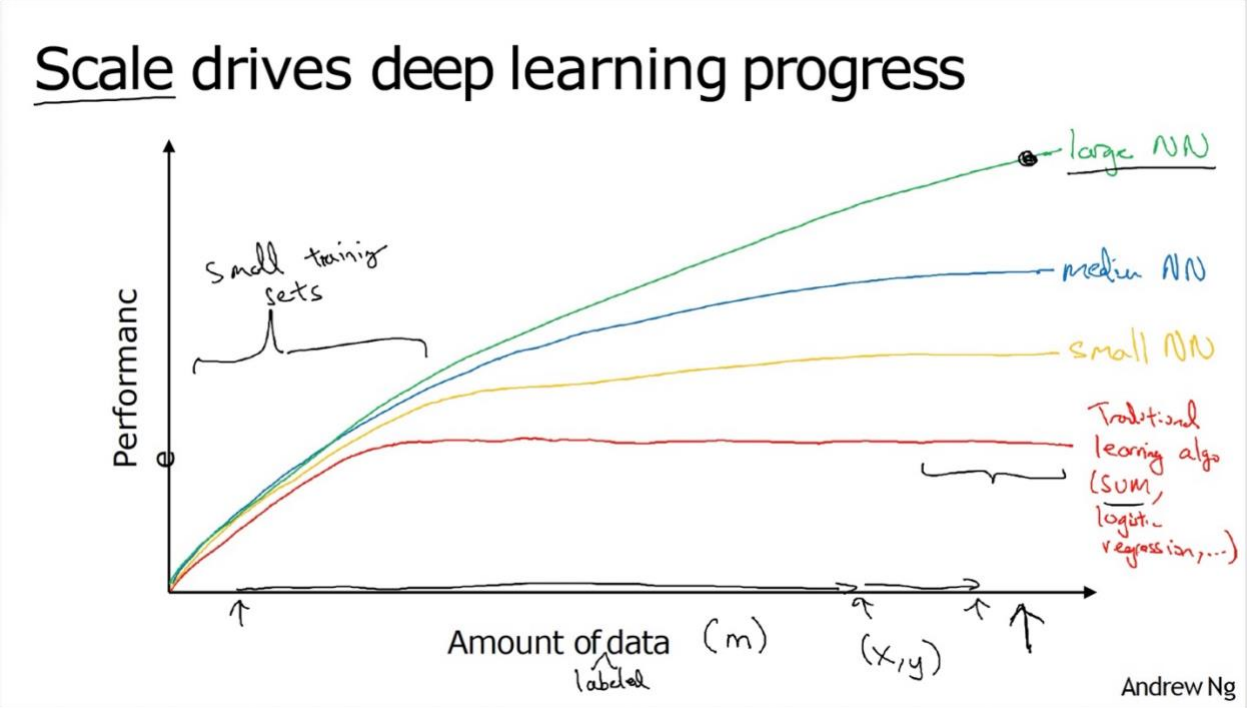
Deep Learning

1. Neural Network Definition

- **Detailed Explanation:** A neural network is a function approximator at its core. It learns to map inputs to outputs by adjusting the connections between neurons. These connections have weights, and the neurons apply mathematical operations to the inputs they receive. The network's architecture (number of layers, number of neurons per layer, how neurons are connected) and the values of the weights determine its behavior. Through a learning process, the network adjusts these weights to improve its performance on a specific task.

2. Why Deep Learning is Taking Off (Explanation of Graph Trends)

- **Graph Trend:** Imagine a graph where the x-axis represents the amount of training data you feed into a learning algorithm, and the y-axis represents the algorithm's performance (e.g., accuracy).



- **Traditional Algorithms:**
  - For traditional machine learning algorithms (like logistic regression and support vector machines), the graph typically shows an initial increase in performance as you provide more training data.
  - However, at some point, the performance improvement plateaus. Adding significantly more data beyond this point doesn't lead to much better results. These algorithms have limitations in their ability to extract complex patterns from very large datasets.
- **Neural Networks:**
  - Small neural networks might behave similarly to traditional algorithms.
  - Deeper (more layers) and larger (more neurons) neural networks can leverage much larger amounts of data. Their performance tends to increase more consistently with data size.
  - Deep learning models can learn hierarchical representations of data, automatically extracting relevant features, which is a key advantage when dealing with complex, high-dimensional data.
- **Key Takeaways:**
  - Deep learning's power comes from its ability to exploit massive datasets.

- The availability of big data (due to increased digitization), combined with increased computing power, has fueled the deep learning revolution.
- Deep learning models can automatically learn intricate features from the data, reducing the need for manual feature engineering.

3. ReLU Function

- **Definition:** ReLU (Rectified Linear Unit) is an activation function widely used in neural networks. It's a simple yet effective function that introduces non-linearity, which is crucial for neural networks to learn complex relationships.
- **Mathematical Representation:** The ReLU function is defined as:
  - $f(x) = \max(0, x)$

  This means:

  - If $x > 0$, then $f(x) = x$
  - If $x \leq 0$, then $f(x) = 0$

How ReLU Improved Gradient Descent:

1. Neural Networks and Backpropagation

- **Neural Network Structure:** Neural networks consist of layers of interconnected nodes (neurons). Information flows from the input layer through hidden layers to the output layer to make a prediction.
- **Backpropagation:** The primary algorithm for training neural networks is backpropagation. It works by:
  1. Calculating the "error" (difference between the network's prediction and the actual value).
  2. Propagating this error backward through the network, layer by layer.
  3. Calculating the gradient of the error concerning each weight in the network. The gradient indicates how much each weight needs to be adjusted to reduce the error.
  4. Updating the weights based on these gradients, typically using an optimization algorithm like gradient descent.

2. The Role of Activation Functions

- **Non-linearity:** Activation functions introduce non-linearity into the network. This is essential because real-world data is often non-linear, and neural networks need to learn complex, non-linear relationships. Without non-linearity, multiple layers would be equivalent to a single layer, limiting the network's power.
- **Common Activation Functions:** Historically, sigmoid and tanh were popular activation functions.
  - Sigmoid: $\sigma(x) = 1 / (1 + \exp(-x))$
  - Tanh: $\tanh(x) = (\exp(x) \exp(-x)) / (\exp(x) + \exp(-x))$

3. What is the Vanishing Gradient Problem?

- **Gradients in Deep Networks:** In deep networks (networks with many layers), backpropagation involves multiplying many gradients together, layer by layer, as the error signal is passed backward.
- **Sigmoid and Tanh Saturation:** Sigmoid and tanh functions have a key characteristic: their gradients are close to zero when the input is very large or very small. We say these functions "saturate" in these regions.
  - For Sigmoid, the maximum value of the derivative is 0.25
  - For Tanh, the maximum value of the derivative is 1
- **The Multiplication Effect:** If the gradients in each layer are less than 1, multiplying many of these small gradients together results in an exponentially decreasing gradient as we move backward through the layers. In very deep networks, the gradients in the earlier layers can become extremely close to zero.
- **Slow Learning or Stalling:** When gradients are very small, the weights in the earlier layers are updated very slowly or not at all. These layers effectively stop learning, even though they are crucial for the network to learn complex features. The network becomes difficult to train effectively.

4. Why ReLU Helps

- **ReLU's Linear Region:** ReLU's key advantage is that for positive inputs, its gradient is always 1. This prevents the gradient from shrinking as it passes through ReLU layers.
- **Maintaining Gradient Flow:** By maintaining a stronger gradient, ReLU helps the error signal propagate more effectively to the earlier layers, allowing them to continue learning.
- **Improved Training:** This leads to significantly faster and more effective training of deep neural networks.

In Summary

The vanishing gradient problem arises from the repeated multiplication of small gradients during backpropagation in deep networks, especially when using saturating activation functions like sigmoid or tanh. This hinders the learning of earlier layers. ReLU mitigates this by having a constant gradient for positive inputs, facilitating better gradient flow and enabling the training of much deeper architectures.

The Dying ReLU Problem:

The "Dying ReLU" problem is a specific issue encountered when using the ReLU (Rectified Linear Unit) activation function in neural networks. Here's a detailed explanation:

1. ReLU Basics

- As a reminder, the ReLU activation function is defined as:
    - $f(x) = \max(0, x)$
- This means:
    - If $x > 0$, then $f(x) = x$
    - If $x \leq 0$, then $f(x) = 0$
- ReLU is computationally efficient and helps mitigate the vanishing gradient problem for positive inputs.

2. The Dying ReLU Phenomenon

- **Zero Gradient for Negative Inputs:** The crucial point is that for any negative input ($x < 0$), the output of ReLU is zero, and the gradient (the slope of the function) is also zero.
- **Neurons Stuck in Zero State:** During training, if a neuron's weights are updated in a way that causes its input to ReLU to consistently become negative, the neuron will continuously output zero. Because the gradient is zero, there will be no further weight updates for that neuron.
- **Inactivity and "Death":** The neuron essentially becomes inactive, or "dead," as it no longer contributes to the learning process. It's stuck in a state where it always outputs zero, regardless of the input.
- **Impact on the Network:** If a significant portion of neurons in a layer "die," the network's ability to learn and represent complex patterns is severely diminished.

3. Causes of Dying ReLU

- **Large Negative Bias:** If a neuron has a large negative bias, its input may consistently fall into the negative region, leading to death.
- **High Learning Rates:** Large learning rates during training can cause drastic weight updates, potentially pushing neurons into the negative region and causing them to die.
- **Unsuitable Initialization:** Poor weight initialization can also contribute to neurons becoming inactive early in training.

4. Mitigation Strategies

- **Leaky ReLU:** Leaky ReLU is a variant that addresses the dying ReLU problem. It introduces a small, non-zero slope for negative inputs:
    - $f(x) = x$ if $x > 0$
    - $f(x) = \alpha x$ if $x \leq 0$ (where α is a small constant, e.g., 0.01)
    - This ensures that even for negative inputs, there is a small gradient, preventing neurons from becoming completely inactive.
- **Parametric ReLU (PReLU):** PReLU is similar to Leaky ReLU, but the slope for negative inputs (α) is learned as a parameter during training.

- **Careful Initialization:** Using appropriate weight initialization techniques (e.g., He initialization) can help prevent neurons from dying early in training.
- **Adaptive Learning Rates:** Employing adaptive learning rate algorithms (e.g., Adam, RMSprop) can help regulate weight updates and reduce the likelihood of neurons dying.

In Summary

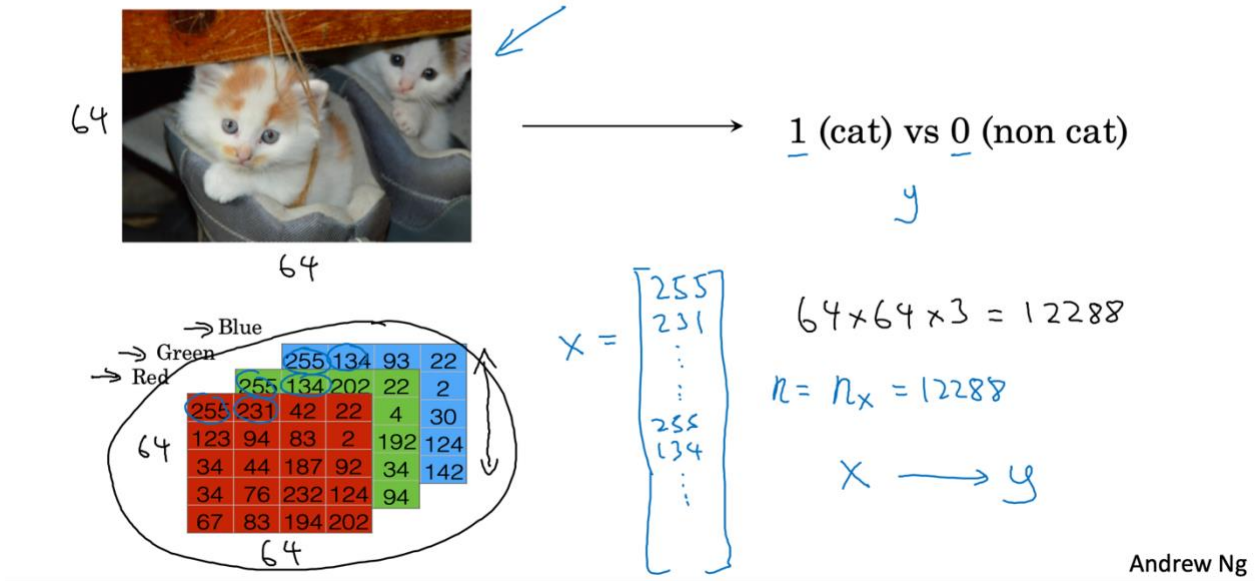
The Dying ReLU problem is a scenario where neurons using ReLU become permanently inactive due to consistently receiving negative inputs, leading to a zero gradient and preventing further learning. Leaky ReLU and other variants, along with careful initialization and learning rate strategies, can help mitigate this issue.

ReLU vs. Sigmoid:

Feature	**ReLU (Rectified Linear Unit)**	**Sigmoid**
Mathematical Representation	f(x) = max(0, x)	σ(x) = 1 / (1 + exp(-x))
Output Range	[0, ∞)	(0, 1)
Gradient Flow	Constant gradient of 1 for positive inputs, avoiding vanishing gradients. Can experience "dying ReLU" problem (zero gradient for negative inputs).	Prone to vanishing gradients, especially for very large or small inputs. Gradient is always between 0 and 0.25
Computation	Computationally efficient (simple max operation).	Computationally more expensive (involves exponentiation).
Sparsity	Introduces sparsity (many zero activations), which can improve efficiency.	Produces dense activations (mostly non-zero).
Vanishing Gradient Problem	Mitigates for positive inputs.	Susceptible.
"Dying ReLU" Problem	can cause it.	Does not occur.
Use Cases (Hidden Layers)	Highly favored in hidden layers of deep neural networks.	Historically used, but less common in deep networks due to vanishing gradients.
Use cases (output layers)	not often for final layers, unless the regression output must be a positive number.	useful when the output needs to be a probability between 0 and 1.
Derivative Function	1 when x > 0 else 0	σ(x)*(1-σ(x))

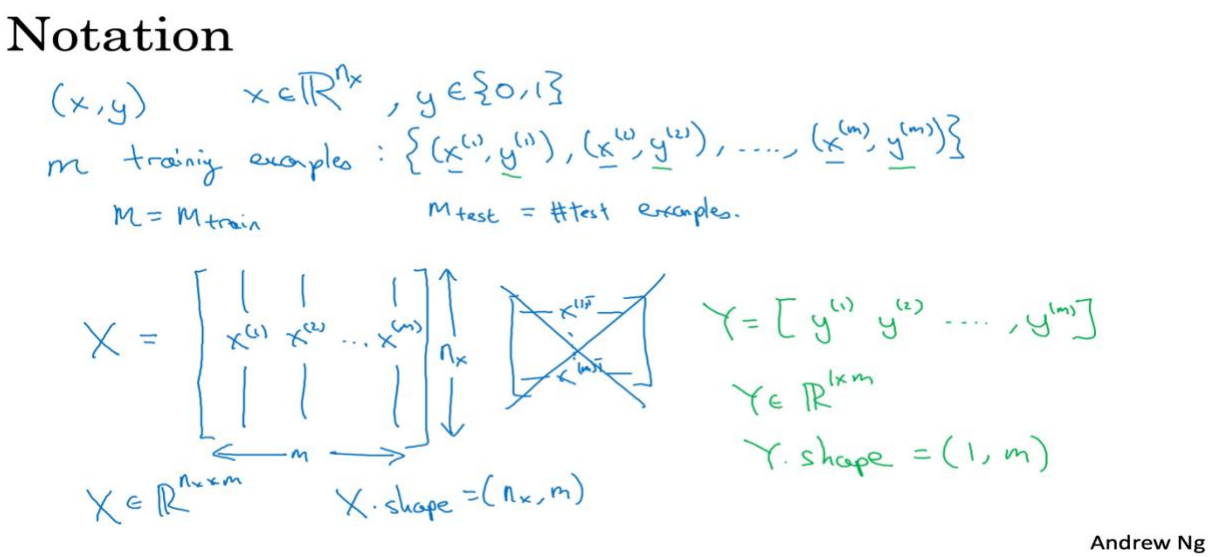
Logistic Regression for Binary Classification

- **Binary Classification**: Logistic regression is an algorithm designed for binary classification problems. In this context, the goal is to categorize an input into one of two possible outcomes.
- **Example:** A classic example is image recognition, where the task is to classify an image as either belonging to a specific category (e.g., "cat") or not.
    - Input: An image.
    - Output: A label (Y) indicating the category: 1 for "cat," 0 for "not-cat."
- **Image Representation:**
    - Images are represented digitally as three matrices, corresponding to the red, green, and blue color channels.
    - Each matrix stores pixel intensity values.
    - If an image has dimensions of 64 pixels by 64 pixels, each color channel is a 64x64 matrix.



