MACHINE LEARNING

BUILDING OUR FIRST NEURAL NETWORK

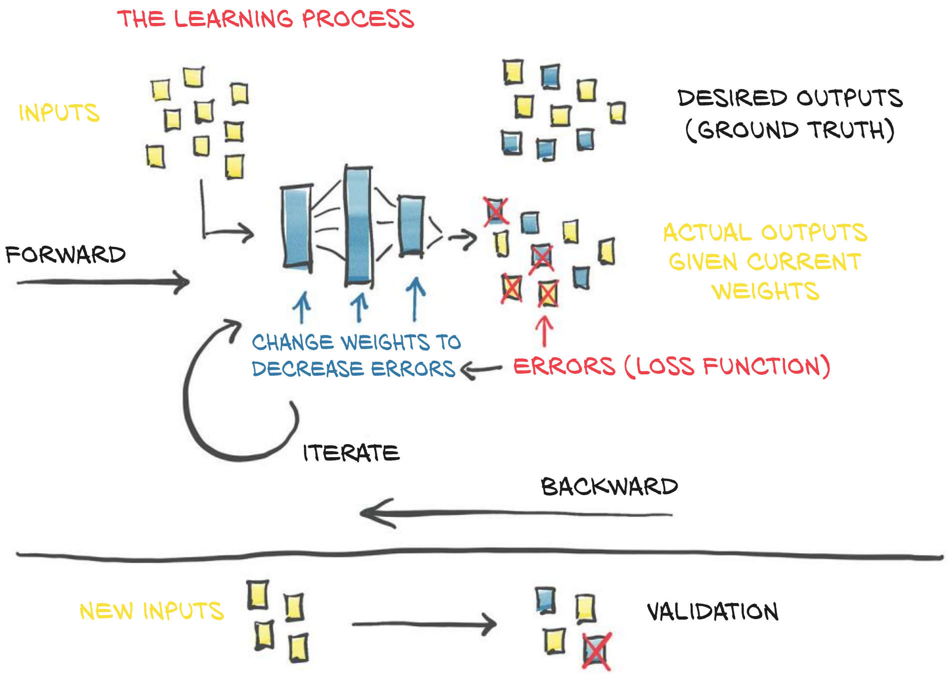
Reminder, Forward step and Backward Step

Forward pass

Backward pass

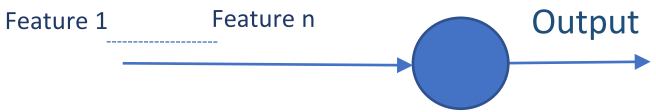
Optimization

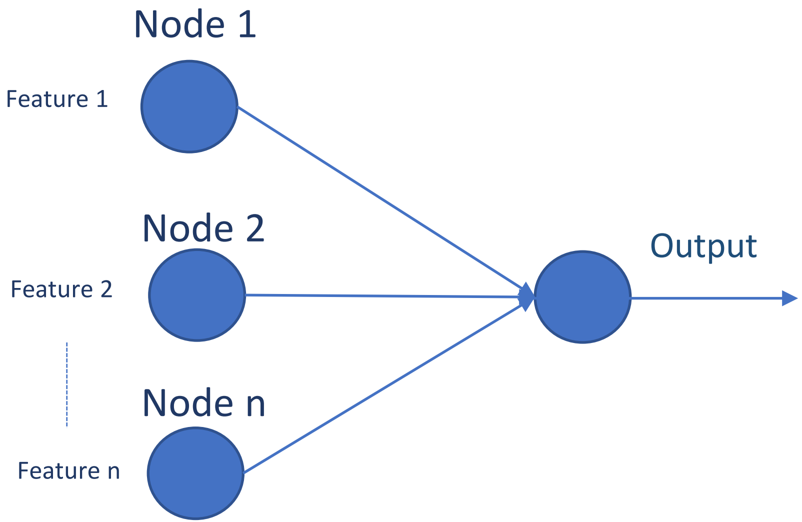
Repetition Forward pass



Overview

We will move from a Linear Regression to Neural Network





We will go through this evolution using a binary classification problem

Binary Classification

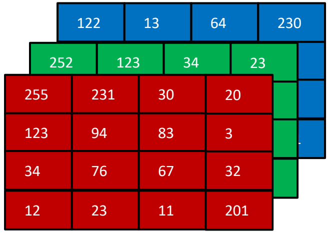
Problem: to classify pictures as “picture” or “Non-Cat” picture

1 (cat)

0 (cat)

y (label)

How is a picture represented in a computer ?



