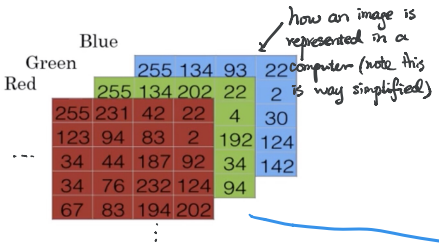
## Binary Classification


output for binary classification only has 2 classes, e.g. 0 or 1 (non-cat or cat)
i.e. $y$ = 0 (non-cat) or 1 (cat)




an image is represented as 3 matrixes with the size equal to the pixels of the image. For example, if the image is 64 pixels by 64 pixels, then it's represented as 3 64 by 64 matrixes corresponding to the red, green and blue pixel intensity value for the image. So to unroll these pixel intensity values into feature vector, we will define a vector $x$ to store all these values in an orderly fashion.

i.e. $x = \begin{bmatrix} 255 \\ 231 \\ \vdots \\ 255 \\ 134 \\ \vdots \\ 256 \\ 134 \\ \vdots \end{bmatrix}$ then the dimension of of $x$ is $64 \times 64 \times 3 = 12288$

$\Rightarrow n_x$ or $n = 12288$

### Notation:

- a single training example $(x, y)$, where $x \in \mathbb{R}^{n_x}$, $y \in \{0, 1\}$
- $m$ training examples: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots (x^{(m)}, y^{(m)})\}$, where $m$ = # training examples. or $m = m_{train}$
- $m_{test}$ = # test examples
- input matrix: $X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix}_{n \times m}$ $X \in \mathbb{R}^{n \times m}$, $X.shape = (n, m)$

output matrix: $Y = [y^{(1)} \; y^{(2)} \; \dots \; y^{(m)}]_{1 \times m}$ $Y \in \mathbb{R}^{1 \times m}$, $Y.shape = (1, m)$

## Algorithm used for binary classification: logistic Regression

For the $i$-th example,

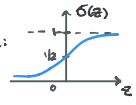
- Given $x^{(i)}$, estimate of output $y^{(i)}$ is $\hat{y}^{(i)} = P_r(y^{(i)}=1 | x^{(i)})$, i.e. what is the probability of the image being a cat given $x^{(i)}$?

→ where does it come from: $P_r(y|x) = \hat{y}^y (1-\hat{y})^{1-y}$

so $P_r(y=1|x) = \hat{y}$, $P_r(y=0|x) = 1-\hat{y}$ $\Rightarrow$ { if $y=1$, $P_r(y=1|x) = \hat{y} \approx 1$; if $y=0$, $P_r(y=0|x) = 1-\hat{y} \approx 1$

- Suppose we have parameters $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$, and we modeled the estimated output as:

$\hat{y}^{(i)} = \sigma(w^T x^{(i)} + b)$, where $\sigma$ is sigmoid function:




which is a single layer NN, later we will show output of more layers NN

Sometimes there might be other notations:
$x_0 = 1$, $x \in \mathbb{R}^{n+1}$
$\hat{y} = \sigma(\theta^T x)$, where
$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}$ $\theta_0 \to b$, $\begin{bmatrix} \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} \to w$

- Using $z^{(i)}$ to denote $w^T x^{(i)} + b$, $\hat{y}^{(i)} = \sigma(z^{(i)}) = \frac{1}{1+e^{-z^{(i)}}}$, where $z^{(i)} = w^T x^{(i)} + b$ { if $z$ is large, $\sigma(z) \to 1$; if $z$ is small, $\sigma(z) \to 0$

- To measure the performance of a single prediction, we define loss function:

$L(\hat{y}, y) = -(y \log \hat{y} + (1-y) \log(1-\hat{y}))$ { if $y=1$, $L(\hat{y}, y) = -\log \hat{y}$ → want $\hat{y} \approx 1$ $\Rightarrow$ want $\log \hat{y}$ large $\Rightarrow$ want $-\log \hat{y}$ small; if $y=0$, $L(\hat{y}, y) = -\log(1-\hat{y})$ → want $\hat{y} \approx 0$ $\Rightarrow$ want $\log(1-\hat{y})$ large $\Rightarrow$ want $-\log(1-\hat{y})$ small

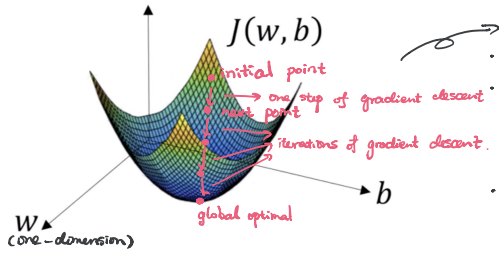
$\Rightarrow$ want $L(\hat{y}, y)$ small

- Then for the entire dataset, we can measure the performance by cost function:

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

→ it's actually the log-likelihood function of $y|x$, i.e. $\sum_{i=1}^{m} \log P_r(y^{(i)}|x^{(i)}) = \sum_{i=1}^{m} [y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log(1-\hat{y}^{(i)})]$

Now we use an algorithm called **Gradient Descent** to find w and b that minimise cost function.




→ how it works:

- J(w,b) has to be a convex function so we can find the global optimal.
- Starting from the initial point (usually w=0, b=0), one step of gradient descent takes a step downhill in the direction of the steepest descent and takes J(w,b) to the next point (new w and b).
- Then by iterating gradient descent, J(w,b) will eventually hit global optimal or near global optimal.