- **Feature Vector (X):**
  - To process an image with a learning algorithm, the pixel intensity values from the color channel matrices are unrolled into a feature vector (x). This involves concatenating the rows (or columns) of each color channel matrix into a single long vector.
  - For a 64×64 pixel image with 3 color channels (RGB), the feature vector x has a dimension of: $n_x = 64 \times 64 \times 3 = 12{,}288$
  - Notation: $n_x = 12{,}288$: Dimension of the input feature vector. n (lowercase) is sometimes used interchangeably with n_x.

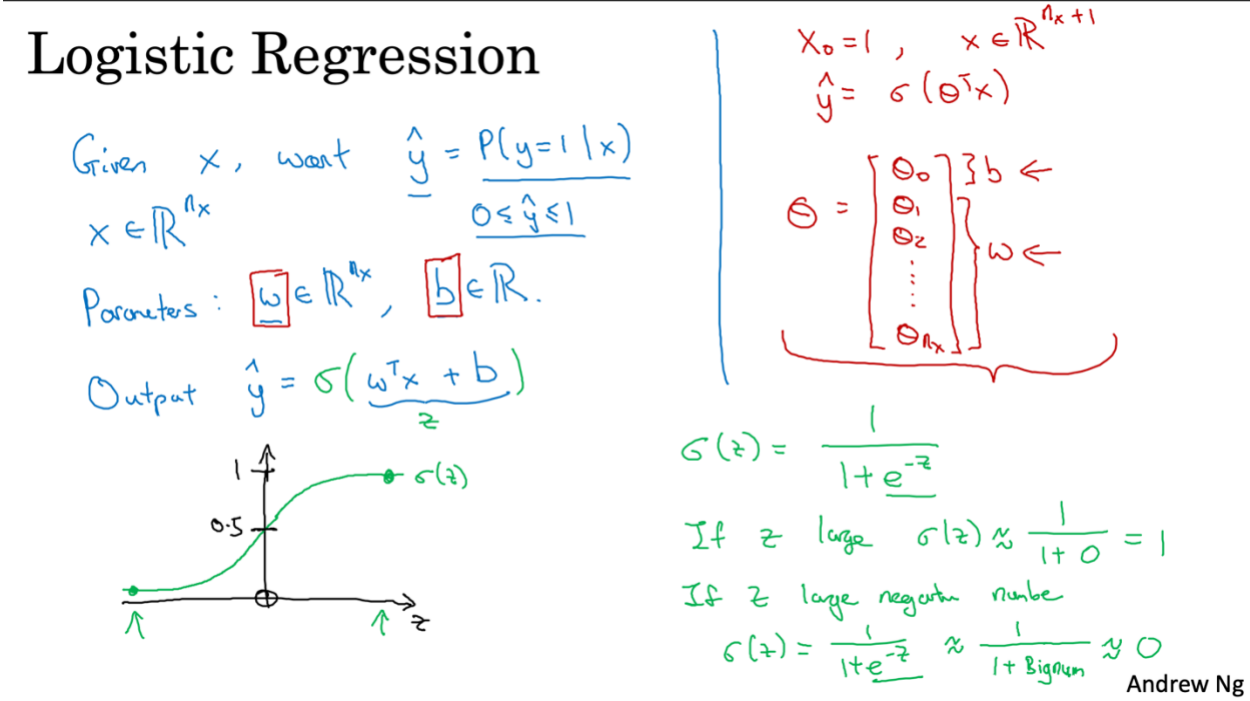
Notation:



- **Training Example:**
  - A single training example is a pair (x, y), where:
  - $x \in \mathbb{R}^{n_x}$ is the feature vector.(i.e, n_x dimensional)
    $y \in \{0, 1\}$ is the label.
- **Training Set:**
  A training set consists of m training examples:
  $\{ (x^1, y^1), (x^2, y^2), ..., (x^m, y^m) \}$
  where m: Number of training examples.
  m_{train}: Number of training examples in the training set.
  m_{test}: Number of examples in the test set.

- **Input Matrix (X):**
  The input feature vectors are stacked column-wise to form the matrix X:
  $X = [x^1 \ x^2 \ ... \ x^m]$
  Shape: $X \in \mathbb{R}^{n_x \times m}$
  Python Notation: X.shape = (n_x, m)
- **Output Matrix (Y):**
  The labels are organized into a row vector Y:
  $Y = [y^1 \ y^2 \ ... \ y^m]$
  Shape: $Y \in \mathbb{R}^{1 \times m}$
  Python Notation: Y.shape = $(1, m)$

Logistic Regression:



Binary Classification: Logistic regression is a learning algorithm used for binary classification problems, where the output label y can only take two values: 0 or 1.

Prediction (ŷ): The algorithm aims to output a prediction (denoted as ŷ), which represents the estimated probability that the output y is equal to 1, given the input features x.
Formally, ŷ = P(y = 1 | x)

Input Features (x): The input to the algorithm is a feature vector x, which can represent various data types, such as pixel values of an image.

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

Parameters:

The parameters of logistic regression consist of:

$w \in \mathbb{R}^{n_x}$: A weight vector

$b \in \mathbb{R}$: A bias term

Output Calculation: A linear combination of the input features and weights is calculated: $z = w^T x + b$

The sigmoid function (σ) is applied to this linear combination to produce the output ŷ: $\hat{y} = \sigma(z)$

Sigmoid Function: The sigmoid function (σ) is defined as: $\sigma(z) = 1 / (1 + e^{-z})$

It maps any real number to a value between 0 and 1, making it suitable for representing probabilities.

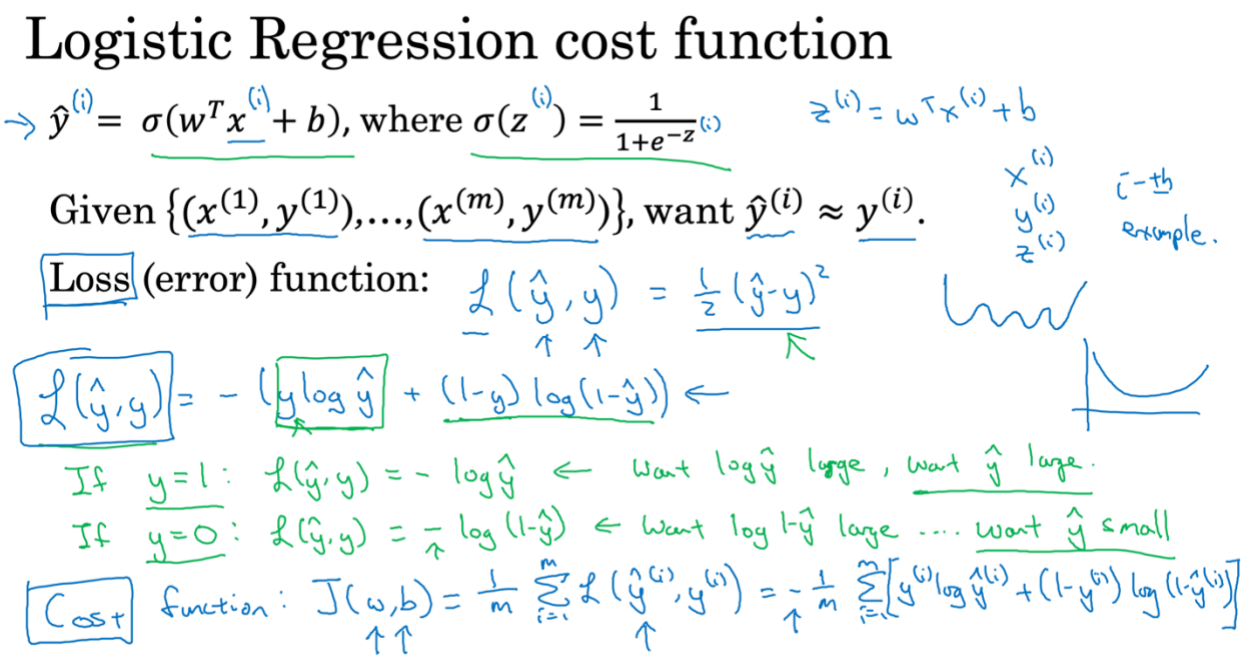
When z is very large, $\sigma(z) \approx 1$

When z is very small (a large negative number), $\sigma(z) \approx 0$

Alternative Notation: By defining $x_0 = 1$ and $\Theta \in \mathbb{R}^{n_x+1}$ (where $\Theta = [b, w_1, w_2, ..., w_n^x]$), we can write:

$\hat{y} = \sigma(\Theta^T x)$

Cost Function in Logistic Regression



Andrew Ng

The prediction ŷ for a given input feature vector x is calculated as: $\hat{y} = \sigma(w^T x + b)$, where σ is the sigmoid function. $\sigma(z) = 1 / (1 + e^{-z})$

Goal: The goal is to learn the parameters w (weight vector) and b (bias) so that the predictions $\hat{y}^{(i)}$ on the training set are as close as possible to the true labels $y^{(i)}$.

Loss Function: The loss function L(ŷ, y) measures how well the algorithm performs on a single training example (x, y).

In logistic regression, the loss function used is:

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

If y = 1, the loss function tries to make ŷ as large as possible.

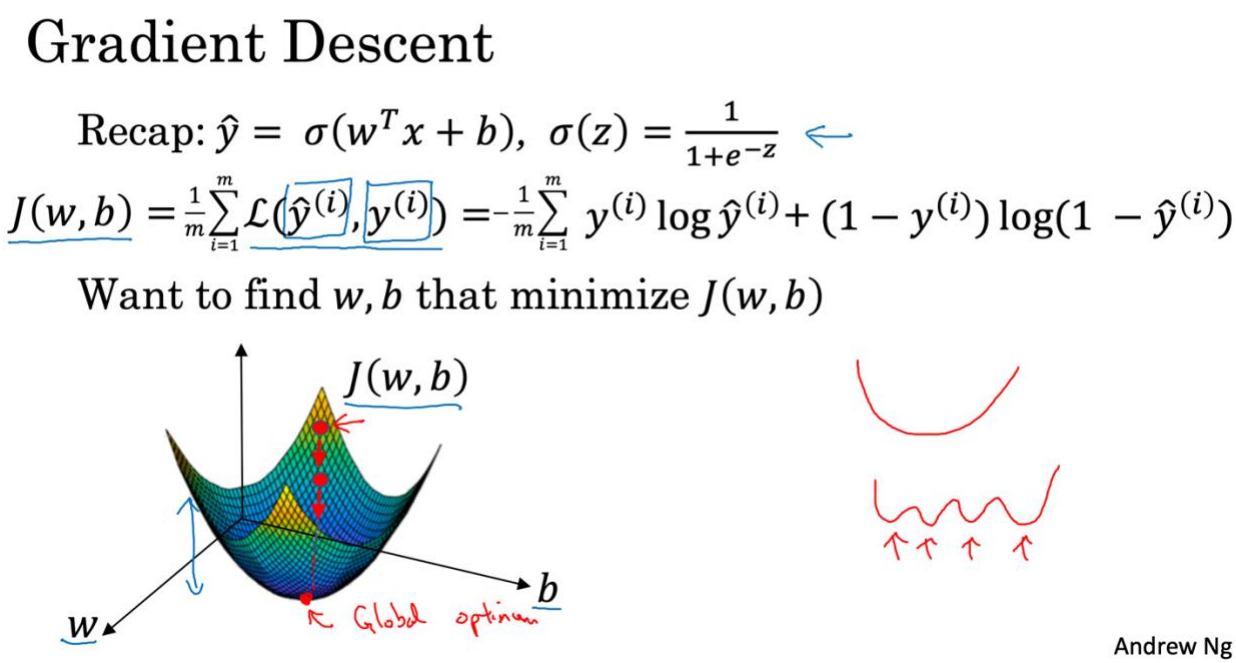
If y = 0, the loss function tries to make ŷ as small as possible.

Cost Function: The cost function J(w, b) measures the overall performance of the algorithm on the entire training set. It is the average of the loss functions over all m training examples:

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

Training: The objective of training a logistic regression model is to find the parameters w and b that minimize the cost function J(w, b).

Gradient Descent



Objective: The goal of gradient descent is to find the values of parameters w and b that minimize the cost function J(w, b).

Cost Function: The cost function J(w, b) measures how well the parameters w and b perform on the entire training set. It's calculated as the average of the loss function over all training examples.

The loss function measures how well the algorithm's prediction $\hat{y}^{(i)}$ compares to the true label $y^{(i)}$ for a single training example.

