Deep Learning

Goals

- Demystify Deep Learning
- Be autonomous & prototype easily
- Work on different data types: vector, image, text, time-series, ...

Deep Learning is about practice, *practice*, ***practice***!

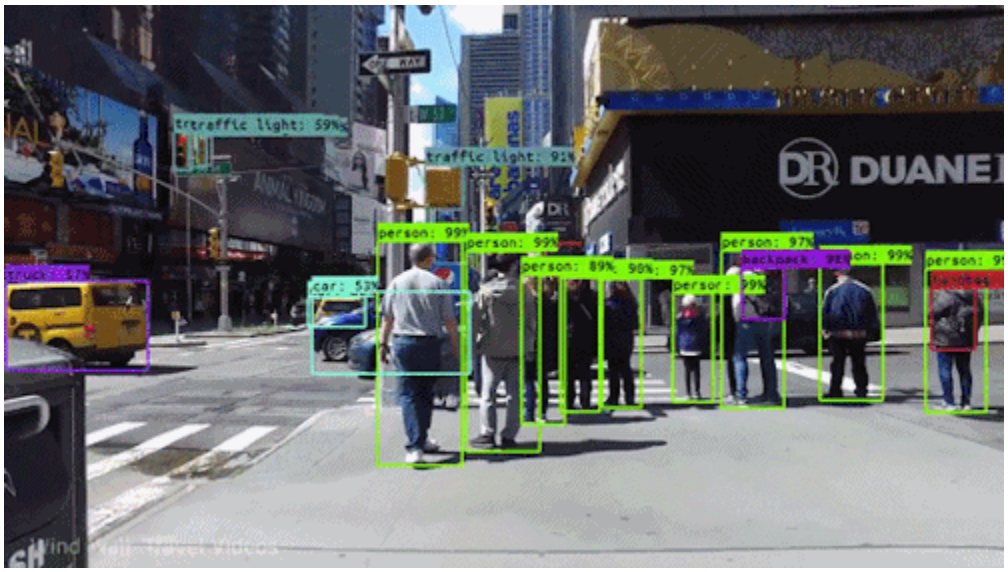
Plan of the module

- Lecture 1 : Fundamentals of Deep Learning
- Lecture 2 : Optimizers, Loss & Fitting
- Lecture 3 : Convolutional Neural Networks
- Lecture 4 : RNNs & NLP
- Lecture 5 : Transformers

Fundamentals of Deep Learning

1. Deep Learning is Everywhere

Image detection & Image recognition

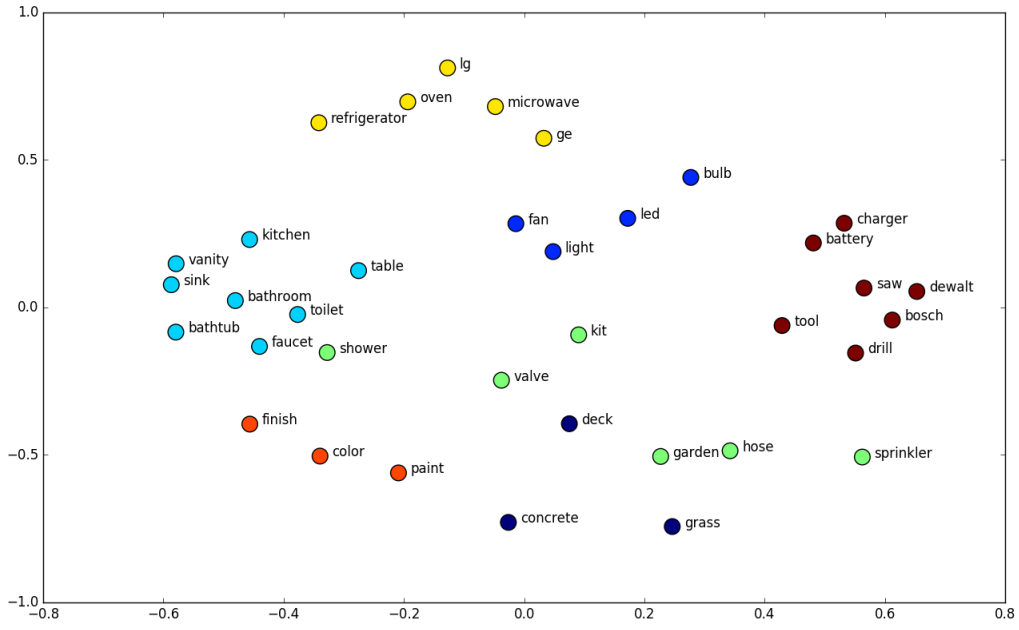


Autonomous cars, violence detection on public transportation, unlocking phones, person tracking, gesture detection, ...

Pose estimation

Source: [BioMechanic - Pose Detection with OpenPose]

Natural Language Processing (In general)

