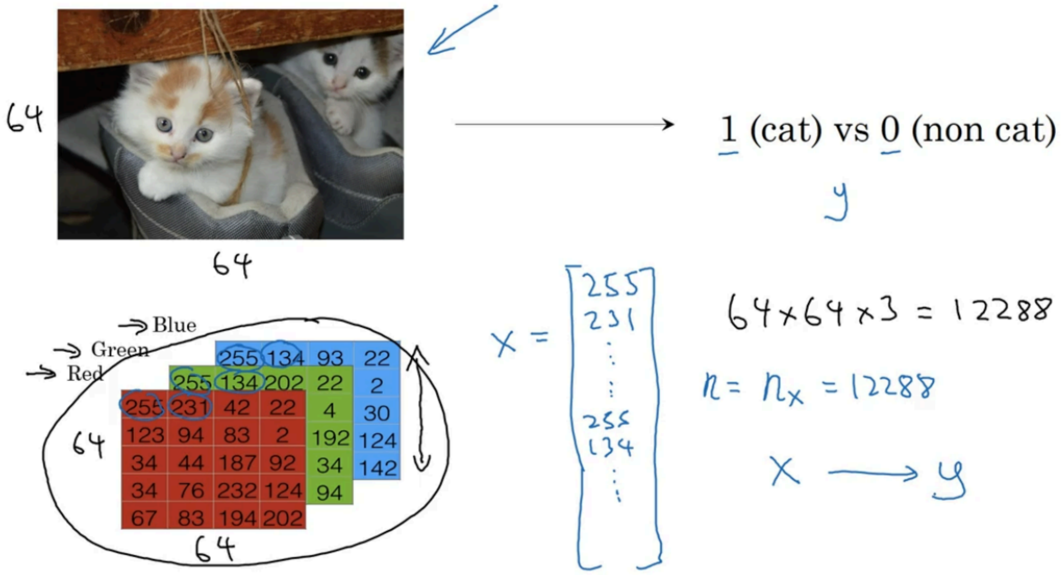
Week 2: Basics of Neural Network Programming

Binary Classification:



- An image of size **64 × 64** with 3 color channels (RGB) can be represented as a 3D array (tensor):

$$Image\ size = 64 \times 64 \times 3$$

- Each pixel has **3 values** (Red, Green, Blue), ranging from **0–255**.
- To use in ML/DL models, we flatten the image into a vector:

$$n_x = 64 \times 64 \times 3 = 12288$$

So each image is represented as:

