2 - Overview of the Problem set

Problem Statement: You are given a dataset ("data.h5") containing:

```
- a training set of m_train images labeled as cat (y=1) or non-cat (y=0)
- a test set of m_test images labeled as cat or non-cat
- each image is of shape (num_px, num_px, 3) where 3 is for the 3 channels (RGB). Thus, each image is square
(height = num_px) and (width = num_px).
```

You will build a simple image-recognition algorithm that can correctly classify pictures as cat or non-cat.

Let's get more familiar with the dataset. Load the data by running the following code.

(num_px, num_px, 3) ↳ 3 channels (RGB)

square images

Exercise 2

Reshape the training and test data sets so that images of size (num_px, num_px, 3) are flattened into single vectors of shape (num_px ∗ num_px ∗ 3, 1).

A trick when you want to flatten a matrix X of shape (a,b,c,d) to a matrix X_flatten of shape (b∗c∗d, a) is to use:

```
X_flatten = X.reshape(X.shape[0], -1).T      # X.T is the transpose of X
```

In [6]:

```
# Reshape the training and test examples
#(≈ 2 lines of code)
# train_set_x_flatten = ...
# test_set_x_flatten = ...
# YOUR CODE STARTS HERE
train_set_x_flatten = train_set_x_orig.reshape(train_set_x_orig.shape[0], -1).T
test_set_x_flatten = test_set_x_orig.reshape(test_set_x_orig.shape[0], -1).T

# YOUR CODE ENDS HERE

# Check that the first 10 pixels of the second image are in the correct place
assert np.alltrue(train_set_x_flatten[0:10, 1] == [196, 192, 190, 193, 186, 182, 188, 179, 174, 213]), "Wrong soluti
assert np.alltrue(test_set_x_flatten[0:10, 1] == [115, 110, 111, 137, 129, 129, 155, 146, 145, 159]), "Wrong solutio

print ("train_set_x_flatten shape: " + str(train_set_x_flatten.shape))
print ("train_set_y shape: " + str(train_set_y.shape))
print ("test_set_x_flatten shape: " + str(test_set_x_flatten.shape))
print ("test_set_y shape: " + str(test_set_y.shape))
```

```
train_set_x_flatten shape: (12288, 209)
train_set_y shape: (1, 209)
test_set_x_flatten shape: (12288, 50)
test_set_y shape: (1, 50)
```

train_set_x_flatten => (m, num_px, num_px, 3)
↳ RGB

reshape(m, -1).T

↳ automatically calculate product of the remaining dimensions.

(px × px × 3)

$.T$

↳ (features, examples).

now col.

$$\begin{array}{rc} & \begin{array}{cccc} ex_1 & ex_2 & ex_3 & \\ \downarrow & & \downarrow & \end{array} \\ \begin{array}{r} feature_1 \rightarrow \\ feature_2 \rightarrow \\ feature_3 \rightarrow \\ \rightarrow \end{array} & \begin{bmatrix} [\,| & | & | & |\,], \\ [\,x_1 & x_2 & x_3 & x_4\,], \\ [\,| & | & | & |\,], \\ [\, & & & \,], \end{bmatrix} \end{array}$$

Preprocessing Steps: → figure out dimensions + shapes (m, px, px, batch)

→ reshape dataset 3,T.

each example is (px × px × 3, 1)