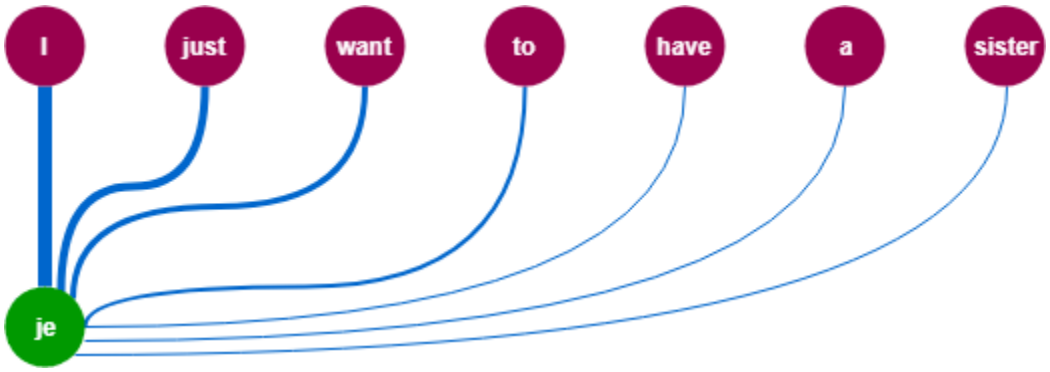


Retrieval of information from a text input

Sentiment Analysis

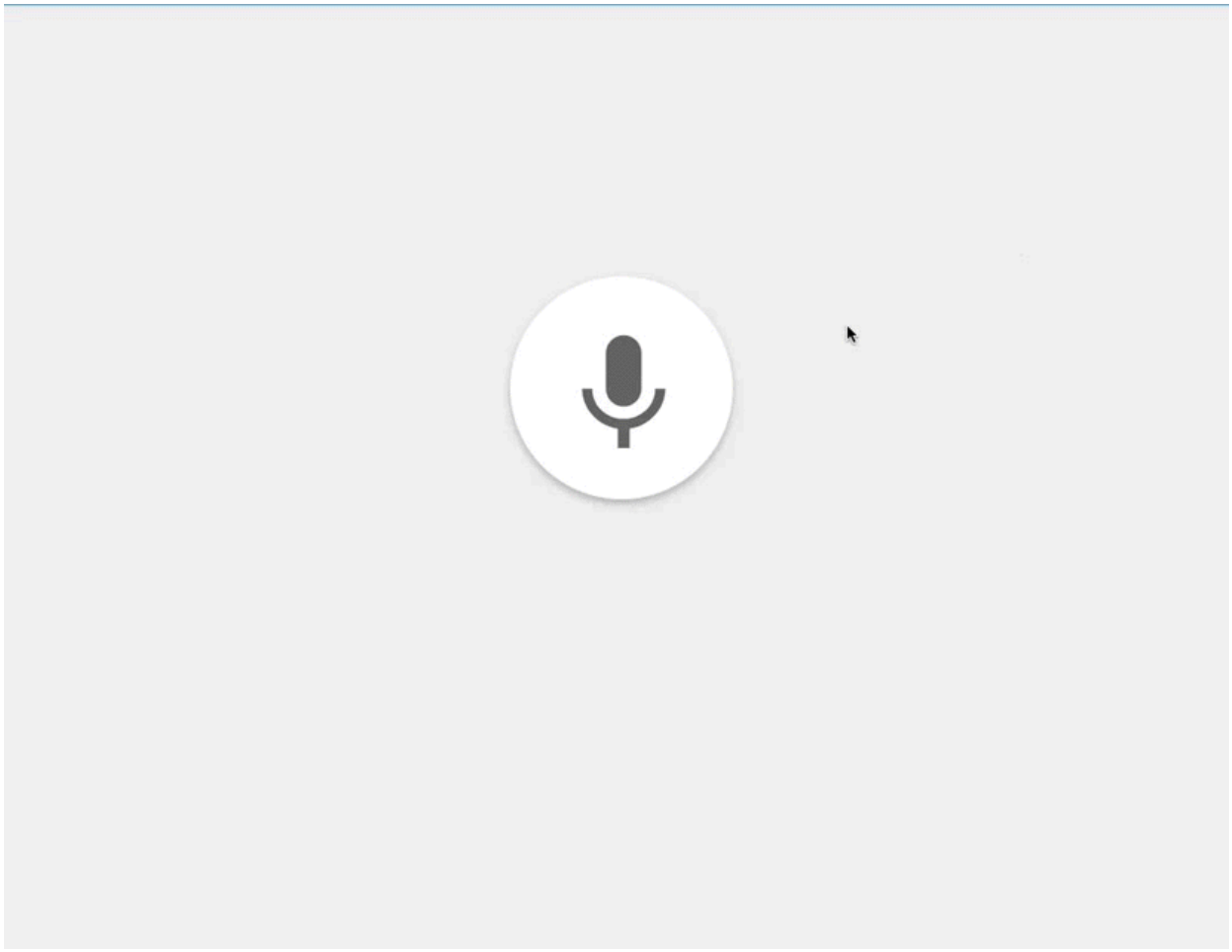
Sentiment analysis is part of NLP

Translation

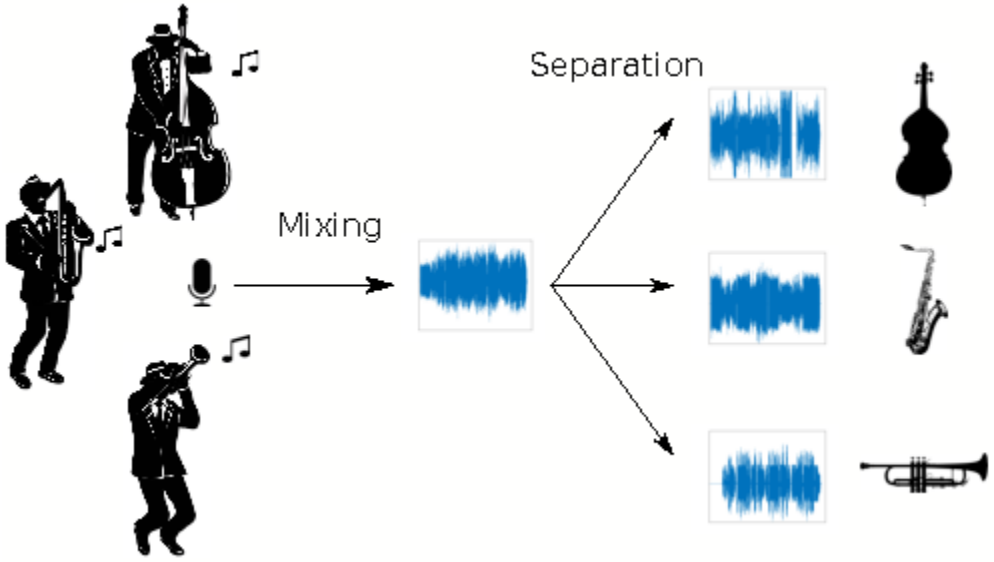


Translation is part of NLP

Speech to text / Speech Recognition



Detection of multiple sources in sounds / voices / signals



Captioning (Picture to text)

Generation of fake pictures, videos, sounds (1/2)

Click on the person who is real.

www.whichfaceisreal.com

Generation of fake pictures, videos, sounds (2/2)

Called DeepFakes
And many many many more!

⚠️ But don't be lured, many are coooool but not always useful (or leverageable in a business context).

So why is Deep Learning taking the lead on so many subjects?

2. Basic architecture

Let's get some data!

```
# Target y (classification task 0/1, e.g. cat/dog)
y = 1

# Input X = one single observation, 4 features (x1, x2, x3, x4)
# (e.g. eyes color, ears_lenghts, ...)
X = [1., -3.1, -7.2, 2.1]
```