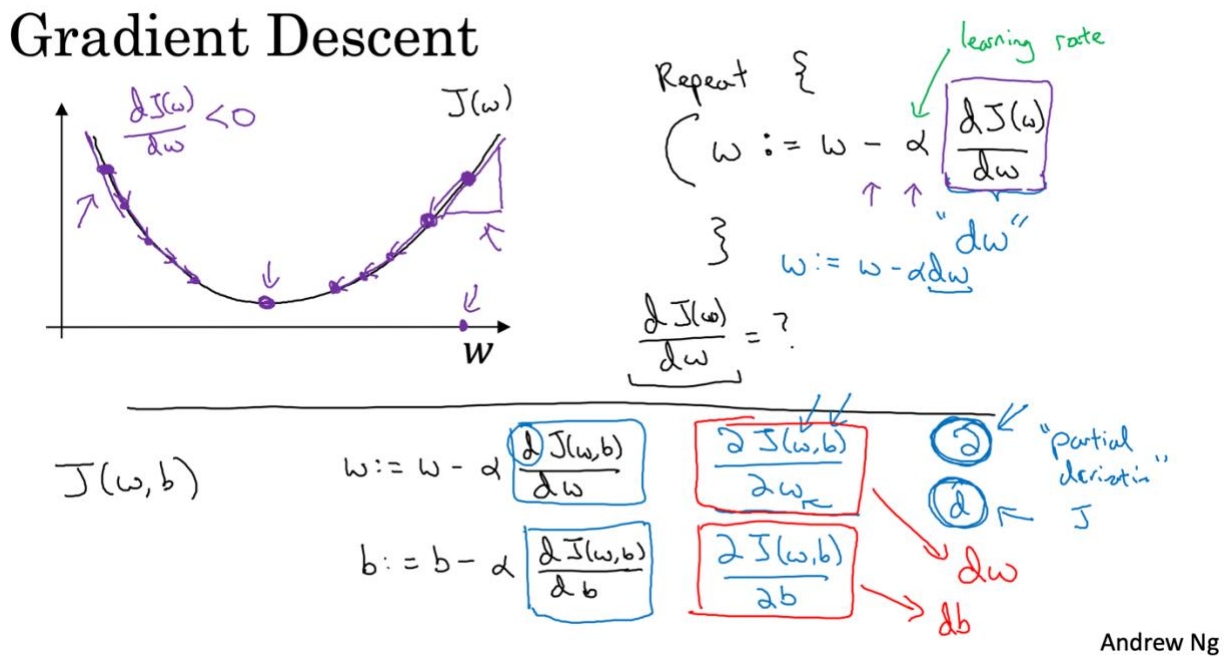
Visualization: The cost function J(w, b) can be visualized as a surface where the horizontal axes represent the parameters w and b, and the vertical axis represents the value of J.

For logistic regression, this surface is a convex function (a bowl shape), meaning it has a single global minimum.

Algorithm: Gradient descent starts by initializing w and b to some initial values.

Then, it iteratively updates w and b by taking steps in the direction of the steepest downhill slope of the cost

function.



Each step moves the parameters closer to the global minimum of J.

Update Rule: The update rule for gradient descent is:

w := w α (∂J(w, b) / ∂w)

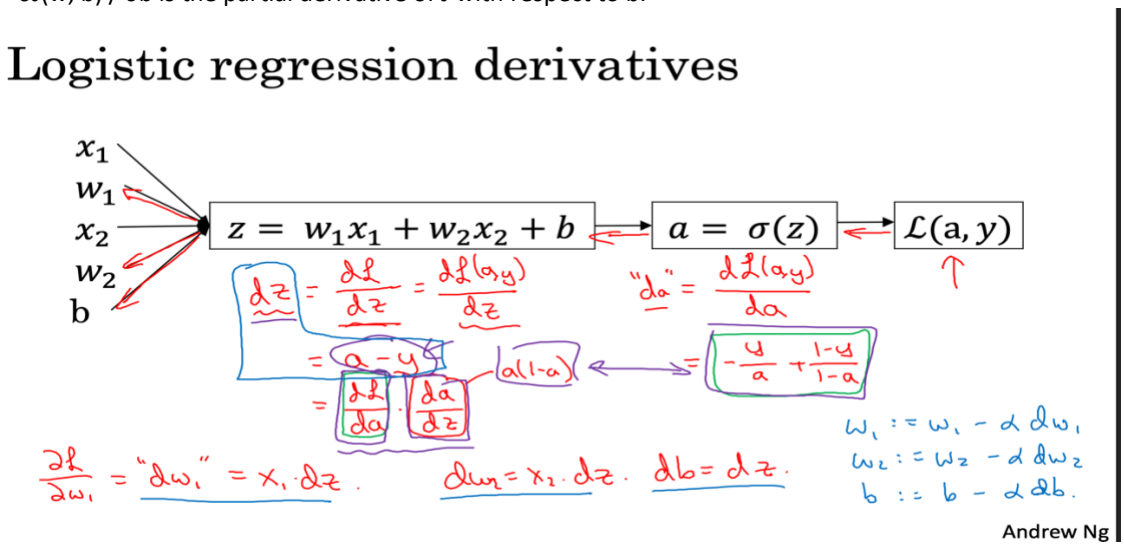
b := b α (∂J(w, b) / ∂b)

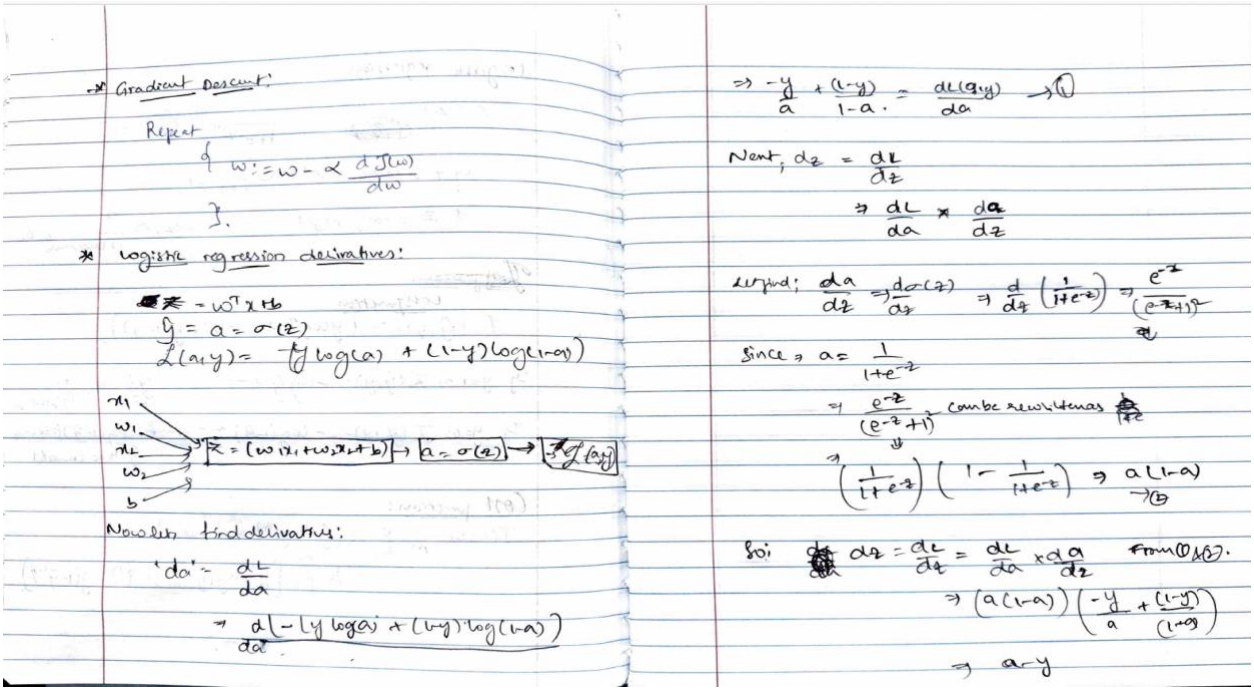
Where:

α is the learning rate, controlling the step size.

∂J(w, b) / ∂w is the partial derivative of J with respect to w.

∂J(w, b) / ∂b is the partial derivative of J with respect to b.

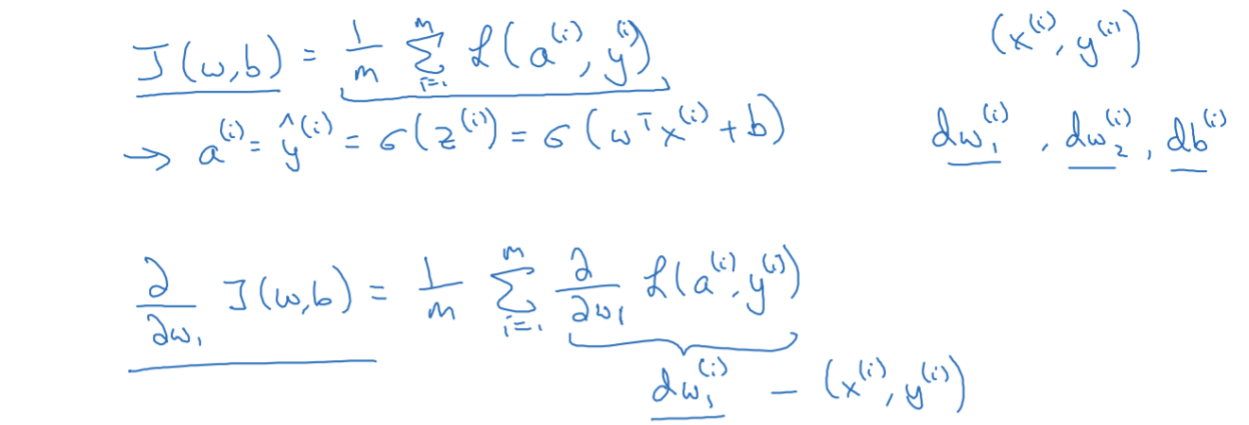




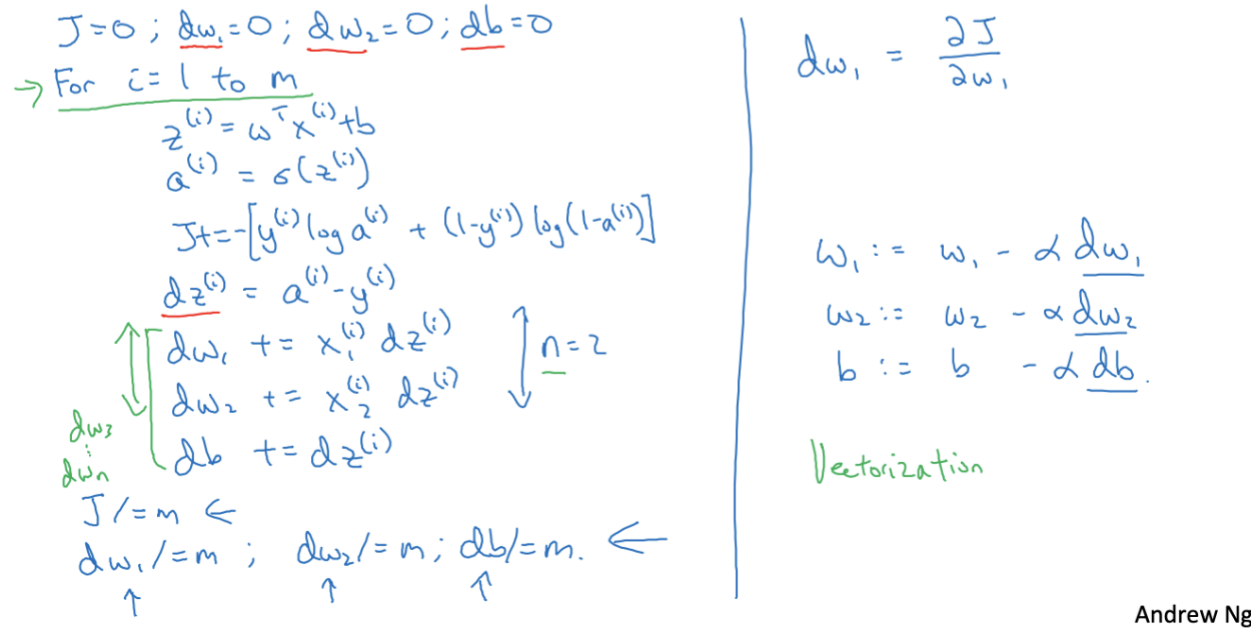
So, if we think Logistic Regression with a Neural Network mindset, here is what is happening:

```
# 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)) ))

# YOUR CODE ENDS HERE
# BACKWARD PROPAGATION (TO FIND GRAD)
#(≈ 2 lines of code)
# dw = ...
# db = ...
# YOUR CODE STARTS HERE
dw = 1/m*(np.dot(X,(A-Y).T))
db = 1/m*(np.sum(A-Y))
# YOUR CODE ENDS HERE
cost = np.squeeze(np.array(cost))
```



Andrew Ng



Andrew Ng

Vectorization

Definition:

Vectorization is the technique of eliminating explicit for loops in code.

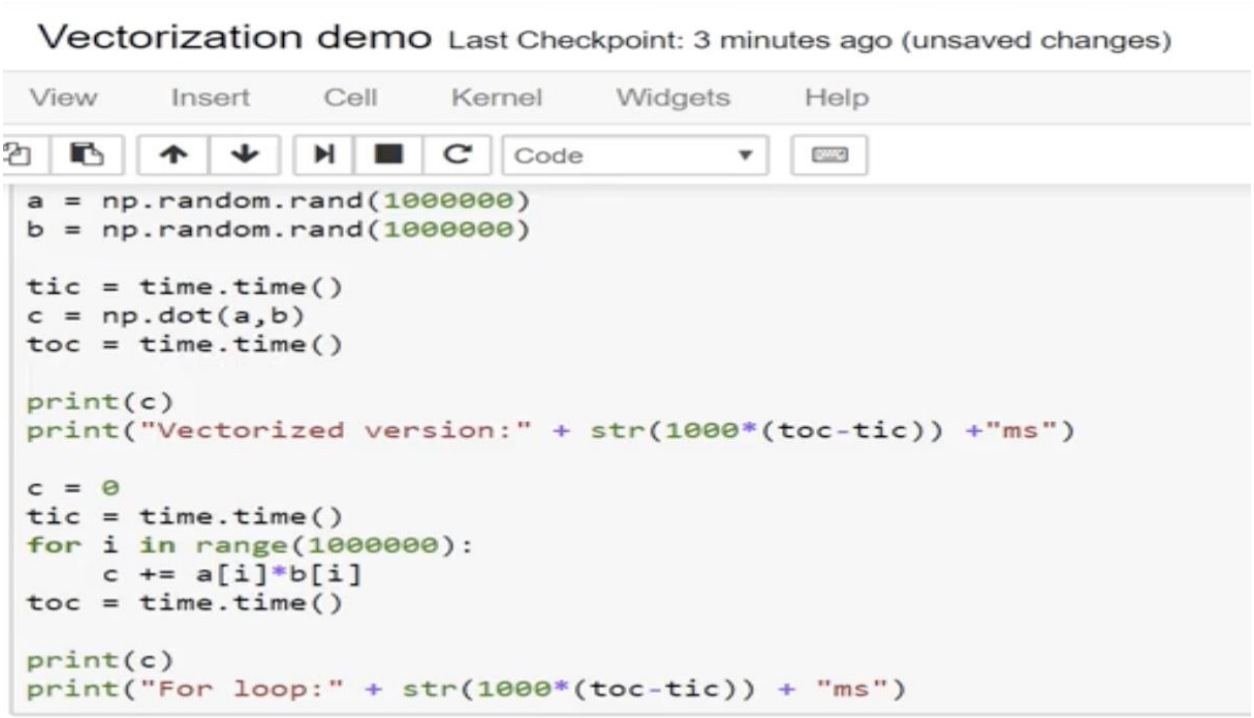
Importance: In deep learning, training is often performed on large datasets. Vectorization makes code run much faster, which is crucial for efficient training on these datasets.