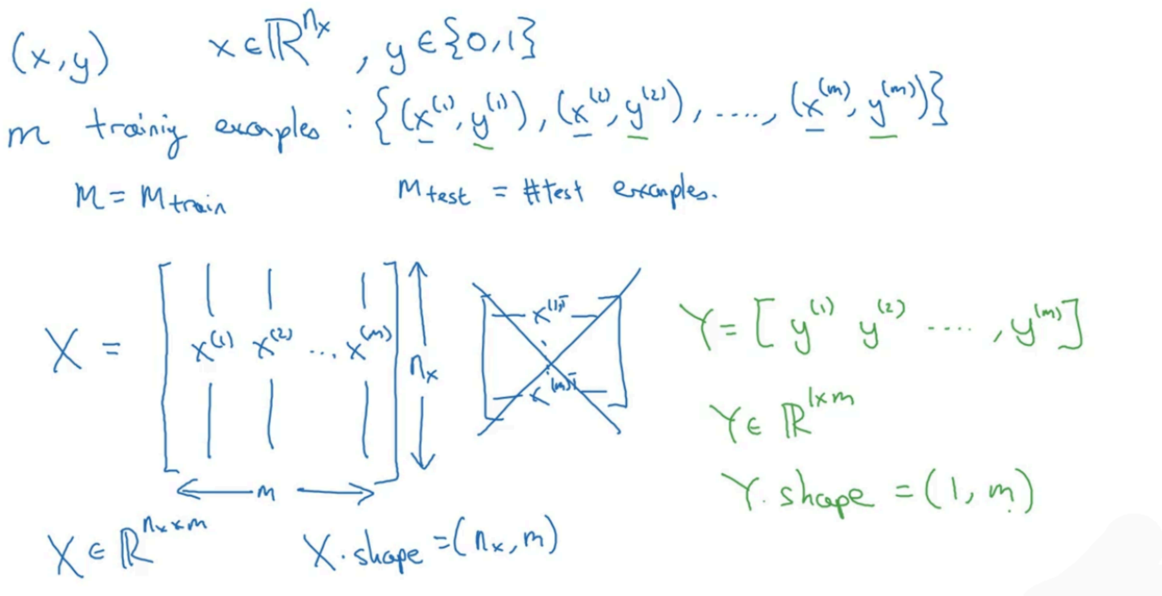
$$x \in \mathbb{R}^{12288}$$

- Label y:
  - $y = 1 \rightarrow Cat$
  - $y = 0 \rightarrow Non - cat$

Thus, the task becomes:

$$x \longrightarrow y$$

Training Data Representation



- Each training example:

$$(x^{(i)}, y^{(i)})$$

where

$$x^{(i)} \in \mathbb{R}^{n_x}, \quad y^{(i)} \in \{0, 1\}$$

- For m training examples:

$$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$$

Matrix Representation

- Collect all features in one matrix X:

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

$$X \in \mathbb{R}^{n_x \times m}, \quad X.\text{shape} = (n_x, m)$$

- Collect all labels in one row vector Y:

$$Y = [y^{(1)}, y^{(2)}, \dots, y^{(m)}]$$

$$Y \in \mathbb{R}^{1 \times m}, \quad Y.\text{shape} = (1, m)$$

Neural Network Notation

- **Weights & Biases**
  - w: Weight vector
  - b: Bias term
- **Forward Propagation**
  - Linear component:

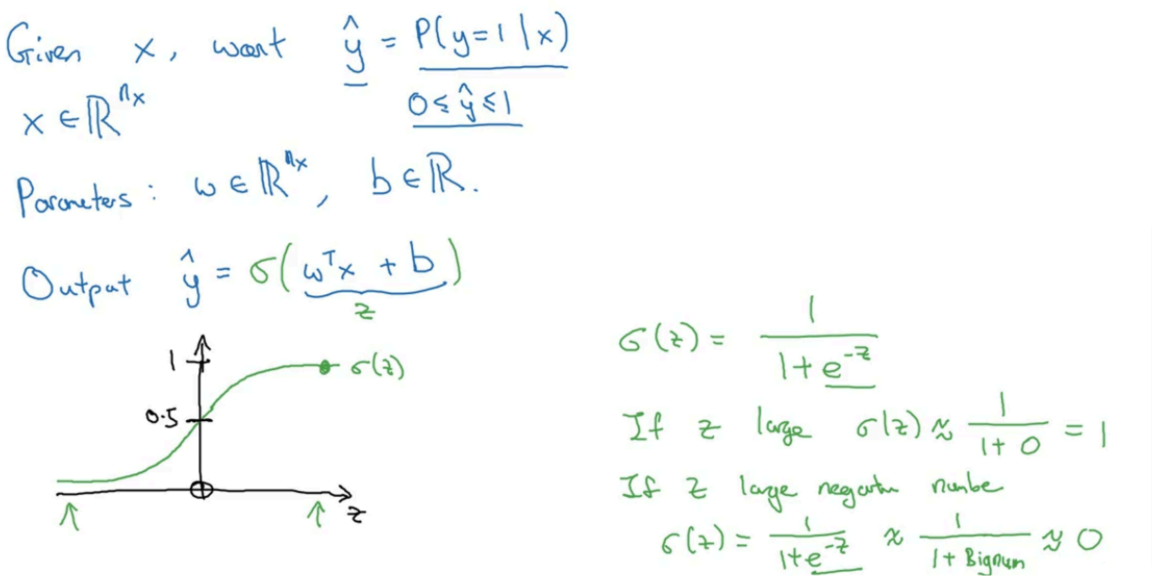
$$z = w^T x + b$$

  - Activation (using sigmoid as example):

$$a = \sigma(z) = \frac{1}{1 + e^{-z}}$$

- **Network Structure**
  - L: Total number of layers in the network
  - $n^{[l]}$: Number of units in layer l
  - $w^{[l]}$: Weight matrix of layer l
  - $b^{[l]}$: Bias vector of layer l
  - $a^{[l]}$: Activations of layer l