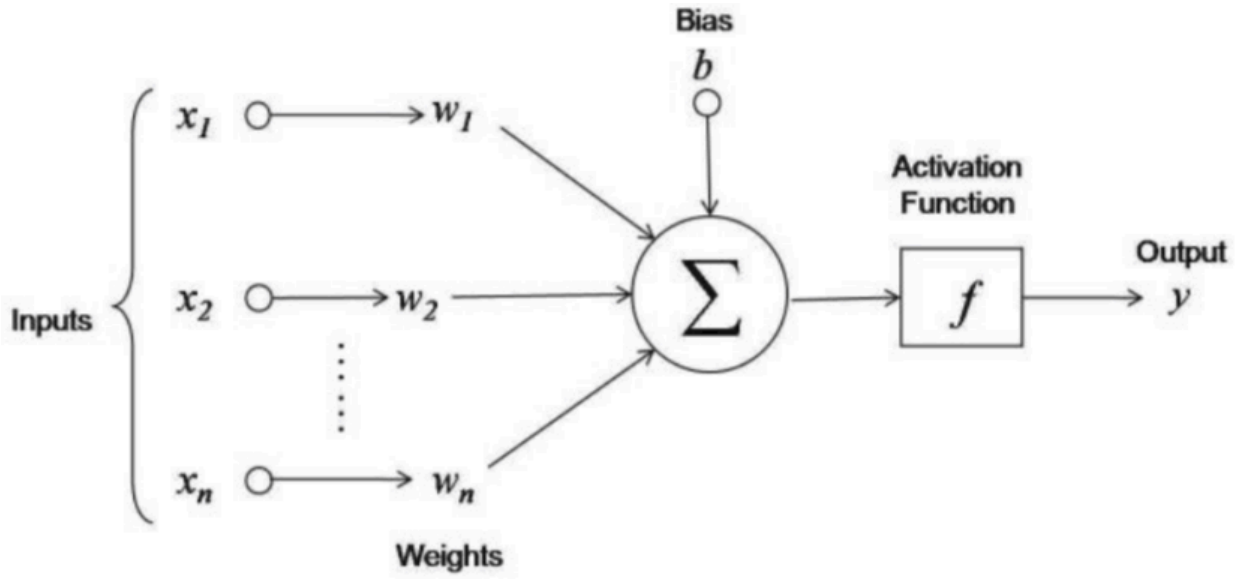
Imagine you have a linear regression with some weights

```
def linreg_1(X):
    return -3 + 2.1*X[0] - 1.2*X[1] + 0.3*X[2] + 1.3*X[3]

out_1 = linreg_1(X)
```

And you transform it's output

```
def activation(x):
    if x > 0:
        return x
    else:
        return 0

out_1 = activation(out_1)
```

Let's call this operation {linear regression, activation} a *NEURON*



Given an input

$X = (x_1, x_2, \ldots, x_n)$

)

, a neuron is the concatenation of:

- A linear combination of the input with the weights
- w
- k
- plus a bias
- b
- , that outputs
- ∑
- n
- k
- =
- 1
- w
- k
- x
- k
- +
- b
-
- A non-linear modification
- f
- of that sum

Therefore, the output of a neuron is

o

u

t

p

u

t

=

f

(

∑

n

k

=

1

w

k

x

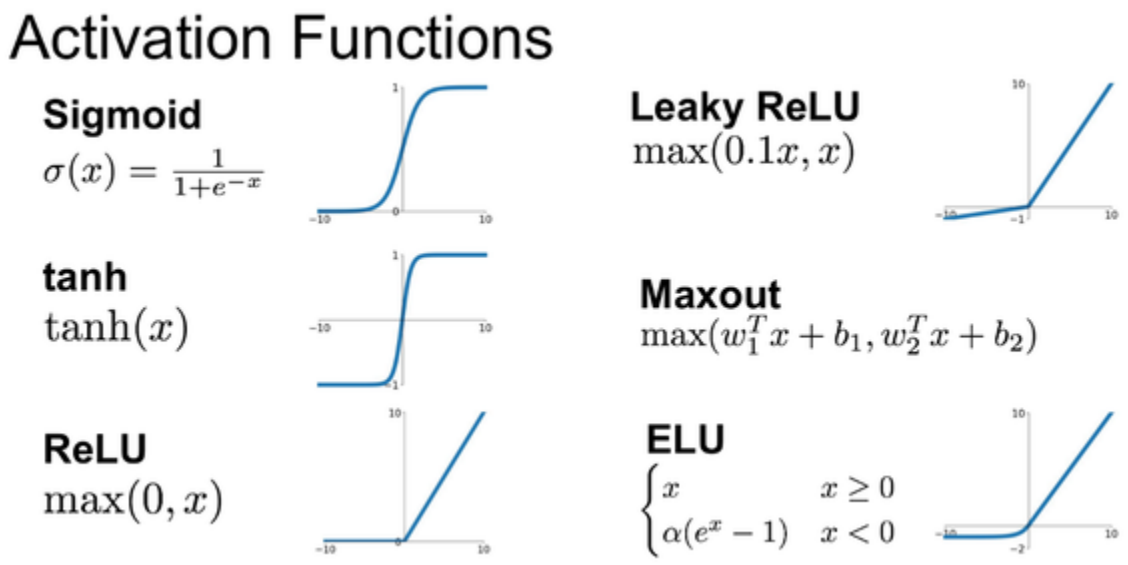
k

+

b

)

Activation Functions (well known examples)



❓ Which of these activations did we just code?
Imagine now that you produce another output by:

- applying *another* linear regression to the same input X
- followed by the *same* activation function

```
def linreg_2(X):
    return -5 - 0.1*X[0] + 1.2*X[1] + 4.9*X[2] - 3.1*X[3]

out_2 = activation(linreg_2(X))
```

And a third one

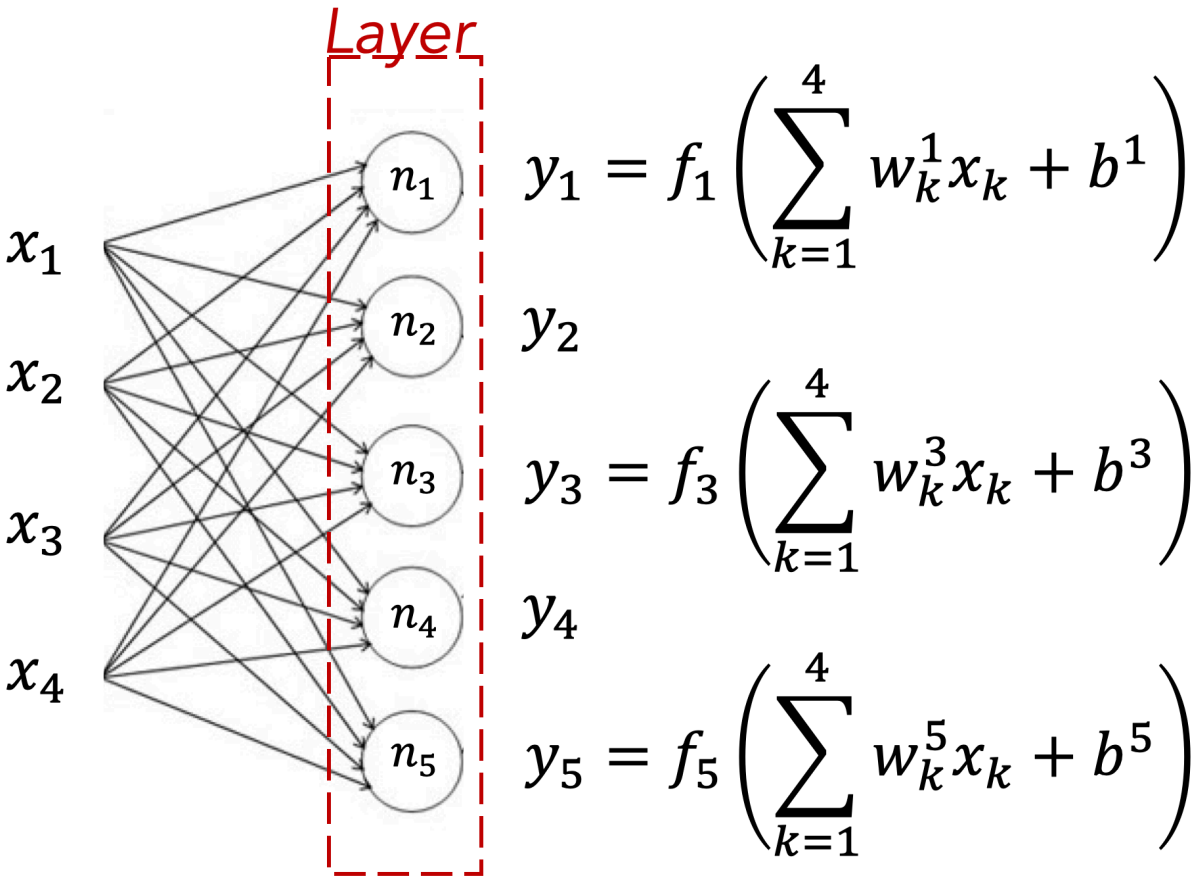
```
def linreg_3(X):
    return -8 + 0.4*X[0] + 2.6*X[1] +- 2.5*X[2] + 3.8*X[3]

out_3 = activation(linreg_3(X))
```