Example: In logistic regression, a key computation is calculating $z = w^T x + b$, where w and x are vectors.

Non-vectorized implementation:

z = 0
For i in range(n_x):
    z += w_i * x_i
z += b

Vectorized implementation:
In Python (using numpy):
z = np.dot(w, x) + b
This avoids loops and is much faster.
Demonstration:



Underlying Principle:
CPUs and GPUs have Single Instruction, Multiple Data (SIMD) instructions that allow for parallel computations. Vectorized code leverages these instructions to perform operations on multiple data elements simultaneously, greatly improving efficiency.
Rule of Thumb:
Avoid explicit for loops whenever possible to write efficient code.

Vectorization of Gradient Descent in Logistic Regression

1. Vectorization in Logistic Regression
Goal: To efficiently compute logistic regression calculations over an entire training set without explicit for-loops.
Benefit: Significantly speeds up code execution, crucial for large datasets.
2. Vectorizing Forward Propagation
Input Matrix (X): Training inputs are stacked column-wise into matrix X (n_x × m):
X = np.array([[x1], [x2], ..., [xm]]) # Conceptual
X.shape # (nx, m)
Calculating Z:

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

Calculating A (Activations):

A = sigmoid(Z) # Sigmoid applied element-wise

3. Vectorizing Backward Propagation

Calculate dZ:

dZ = A Y

4. Vectorized Gradient Descent

Update Parameters: (Using gradient descent)

w = w alpha * dw

b = b alpha * db

Calculate db:

db = (1 / m) * np.sum(dZ)

Calculate dw:

dw = (1 / m) * np.dot(X, dZ.T)

6. Initial (Non-Vectorized) Implementation

```
def initial_logistic_regression(X, Y, w, b, alpha):
    m = X.shape[1]
    nx = X.shape[0]
    dw = np.zeros((nx, 1))
    db = 0
    for i in range(m):
        z = np.dot(w.T, X[:, i]) + b
        a = sigmoid(z)
        dz_i = a - Y[0, i]
        for j in range(nx):
            dw[j, 0] += X[j, i] * dz_i
        db += dz_i
    dw = (1 / m) * dw
    db = (1 / m) * db
    w = w - alpha * dw
    b = b - alpha * db
    return dw, db, w, b
```

How it works:

Python automatically expands the smaller array to match the shape of the larger array

Then, it performs the operation element-wise

Specific Broadcasting Rules:

If you have an (m, n) matrix and operate with a (1, n) matrix, Python copies the (1, n) matrix m times vertically

If you have an (m, n) matrix and operate with an (m, 1) matrix, Python copies the (m, 1) matrix n times horizontally

If you operate with a (1, 1) matrix (a scalar), Python copies it to match the dimensions of the other array

Benefits:

Avoids explicit for loops, leading to faster code

Makes code more readable

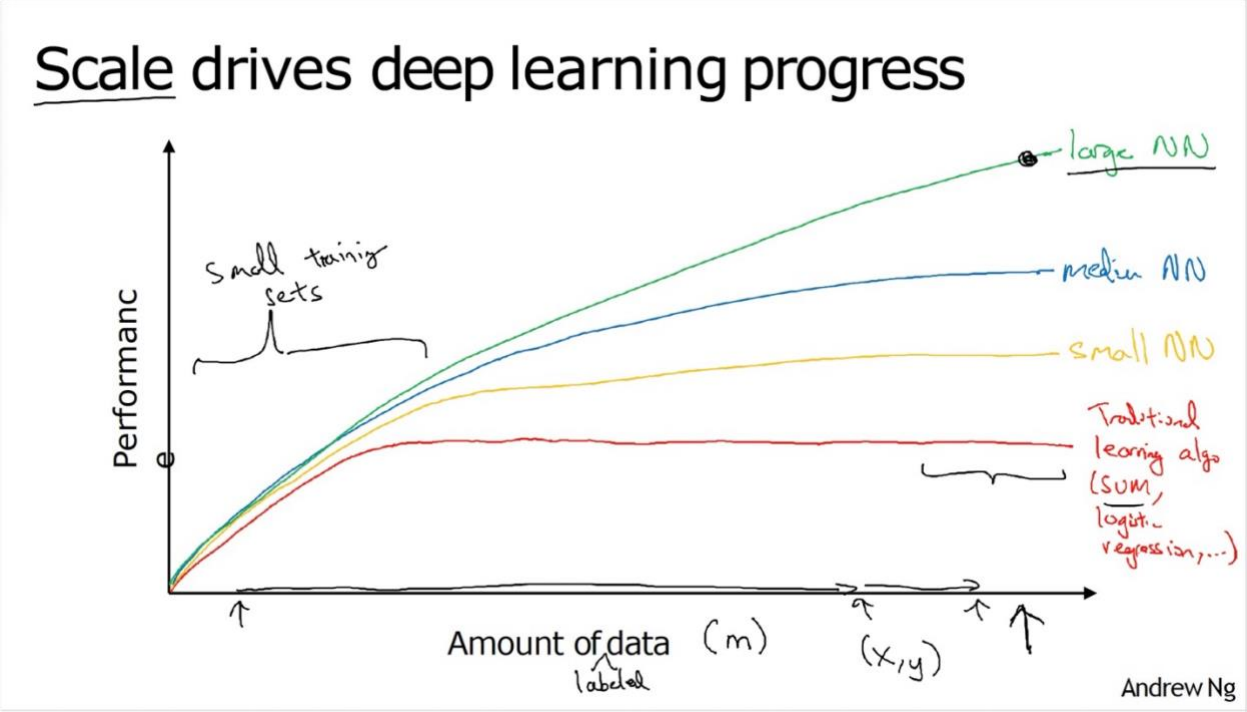
Deep Learning

1. Neural Network Definition

- **Detailed Explanation:** A neural network is a function approximator at its core. It learns to map inputs to outputs by adjusting the connections between neurons. These connections have weights, and the neurons apply mathematical operations to the inputs they receive. The network's architecture (number of layers, number of neurons per layer, how neurons are connected) and the values of the weights determine its behavior. Through a learning process, the network adjusts these weights to improve its performance on a specific task.

2. Why Deep Learning is Taking Off (Explanation of Graph Trends)

- **Graph Trend:** Imagine a graph where the x-axis represents the amount of training data you feed into a learning algorithm, and the y-axis represents the algorithm's performance (e.g., accuracy).



- **Traditional Algorithms:**
  - For traditional machine learning algorithms (like logistic regression, and support vector machines), the graph typically shows an initial increase in performance as you provide more training data.
  - However, at some point, the performance improvement plateaus. Adding significantly more data beyond this point doesn't lead to much better results. These algorithms have limitations in their ability to extract complex patterns from very large datasets.
- **Neural Networks:**
  - Small neural networks might behave similarly to traditional algorithms.
  - Deeper (more layers) and larger (more neurons) neural networks can leverage much larger amounts of data. Their performance tends to increase more consistently with data size.
  - Deep learning models can learn hierarchical representations of data, automatically extracting relevant features, which is a key advantage when dealing with complex, high-dimensional data.
- **Key Takeaways:**
  - Deep learning's power comes from its ability to exploit massive datasets.
  - The availability of big data (due to increased digitization), combined with increased computing power, has fueled the deep learning revolution.
  - Deep learning models can automatically learn intricate features from the data, reducing the need for manual feature engineering.

3. ReLU Function

- **Definition:** ReLU (Rectified Linear Unit) is an activation function widely used in neural networks. It's a simple yet effective function that introduces non-linearity, which is crucial for neural networks to learn complex relationships.
- **Mathematical Representation:** The ReLU function is defined as:
    - $f(x) = \max(0, x)$

  This means:

    - If $x > 0$, then $f(x) = x$
    - If $x \leq 0$, then $f(x) = 0$

How ReLU Improved Gradient Descent:

1. Neural Networks and Backpropagation

- **Neural Network Structure:** Neural networks consist of layers of interconnected nodes (neurons). Information flows from the input layer through hidden layers to the output layer to make a prediction.
- **Backpropagation:** The primary algorithm for training neural networks is backpropagation. It works by:
    1. Calculating the "error" (difference between the network's prediction and the actual value).
    2. Propagating this error backward through the network, layer by layer.
    3. Calculating the gradient of the error concerning each weight in the network. The gradient indicates how much each weight needs to be adjusted to reduce the error.
    4. Updating the weights based on these gradients, typically using an optimization algorithm like gradient descent.

2. The Role of Activation Functions

- **Non-linearity:** Activation functions introduce non-linearity into the network. This is essential because real-world data is often non-linear, and neural networks need to learn complex, non-linear relationships. Without non-linearity, multiple layers would be equivalent to a single layer, limiting the network's power.
- **Common Activation Functions:** Historically, sigmoid and tanh were popular activation functions.
    - Sigmoid: $\sigma(x) = 1 / (1 + \exp(-x))$
    - Tanh: $\tanh(x) = (\exp(x) \exp(-x)) / (\exp(x) + \exp(-x))$

3. What is the Vanishing Gradient Problem?

- **Gradients in Deep Networks:** In deep networks (networks with many layers), backpropagation involves multiplying many gradients together, layer by layer, as the error signal is passed backward.
- **Sigmoid and Tanh Saturation:** Sigmoid and tanh functions have a key characteristic: their gradients are close to zero when the input is very large or very small. We say these functions "saturate" in these regions.
    - For Sigmoid, the maximum value of the derivative is 0.25
    - For Tanh, the maximum value of the derivative is 1
- **The Multiplication Effect:** If the gradients in each layer are less than 1, multiplying many of these small gradients together results in an exponentially decreasing gradient as we move backward through the layers. In very deep networks, the gradients in the earlier layers can become extremely close to zero.
- **Slow Learning or Stalling:** When gradients are very small, the weights in the earlier layers are updated very slowly or not at all. These layers effectively stop learning, even though they are crucial for the network to learn complex features. The network becomes difficult to train effectively.

4. Why ReLU Helps

- **ReLU's Linear Region:** ReLU's key advantage is that for positive inputs, its gradient is always 1. This prevents the gradient from shrinking as it passes through ReLU layers.
- **Maintaining Gradient Flow:** By maintaining a stronger gradient, ReLU helps the error signal propagate more effectively to the earlier layers, allowing them to continue learning.
- **Improved Training:** This leads to significantly faster and more effective training of deep neural networks.

In Summary

The vanishing gradient problem arises from the repeated multiplication of small gradients during backpropagation in deep networks, especially when using saturating activation functions like sigmoid or tanh. This hinders the learning of earlier layers. ReLU mitigates this by having a constant gradient for positive inputs, facilitating better gradient flow and enabling the training of much deeper architectures.

The Dying ReLU Problem:
The "Dying ReLU" problem is a specific issue encountered when using the ReLU (Rectified Linear Unit) activation function in neural networks. Here's a detailed explanation:

1. ReLU Basics

- As a reminder, the ReLU activation function is defined as:
    - $f(x) = \max(0, x)$
- This means:
    - If $x > 0$, then $f(x) = x$
    - If $x \leq 0$, then $f(x) = 0$
- ReLU is computationally efficient and helps mitigate the vanishing gradient problem for positive inputs.

2. The Dying ReLU Phenomenon

- **Zero Gradient for Negative Inputs:** The crucial point is that for any negative input ($x < 0$), the output of ReLU is zero, and the gradient (the slope of the function) is also zero.
- **Neurons Stuck in Zero State:** During training, if a neuron's weights are updated in a way that causes its input to ReLU to consistently become negative, the neuron will continuously output zero. Because the gradient is zero, there will be no further weight updates for that neuron.
- **Inactivity and "Death":** The neuron essentially becomes inactive, or "dead," as it no longer contributes to the learning process. It's stuck in a state where it always outputs zero, regardless of the input.
- **Impact on the Network:** If a significant portion of neurons in a layer "die," the network's ability to learn and represent complex patterns is severely diminished.

3. Causes of Dying ReLU

- **Large Negative Bias:** If a neuron has a large negative bias, its input may consistently fall into the negative region, leading to death.
- **High Learning Rates:** Large learning rates during training can cause drastic weight updates, potentially pushing neurons into the negative region and causing them to die.
- **Unsuitable Initialization:** Poor weight initialization can also contribute to neurons becoming inactive early in training.

4. Mitigation Strategies

- **Leaky ReLU:** Leaky ReLU is a variant that addresses the dying ReLU problem. It introduces a small, non-zero slope for negative inputs:
    - $f(x) = x$ if $x > 0$
    - $f(x) = \alpha x$ if $x \leq 0$ (where α is a small constant, e.g., 0.01)
    - This ensures that even for negative inputs, there is a small gradient, preventing neurons from becoming completely inactive.
- **Parametric ReLU (PReLU):** PReLU is similar to Leaky ReLU, but the slope for negative inputs (α) is learned as a parameter during training.
- **Careful Initialization:** Using appropriate weight initialization techniques (e.g., He initialization) can help prevent neurons from dying early in training.
- **Adaptive Learning Rates:** Employing adaptive learning rate algorithms (e.g., Adam, RMSprop) can help regulate weight updates and reduce the likelihood of neurons dying.

In Summary

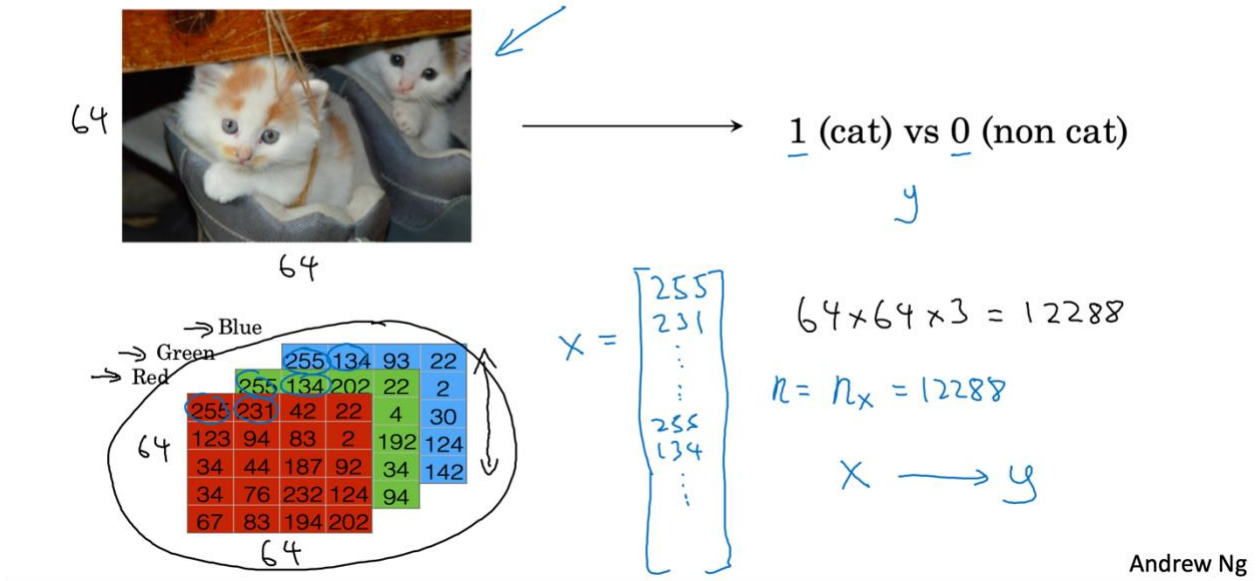
The Dying ReLU problem is a scenario where neurons using ReLU become permanently inactive due to consistently receiving negative inputs, leading to a zero gradient and preventing further learning. Leaky ReLU and other variants, along with careful initialization and learning rate strategies, can help mitigate this issue.

