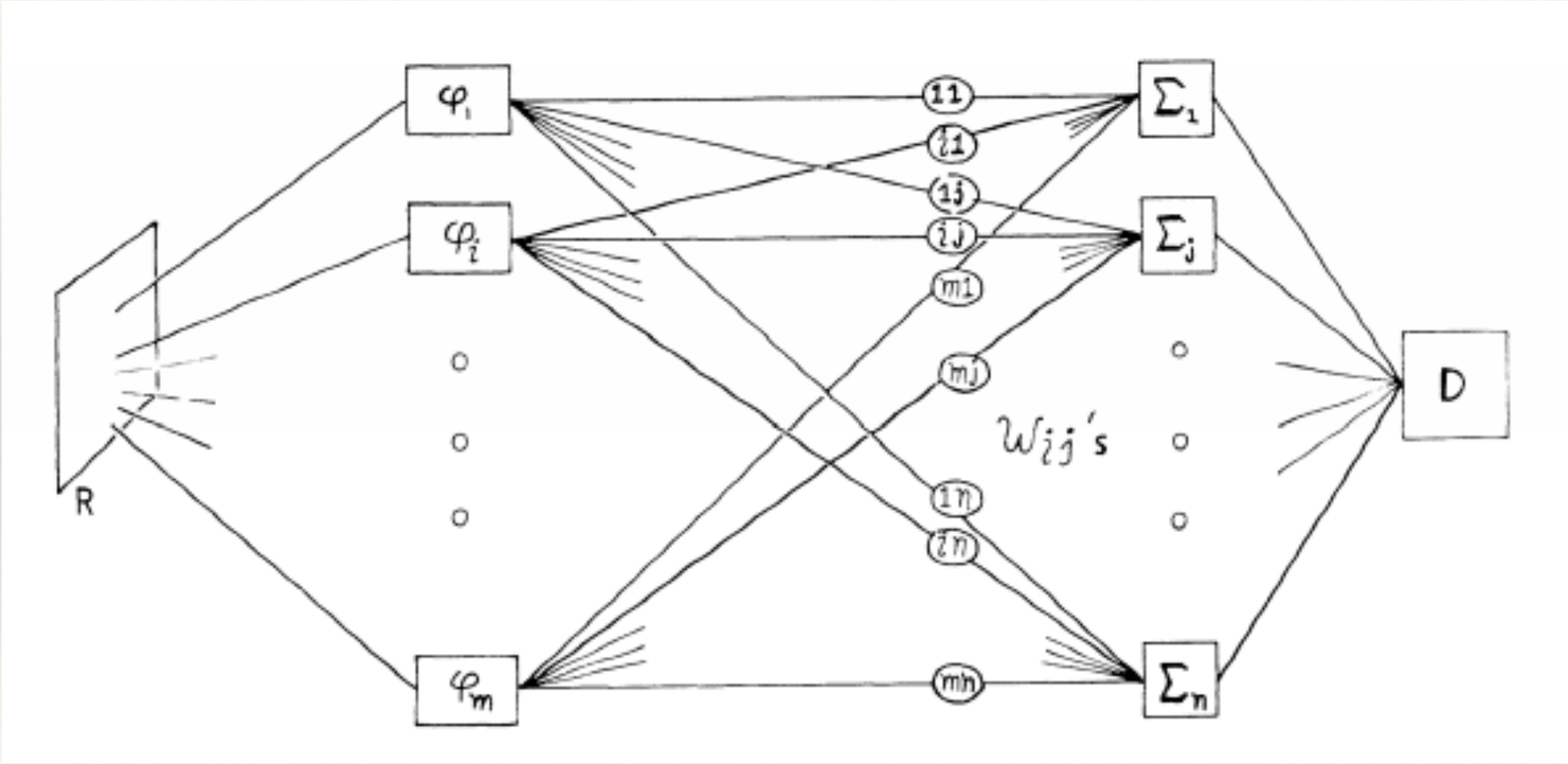
$w_{i,j}$ is the connection weight between the $i^{\text{th}}$ input neuron and the $j^{\text{th}}$ output neuron.
$x_i$ is the $i^{\text{th}}$ input value of the current training instance.
$\hat{y}_j$ is the output of the $j^{\text{th}}$ output neuron for the current training instance.
$y_j$ is the target output of the $j^{\text{th}}$ ourput neuron for the current training instance.
$\eta$ is the learning rate.

The decision boundary of each output neuron is **linear**..
so Perceptrons are incapable of learning complex patterns…

Note: you can implement a Perceptron in sklearn, and compare it to LogisticRegression.
Works like charm - but it does not output class probabilities. Nice idea, but weak.. yet..

# Multi-Layer Perceptron (MLP)

*1969, Minsky/Papert, "Perceptrons", the monograph* [Ref-MinskyPapert]



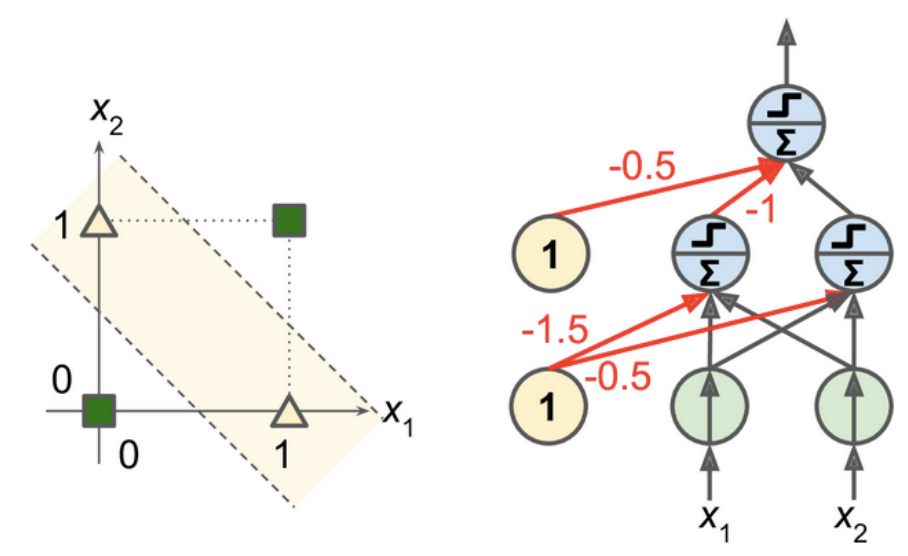
# Multi-Layer Perceptron (MLP)

Minsky/Papert:

- serious weaknesses of Perceptrons highlighted, in particular its incapability of solving relatively trivial problems, e.g. the exclusive-OR (XOR) classification problem
- Well, true of any other linear classification model (e.g. Logistic Regression classifiers).. but **researchers had expected much more from Perceptrons!** Great disappointment. People dropping off the field..

But.. it soon turned out most limitations of Perceptrons could be eliminated by **stacking multiple Perceptrons**! → first concept of "**layers**"!