We just wrote a *LAYER* of neurons

An input

$$x = (x_1, \dots, x_n)$$

does not pass through a single neuron but through many different neurons.



❗ Remark ❗ Each neuron can have a different activation function (

f

1

≠

f

2

.

.

.

≠

f

5

) but in practice, each layer only uses one type.

What if we use the 3 outputs of this layer as input of another layer, again?

```
def linreg_next_layer(X):
    return 5.1 + 1.1*X[0] - 4.1*X[1] - 0.7*X[2]

def activation_next_layer(x):
    # sigmoid activation for clasification task!
    return 1. / (1 + np.exp(-x))

def neural_net_predictor(X):

    out_1 = activation(linreg_1(X))
    out_2 = activation(linreg_2(X))
    out_3 = activation(linreg_3(X))

    outs = [out_1, out_2, out_3]

    y_pred = activation_next_layer(linreg_next_layer(outs))

    return y_pred
```

```
# Final prediction
y_pred = neural_net_predictor(X)

print(f' Probability of being a dog: {y_pred}')
```

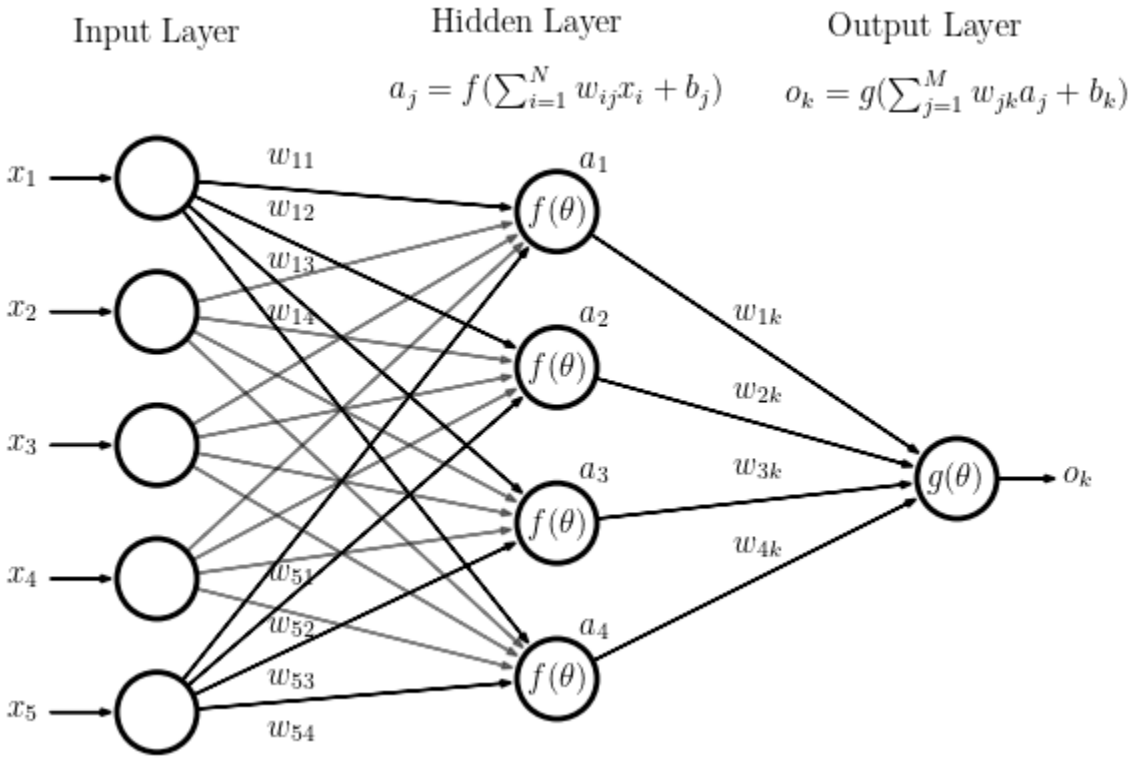
```
Probability of being a dog: 0.832716044461517
```

🎉 **Congrats! You just built your first (artificial) neural network.**

So, what is a Neural Network?