ReLU vs. Sigmoid:

Feature	**ReLU (Rectified Linear Unit)**	**Sigmoid**
Mathematical Representation	$f(x) = \max(0, x)$	$\sigma(x) = 1 / (1 + \exp(-x))$
Output Range	$[0, \infty)$	(0, 1)
Gradient Flow	Constant gradient of 1 for positive inputs, avoiding vanishing gradients. Can experience "dying ReLU" problem (zero gradient for negative inputs).	Prone to vanishing gradients, especially for very large or small inputs. Gradient is always between 0 and 0.25
Computation	Computationally efficient (simple max operation).	Computationally more expensive (involves exponentiation).
Sparsity	Introduces sparsity (many zero activations), which can improve efficiency.	Produces dense activations (mostly non-zero).
Vanishing Gradient Problem	Mitigates for positive inputs.	Susceptible.
"Dying ReLU" Problem	can cause it.	Does not occur.
Use Cases (Hidden Layers)	Highly favored in hidden layers of deep neural networks.	Historically used, but less common in deep networks due to vanishing gradients.
Use cases (output layers)	not often for final layers, unless the regression output must be a positive number.	useful when the output needs to be a probability between 0 and 1.
Derivative Function	1 when $x > 0$ else 0	$\sigma(x)*(1-\sigma(x))$

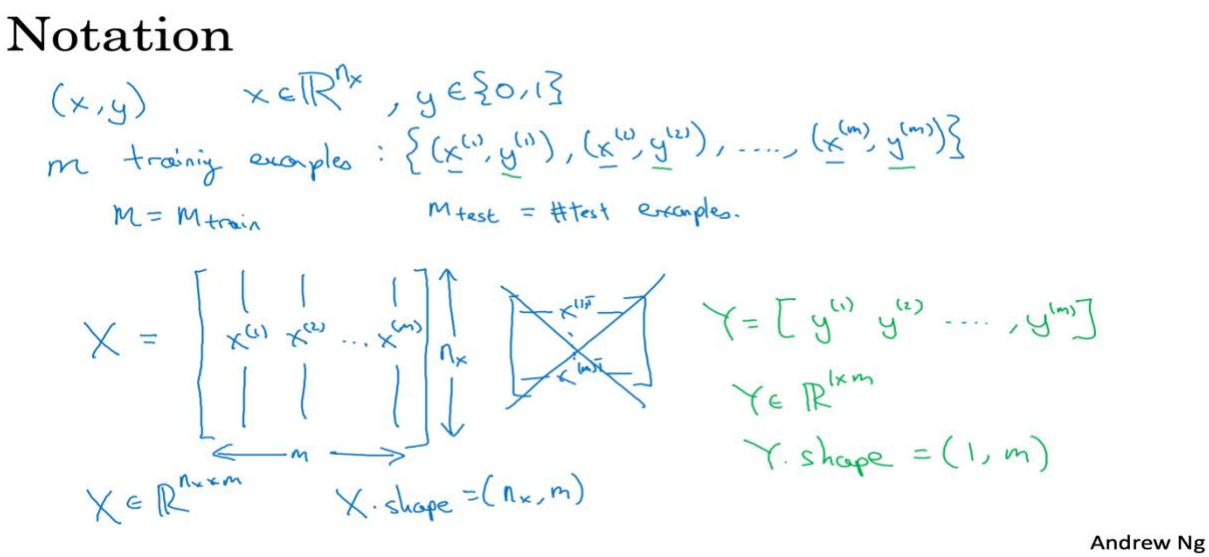
Logistic Regression for Binary Classification

- **Binary Classification**: Logistic regression is an algorithm designed for binary classification problems. In this context, the goal is to categorize an input into one of two possible outcomes.
- **Example:** A classic example is image recognition, where the task is to classify an image as either belonging to a specific category (e.g., "cat") or not.
    - Input: An image.
    - Output: A label (Y) indicating the category: 1 for "cat," 0 for "not-cat."
- **Image Representation:**
    - Images are represented digitally as three matrices, corresponding to the red, green, and blue color channels.
    - Each matrix stores pixel intensity values.
    - If an image has dimensions of 64 pixels by 64 pixels, each color channel is a 64x64 matrix.



- **Feature Vector (X):**
  - To process an image with a learning algorithm, the pixel intensity values from the color channel matrices are unrolled into a feature vector (x). This involves concatenating the rows (or columns) of each color channel matrix into a single long vector.
  - For a 64×64 pixel image with 3 color channels (RGB), the feature vector x has a dimension of: $n_x = 64 \times 64 \times 3 = 12{,}288$
  - Notation: $n_x = 12{,}288$: Dimension of the input feature vector. n (lowercase) is sometimes used interchangeably with n_x.

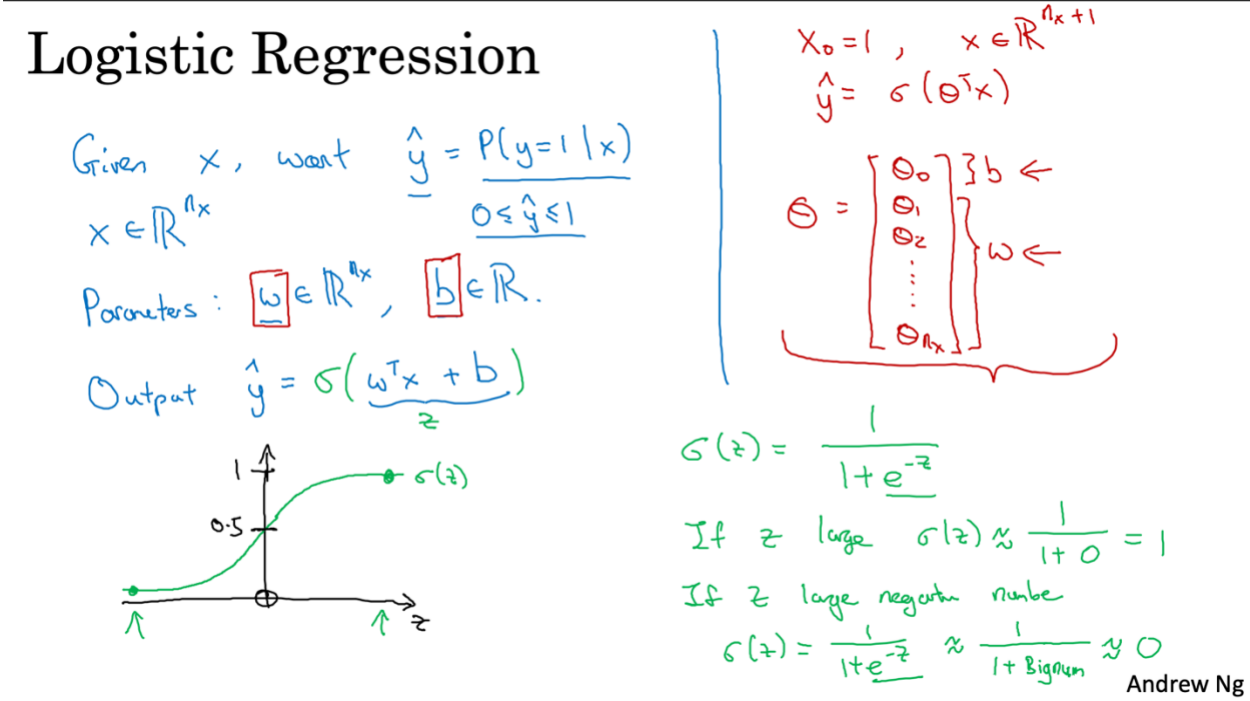
Notation:



- **Training Example:**
  - A single training example is a pair (x, y), where:
  - $x \in \mathbb{R}^{n_x}$ is the feature vector.(i.e, n_x dimensional)
    $y \in \{0, 1\}$ is the label.
- **Training Set:**
  A training set consists of m training examples:
  $\{ (x^1, y^1), (x^2, y^2), ..., (x^m, y^m) \}$
  where m: Number of training examples.
  m_{train}: Number of training examples in the training set.
  m_{test}: Number of examples in the test set.

- **Input Matrix (X):**
  The input feature vectors are stacked column-wise to form the matrix X:
  $X = [x^1 \ x^2 \ ... \ x^m]$
  Shape: $X \in \mathbb{R}^{n_x \times m}$
  Python Notation: X.shape = (n_x, m)
- **Output Matrix (Y):**
  The labels are organized into a row vector Y:
  $Y = [y^1 \ y^2 \ ... \ y^m]$
  Shape: $Y \in \mathbb{R}^{1 \times m}$
  Python Notation: Y.shape = $(1, m)$

Logistic Regression:



Andrew Ng

Binary Classification: Logistic regression is a learning algorithm used for binary classification problems, where the output label y can only take two values: 0 or 1.

Prediction (ŷ): The algorithm aims to output a prediction (denoted as ŷ), which represents the estimated probability that the output y is equal to 1, given the input features x.
Formally, ŷ = P(y = 1 | x)

Input Features (x): The input to the algorithm is a feature vector x, which can represent various data types, such as pixel values of an image.

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

Parameters:

The parameters of logistic regression consist of:

$w \in \mathbb{R}^{n_x}$: A weight vector

$b \in \mathbb{R}$: A bias term

Output Calculation: A linear combination of the input features and weights is calculated: $z = w^T x + b$

The sigmoid function (σ) is applied to this linear combination to produce the output ŷ: $\hat{y} = \sigma(z)$

Sigmoid Function: The sigmoid function (σ) is defined as: $\sigma(z) = 1 / (1 + e^{-z})$

It maps any real number to a value between 0 and 1, making it suitable for representing probabilities.

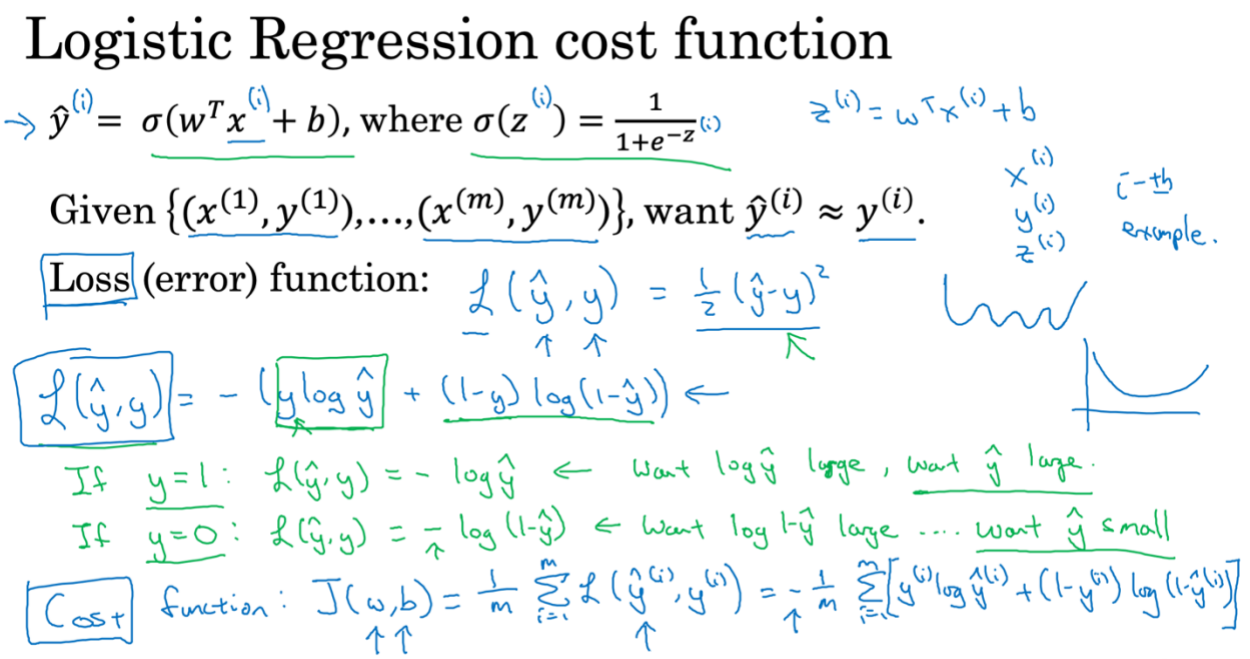
When z is very large, $\sigma(z) \approx 1$

When z is very small (a large negative number), $\sigma(z) \approx 0$

Alternative Notation: By defining $x_0 = 1$ and $\Theta \in \mathbb{R}^{n_x+1}$ (where $\Theta = [b, w_1, w_2, ..., w_n^x]$), we can write:

$\hat{y} = \sigma(\Theta^T x)$

Cost Function in Logistic Regression



Andrew Ng

The prediction ŷ for a given input feature vector x is calculated as: $\hat{y} = \sigma(w^T x + b)$, where σ is the sigmoid function. $\sigma(z) = 1 / (1 + e^{-z})$

Goal: The goal is to learn the parameters w (weight vector) and b (bias) so that the predictions $\hat{y}^{(i)}$ on the training set are as close as possible to the true labels $y^{(i)}$.

Loss Function: The loss function L(ŷ, y) measures how well the algorithm performs on a single training example (x, y).

In logistic regression, the loss function used is:

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

If y = 1, the loss function tries to make ŷ as large as possible.