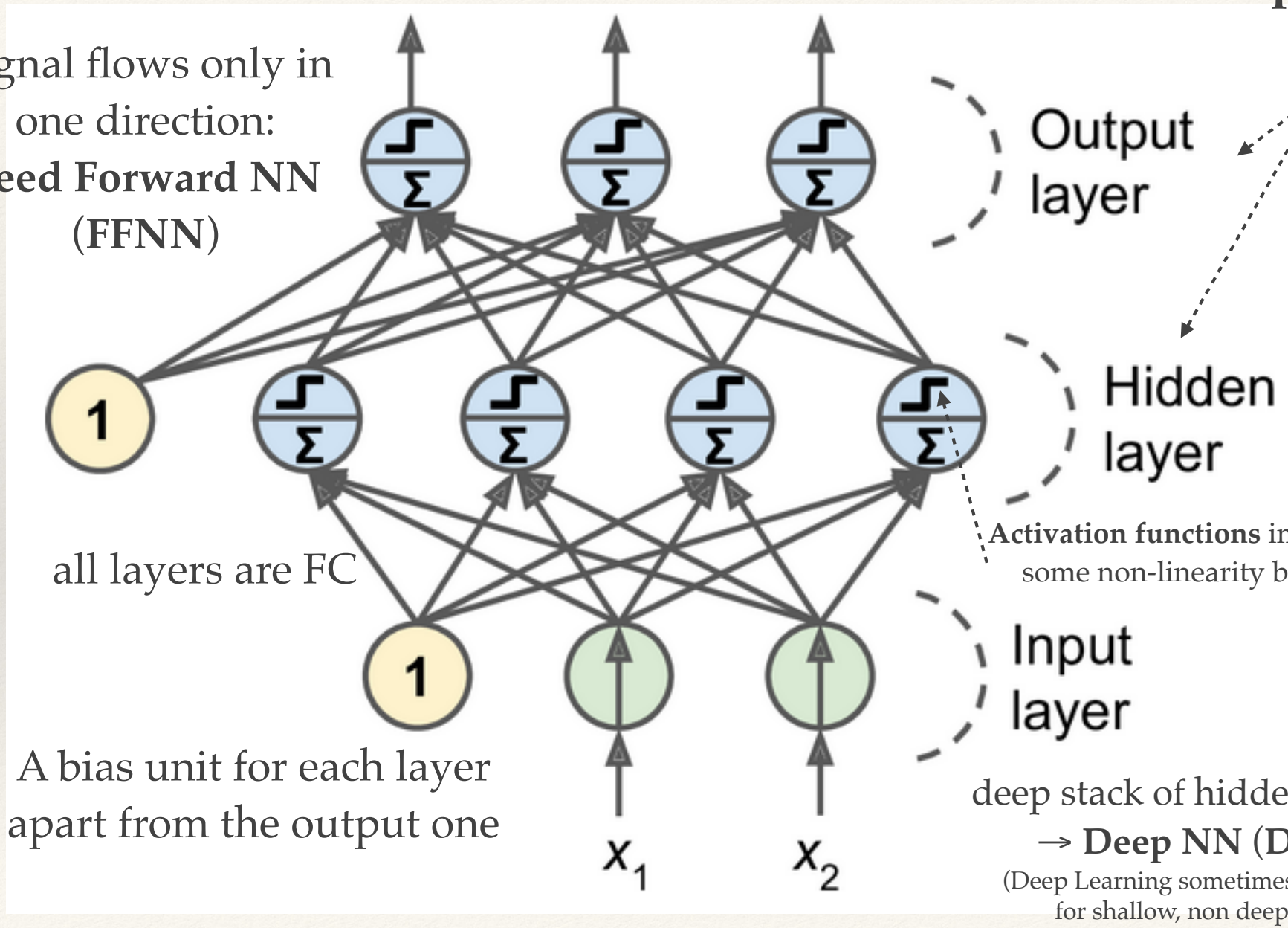
- the resulting ANN is called a **Multi-Layer Perceptron (MLP)**




Note that **you added an entire brand new layer** to the original Perceptron architecture

(see next)

# MLP - FCNN - FFNN

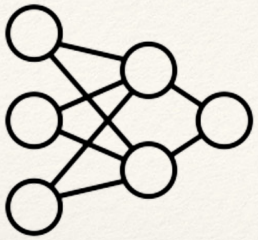


# Applied Machine Learning - Advanced

Prof. Daniele Bonacorsi

## Hands-on:
## Tensorflow playground

Data Science and Computation PhD + Master in Bioinformatics
**University of Bologna**