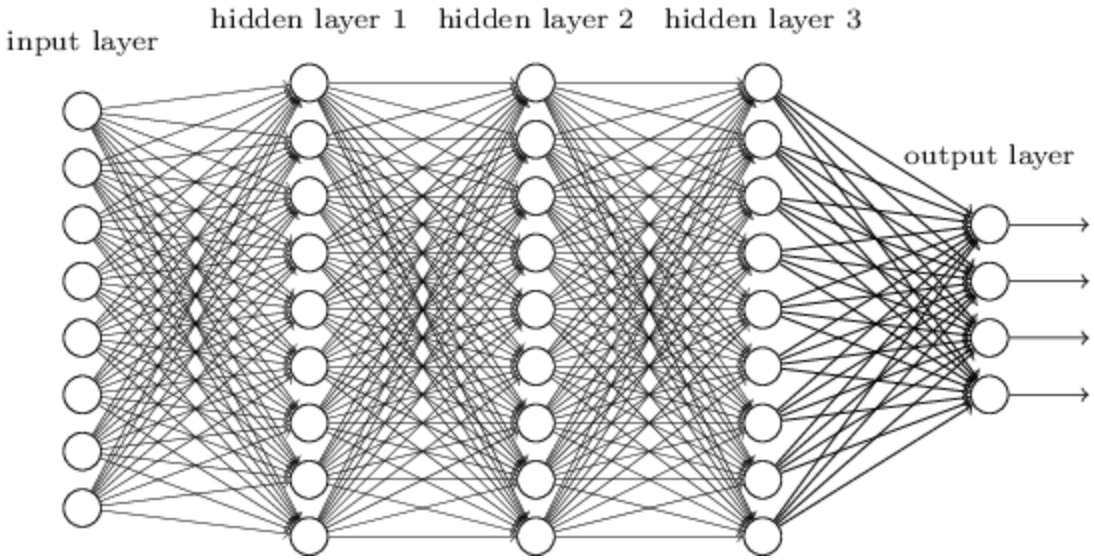
- nothing more than a fancy function
- f
- θ
- that computes
- ^
- y
- =
- f
- θ
- (

- x
-)
-
- where
- θ
- are the **weights** of all the linear regressions that take place within the neurons



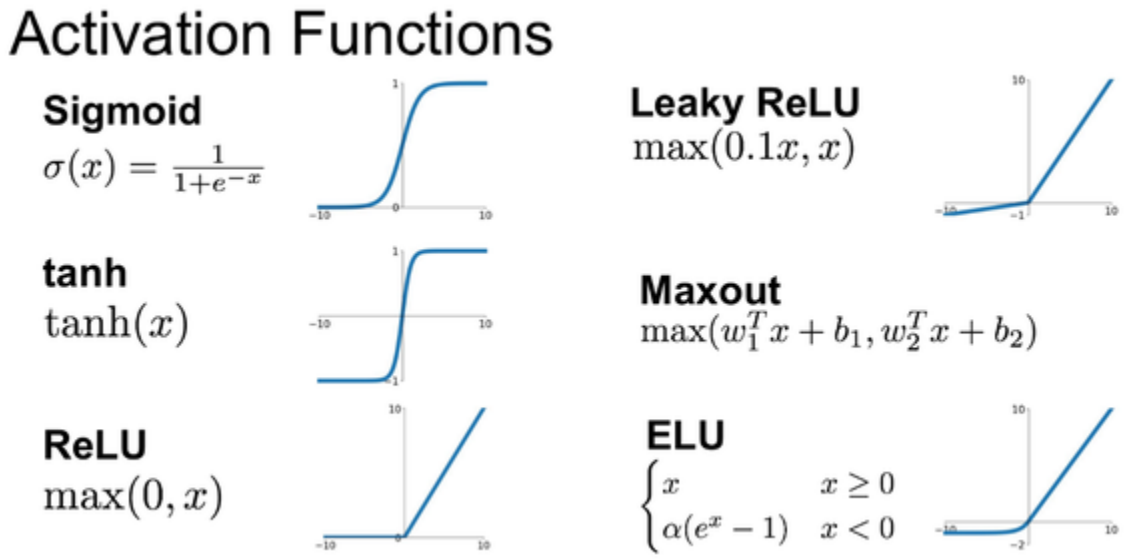
All these neurons and layers make up the **architecture** of the neural network.
❓ **Question:** How many weights are trained in layer 1? In layer 2?

***Deep Learning* simply refers to Neural Networks (that have many layers)**



Yes. Deep Learning is nothing more than that. Amazed? Disappointed? Intrigued?
❓ What's the dimension of the prediction?

🤔 **Why using activation function at all?**



- To introduce **non-linarities**!
- Without them, our Neural Network would be a simple linear model!

A

(

a

1

x

1

+

a

2

x

2

)

+

B

(

b

1

x

1

+

b

2

x

2

)

=

(

A

a

1

+

B

b

1

)

x

1

+

(

A

a

2

+

B

b

2

)

x

2

💻 Let's visualize a Neural Network on Playground

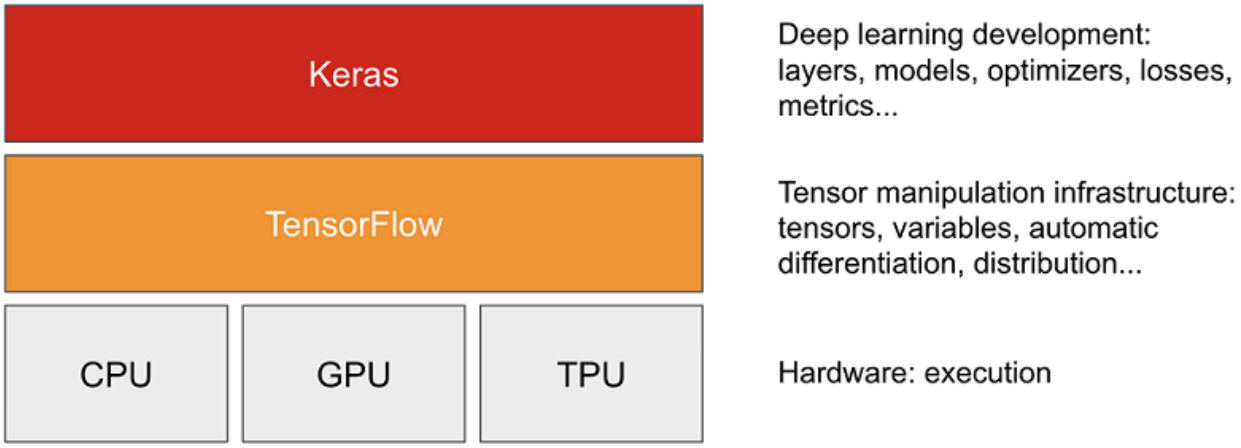
- Look familiar? Hit the play button ▶️ and see what happens!
- Can you fit this non-linear problem with only **linear activations** ? 🤔
- Try to fit the other datasets, keeping only one layer
- Try a regression task!

3. My first Neural Network in Keras

3.1 Keras

TensorFlow is an open-source machine learning framework developed by Google for building and deploying machine learning and deep learning models.

Keras is an open-source deep learning API that runs on top of TensorFlow, providing a user-friendly interface for building and training neural networks.