RGB (Red, Green, Blue)

64 pixels

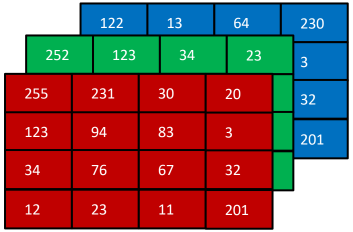
64 pixels

3 Arrays (64x64)

x (feature)

Feature formatting

We will format our 3-dimensional array into a single vector x



$$x = \begin{bmatrix} 255 \\ 231 \\ 30 \\ \vdots \\ 201 \\ 252 \\ 123 \\ \vdots \\ 201 \end{bmatrix} \qquad y = 1$$

x Vector with $64 \cdot 64 \cdot 3 = 12288$ ***components***

y Vector with **just 1*component***

Notation

Given a sample (x,y) $\longrightarrow x \in \mathbb{R}^{n_x}, y \in \{0,1\}$

$m\ training\ samples: \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$

$m = m_{train} \qquad m_{test}$

$$\chi = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(1)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \; n_x \qquad \Upsilon = [y^{(1)}, y^{(2)}, \dots, y^{(m)}]$$

m

$\chi \epsilon\ \mathbb{R}^{n_x x\ m}$ $\qquad$ $\Upsilon \epsilon\ \mathbb{R}^{1x\ m}$

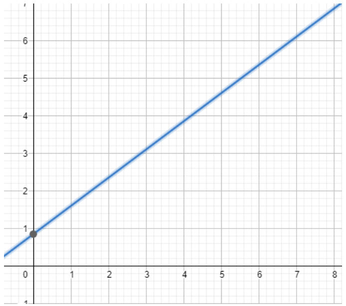
$\chi.shape = (n_{x,}m)$ $\qquad$ $\text{y}.shape = (1, m)$

Logistic Regression

A logistic regression is a Neural Network used for binary classification

Our problem: Given a sample x, we want to $\hat{y}$ as the probability of being a cat $\hat{y} = P(y = 1|x) \quad 0 \le \hat{y} \le 1$

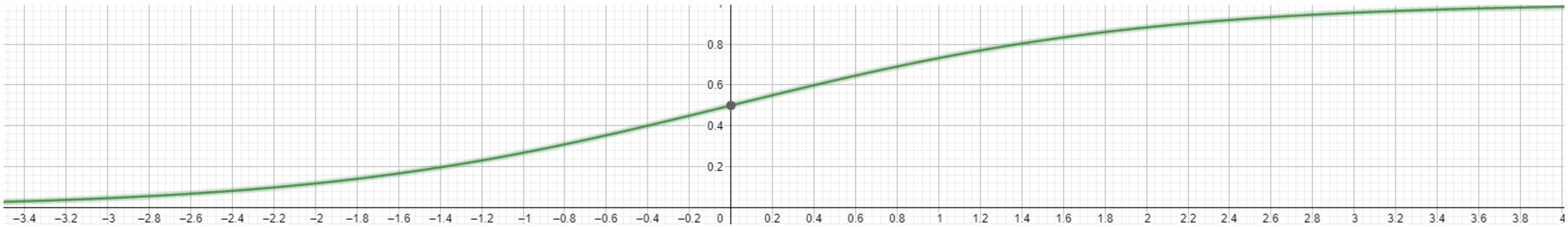
Our model: $\boldsymbol{z = w^T \cdot x + b, where\, w \in \mathbb{R}^{n_x}, b \in \mathbb{R}}$



But… , this function returns values greater than 1 !!!

And.., probability should be a real number between, and 1

Here it comes... the neuron!



$We\ call\ z = w^T + b$

$$\hat{y} = \sigma(z) \qquad \sigma(z) = \frac{1}{1+e^{-z}}$$

If z is large + ... $\sigma(z) \approx \dfrac{1}{1+0} \approx 1$

If z is large -... $\sigma(z) \approx \dfrac{1}{1+big} \approx 0$

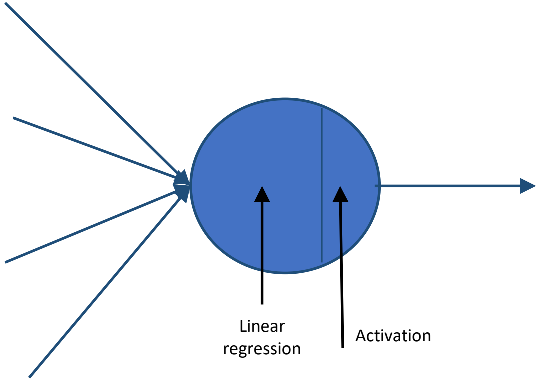
Now our output is number between 0 and 1

What is a neuron ?