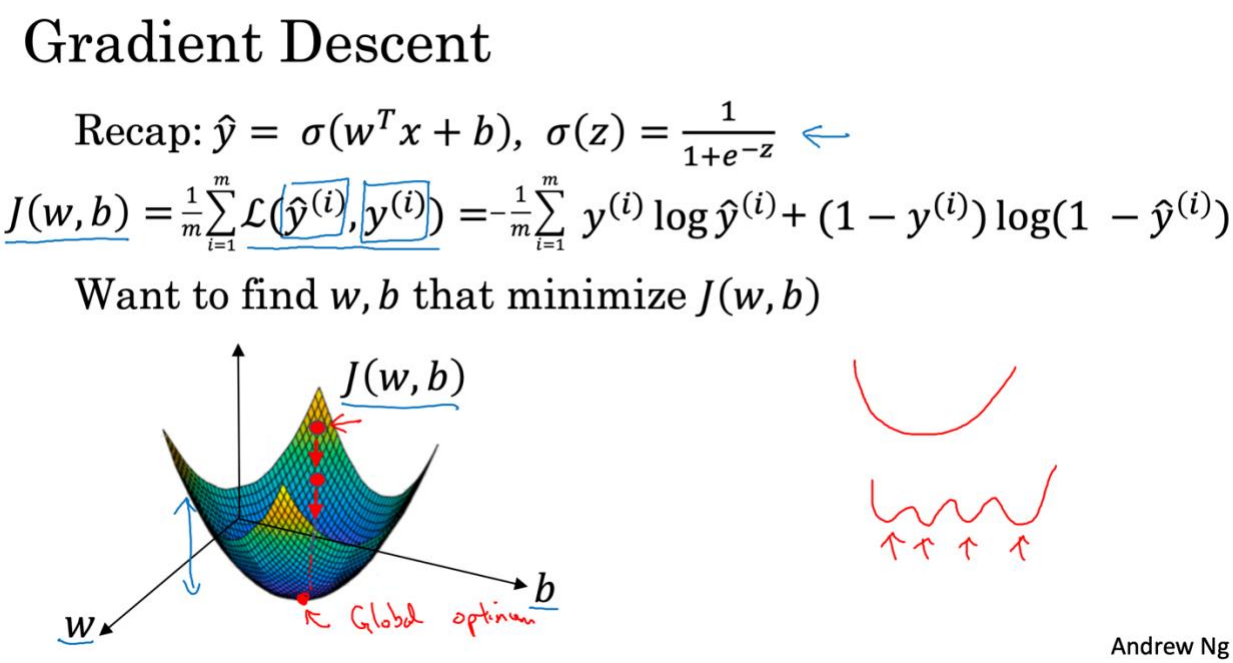
If y = 0, the loss function tries to make ŷ as small as possible.

Cost Function: The cost function J(w, b) measures the overall performance of the algorithm on the entire training set. It is the average of the loss functions over all m training examples:

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

Training: The objective of training a logistic regression model is to find the parameters w and b that minimize the cost function J(w, b).

Gradient Descent



Andrew Ng

Objective: The goal of gradient descent is to find the values of parameters w and b that minimize the cost function J(w, b).

Cost Function: The cost function J(w, b) measures how well the parameters w and b perform on the entire training set. It's calculated as the average of the loss function over all training examples.

The loss function measures how well the algorithm's prediction $\hat{y}^{(i)}$ compares to the true label $y^{(i)}$ for a single training example.

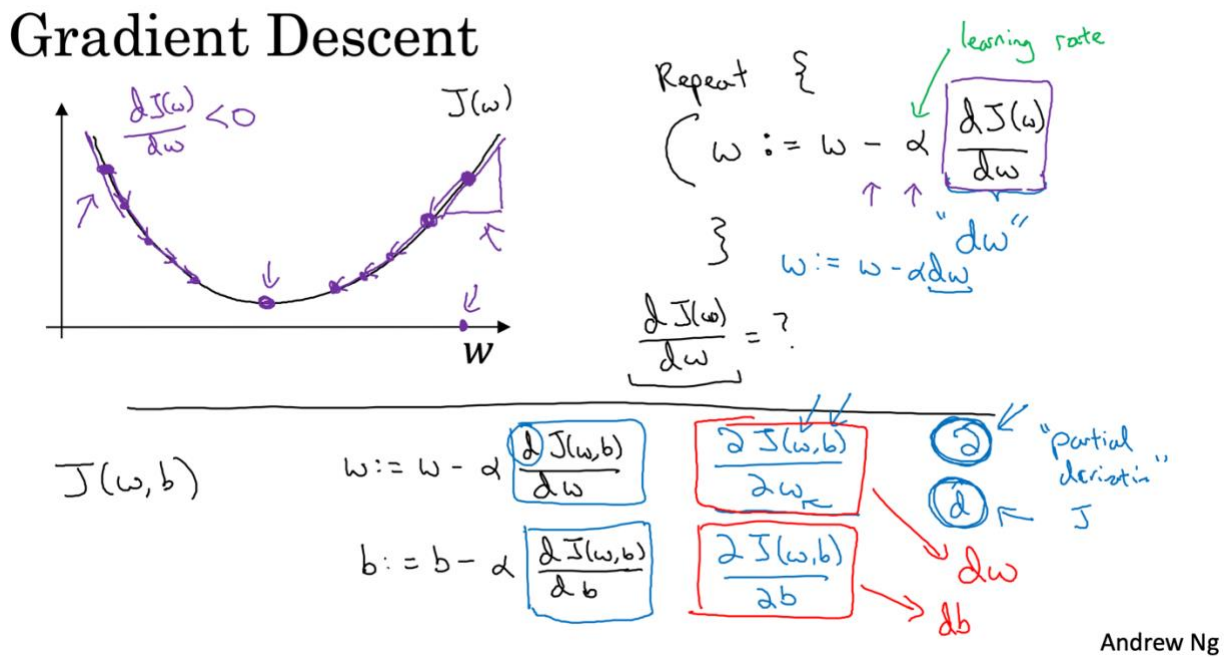
Visualization: The cost function J(w, b) can be visualized as a surface where the horizontal axes represent the parameters w and b, and the vertical axis represents the value of J.

For logistic regression, this surface is a convex function (a bowl shape), meaning it has a single global minimum.

Algorithm: Gradient descent starts by initializing w and b to some initial values.

Then, it iteratively updates w and b by taking steps in the direction of the steepest downhill slope of the cost

function.



Each step moves the parameters closer to the global minimum of J.

Update Rule: The update rule for gradient descent is:

w := w α (∂J(w, b) / ∂w)

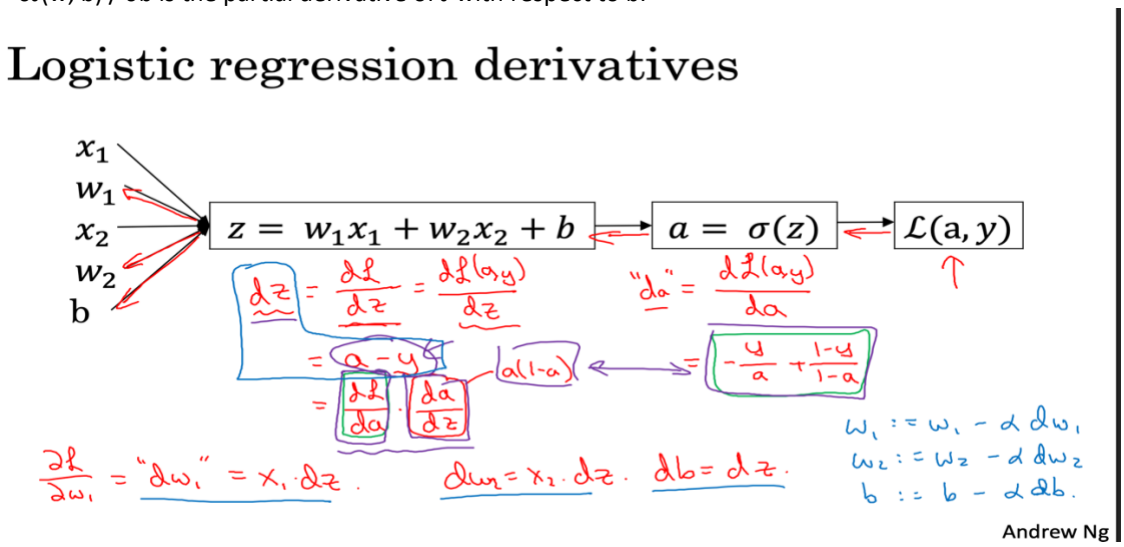
b := b α (∂J(w, b) / ∂b)

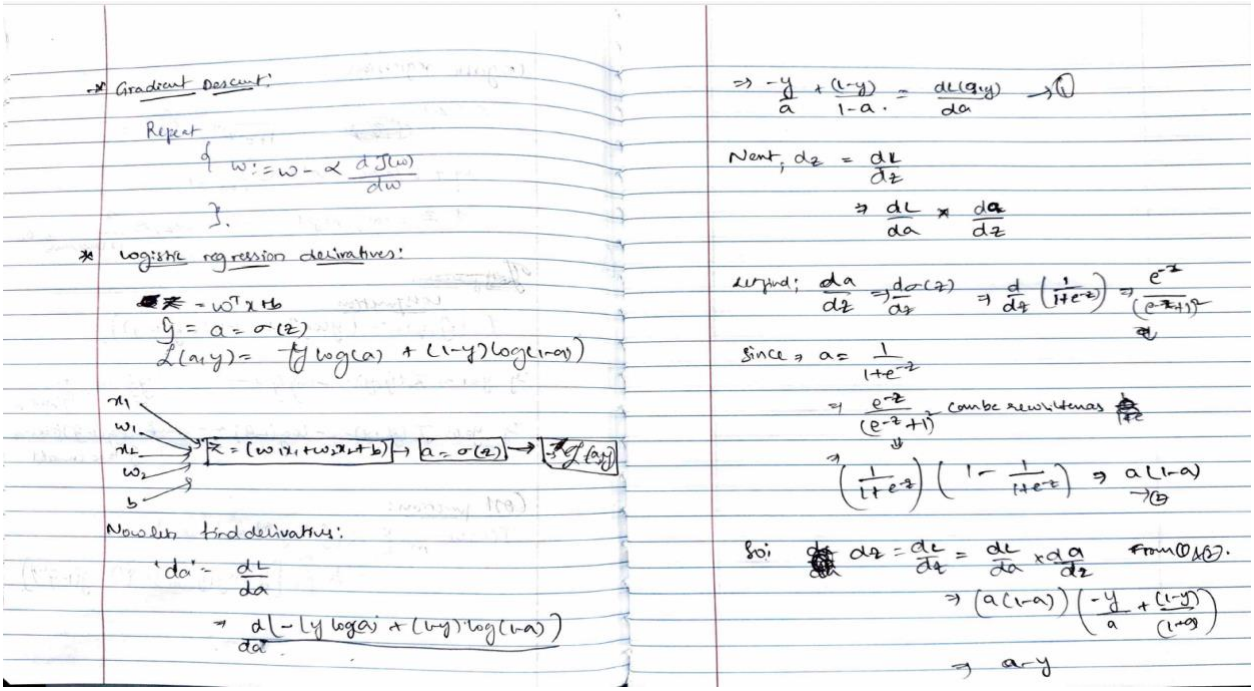
Where:

α is the learning rate, controlling the step size.

∂J(w, b) / ∂w is the partial derivative of J with respect to w.

∂J(w, b) / ∂b is the partial derivative of J with respect to b.

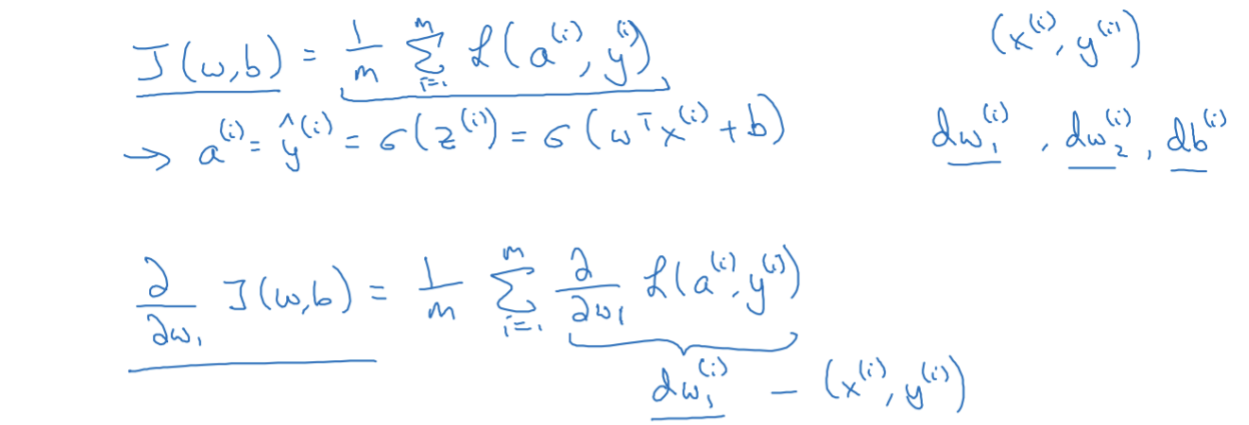




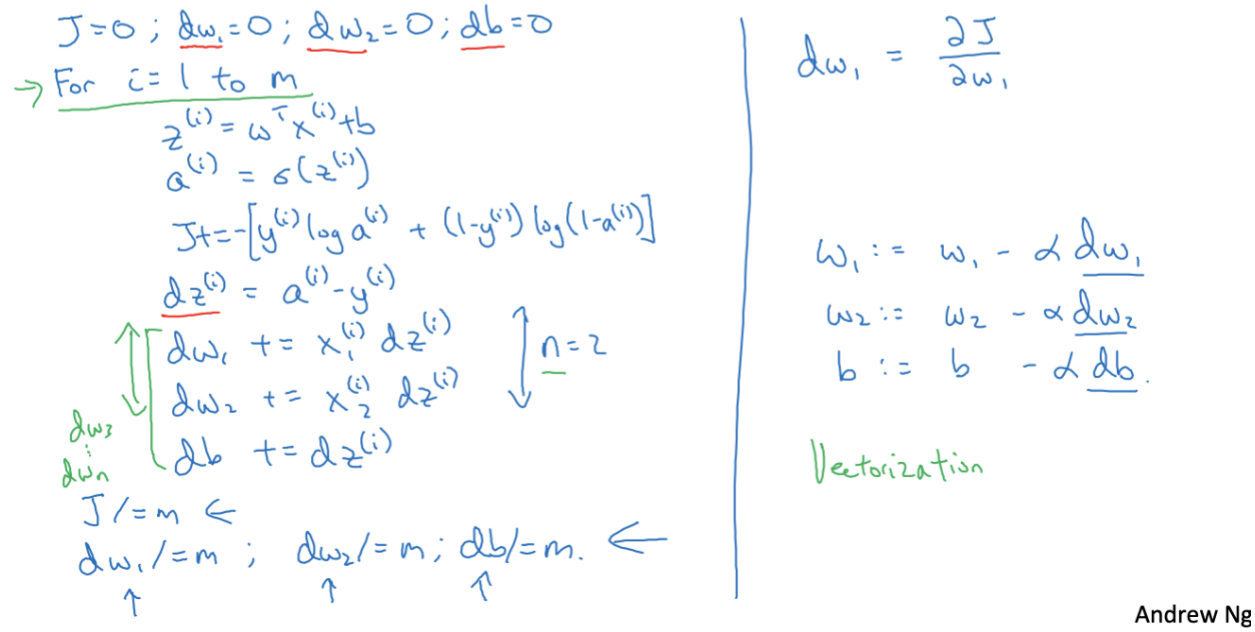
So, if we think Logistic Regression with a Neural Network mindset, here is what is happening:

```
# 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)) ))

# YOUR CODE ENDS HERE
# BACKWARD PROPAGATION (TO FIND GRAD)
#(≈ 2 lines of code)
# dw = ...
# db = ...
# YOUR CODE STARTS HERE
dw = 1/m*(np.dot(X,(A-Y).T))
db = 1/m*(np.sum(A-Y))
# YOUR CODE ENDS HERE
cost = np.squeeze(np.array(cost))
```



Andrew Ng



Andrew Ng

Vectorization

Definition:

Vectorization is the technique of eliminating explicit for loops in code.