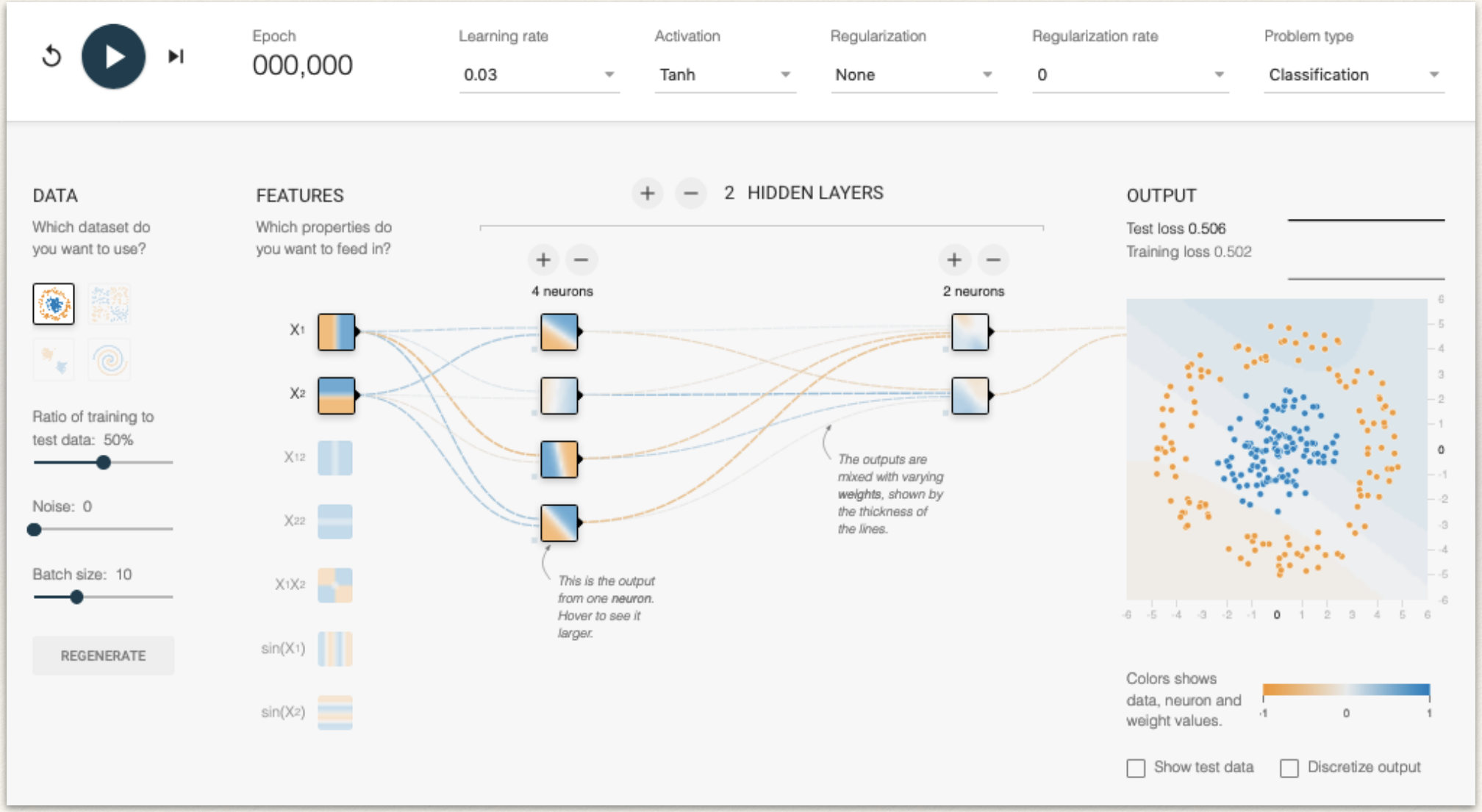


# Hands-on: Tensorflow playground

## **Getting familiar with NNs w/o code**

# Tensorflow Playground

https://playground.tensorflow.org/



# Tensorflow Playground

It is a handy NN built by the Google TensorFlow team

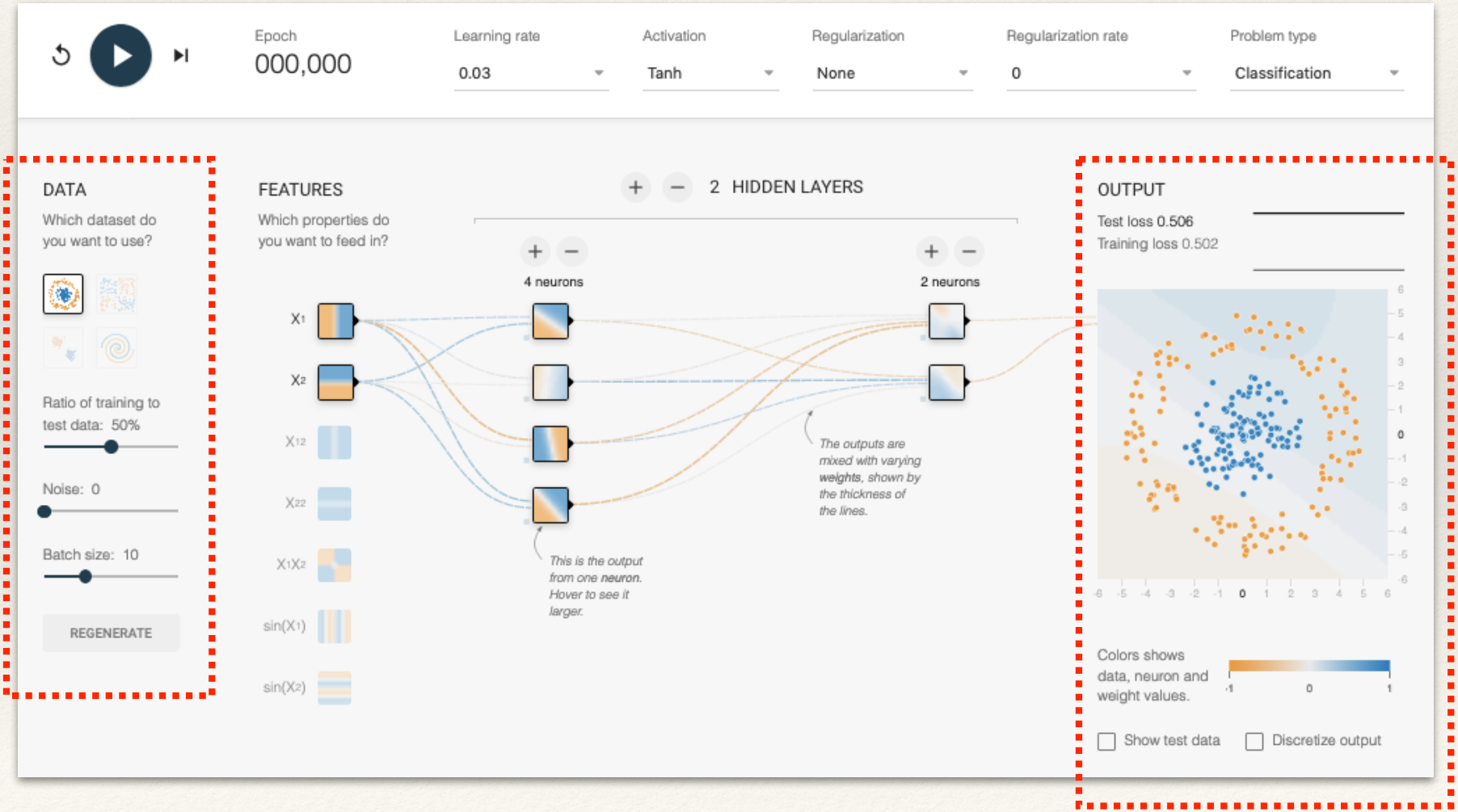
- you can try several binary classifiers in few clicks, tweak the model's architecture and its hyperparameters to gain some intuition on how NN actually work and what the role of their hyperparameters is

*My suggestion:*

- *Explore yourself (even offline) and ask questions if any*
- *In the following, I would give you some guided exercises/observations..*

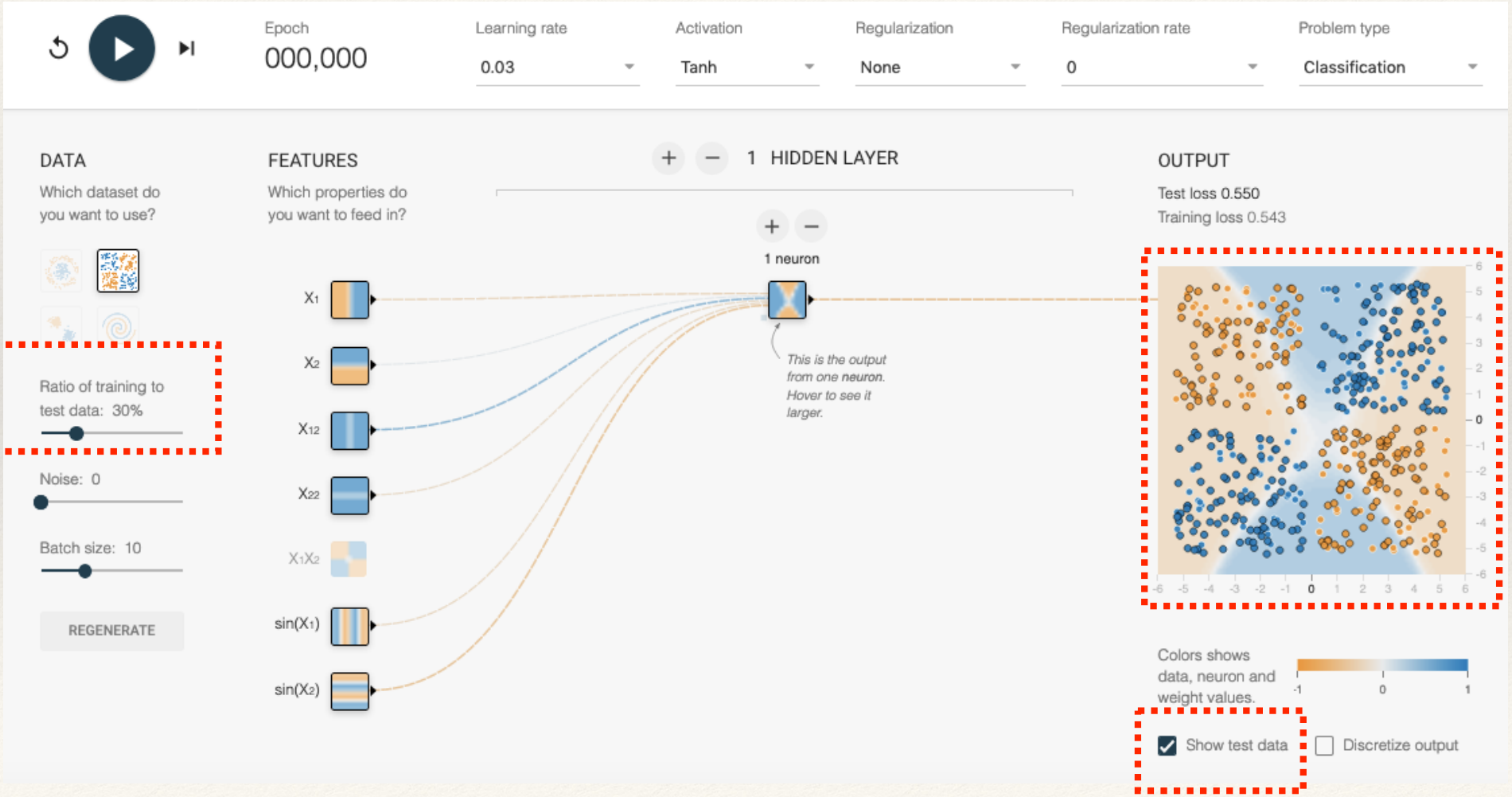
# Tensorflow Playground - Pre-exercise A

Select the **DATA** (dataset) you want to use and click multiple times on **REGENERATE**, then look at the right for the effect



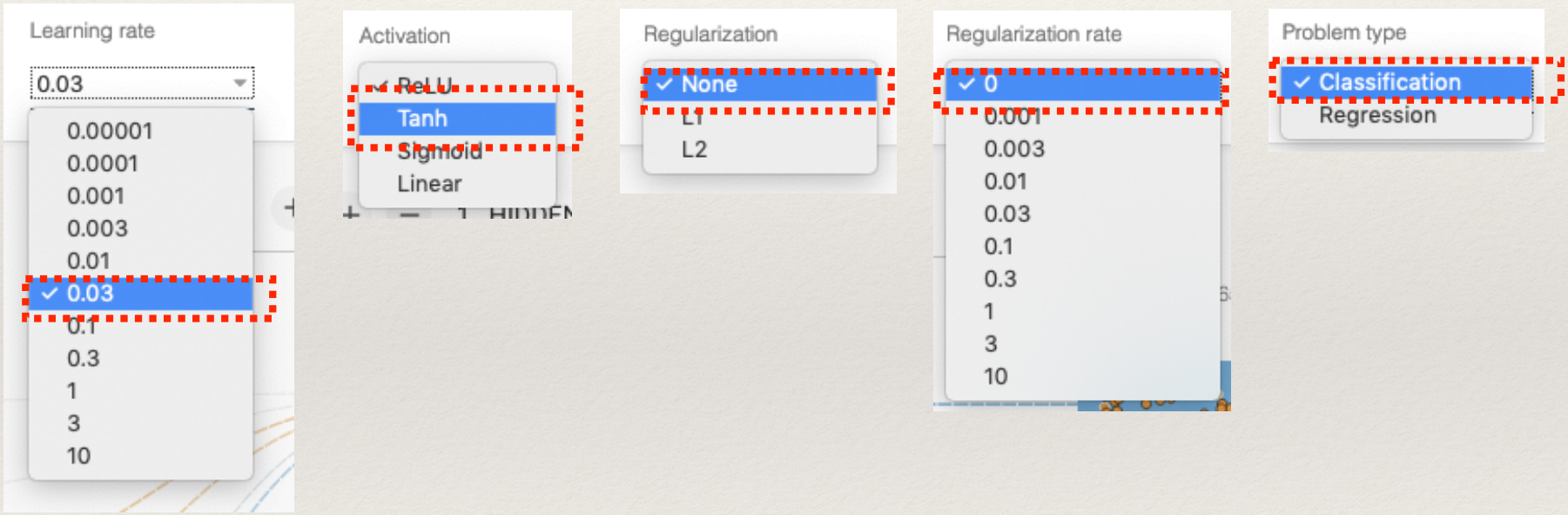
# Tensorflow Playground - Pre-exercise B

Change the **TRAINING/TEST DATA RATIO** after having selected the **SHOW TEST DATA** radio button, and check its effect visually



# Tensorflow Playground - Pre-exercise C

Check the default hyper-parameters, and the options you have beyond the defaults.