↓

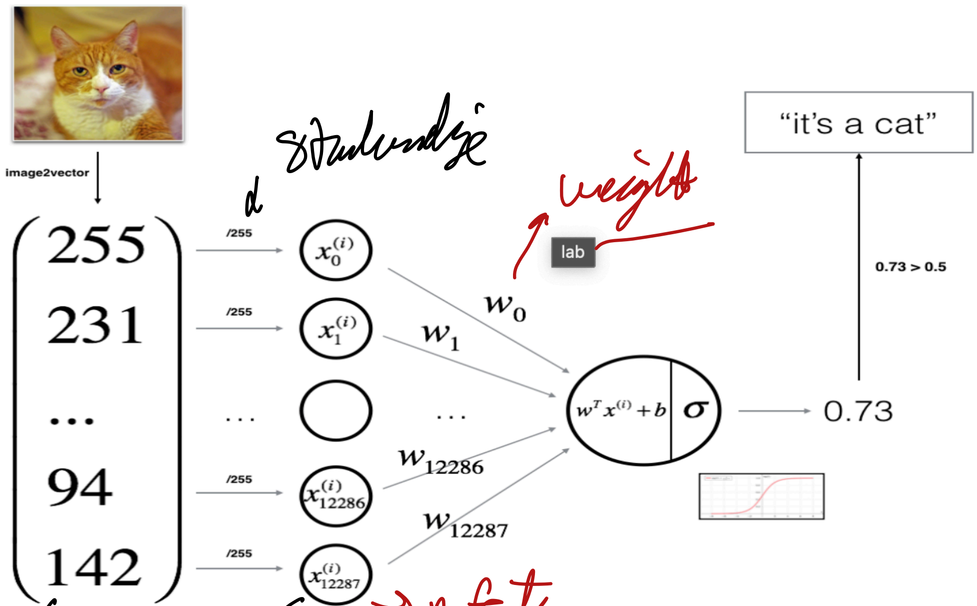
col vector

→ standardize → divide by mean, or here divide by max value 255.

3 - General Architecture of the learning algorithm

It's time to design a simple algorithm to distinguish cat images from non-cat images.

You will build a Logistic Regression, using a Neural Network mindset. The following Figure explains why **Logistic Regression is actually a very simple Neural Network!**



standardize

weights

→ n features

flatten the

$(P_x, P_x, 3) \to (P_x * P_x * 3, 1)$

x data input

$y = \sigma(w^T x^{(i)} + b) \in \mathbb{R}.$

n features.

$\langle 1, n \rangle$ $\langle n, 1 \rangle$

$w = [\;] \to w^T = [\;]$

$n \times 1$ $1 \times n$

$y = 1$ for cate

$y = 0$ for not cat.

→ for example $x^{(i)}$

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

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

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

↓
Loss.

Cost ⇒ $J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)})$

→ init the parameters of the model
→ learn parameters for model BY MINIMIZING cost.
→ use parameters to predict
→ analyze results

$$X = \begin{array}{c|c} & \xrightarrow{\quad m \quad} \\ \begin{array}{c} n \\ \text{features} \end{array} & \begin{bmatrix} | & | \\ x_1 & x_2 \\ | & | \end{bmatrix} \end{array}$$

$$w = \begin{array}{c|c} & 1 \\ \begin{array}{c} n \\ \text{features} \end{array} & \begin{bmatrix} \vdots \\ \vdots \end{bmatrix} \end{array}$$

$$A = \sigma(w.T\, X + b)$$

np.dot(w.T, X)

$[w_0 \ w_1 \ w_2 \ w_3 \ w_4]$
$1 \times n$

$[\quad\quad] + b$

$1 \times m$

$\to 1 \times m$.

$A = 1 \times M$.

m examples.

$$\text{Cost} = \left(-\frac{1}{m}\right) \sum_{i=1}^{m} Y \log(A) + (1-Y) \log(A)$$

$1 \times m \quad 1 \times m$

= element wise multiplicate

Sum over them and then divide.

lab

```
# FORWARD PROPAGATION (FROM X TO COST)
#(≈ 2 lines of code)
# compute activation
# A = ...
# compute cost by using np.dot to perform multiplication.
# And don't use loops for the sum.
# cost = ...
# YOUR CODE STARTS HERE
A = sigmoid(np.dot(w.T,X) + b)
cost = (-1/m)*np.sum(Y*np.log(A) + (1-Y)*np.log(1-A))
```

$w = \begin{bmatrix} \\ \\ \\ \end{bmatrix} \quad \langle n, 1 \rangle \Rightarrow w^T = [\quad\quad] \ \langle 1, n \rangle$

$$X = \begin{array}{cc} & m \\ n & \begin{bmatrix} | & | & & & \\ x_1 & x_2 & - & - & - \\ | & | & & & \end{bmatrix} \end{array} \quad \langle n, m \rangle.$$

$w^T X = \langle 1, n \rangle \langle n, m \rangle \Rightarrow \langle 1, m \rangle.$

applying appropriate weight to appropriate input's feature.

b Scalar $\Rightarrow$ broadcasted to $\langle 1, m \rangle$

which is for the pixel vectorized

$\Rightarrow A : \langle 1, m \rangle \quad [\quad\quad]$

↳ prediction for every feature in the m samples.