Documentation

📚 tensorflow.keras official docs

📚 keras official docs

🖥️ **Installation:**

```
pip install tensorflow
```

This will also install a compatible version of Keras

In 🐍 Python to code:

```
from tensorflow.keras import *
```

```
from keras import *
```

- Functionally both are the same
- The second option has better documentation pop-up support in VS Code

Defining & *Training* a Neural Network in three steps:
1. Define the architecture
f

of your model that, given an input X,it outputs $y_{pred} = f_{\theta_0}(X)$ (Here, the weights θ_0 are random)

```
from tensorflow.keras import Sequential
model = Sequential()
model.add(...)
```

2. Define the methods to estimate the best θ possible, which means having y_{pred} close to the real output y

```
model.compile(...)
```

3. Fitting on on data: Going from θ_0 => θ based on the data X and y

```
model.fit(X, y, ...)
```

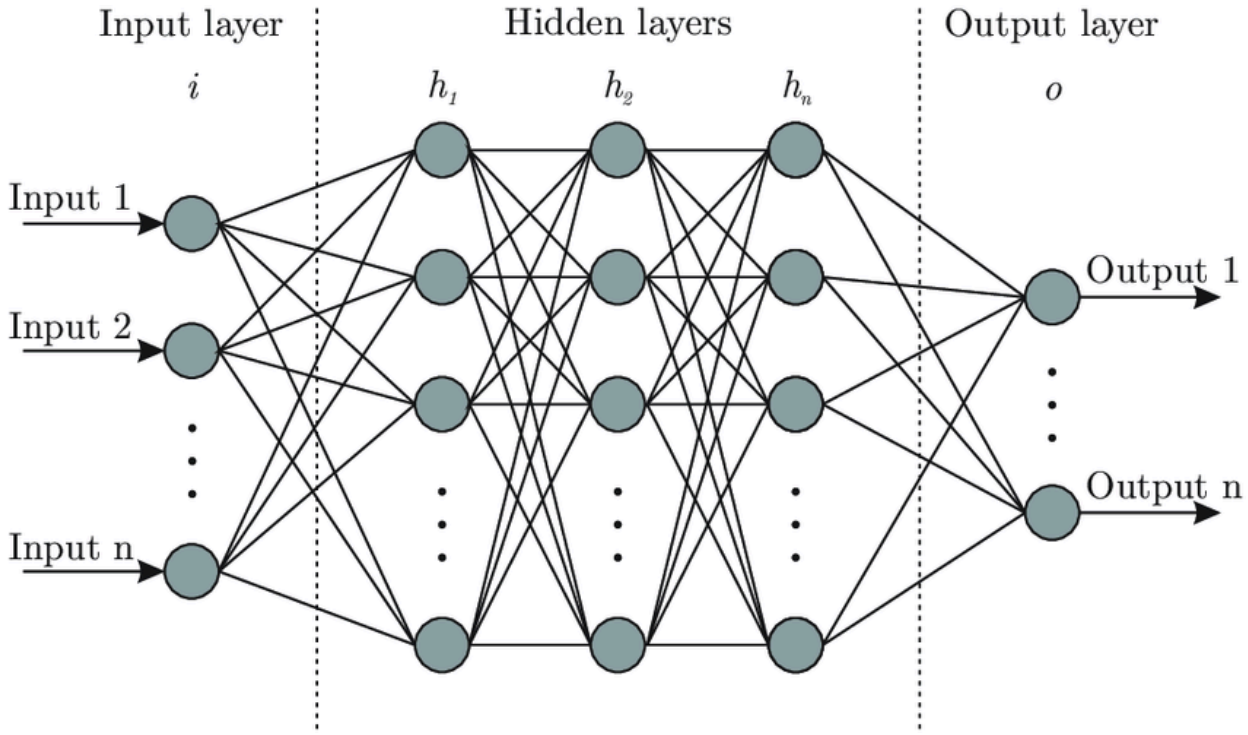
Then, you can use it
y_new = model.predict(X_new)

❓ Here is the perfect time to ask questions.

🤔 Ok, but what are we hiding in the `...`?

3.2 Sequential architecture : `model.add(...)`

It corresponds to the definition of the architecture.



Each layer has to be added with the `model.add(...)`
💻 Let's code it

```
from keras import Sequential, layers

# Basically, it will look like a sequence of layers
model = Sequential()

# First layer: 10 neurons and ReLU as the activation function
model.add(layers.Dense(10, activation='relu'))

# Disclaimer: The standard layers are called Fully Connected (Dense in Keras)
```

```
# You can go for two fully connected layers
model = Sequential()
model.add(layers.Dense(10, activation='relu'))
model.add(layers.Dense(20, activation='tanh'))
```

```
# You can also go for many, many, many more ...

model = Sequential()
model.add(layers.Dense(10, activation='relu'))
model.add(layers.Dense(20, activation='tanh'))
model.add(layers.Dense(10, activation='linear'))
model.add(layers.Dense(100, activation='sigmoid'))
model.add(layers.Dense(40, activation='softmax'))
model.add(layers.Dense(10, activation='tanh'))
model.add(layers.Dense(3, activation='relu'))
model.add(layers.Dense(9, activation='tanh'))
model.add(layers.Dense(8900, activation='relu'))
model.add(layers.Dense(1000, activation='tanh'))
```

Decision Rules

🤔 Ok... but how am I supposed to choose?

- Most decisions rely on practice
- Nonetheless, some are directly related to the problem/task you are tackling!

Rule 1: Tell the model the shape of your inputs

```
from keras import Input

model = Sequential()

# Imagine each observation has 4 features (x1, x2, x3, x4)
model.add(Input(shape=(4,)))  # Shape has to be a tuple
```

- The Input's shape directly impacts the **number of weights** in the first layer
- Without specifying it the model would infer the size the first time it sees a training batch
- It's better to specify it upfront: Keras can already build the model and allocate memory

Rule 2: The last layer is dictated by the task

Regression tasks require a **linear** activation function

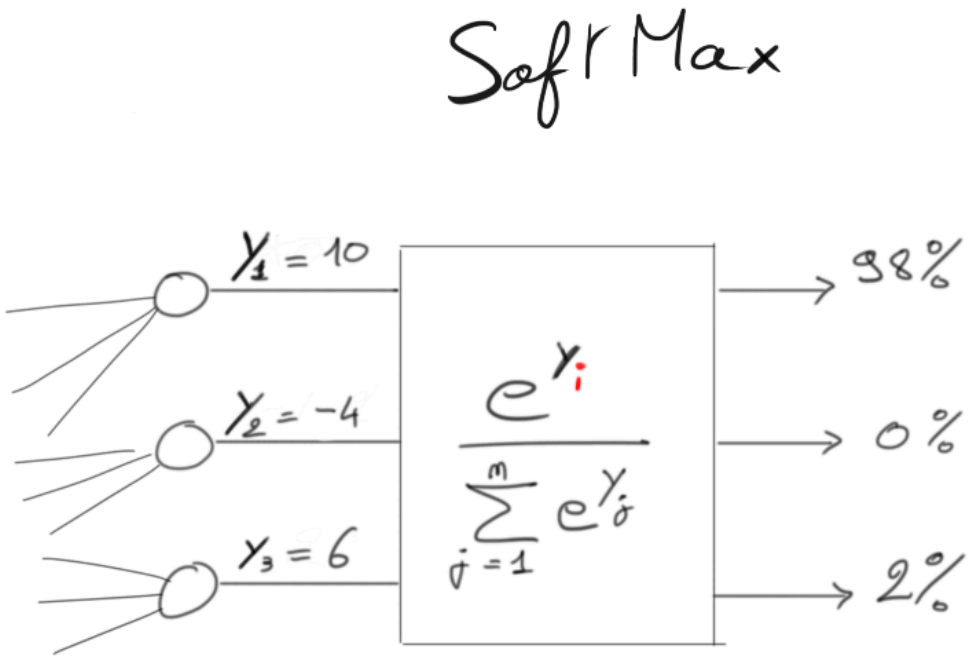
```
### size 1 (predict one value):
model.add(layers.Dense(1, activation='linear'))
```

```
# OR

### size 13 (y_pred.shape=(13,))
model.add(layers.Dense(13, activation='linear'))
```