A neuron in Machine Learning is a function composition between:

- linear equation (regression function)
- and a non-linear equation (activation function)

$$\hat{y} = g \circ f(w,b) = g(f(w,b))$$



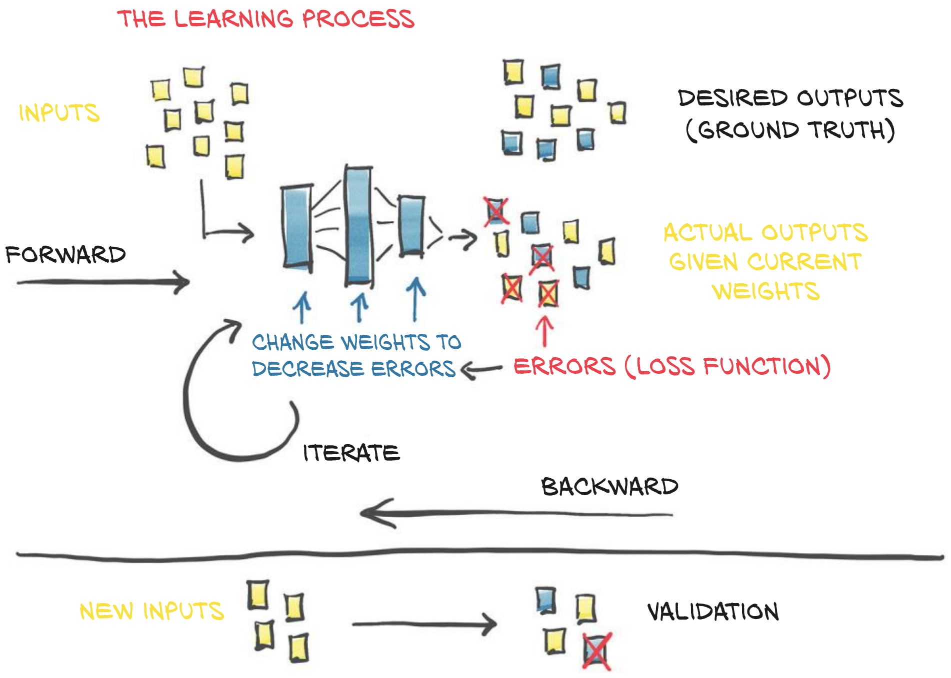
In our Cat classifier (binary classification):

$$\hat{y} = \sigma \circ z(w,b) = \sigma\big(z(w,b)\big) = \frac{1}{1+e^{-z}} = \frac{1}{1+e^{-(w^T x+b)}}$$

w^T *is vector*, b *is a real number*

Loss Function

- ✓ Forward pass
- ➡ Backward pass
- Optimization
- Repetition Forward pass

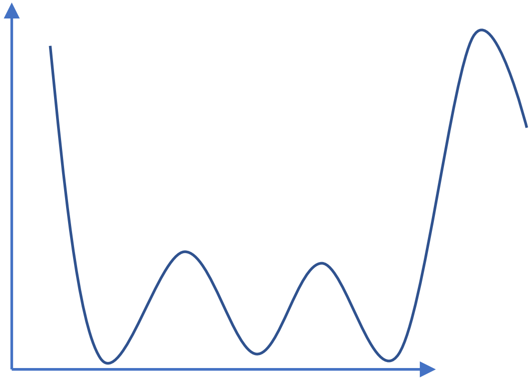


Logistic Regression Cost Function

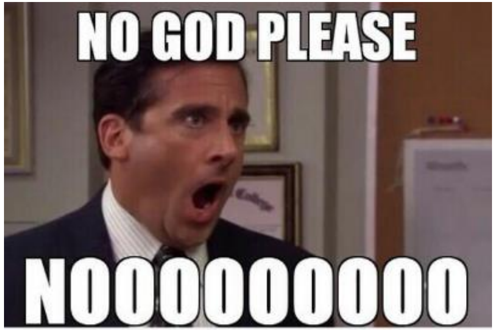
i is the i-th example

$Given\ \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\},$ we want $\hat{y}^{(i)} \approx y^{(i)}$

$Our\ cost\ function\ should\ be\ \mathcal{L}(\hat{y}, y) = (\hat{y} - y)^2$



Local minimum issue



Gradient Descent doesn't work well with local minimum

Loss function for Logistic Regression

Specific Loss Function for Logistic Regression

Loss function $\Rightarrow$ $\mathcal{L} = (\hat{y}, y) - (y \cdot \log(\hat{y}) - (1 - y) \cdot \log(1 - \hat{y}))$

$If\ y = 1, \mathcal{L}(\hat{y}, y) = -\log(\hat{y})$ $\quad$ $If\ \hat{y}\ close\ to\ 1, \mathcal{L}\ \approx 0$