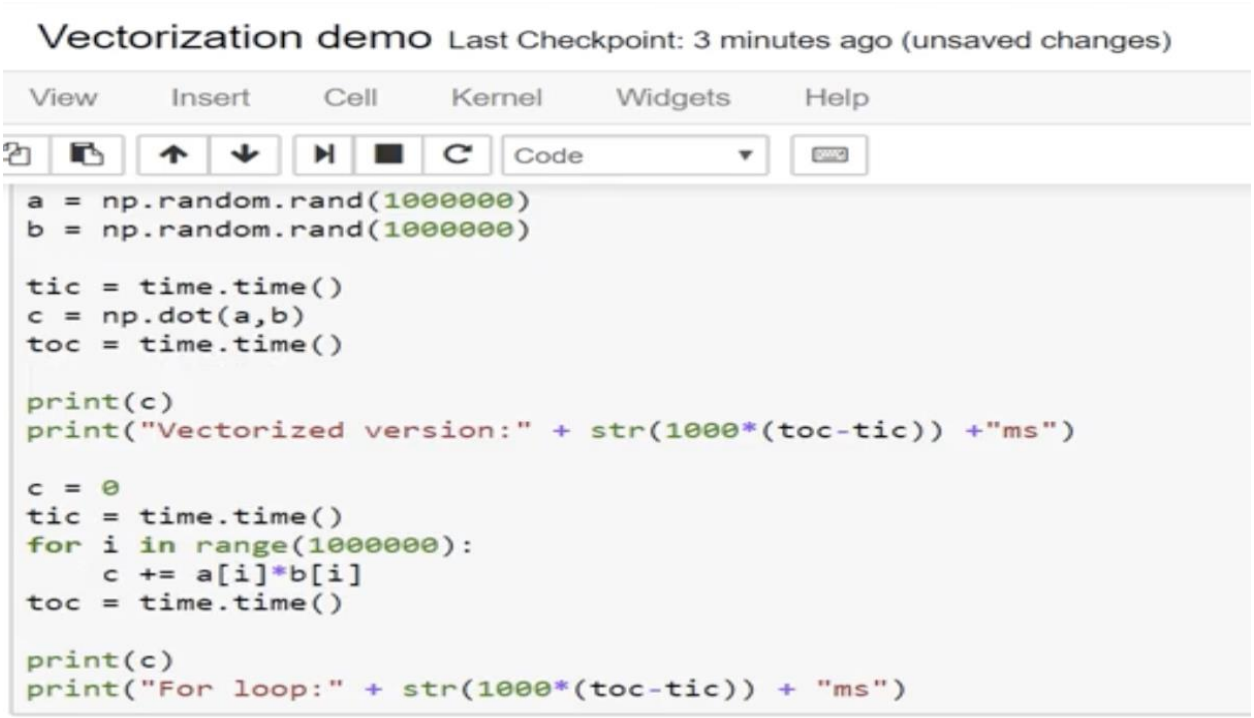
Importance: In deep learning, training is often performed on large datasets. Vectorization makes code run much faster, which is crucial for efficient training on these datasets.

Example: In logistic regression, a key computation is calculating $z = w^T x + b$, where w and x are vectors.

Non-vectorized implementation:

z = 0
For i in range(n_x):
    z += w_i * x_i
z += b

Vectorized implementation:
In Python (using numpy):
z = np.dot(w, x) + b
This avoids loops and is much faster.
Demonstration:



Underlying Principle:
CPUs and GPUs have Single Instruction, Multiple Data (SIMD) instructions that allow for parallel computations. Vectorized code leverages these instructions to perform operations on multiple data elements simultaneously, greatly improving efficiency.
Rule of Thumb:
Avoid explicit for loops whenever possible to write efficient code.

Vectorization of Gradient Descent in Logistic Regression

1. Vectorization in Logistic Regression
Goal: To efficiently compute logistic regression calculations over an entire training set without explicit for-loops.
Benefit: Significantly speeds up code execution, crucial for large datasets.
2. Vectorizing Forward Propagation
Input Matrix (X): Training inputs are stacked column-wise into matrix X (n_x × m):
X = np.array([[x1], [x2], ..., [xm]]) # Conceptual
X.shape # (nx, m)
Calculating Z:

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

Calculating A (Activations):

A = sigmoid(Z) # Sigmoid applied element-wise

3. Vectorizing Backward Propagation

Calculate dZ:

dZ = A Y

4. Vectorized Gradient Descent

Update Parameters: (Using gradient descent)

w = w alpha * dw

b = b alpha * db

Calculate db:

db = (1 / m) * np.sum(dZ)

Calculate dw:

dw = (1 / m) * np.dot(X, dZ.T)

6. Initial (Non-Vectorized) Implementation

```
def initial_logistic_regression(X, Y, w, b, alpha):
    m = X.shape[1]
    nx = X.shape[0]
    dw = np.zeros((nx, 1))
    db = 0
    for i in range(m):
        z = np.dot(w.T, X[:, i]) + b
        a = sigmoid(z)
        dz_i = a - Y[0, i]
        for j in range(nx):
            dw[j, 0] += X[j, i] * dz_i
        db += dz_i
    dw = (1 / m) * dw
    db = (1 / m) * db
    w = w - alpha * dw
    b = b - alpha * db
    return dw, db, w, b
```

How it works:

Python automatically expands the smaller array to match the shape of the larger array

Then, it performs the operation element-wise

Specific Broadcasting Rules:

If you have an (m, n) matrix and operate with a (1, n) matrix, Python copies the (1, n) matrix m times vertically

If you have an (m, n) matrix and operate with an (m, 1) matrix, Python copies the (m, 1) matrix n times horizontally

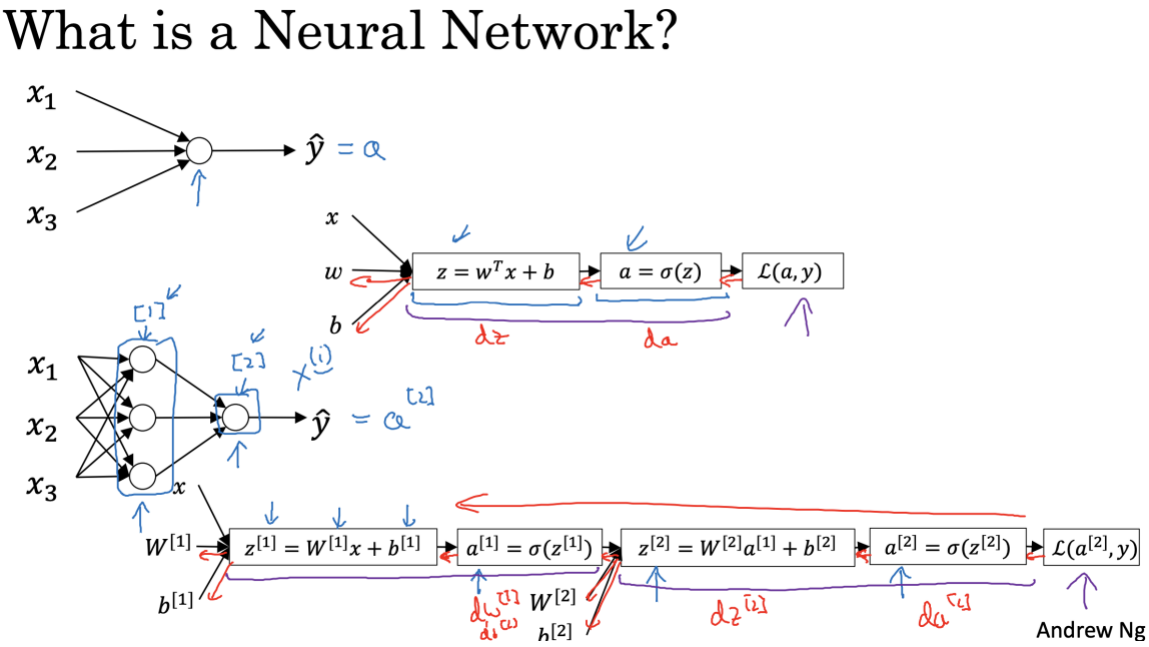
If you operate with a (1, 1) matrix (a scalar), Python copies it to match the dimensions of the other array

Benefits:

Avoids explicit for loops, leading to faster code

Makes code more readable

What is a Neural Network?



1. Core Concept

- A neural network can be formed by stacking together multiple logistic regression units. Where logistic regression involves a single calculation of z, followed by a calculation of a, a neural network repeats this process multiple times.
- In a neural network, you have layers of interconnected nodes, where each node performs a z-like calculation (linear combination of inputs) and an a-like calculation (applying an **activation function**).

2. Forward Propagation

- The input features (x), along with parameters w and b, are used to compute $z^{(1)}$ in the first layer.
- $a^{(1)}$ is then computed by applying an activation function (like the sigmoid function) to $z^{(1)}$.
- This process is repeated for subsequent layers: z_2 is calculated using $a^{(1)}$ and new parameters, and $a^{(2)}$ is calculated from $z^{(2)}$
- $a^{(2)}$ represents the final output (ŷ) of the neural network.

3. Backward Propagation

- Similar to logistic regression, neural networks use a backward pass to compute derivatives and update parameters.
- This involves calculating $da^{(2)}$, $dz^{(2)}$, $dw^{(2)}$, $db^{(2)}$, and so on, working from right to left through the network.

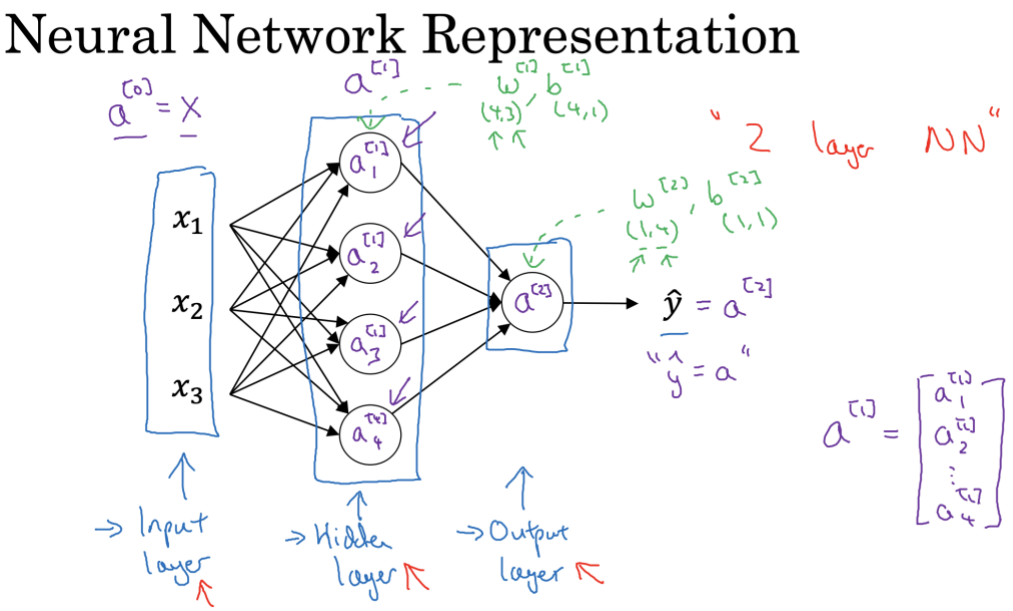
4. Key Differences from Logistic Regression

- **Multiple Layers:** Neural networks have multiple layers, allowing them to learn more complex functions than logistic regression.
- **Repeated Calculations:** The z and a calculations are repeated for each layer.

5. Purpose

- Neural networks are used for various tasks, including housing price prediction.
- They can learn to approximate functions and make predictions based on input data. In essence, a neural network is an extension of logistic regression, with multiple layers enabling it to learn more intricate patterns in data.

Neural Network Representation



This is a 2-layer NN. Generally, the input layer is layer 0.

1. Core Concept

- A neural network can be formed by stacking together multiple logistic regression units.
- While logistic regression computes a single $z = w^T x + b$ followed by $a = \sigma(z)$, a neural network performs this computation in multiple layers.
- Each neuron computes a linear transformation (z) and passes it through an activation function (a).

For example:

$z^{(1)} = W^{(1)}x + b^{(1)}$

$a^{(1)} = \sigma(z^{(1)})$

$z^{(2)} = W^{(2)}.Ta^{(1)} + b^{(2)}$

$a^{(2)} = \sigma(z^{(2)}) = \hat{y}$

2. Forward Propagation

- The input vector X is denoted as $a^{(0)}$, i.e., $a^{(0)} = X$

Compute:

$z^{(1)} = W^{(1)}a^{(0)} + b^{(1)}$

$a^{(1)} = \sigma(z^{(1)})$

Then:

$z^{(2)} = W^{(2)}a^{(1)} + b^{(2)}$

$a^{(2)} = \sigma(z^{(2)})$

Final output: $\hat{y} = a^{(2)}$

3. Backward Propagation

- After computing the output $\hat{y} = a^{(2)}$, the loss $L(a^{(2)}, y)$ is calculated.
- Gradients are computed by chaining back through the layers:

$dL/da^{(2)}$, $da^{(2)}/dz^{(2)} \rightarrow dz^{(2)} \rightarrow dW^{(2)}, db^{(2)}$

Similarly for layer 1: $dz^{(1)} \rightarrow dW^{(1)}, db^{(1)}$

4. Key Differences from Logistic Regression

- Logistic regression has one layer, while a neural network like this example has two (1 hidden + 1 output).
- A neural network can model non-linear and more complex relationships.

5. Purpose

Neural networks are powerful models used in applications such as housing price prediction. They can approximate complex functions and generalize patterns from data.

Activation Functions in Neural Networks

In a neural network, activation functions are mathematical functions applied to the outputs of each layer. They introduce non-linearity, which allows the network to learn and represent more complex functions.