# Tensorflow Playground - Exercise 1

**Practice with the patterns learned by a NN.**

Try training the default NN by clicking the Run button (top left). Notice how it quickly finds a good solution for the classification task.

- The neurons in the first hidden layer have learned simple patterns…
- … while the neurons in the second hidden layer have learned to combine the simple patterns of the first hidden layer into more complex patterns
- In general, the more layers there are, the more complex the patterns can be

# Tensorflow Playground - Exercise 2

**Practice with activation functions.**

Try replacing e.g. the tanh activation function with a ReLU activation function, and train the network again.

- Notice that it finds a solution even faster, but this time the boundaries are linear. This is due to the shape of the ReLU function

# Tensorflow Playground - Exercise 3

**Practice with the risk of local minima.**

Modify the NN architecture to have just one hidden layer with 3 neurons.

- Train it multiple times (to reset the network weights, click the Reset button next to the Play button).
- Notice that the training time varies a lot, and sometimes it even gets stuck in a local minimum.

# Tensorflow Playground - Exercise 4

**Practice with what happens when NNs are too small.**

Remove 1 neuron to keep just 2.

- Notice that the NN is now incapable of finding a good solution, even if you try multiple times
- The model has too few parameters and systematically underfits the training set

# Tensorflow Playground - Exercise 5

**Practice with what happens when NNs are large enough.**

Set the number of neurons to 8, and train the NN several times.

- Notice that it is now consistently fast and never gets stuck
- This highlights an important finding in NN theory: large neural networks almost never get stuck in local minima, and even when they do these local optima are almost as good as the global optimum. However, they can still get stuck on long plateaus for a long time

# Tensorflow Playground - Exercise 6

**Practice with the risk of vanishing gradients in deep NNs.**

Select the spiral dataset (the bottom-right dataset under "DATA"), and change the NN architecture to have 4 hidden layers with 8 neurons each.

- Notice that training takes much longer and often gets stuck on plateaus for long periods of time.
- Also notice that the neurons in the highest layers (on the right) tend to evolve faster than the neurons in the lowest layers (on the left). This problem, called the "vanishing gradients" problem, can be alleviated with better weight initialization and other techniques, better optimizers (such as AdaGrad or Adam), or Batch Normalization..

# Tensorflow Playground - Exercise 7

**Practice more!**

My suggestion: take 1 hour or so to play around with other parameters and get a feel for what they do, to build as much as possible come intuitive understanding about NNs

- we did it in various ways, now also **do it with the support of a visualisation!**

# NNs - representation

Let's see how to represent a NN with some formalism.

- I.e. how to represent our hypothesis, first of all.

# Some jargon

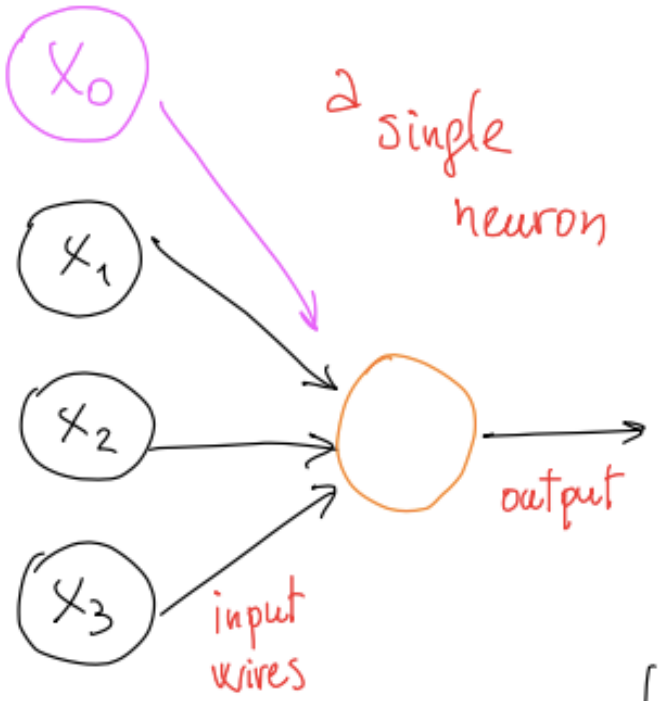
We need an extra node $x_0$ (**bias unit**, or **bias neuron**) =1

I will refer to artificial neutrons with a sigmoid or logistic **activation function** (the non-linear g(z) sigmoid)

I will refer to the $\theta$ parameters, from now on, as **weights** of the model

Let's review all this formally but visually.

# An artificial neuron as a logistic unit




neuron model as logistic unit

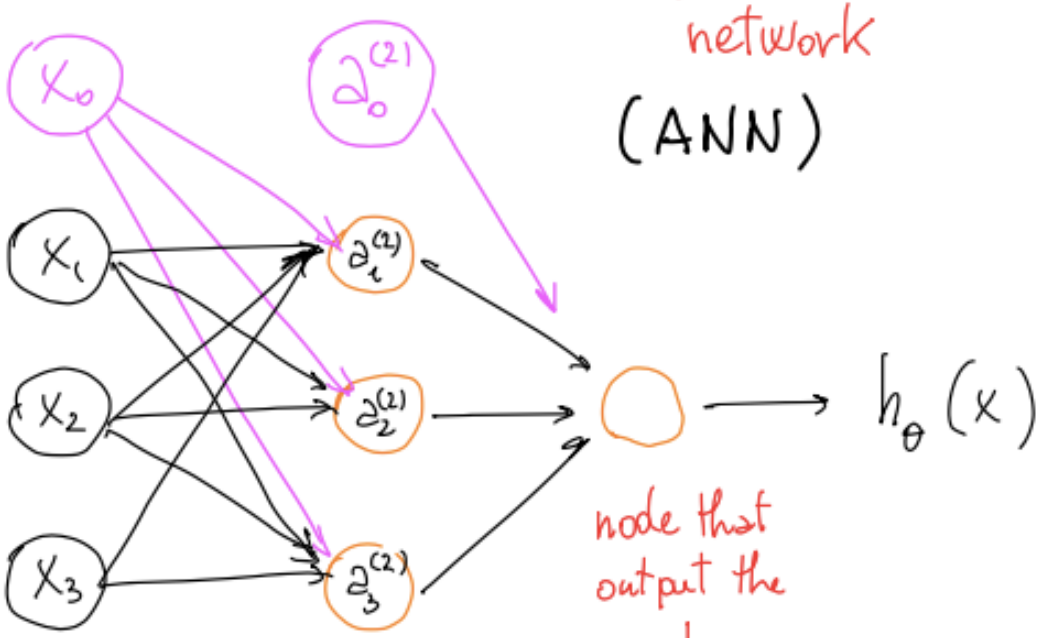
$$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$$

"ACTIVATION FUNCTION"

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \quad ; \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}$$

"parameters or WEIGHTS of a model"

# An Artificial NN (ANN)




a neural network (ANN)

a group of interconnected neurons

node that output the h

input units

3 neurons

LAYER 1 "INPUT LAYER" X

LAYER 2 "HIDDEN LAYER" not $X_1$ not Y

LAYER 3 "OUTPUT LAYER" Y

Let's explain the computation this ANN does

# ANN notation

$a_i^{(j)}$ = "ACTIVATION" of unit $i$ in layer $j$ (neuron)

value that's computed and pushed forward

($a_1^{(2)}$ is the activation of the 1st unit in layer 2)

$\Theta^{(j)}$ = matrix of weights controlling the function mapping from layer $j$ to layer $j+1$

# ANN notation

$a_i^{(j)}$ = "ACTIVATION" of unit (neuron) $i$ in layer $j$

value that's computed and pushed forward

($a_1^{(2)}$ is the activation of the 1st unit in layer 2)

$\Theta^{(j)}$ = matrix of weights controlling the function mapping from layer $j$ to layer $j+1$