$If\ y = 0, \mathcal{L}(\hat{y}, y) = -\log(1 - \hat{y})$ $\quad$ $If\ \hat{y}\ close\ to\ 0, \mathcal{L}\ \approx 0$

$$\Im(w, b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) = -\frac{1}{m}\sum_{i=1}^{m} [y^{(i)} log \hat{y}^{(i)} + \left(1 - y^{(i)}\right)\log(1 - \hat{y}^{(i)})]$$

Cost function

Universidad Francisco de Vitoria
UFV Madrid

Loss Function versus Cost Function

Loss function measures de accuracy in a single example

$$\mathcal{L}(\hat{y}, y) = -(y \cdot \log(\hat{y}) - (1 - y) \cdot \log(1 - \hat{y}))$$

Cost function measures de accuracy on the whole dataset

$$\Im(w, b) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)} log\hat{y}^{(i)} + \left(1 - y^{(i)}\right)\log(1 - \hat{y}^{(i)})]$$

Backward pass

Universidad Francisco de Vitoria
UFV Madrid

We're going to suppose that we only have 1 sample with two features

x_1
w_1
x_2
w_2
b

$$z = w_1 \cdot x_1 + w_2 \cdot x_2 + \mathrm{b} \quad \Rightarrow \quad a = \sigma(z) \quad \Rightarrow \quad \mathcal{L}(a, y)$$

First backward step

$$\frac{d\mathcal{L}(a,y)}{da}$$

Second backward step

$$\frac{d\mathcal{L}(a,y)}{dz}$$

Third backward step

$$\left[\frac{\partial\mathcal{L}(a,y)}{\partial w_1}, \frac{\partial\mathcal{L}(a,y)}{\partial w_w}, \frac{\partial\mathcal{L}(a,y)}{\partial b}\right]$$

Ground truth

output

feature

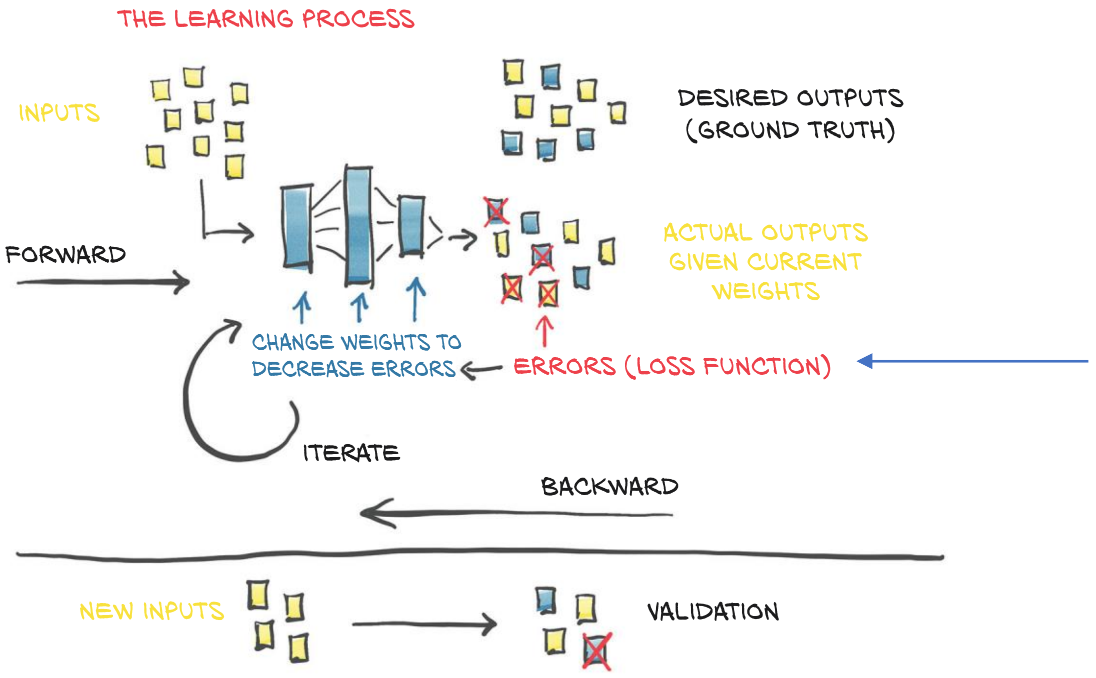
$$\frac{\partial\mathcal{L}(a,y)}{\partial w_1} = \frac{dz}{dw_1} \cdot \frac{da}{dz} \cdot \frac{d\mathcal{L}(a,y)}{dy} = x_1 \cdot a \cdot (1-a) \cdot \left(-\frac{y}{a} + \frac{1-y}{1-a}\right) = x_1(a-y)$$