$Cost = \left(-\frac{1}{m}\right).$

$Y = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix} \langle 1, m \rangle$ $\log(A) \Rightarrow \langle 1, m \rangle$ [ ]

$Y * \log(A) \Rightarrow \langle 1, m \rangle$ element wise multiplication.

Sum over the $\langle 1, m \rangle \Rightarrow$ Scalar cost

which is the -ve log likelihood cost.

b/c we want to minimize by reducing our likelihood max → that we match multiplication.

Calculus prop for the gradient descent

$$\frac{\partial J}{\partial w} = \frac{1}{m} X(A-Y)^T$$

tells us how much to change.

$$\frac{\partial J}{\partial b} = \frac{1}{m} \sum (a^{(i)} - y^{(i)})$$

$A \to \langle 1, m \rangle$

$Y \to \langle 1, m \rangle$ $\to T$ $\langle m, 1 \rangle$.

$X : \langle n, m \rangle$ $X \cdot (A-Y)^T = \langle n, 1 \rangle$

how much to change each one of the n weights

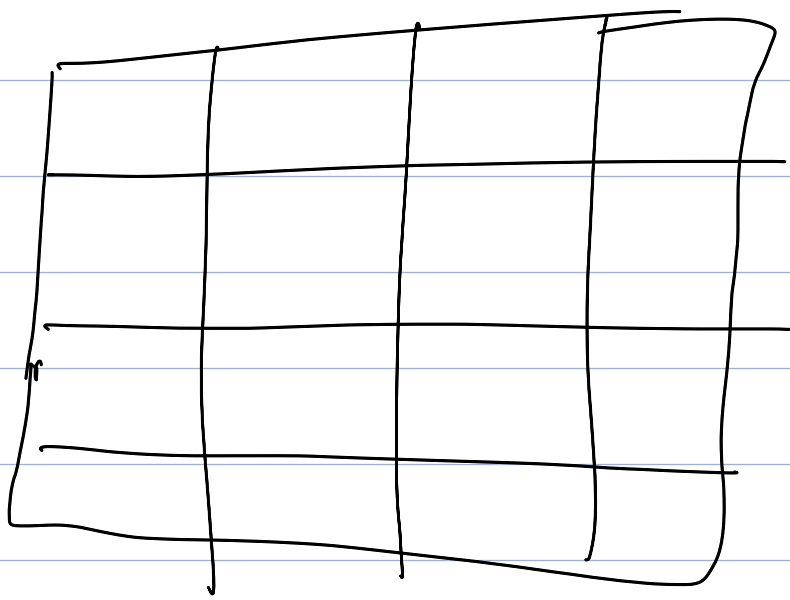
$$\frac{\partial J}{\partial b} = \frac{1}{m} \sum (a^{(i)} - y^{(i)}) + \frac{1}{m} \text{np.sum}(A, Y)$$

$\langle 1, m \rangle, \quad \langle 1, m \rangle.$

↳ how to change bias

Now we can compute cost and the gradient.

→ want to update parameters using gradient descent.



optimization fxn: learn w and b by minimize J (cost).

For $\theta = \theta - \alpha \, d\theta$

↳ learning rate.

$w \Rightarrow \begin{bmatrix} w_0 \\ \vdots \\ w_n \end{bmatrix}$ $\langle px * px * 3, 1 \rangle$

$n = px * px * 3$

$m = \#$ of examples

$$X = \overset{\longleftarrow\ m\ \longrightarrow}{\begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}} \updownarrow n \quad \langle n, m \rangle$$

Return w. and b.

grads →