To put this into mathematical representation:

$w := w - \alpha \frac{\partial J(w,b)}{\partial w}$ ⟩ $dw < 0 \Rightarrow$ as $w\uparrow$, $J(w)\downarrow \Rightarrow$ want $w\uparrow \Rightarrow -dw > 0 \Rightarrow$ new $w\uparrow$
$dw > 0 \Rightarrow$ as $w\uparrow$, $J(w)\uparrow \Rightarrow$ want $w\downarrow \Rightarrow -dw < 0 \Rightarrow$ new $w\downarrow$

"learning rate" decides how big a step each iteration takes — represented as "dw", sets the direction of the steepest descent to reach global optimal

$b := b - \alpha \frac{\partial J(w,b)}{\partial b}$ → "db"

To have an explicit expression for dw and db:

Let $a = \sigma(z) = \frac{1}{1+e^{-z}}$ | $\frac{\partial a}{\partial z} = \frac{1}{(1+e^{-z})^2} \cdot e^{-z} = \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}} = a(1-a)$

$$\frac{\partial \mathcal{L}}{\partial a} = \frac{\partial}{\partial a} -(y \log a + (1-y)\log(1-a)) = -\frac{y}{a} + \frac{1-y}{1-a}$$

$$\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial z} = \left(-\frac{y}{a} + \frac{1-y}{1-a}\right) a(1-a) = -y(1-a) + (1-y)a = a - y$$

Then

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial w_p} = (a-y)\cdot x_p, \quad p = 1, 2, \dots n$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial z}\cdot\frac{\partial z}{\partial b} = (a-y)$$

Finally,

$$dw = \frac{\partial J}{\partial w} = \frac{\partial}{\partial w}\frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial w}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = \frac{1}{m}\sum_{i=1}^{m}(a^{(i)} - y^{(i)})\cdot x_p$$

$$db = \frac{\partial J}{\partial b} = \frac{\partial}{\partial b}\frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial b}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = \frac{1}{m}\sum_{i=1}^{m}(a^{(i)} - y^{(i)})$$

Finally, we've come to the exciting **computing part**. Here we will show 2 methods of how to put logistic regression and Gradient Descent algorithm into coding:

- Method 1 for-loop:

initialize with $J = 0$, $dw_1 = 0$, $dw_2 = 0$, ... $dw_n = 0$, $db = 0$

For $i = 1$ to $m$,

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

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

$J \mathrel{+}= -(y^{(i)} \log a^{(i)} + (1-y^{(i)})\log(1-a^{(i)}))$ ← "+=" is the sum of m examples

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

$dw_1 \mathrel{+}= dz^{(i)} x_1$

$dw_2 \mathrel{+}= dz^{(i)} x_2$

⋮

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

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

$J = J/m$ and similarly for $dw_1$ ... $db$

} known as "feedforward"

} known as "backpropagation"

$w_1 := w_1 - \alpha dw_1$

$w_2 := w_2 - \alpha dw_2$

⋮

$w_n := w_n - \alpha dw_n$

$b := b - \alpha db$

- Method 2 vectorization:

Recall $X = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(m)} \\ \vdots & \vdots & & \vdots \\ x_n^{(1)} & x_n^{(2)} & \cdots & x_n^{(m)} \end{bmatrix}_{n \times m} \in \mathbb{R}^{n \times m}$, $w = \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}_{n \times 1} \in \mathbb{R}^n$, $Y = [\, y^{(1)} \cdots y^{(m)} \,]_{1 \times m}$

Then replacing the above codes, we have:

$J = 0$, $dw = np.zeros((n_x, 1))$, $db = 0$

$Z = \underbrace{np.dot(w^T, X)}_{w^T x} + b = [\, z^{(1)} \cdots z^{(m)} \,]_{1 \times m}$ (note $b \in \mathbb{R}$ but this code still works thank to Broadcasting in Python)

$A = \sigma(Z) = [\, a^{(1)} \cdots a^{(m)} \,]_{1 \times m}$

$dZ = A - Y = [\, a^{(1)} - y^{(1)} \cdots a^{(m)} - y^{(m)} \,]_{1 \times m} = [\, dz^{(1)} \cdots dz^{(m)} \,]_{1 \times m}$

$dW = \frac{1}{m} X dZ^T = \frac{1}{m} \begin{bmatrix} x_1^{(1)} & \cdots & x_1^{(m)} \\ \vdots & & \vdots \\ x_n^{(1)} & \cdots & x_n^{(m)} \end{bmatrix}_{n \times m} \begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix}_{m \times 1} = \frac{1}{m} \begin{bmatrix} \sum_i x_1^{(i)} dz^{(i)} \\ \vdots \\ \sum_i x_n^{(i)} dz^{(i)} \end{bmatrix}_{n \times 1} = \begin{bmatrix} dw_1 \\ \vdots \\ dw_n \end{bmatrix}_{n \times 1}$

$db = \frac{1}{m} \cdot np.sum(dZ)$

$w := w - dw$

$b := b - db$

$J = -\frac{1}{m} \cdot (np.dot(Y, (\log A)^T) + np.dot(1-Y, (\log(1-A))^T))$

Important: Always try to avoid coding with explicit expressions (i.e. for-loops) because it's too inefficient comparing to vectorization. For more details about vectorization, please view "Vectorization_and_Broadcasting.ipynb".