Therefore, the gradinet vector for the cost function is

$$\nabla\mathcal{L}[w_1, w_2, b] = [x_1(a-y), x_2(a-y), (a-y)]$$

Loss Function

- ✓ Forward pass
- ✓ Backward pass
- ➡ Optimization
- Repetition Forward pass



Universidad Francisco de Vitoria UFV Madrid

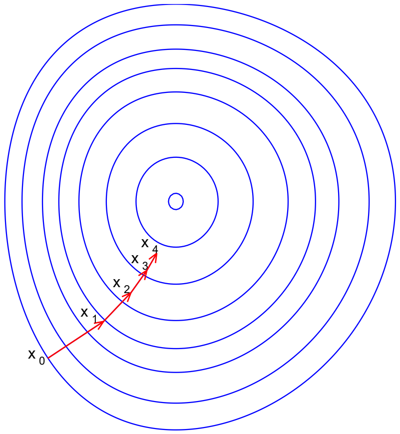
Remind the gradient descent algorithm used in the linear regression practice

$$\boldsymbol{z}_{n+1} = \boldsymbol{z}_n - \gamma \nabla \mathcal{L}(\boldsymbol{z}_n)$$

$$w_{1_{n+1}} = w_{1_n} - \gamma \nabla \mathcal{L}\left(w_{1_n}\right) = w_{1_n} - \gamma \cdot x_n(a_n - y_n)$$

$$w_{2_{n+1}} = w_{2_n} - \gamma \nabla \mathcal{L}\left(w_{2_n}\right) = w_{2_n} - \gamma \cdot x_n(a_n - y_n)$$

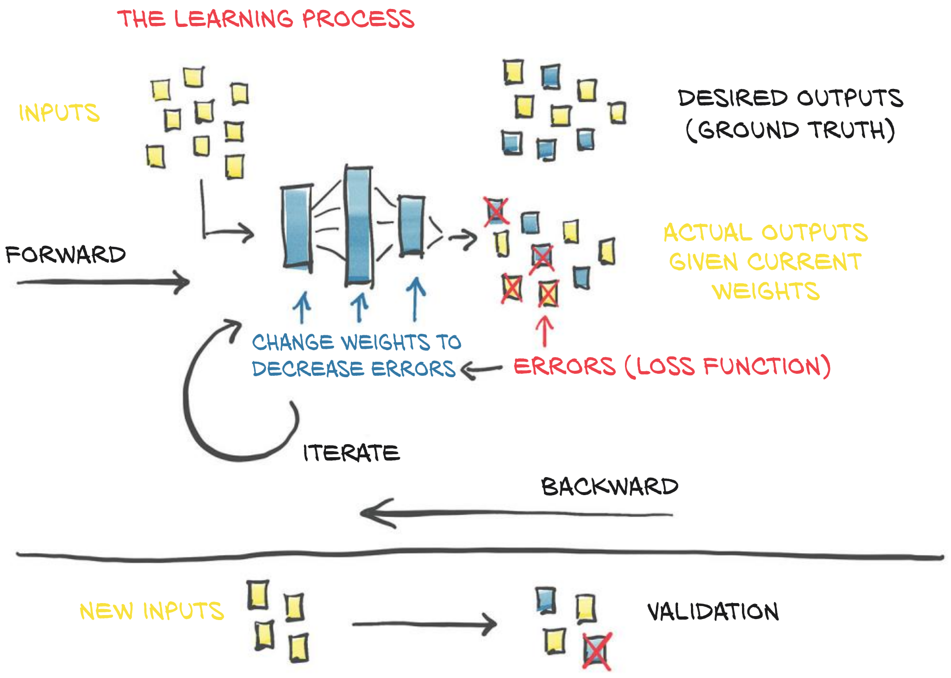
$$b_{n+1} = b_n - \gamma \nabla \mathcal{L}(b_n) = b_n - \gamma \cdot (a_n - y_n)$$