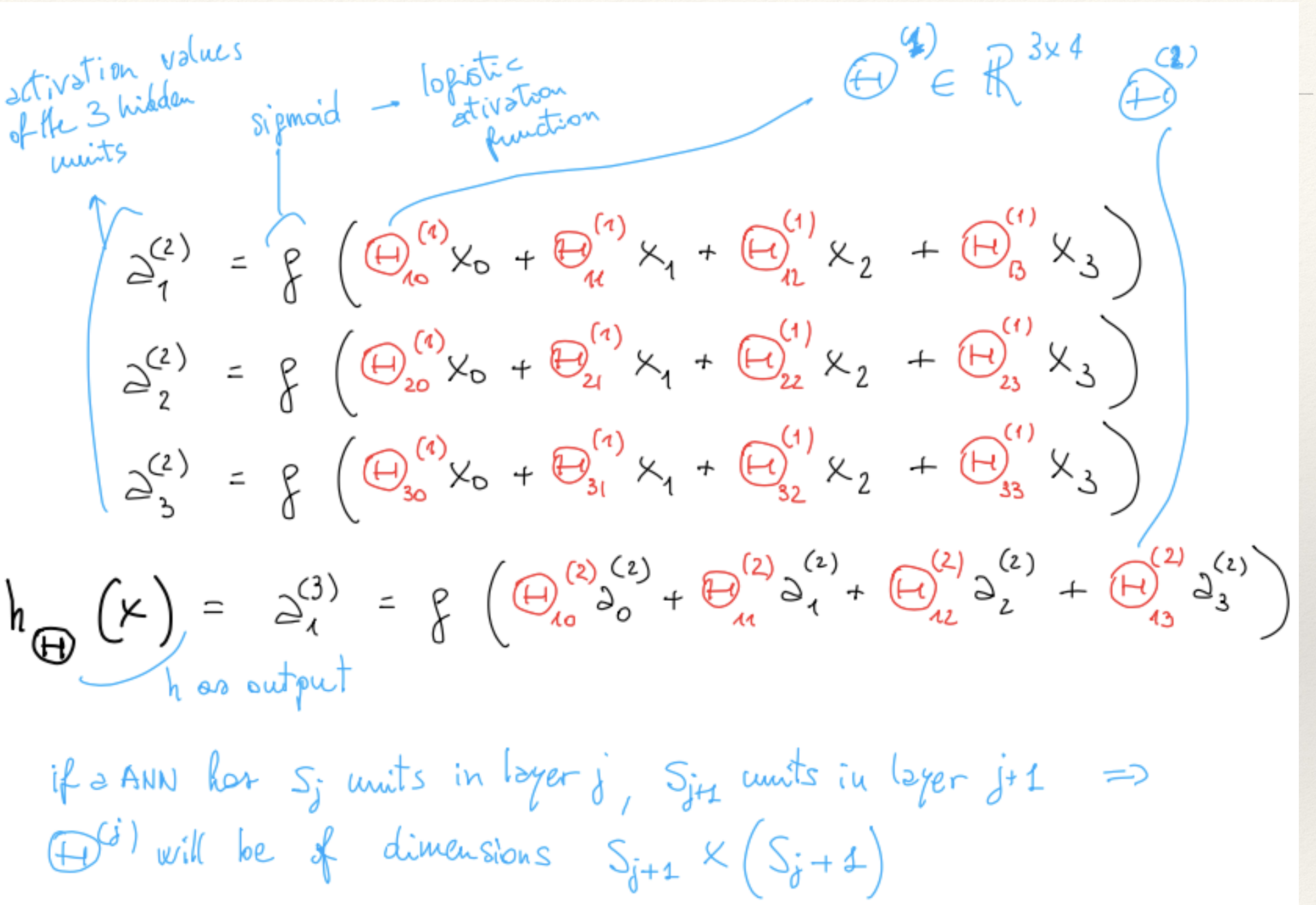
Here is the computation represented by the diagram:

$$a_1^{(2)} = g\left(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3\right)$$

$$a_2^{(2)} = g\left(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3\right)$$

$$a_3^{(2)} = g\left(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3\right)$$

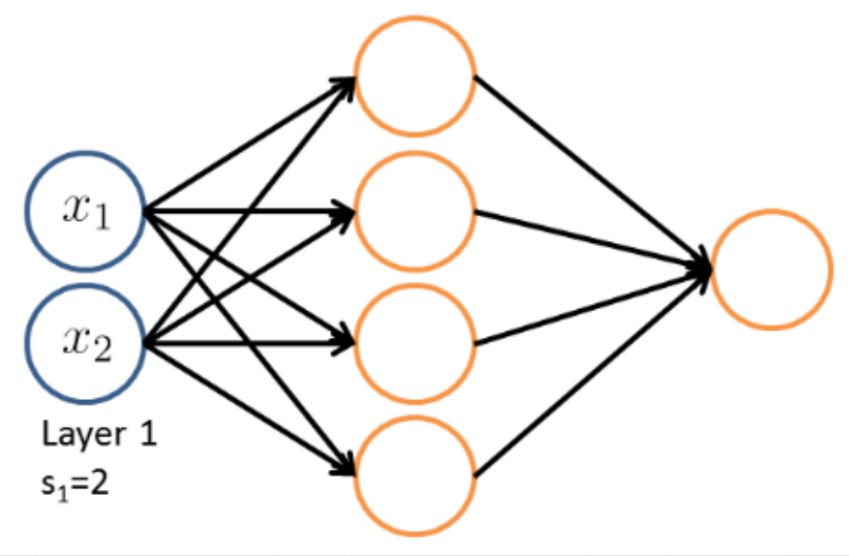
$$h_\theta(x) = a_1^{(3)} = g\left(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)}\right)$$




*( e.g. $j=1 \rightarrow s_j=s_1=3 \rightarrow s_{j+1}=s_2=1 \rightarrow dim = 1x(3+1) = 1x4$ )*

# C.Y.U.

Quiz

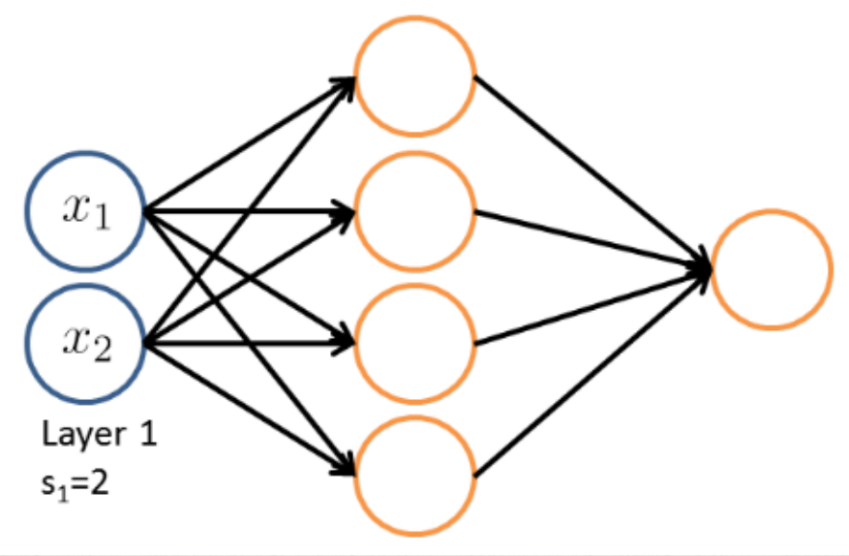



Consider an ANN like the one in the figure. What is the dimension of $\Theta^{(1)}$?

1. 2 x 4
2. 4 x 2
3. 3 x 4
4. 4 x 3

# C.Y.U.

Quiz




Consider an ANN like the one in the figure. What is the dimension of $\Theta^{(1)}$?