Classification tasks require a **sigmoid** or **softmax** activation function

```
### 2 classes (binary)
model.add(layers.Dense(1, activation='sigmoid'))

# OR

### 8 classes (y_pred.shape=(8,))
model.add(layers.Dense(8, activation='softmax'))
```

Softmax turns numbers into **probabilities** that sum up to 1:



☝ Softmax (2 classes) = Sigmoid

Full Model Recap

```
### Regression of size 1
model = Sequential()
model.add(Input(shape=(100,)))
model.add(layers.Dense(10, activation='relu'))
#model.add(...)
```

```
model.add(layers.Dense(1, activation='linear'))


### Regression of size 13
model = Sequential()
model.add(Input(shape=(100,)))
model.add(layers.Dense(10, activation='relu'))
#model.add(...)
model.add(layers.Dense(13, activation='linear'))


### Classification with 2 classes
model = Sequential()
model.add(Input(shape=(100,)))
model.add(layers.Dense(10, activation='relu'))
#model.add(...)
model.add(layers.Dense(1, activation='sigmoid'))


### Classification with 8 classes
model = Sequential()
model.add(Input(shape=(100,)))
model.add(layers.Dense(10, activation='relu'))
#model.add(...)
model.add(layers.Dense(8, activation='softmax'))
```

❓ Which task was our first example? Which activation did we use?

Playground challenge

💻 Build a model with three hidden layers where the first has 5 neurons, the second has 4 neurons and the third has 3 neurons.

Then, run it on any dataset.
In practice, apart from the input size and the last layer, you have to choose:

- the number of neurons
- the number of layers
- the activation functions

But how?

=> This is the experimental part. This is where the mastery of Deep Learning comes in.

Practice has shown us that some decisions are best to start with as they usually work better.

💡 For instance, (almost) always **start** with the `relu` activation function - if it is not the last layer!

Counting number of parameters with `model.summary()`

 # Small exercice: how many parameters in this simple regression task:

```
model = Sequential()
model.add(Input(shape=(4,)))
model.add(layers.Dense(10, activation='relu'))
model.add(layers.Dense(1, activation='linear'))
```

```
10*4 + 10 + 10 + 1
```

61

```
model.summary()
```

Model: "sequential_4"

Layer (type)	Output Shape	Param #
dense_14 (Dense)	(None, 10)	50
dense_15 (Dense)	(None, 1)	11

Total params: 61 (244.00 B)

Trainable params: 61 (244.00 B)

Non-trainable params: 0 (0.00 B)

 # Small exercice: how many parameters in this model:

```
model = Sequential()
model.add(Input(shape=(784,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(64, activation='tanh'))
model.add(layers.Dense(10, activation='softmax'))
```

```
# Answer : Use the summary() function to keep track of what you are doing

model = Sequential()
model.add(Input(shape=(784,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(64, activation='tanh'))
model.add(layers.Dense(10, activation='softmax'))

model.summary()
print("manual calculation =", (64*784 + 64) + (64*64 + 64) + (10*64 + 10))
```

```
Model: "sequential_8"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃         Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ dense_25 (Dense)                     │ (None, 64)                  │          50,240 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_26 (Dense)                     │ (None, 64)                  │           4,160 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_27 (Dense)                     │ (None, 10)                  │             650 │
└──────────────────────────────────────┴─────────────────────────────┴─────────────────┘

Total params: 55,050 (215.04 KB)

Trainable params: 55,050 (215.04 KB)

Non-trainable params: 0 (0.00 B)

manual calculation = 55050
```

4. Training: loss & optimization procedure

```
model.compile(...)
model.fit(...)
```

👉 Focus of lecture 2

4.1 Compiling

```
model.compile(loss='mse', optimizer='adam')
```

- The loss is the way you compare
- y
- t
- r
- u
- e
- to
- y
- p
- r
- e
- d
-
- The optimizer is the way you update
- θ
- over the iterations to get closer to
- y
- t
- r
- u
- e
-

  (called "solver" in sklearn)

4.2 Fitting

```
model.fit(X, y, batch_size=32, epochs=10)
```

It tries to find the
θ
that minimizes the loss function.

The learning phase is **iterative** and **stochastic** (remember SGD?).

The intuition is that you give a subset of data, and the algorithm updates the parameters

θ

:

- `batch_size` is the size of the subset given to the neural network to update the parameters
- θ
-
- The entire dataset is split into different batches
- Once the algorithm has seen all the data, it counts as one `epoch`

Check out the docs to see all the arguments of `model.fit()`.

5 Full Example: Face recognition

```
# Load data
from sklearn.datasets import fetch_lfw_people
faces = fetch_lfw_people(min_faces_per_person=200, resize=0.25)

# 766 images of 31 * 23 pixel black & white
print(faces.images.shape)
```