γ = Learning rate

Loss Function

- ✓ Forward pass
- ✓ Backward pass
- ✓ Optimization
- ➡ Repetition Forward pass



Repetition

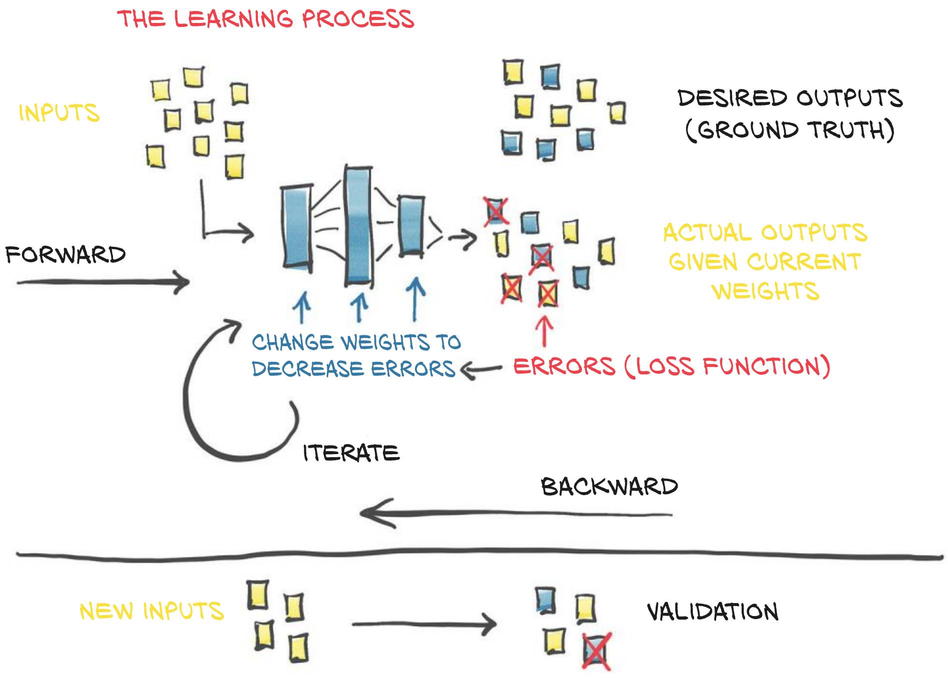
We will use a for loop to iterate as many times as we consider (epochs)

```
for epochs in range(1,n_epochs+1)

     a = compute our output
     Loss = compute our loss
     Gradient = compute our gradient
     Params in epoch n = (params in epoch n-1)- learning rate x Gradient Descent
```

Loss Function

- ✓ Forward pass
- ✓ Backward pass
- ✓ Optimization
- ✓ Repetition Forward pass



Nothing to do with just 1 example

How will our algorithm be with m examples instead of just 1 example?

Logistic regression with m examples instead of 1 example

What happens if I have to compute the algorithm for m examples instead of 1 example??

Let's review our trainin loop

```
for epoch in epochs:
      for i in range(1,m+1):
```

$z^{(i)} = w^T \cdot x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

Forward Function

$\mathcal{L} = \mathcal{L} + (-[y^{(i)} - loga^{(i)} + (1 - y^{(i)})\log(1 - a^{(i)})])$

Loss Function

$dz^{(i)} = a^{(i)} - y^{(i)}$

$db = db + dz^{(i)}$

```
for n in range(1,n_features):
```

$dw_n = x_n^{(i)} \cdot dz^{(i)}$

Gradient Vector

$\mathfrak{J} = {}^{\mathcal{L}}/_{m}, {}^{dw_1}/_{m}, \ldots, {}^{dw_n}/_{m}, {}^{db}/_{m}$

Cost Function

```
for n in range(1,n_features):
```

$w_n = w_n - \gamma \cdot dw_n$

Optimization

$o(n^3)$

MACHINE LEARNING

VECTORIZATION

Why Vectorizing?

$o(n^3)$ algorithms don't escalate, moreover in Machine Learning environments, where data tends to grow exponentially

Non-Vectorized

$z = w^T \cdot x + b$

$z = 0$

for i in range (1,n_features+1)

$\quad z = z + w[i] \cdot x[i]$

Vectorized

$$z = np.dot(w, x)$$

$$w^T x$$

Performance Comparasion

Universidad Francisco de Vitoria
UFV Madrid

```python
# -*- coding: utf-8 -*-
"""
Created on Thu Oct 20 10:59:15 2022

@author: PedroAnquela
"""

import numpy as np
import time

dim = 10000000

w = np.random.rand(dim)
x = np.random.rand(dim)

start = time.time()

a = np.dot(w,x)

end = time.time()

print("Vectorized version: {0:.4f} ms.".format(1000*(end-start)))

##########################################
# Non-Vectorized Version                 #
##########################################
c = 0
start = time.time()

for i in range (dim):
    c += w[i] * x[i]

end = time.time()

print("Non-Vectorized version: {0:.4f} ms.".format(1000*(end-start)))
```

```
In [8]: runfile('C:/Users/PedroAnquela/Documents/Projectos_Local/UFV/prog_ii_2020/ml/vectorization/vectorization.py', wdir='C:/Users/PedroAnquela/Documents/Projectos_Local/UFV/prog_ii_2020/ml/vectorization')
Vectorized version: 9.9742 ms.
Non-Vectorized version: 3460.2816 ms.
```