1. 2 x 4
2. 4 x 2
3. 3 x 4
4. **4 x 3**

If layer-1 has 2 input nodes and layer-2 has 4 activation nodes, it is

$s_j$=2 and $s_{j+1}$=4 ,

and dimension of $\Theta^{(1)}$ is hence going to be

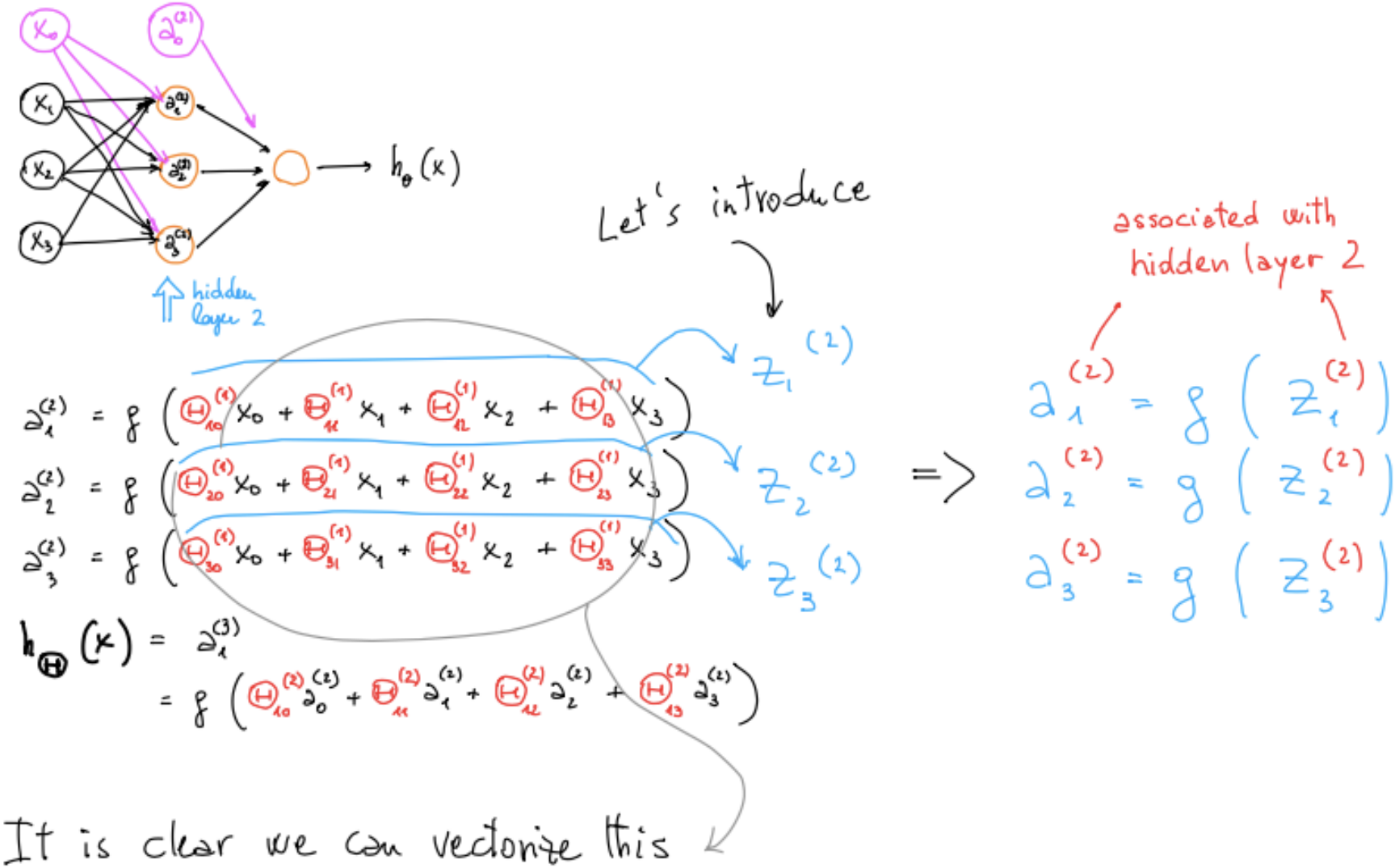
**$(s_{j+1})x(s_j+1) = 4x3$**

We gave a mathematical definition of how to represent (i.e. how to compute the hypotheses used by) a NN.

We need to see:

1. how to actually carry out that computation efficiently (a vectorised implementation).

2. intuition about why these NN representations might help us to learn complex nonlinear hypotheses.


hidden layer 2
Let's introduce
associated with hidden layer 2
It is clear we can vectorize this

$$X = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad x_0 \Rightarrow 1$$

features vector

$$z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(3)} \end{bmatrix} \in \mathbb{R}^3$$

$$\Theta^{(1)} = \begin{bmatrix} \Theta_{10}^{(1)} & \Theta_{11}^{(1)} & \Theta_{12}^{(1)} & \Theta_{13}^{(1)} \\ \Theta_{20}^{(1)} & \Theta_{21}^{(1)} & \Theta_{22}^{(1)} & \Theta_{23}^{(1)} \\ \Theta_{30}^{(1)} & \Theta_{31}^{(1)} & \Theta_{32}^{(1)} & \Theta_{33}^{(1)} \end{bmatrix}$$

$$z^{(2)} = \Theta^{(1)} X$$

$$a^{(2)} = g(z^{(2)}) \quad a^{(2)} \in \mathbb{R}^3$$

the sigmoid activation applies element-wise to $z^{(2)}$

We can consider the input $X$ as the activations of the $1^{st}$ layer, i.e.

$$a^{(1)} = X \Rightarrow z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \Rightarrow 1 \qquad z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(3)} \end{bmatrix} \in \mathbb{R}^3 \qquad \Theta^{(1)} = \begin{bmatrix} \Theta_{10}^{(1)} & \Theta_{11}^{(1)} & \Theta_{12}^{(1)} & \Theta_{13}^{(1)} \\ \Theta_{20}^{(1)} & \Theta_{21}^{(1)} & \Theta_{22}^{(1)} & \Theta_{23}^{(1)} \\ \Theta_{30}^{(1)} & \Theta_{31}^{(1)} & \Theta_{32}^{(1)} & \Theta_{33}^{(1)} \end{bmatrix}$$

features vector

Last bit: the bias unit.

$$a_0^{(2)} = 1 \Rightarrow a^{(2)} = \begin{bmatrix} a_0^{(2)} \\ a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \end{bmatrix} = g\left(\underbrace{z^{(2)}}_{\Theta^{(1)} a^{(1)}}\right)$$

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad x_0 = 1 \qquad z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(3)} \end{bmatrix} \in \mathbb{R}^3 \qquad \Theta^{(1)} = \begin{bmatrix} \Theta_{10}^{(1)} & \Theta_{11}^{(1)} & \Theta_{12}^{(1)} & \Theta_{13}^{(1)} \\ \Theta_{20}^{(1)} & \Theta_{21}^{(1)} & \Theta_{22}^{(1)} & \Theta_{23}^{(1)} \\ \Theta_{30}^{(1)} & \Theta_{31}^{(1)} & \Theta_{32}^{(1)} & \Theta_{33}^{(1)} \end{bmatrix}$$

features vector

Last bit: the bias unit.

$$a_0^{(2)} = 1 \quad \Rightarrow \quad a^{(2)} = \begin{bmatrix} a_0^{(2)} \\ a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \end{bmatrix} = g\left(z^{(2)}\right) = g\left(\Theta^{(1)} a^{(1)}\right)$$

Finally:

$$z^{(3)} = \Theta^{(2)} a^{(2)}$$

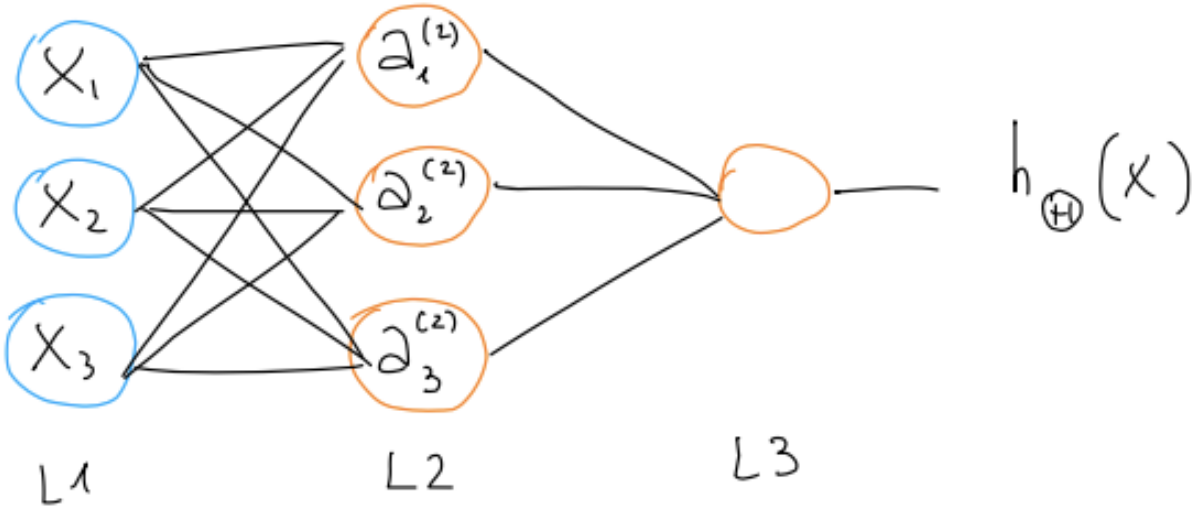
$$h_\Theta(x) = a^{(3)} = g\left(z^{(3)}\right)$$