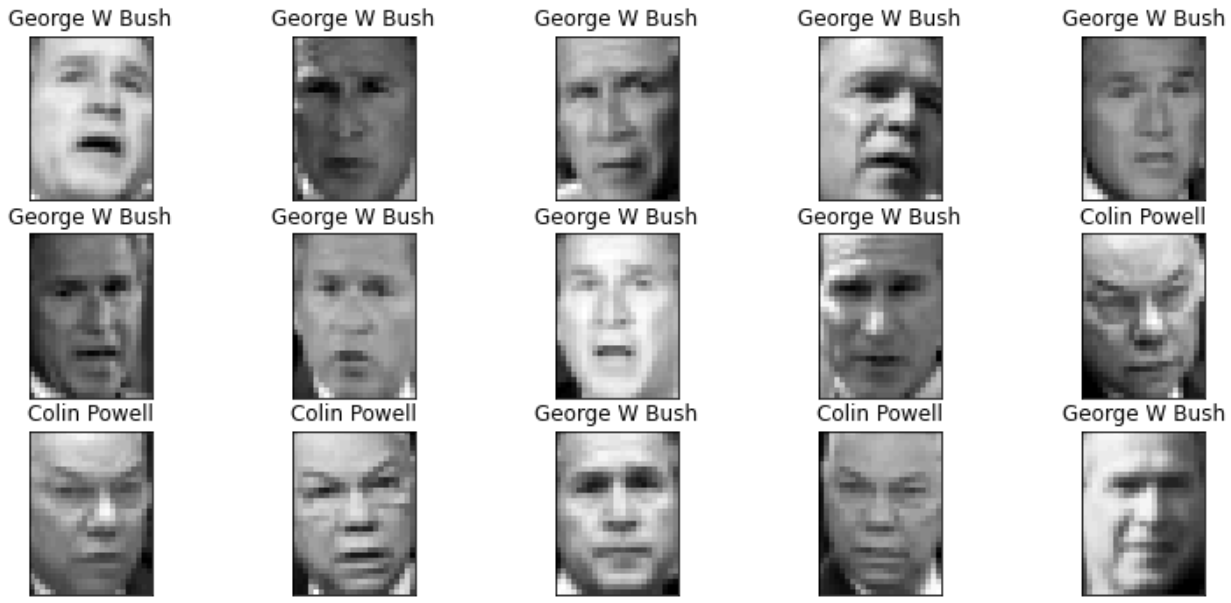
```
(766, 31, 23)
```

```
# 2 different target classes
np.unique(faces.target)
```

```
array([0, 1])
```

Let's visualize some faces:

```
fig = plt.figure(figsize=(13,10))
for i in range(15):
    plt.subplot(5, 5, i + 1)
    plt.title(faces.target_names[faces.target[i]], size=12)
    plt.imshow(faces.images[i], cmap=plt.cm.gray)
    plt.xticks(()); plt.yticks(())
```



Minimal preprocessing

```
# Flatten our 766 images
X = faces.images.reshape(766, 31*23)
X.shape
```

```
(766, 713)
```

```
y = faces.target
y.shape
```

```
(766,)
```

```
# Train test split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=3)
```

```
# Standardize
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
```

Simple model with 2 hidden layers

```
from tensorflow.keras Sequential, layers

# Model definition
model = Sequential()
model.add(Input(shape=(713,)))
model.add(layers.Dense(20, activation='relu'))
```

```
model.add(layers.Dense(10, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
model.summary()
```

```
Model: "sequential_10"
```

Layer (type)	Output Shape	Param #
dense_31 (Dense)	(None, 20)	14,280
dense_32 (Dense)	(None, 10)	210
dense_33 (Dense)	(None, 1)	11

```
Total params: 14,501 (56.64 KB)

Trainable params: 14,501 (56.64 KB)

Non-trainable params: 0 (0.00 B)
```

```
model.compile(
   optimizer='adam',
   loss='binary_crossentropy',
   metrics=['accuracy'])

model.fit(X_train, y_train, batch_size=16, epochs=20)
```

Evaluate performance

```
model.evaluate(scaler.transform(X_test), y_test)
# returns [loss, metrics]
```

```
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9249 - loss: 0.3044
```

[0.3246121108531952, 0.9270833134651184]

🤔 Is it good? What's our baseline?

```
pd.Series(y).value_counts()
```

```
1    530
0    236
Name: count, dtype: int64
```

```
# Baseline score
530 / (530+236)
```

```
0.6919060052219321
```

Let's check our predictions!

```
# Predicted probabilities
model.predict(scaler.transform(X_test))[:10]
```

```
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step

array([[9.9988025e-01],
       [8.7732069e-02],
       [9.9997973e-01],
       [7.0009235e-05],
       [4.4098161e-03],
       [9.9991453e-01],
       [1.3041348e-02],
       [9.9974465e-01],
       [8.2459510e-06],
       [9.9999917e-01]], dtype=float32)
```

Conclusion & Intuition

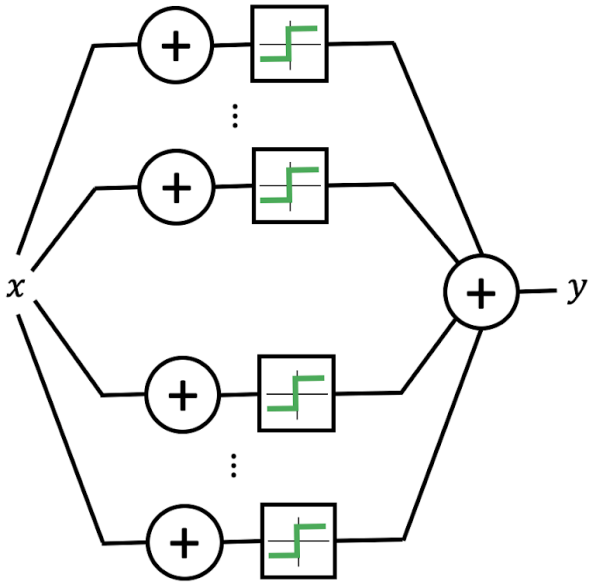
All in all, Deep Learning is nothing more than

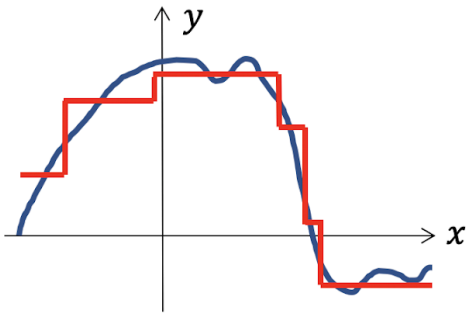
- multiple linear regressions stacked together
- non-linear functions: the activation functions

Why does it work?

A) Mathematically speaking

Dense networks are **universal approximations**: with just one hidden layer, they can approximate any continuous function with arbitrary precision.





☝️ This does not guarantee that you can easily **find** these optimal parameters of your model!

It may require extremely large sample size or computing power.

You will see that dense networks are not always the most appropriate architecture (in particular for images, text...).

B) Intuitively speaking

❗ Contrary to other ML algorithms, Neural Networks are very hard to understand intuitively.

Therefore, the goal is not to understand **why** it works to start with.

The goal is to learn **what** makes it work.

🚀 Let's do that by practicing.

💡 Pro tips:

- Add an Input layer to give the model the size of your input
- Last layer's number of neurons equals the output dimension
- Last layer's activation is linear (regression) or softmax/sigmoid (classification)
- Almost always **start** with the `relu` activation function (except for the last layer)

Bibliography

- How many layers / neurons do I need?
- Activation Functions in Neural Networks [12 Types & Use Cases]