384 times faster when vectorizing

We turn data into numpy vectors

$$A = [a^{(1)}, .. a^{(m)}]$$

$$Y = [y^{(1)}, .. y^{(m)}]$$

$$dZ = A - Y$$

$$db = \frac{1}{m}\sum_{i=1}^{m} dz^{(i)} = \frac{1}{m} np.sum(dZ)$$

$$dw = \frac{1}{m} X \cdot dz^{T} = \frac{1}{m}\left[x^{(1)} \dots x^{(m)}\right]\begin{bmatrix} dz^{(1)} \\ \dots \\ dz^{(m)} \end{bmatrix}$$

$$(1, m) \cdot (m, 1)$$

Translating our vector loop

Let's review our trainin loop

$J = 0, dw_1 = 0, dw_2 = 0, db = 0$

```
for epoch in epochs:
    for i in range(1,m+1):
```

$z^{(i)} = w^T \cdot x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

$\mathcal{L} = \mathcal{L} + (-[y^{(i)} - log a^{(i)} + (1 - y^{(i)})\log(1 - a^{(i)})])$

$dz^{(i)} = a^{(i)} - y^{(i)}$

$db = db + dz^{(i)}$

```
for n in range(1,n_features):
```

$dw_n = x_n^{(i)} \cdot dz^{(i)}$

$\mathfrak{J} = {}^{\mathcal{L}}/_{m}, {}^{dw_1}/_{m}, \dots, {}^{dw_n}/_{m}, {}^{db}/_{m}$

```
for n in range(1,n_features):
```

$w_n = w_n - \gamma \cdot dw_n$

```
for epoch in epochs:
```

$Z = w^T X + b = np.dot(w.T, X) + b$

$A = \sigma(Z)$

$dZ = A - Y$

$db = \frac{1}{m}\sum_{i=1}^{m}\frac{1}{m} np.sum(dZ)$

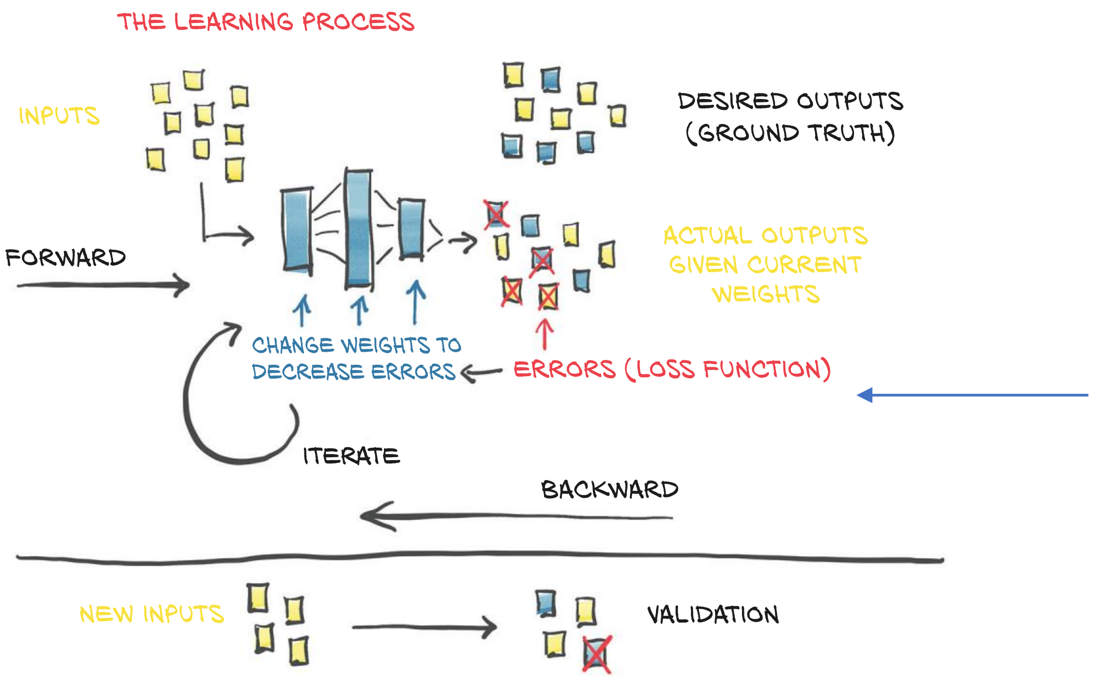
$dw = \frac{1}{m} X \cdot dz^T$

$w = w - \gamma dw$

$b = b - \gamma db$

Model completed!