Logistic Regression:



Cost Function Notation

- **Loss for Single Example:**

  For logistic regression:

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

- **Overall Cost Function:**

$$J = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

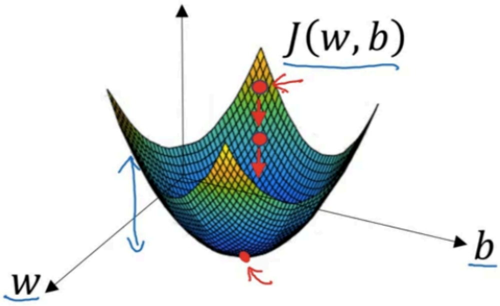
- **Learning Rate:**
  - α: Controls the step size of gradient descent

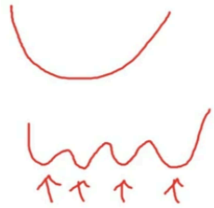
Gradient Descent:

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

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

Want to find w, b that minimize $J(w,b)$





Gradient Descent is an optimization algorithm used to minimize the cost function by iteratively adjusting the parameters (weights and biases) in the direction of steepest descent of the cost function.

Key Concepts of Gradient Descent:

- **Goal:** Find the values of parameters (w, b) that minimize the cost function J(w, b)
- **Algorithm steps:**
  - Initialize parameters (w, b) with random or zero values
  - Calculate the gradient (partial derivatives) of the cost function with respect to each parameter
  - Update each parameter by subtracting the learning rate multiplied by its gradient
  - Repeat until convergence (when the cost function stops decreasing significantly)
- **Update rule:**
  - $w := w - \alpha * \frac{\partial J(w,b)}{\partial w}$
  - $b := b - \alpha * \frac{\partial J(w,b)}{\partial b}$
  - Where α is the learning rate that determines the step size
- **Types of Gradient Descent:**

- **Batch Gradient Descent:** Uses the entire training set to compute gradients in each iteration
- **Stochastic Gradient Descent (SGD):** Uses a single random example to compute gradients in each iteration
- **Mini-batch Gradient Descent:** Uses a small random subset of training examples to compute gradients in each iteration

- **Learning rate (α):**
  - Too small: Slow convergence
  - Too large: May overshoot the minimum or diverge
  - Optimal: Leads to fastest convergence without overshooting
- **Convergence criteria:**
  - Small change in cost function between iterations
  - Small gradients (close to zero)
  - Reaching a maximum number of iterations
- **Challenges:**
  - Local minima: Algorithm may get stuck in a local minimum instead of finding the global minimum
  - Saddle points: Points where gradients are zero but not a minimum
  - Plateaus: Regions where the gradient is very small, causing slow progress

Gradient Descent for Logistic Regression:

$$\frac{\partial J(w,b)}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)}) \cdot x_j^{(i)}$$

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

These derivatives are then used in the update rules to adjust the parameters in the direction that reduces the cost function.

Notation in Python Code

When implementing gradient descent in Python, we typically use the following variable names for derivatives:

- **dw:** Represents the partial derivative of the cost function with respect to weights (∂J/∂w)
- **db:** Represents the partial derivative of the cost function with respect to bias (∂J/∂b)

For neural networks with multiple layers, we often use:

- **dW[l]:** Gradient of the cost function with respect to weights in layer l
- **db[l]:** Gradient of the cost function with respect to biases in layer l
- **dA[l]:** Gradient of the cost function with respect to activations in layer l
- **dZ[l]:** Gradient of the cost function with respect to linear output in layer l

Example of gradient descent update in Python code:

```
# Compute gradients
dw = (1/m) * np.dot(X, (A-Y).T)
db = (1/m) * np.sum(A-Y)

# Update parameters
w = w - learning_rate * dw
b = b - learning_rate * db
```