subscript 1 omitted

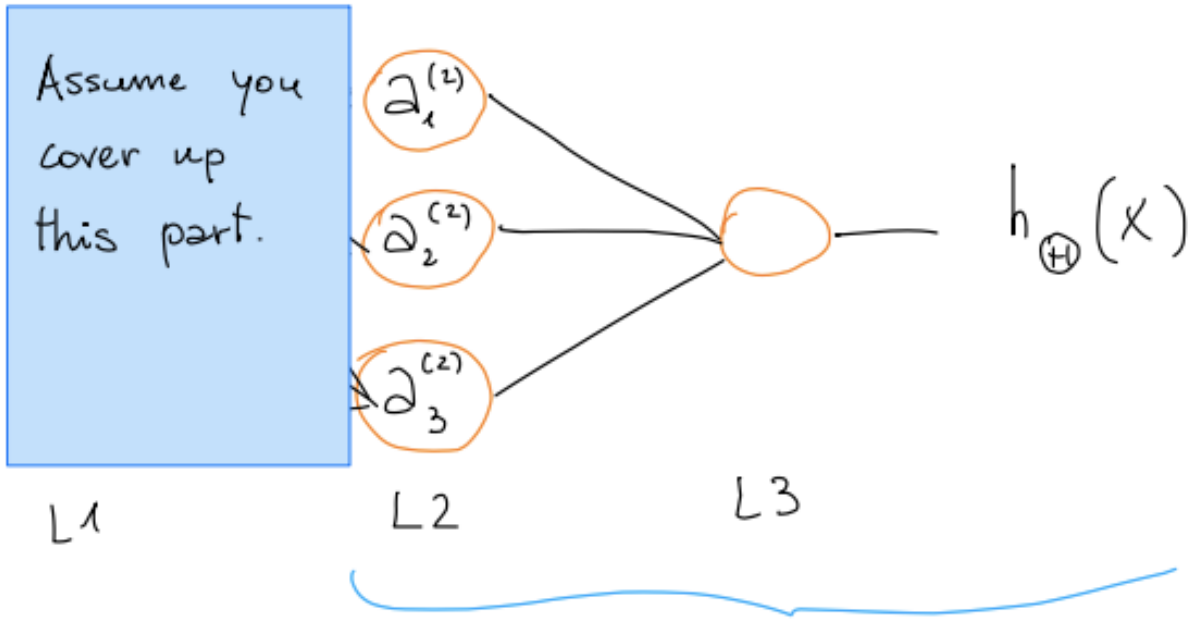
This process of computing $h_\Theta(x)$ is called **FORWARD PROPAGATION** (a vectorized implementation)

start with activation of input units and fwd-propage to the hidden layers and compute the activations of ', then down to output layers.

Fwd - propagation can explain how ANNs work ?



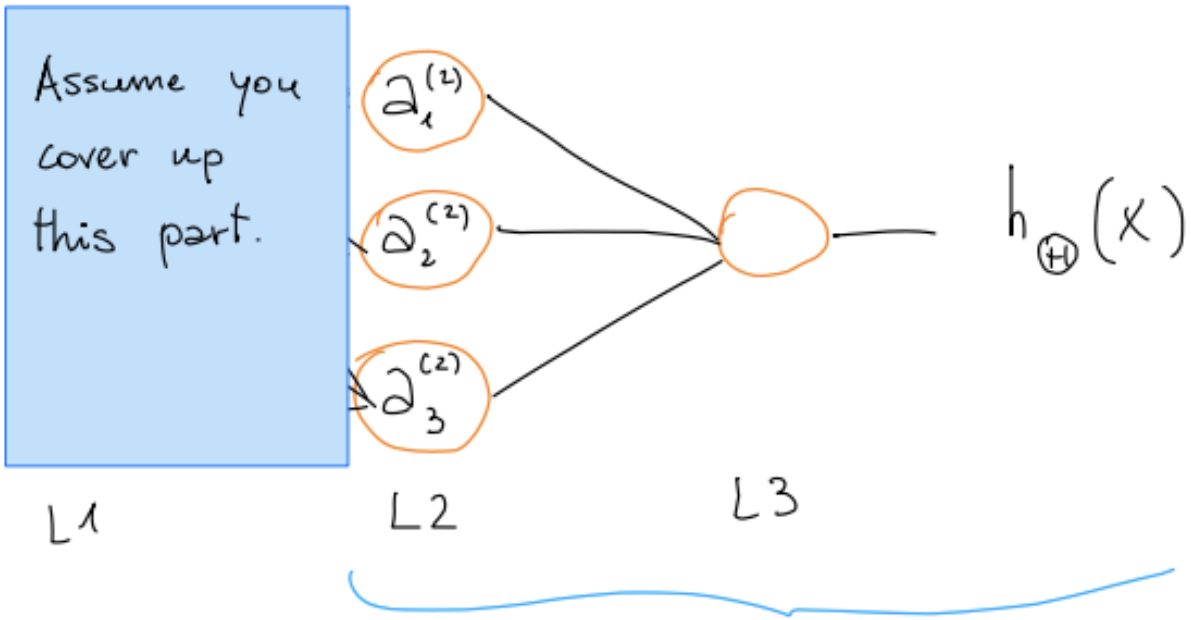
Fwd - propagation can explain how ANNs work ?




This look like LOGISTIC REGRESSION

$$h_\Theta(x) = g\left(\Theta_0 a_0 + \Theta_1 a_1 + \Theta_2 a_2 + \Theta_3 a_3\right)$$

## Fwd - propagation can explain how ANNs work?



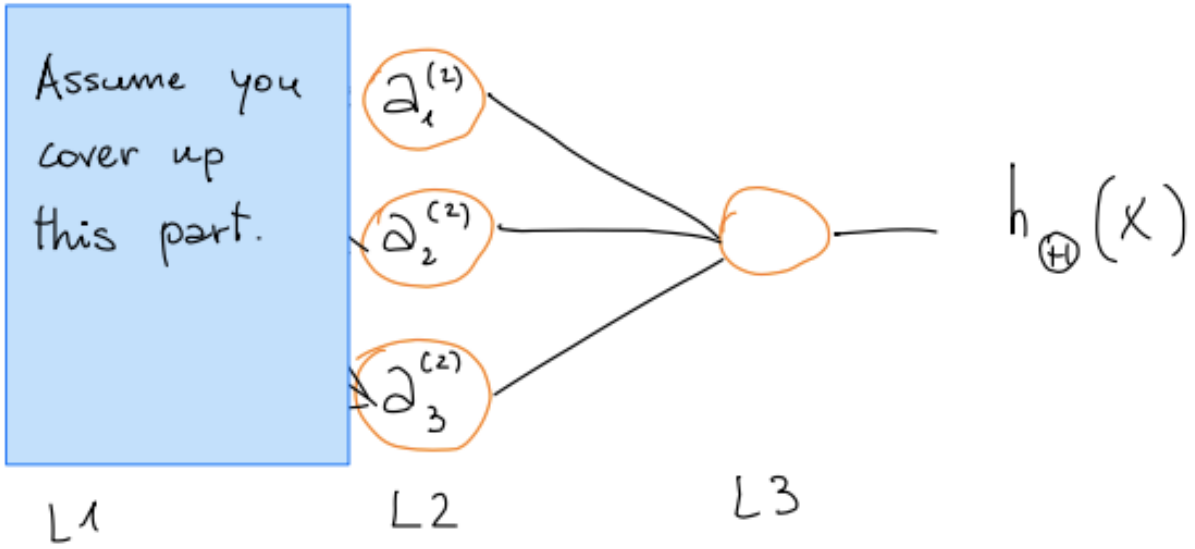

This look like LOGISTIC REGRESSION

to be notation-consistent

$$h_\Theta(x) = g\left(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)}\right)$$