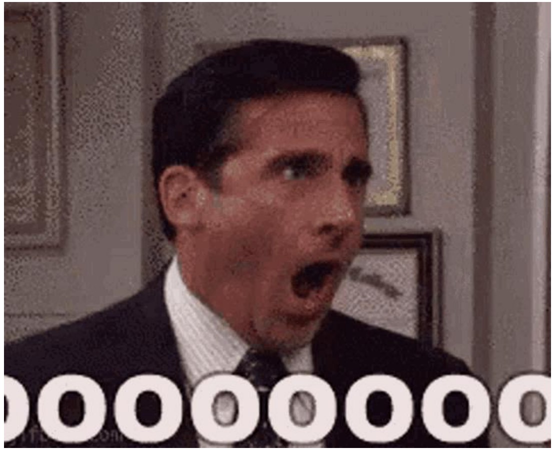
- ✓ Forward pass
- ✓ Backward pass
- ✓ Optimization
- ✓ Repetition Forward pass



Remembering a basic concept of Linear Algebra

$$\begin{bmatrix}100\\101\\102\\103\end{bmatrix} + 100 =$$

Can I do this operation ?



Two matrices must have an equal number of rows and columns to be added.

In which case, the sum of two matrices **A** and **B** will be a matrix which has the same number of rows and columns as **A** and **B**.

The sum of **A** and **B**, denoted **A** + **B**, is computed by adding corresponding elements of **A** and **B**.

Source: https://en.wikipedia.org/wiki/Matrix_addition

Quick snippet

- Open your text editor
- Generate a numpy array with the following dimensions (4,1)
- add the number 100 to your numpy.array.
- Print the result
- Did it work? What was the result?

```
import numpy as np

a = np.array([[100],
              [101],
              [102],
              [103]
              ])

print(a + 100)
```

Python Broadcasting

Universidad Francisco de Vitoria
UFV Madrid

Numpy broadcasts non-adjusted shapes to allow the addition

$$\begin{bmatrix}100\\101\\102\\103\end{bmatrix} + 100 = numpy\ broadcasting = \begin{bmatrix}100\\101\\102\\103\end{bmatrix} + \begin{bmatrix}100\\100\\100\\100\end{bmatrix} = \begin{bmatrix}200\\201\\202\\203\end{bmatrix}$$

$$\begin{bmatrix}1 & 2 & 5\\3 & 4 & 6\end{bmatrix} + \begin{bmatrix}100 & 200 & 300\end{bmatrix} = numpy\ broadcasting = \begin{bmatrix}1 & 2 & 5\\3 & 4 & 6\end{bmatrix} + \begin{bmatrix}100 & 200 & 300\\100 & 200 & 300\end{bmatrix} = \begin{bmatrix}101 & 202 & 305\\103 & 204 & 306\end{bmatrix}$$

Numpy broadcasts non-adjusted shapes to allow the addition

General Rule

Universidad Francisco de Vitoria
UFV Madrid

Numpy broadcasts general principles

$$A_{(m,n)} + B_{(1,n)} \longrightarrow A_{(m,n)} + B_{(m,n)}$$

$$A_{(m,n)} - C_{(m,1)} \longrightarrow A_{(m,n)} + C_{(m,n)}$$

$$A_{(m,n)} + n \in \mathbb{R} \longrightarrow A_{(m,n)} + B_{(m,n)}\,, b_{ij} = n\ \forall b_{ij} \in B$$

Never let Numpy broadcast work in Matrix Multplication

$$A_{(m,n)} \cdot B_{(n,m)}$$

Number of columns of first matrix HAS TO match the number of rows of the second matrix

Matrix multiplication

- Given the following matrix $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$

- And the following vector $\mathrm{B} = [2,3,4]$

- Use numpy to calculate A*B

Trick

Universidad Francisco de Vitoria
UFV Madrid

Use the command numpy.reshape to fix the dimensions you need.

Never let numpy decide by itself dimensions of your matrix

```
import numpy as np

a = np.ones(5)

print(a)
print(a.shape)


a = a.reshape(5,1)

print(a)
print(a.shape)

a = a.reshape(1,5)

print(a)
print(a.shape)
```

```
[1. 1. 1. 1. 1.]
(5,)
```

```
[[1.]
 [1.]
 [1.]
 [1.]
 [1.]]
(5, 1)
```

```
[[1. 1. 1. 1. 1.]]
(1, 5)
```

Program Assignment

Universidad Francisco de Vitoria
UFV Madrid

Cat / non-Cat classifier

- Build a Logistic Regression (Shallow Neural Network) to recognize cats
- Classify your own photo as Cat or Non-Cat
- Loops are not permitted, unless there is an explicit function to do so
- Use Jupyter Notebook to fill up the Assignment (conda install jupyter)
- Clone the Notebook from the following link
- At the top of a Given Block you will find a #Graded Función tag, you will write you own code between the ###Start Solution Here ### and the ###End Solution Here### comments
- Take your time, thing about all the concepts explained in the previous lectures. Check out if your model has converged and it can recognize a cat