I have only 1 output unit

# Fwd - propagation can explain how ANNs work ?

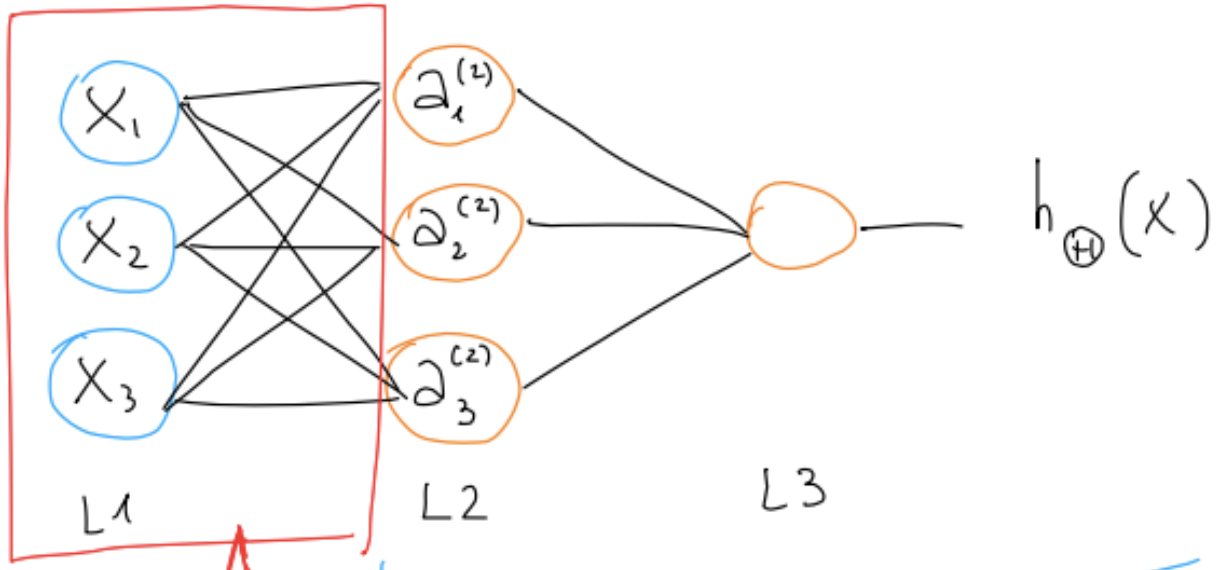



This _ is _ LOGISTIC REGRESSION

where the features fed into logistic regression are the $a_1^{(2)}, a_2^{(2)}, a_3^{(2)}$ computed by the hidden layers.

$x_1, x_2, x_3 \rightarrow a_1^{(2)}, a_2^{(2)}, a_3^{(2)}$

# Fwd - propagation can explain how ANNs work ?



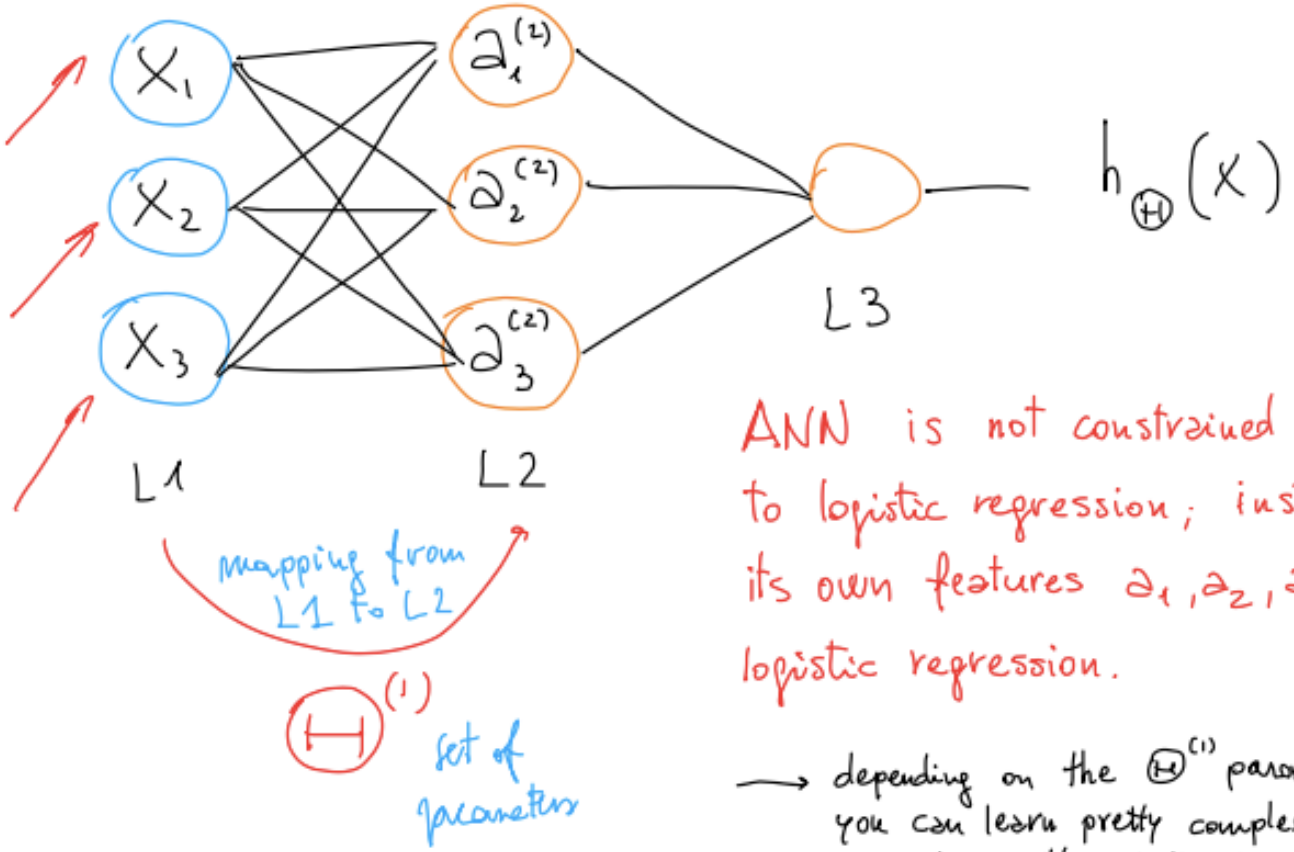

This _ is _ LOGISTIC REGRESSION

where the features fed into logistic regression are the $a_1^{(2)}, a_2^{(2)}, a_3^{(2)}$ computed by the hidden layers.

$x_1, x_2, x_3 \rightarrow a_1^{(2)}, a_2^{(2)}, a_3^{(2)}$

are learnt themselves as function of the input

# Fwd - propagation can explain how ANNs work ?



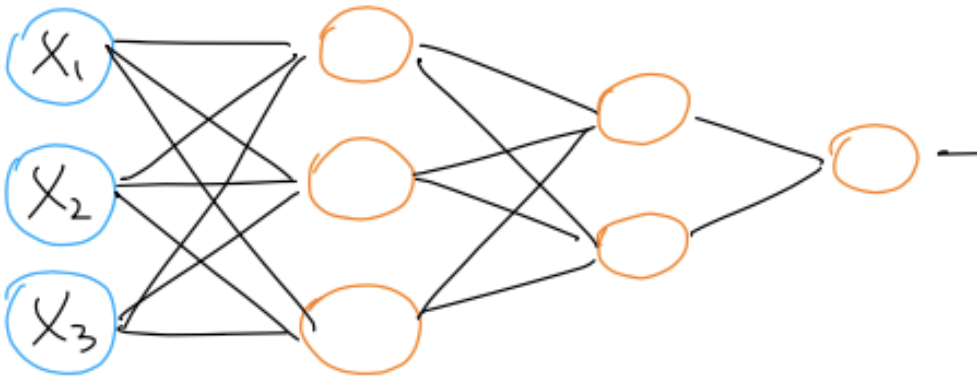

ANN is not constrained to feed $x_1, x_2, x_3$ to logistic regression; instead, it learns its own features $a_1, a_2, a_3$ to feed into logistic regression.

→ depending on the $\Theta^{(1)}$ parameters, you can learn pretty complex features (new features) => end up with a better h w.r.t. what you would have if constrained to $x_1, x_2, x_3$ or to choose the polynomials...

FLEXIBLE !

Other ANN "architectures"

how neurons are connected




very interesting non-linear h

you can compute more complex function of input L1

take L2's features and compute even more complex features

all complexity and flexibility you want

# In summary

**Feed forward propagation in a NN** = start from the activations of the input layer and forward propagate that to the first hidden layer, then the second hidden layer, and then finally the output layer. All layers are computing more complex features than the early layers. Note that in the last step we do exactly the same thing as we did in logistic regression. We can vectorise all computations.

- **FLEXIBILITY** is the keyword!

This feed forward propagation view also helps us to understand what NNs might be doing and why they might help us to learn interesting nonlinear hypotheses.