Forward Propagation

Given input features x, the forward pass in a two-layer neural network is:

$z^1 = W^1x + b^1$ # Linear transformation

$a^1 = g^1(z^1)$ # Activation function

$z^2 = W^2a^1 + b^2$

$\hat{y} = a^2 = g^2(z^2)$

Where:

W^l: Weight matrix for layer l

b^l: Bias vector for layer l

z^l: Linear combination (pre-activation)

a^l: Activation output of layer l

$g^l()$: Activation function

$\hat{y}$: Final prediction

Sigmoid

Formula: a = 1 / (1 + e^-z)

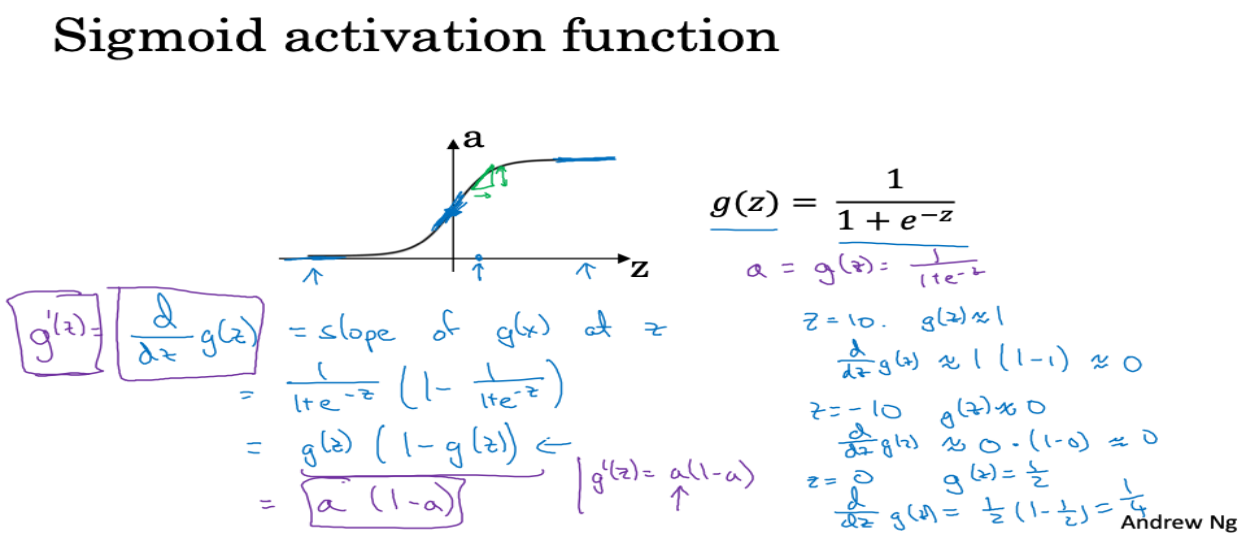
Range: (0, 1)

Use Case: Output layer for binary classification

Pros: Outputs a probability

Cons: Vanishing gradient for large |z|

Not zero-centered (mean ≈ 0.5)



Tanh

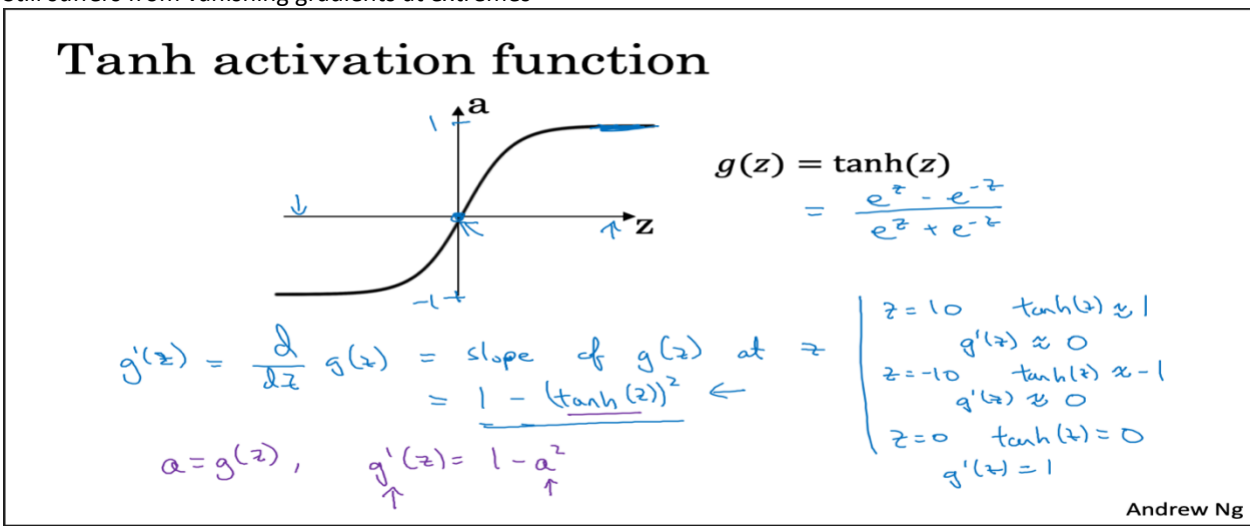
Formula: a = (e^z e^-z) / (e^z + e^-z)

Range: (-1, 1)

Use Case: Hidden layers

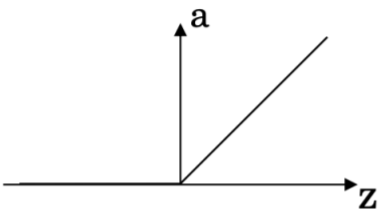
Pros:

Zero-centered
Better learning than sigmoid
Cons:
Still suffers from vanishing gradients at extremes



ReLU
Formula: a = max(0, z)
Range: [0, ∞)
Use Case: Hidden layers (default)
Pros:
Simple
Sparse activation
Faster training
Cons:
Dying ReLU problem (no gradient if z ≤ 0)
Leaky ReLU
Formula: a = max(0.01z, z)
Range: (-∞, ∞)
Use Case: Hidden layers (alternative to ReLU)
Pros:
Prevents dead neurons
Small slope for z < 0
Cons:
Slightly more complex

ReLU and Leaky ReLU



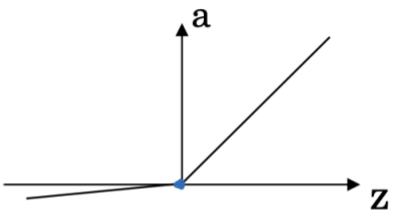
ReLU

$g(z) = \max(0, z)$

$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$

~~undef if z = 0~~

$z = 0.0000\ldots0$



Leaky ReLU

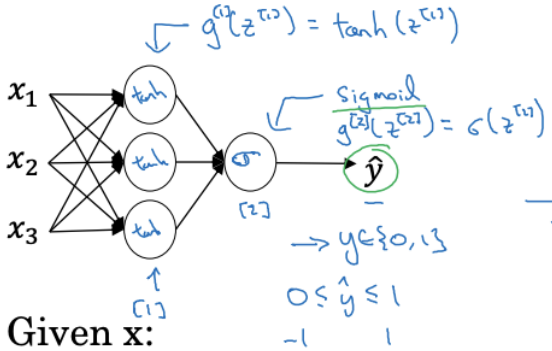
$g(z) = \max(0.01z, z)$

$g'(z) = \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$

Andrew Ng

Graphical Representation

Activation functions



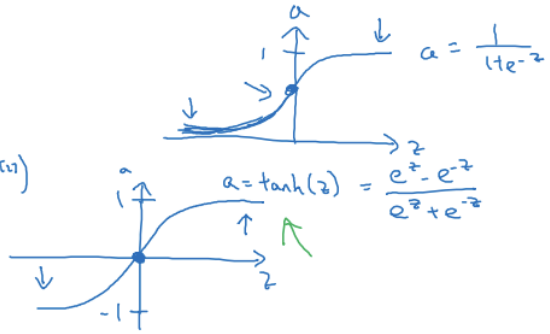
$g^{[1]}(z^{[1]}) = \tanh(z^{[1]})$

Sigmoid $g^{[2]}(z^{[2]}) = \sigma(z^{[1]})$

$\rightarrow y \in \{0, 1\}$

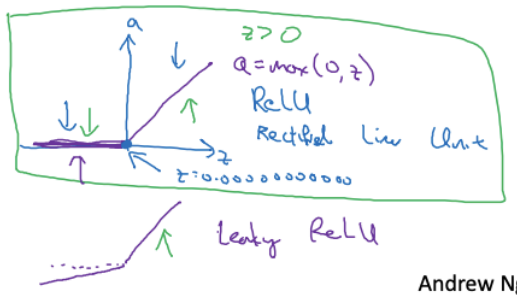
$0 \le \hat{y} \le 1$

$-1 \quad 1$



Given x:

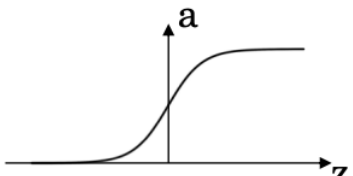
$$z^{[1]} = W^{[1]}x + b^{[1]}$$
$$\rightarrow a^{[1]} = \sigma(z^{[1]}) \quad g^{[1]}(z^{[1]})$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$\rightarrow a^{[2]} = \sigma(z^{[2]}) \quad g^{[2]}(z^{[2]})$$



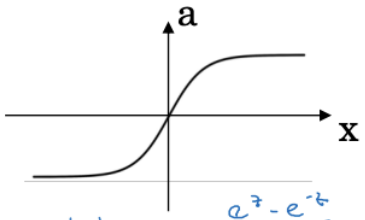
Leaky ReLU

Andrew Ng

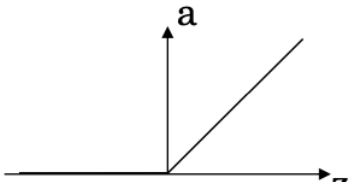
Pros and cons of activation functions



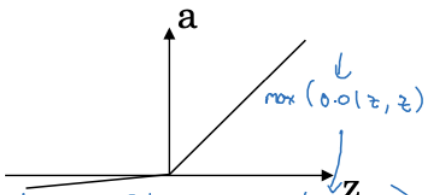
sigmoid: $a = \frac{1}{1+e^{-z}}$



tanh: $a = \frac{e^z - e^{-z}}{e^z + e^{-z}}$



ReLU $a = \max(0, z)$



Leaky ReLU $a = \max(0.01z, z)$

Andrew Ng

Best Practices

Use sigmoid for output in binary classification.

Use ReLU for hidden layers as default.

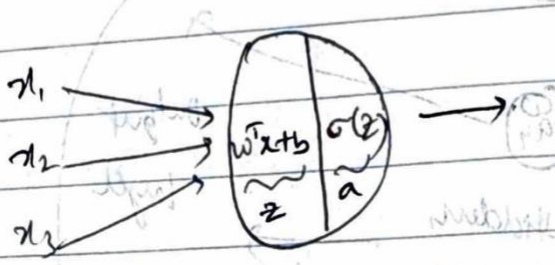
Tanh can be used in hidden layers for zero-centered output.

Leaky ReLU helps avoid dead neurons from ReLU.

* Nueral Network Representation:
* so every activation function represents two steps computation

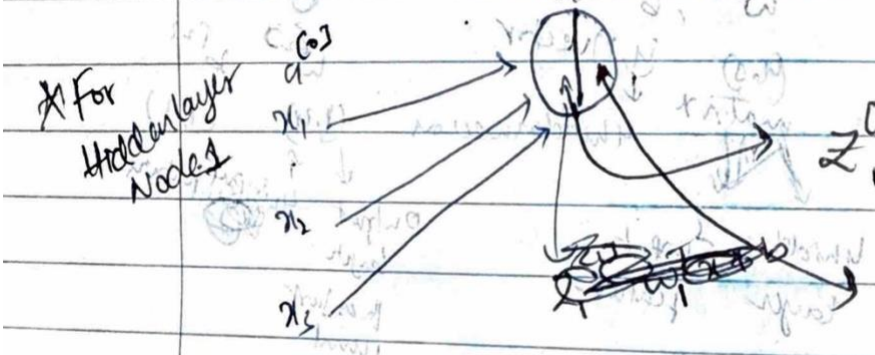
(1) $z = w^T x + b$

(2) $a = \sigma(z)$



* Now we will apply this analogy to our 2NN.

For the first node $a^{[1]}$

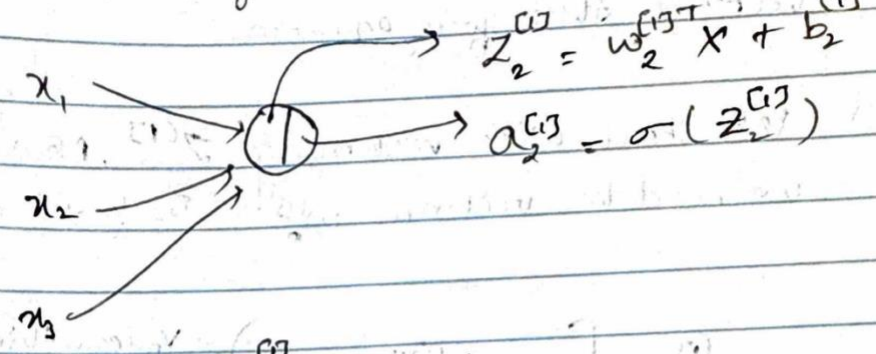


Notation:

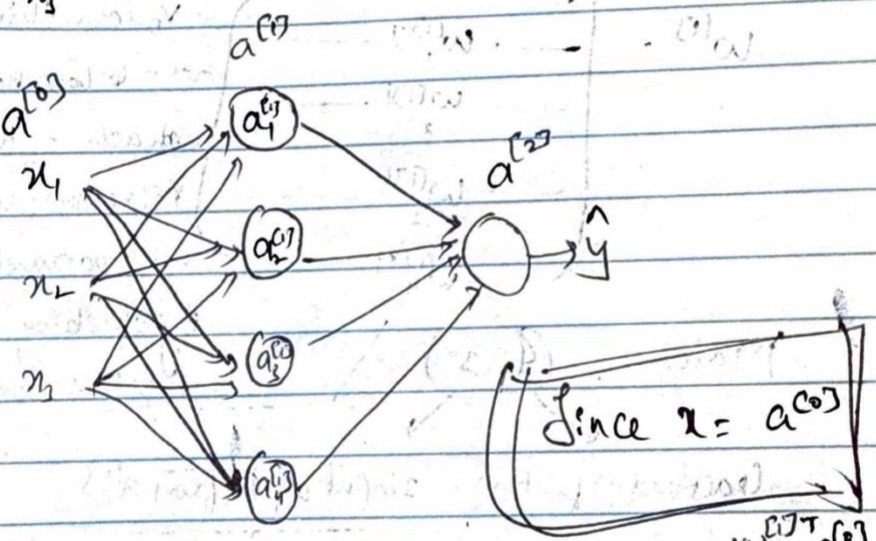
$[l]$ ← layer number

a_i ← node number in that layer

* For Hidden layer Node 2:



⇒



Now we have:

$z_1^{[1]} = w_1^{[1]T} \cdot a^{[0]} + b_1^{[1]}$, $a_1^{[1]} = \sigma(z_1^{[1]})$

$z_2^{[1]} = w_2^{[1]T} \cdot a^{[0]} + b_2^{[1]}$, $a_2^{[1]} = \sigma(z_2^{[1]})$

$z_3^{[1]} = w_3^{[1]T} \cdot a^{[0]} + b_3^{[1]}$, $a_3^{[1]} = \sigma(z_3^{[1]})$

$z_4^{[1]} = w_4^{[1]T} \cdot a^{[0]} + b_4^{[1]}$, $a_4^{[1]} = \sigma(z_4^{[1]})$

* Since writing loops is not so peacable, lets vectorize these four equation.

1) Lets start with vectorizing $z^{[1]}$, Now to vectorize it we need to vectorize $w_1^{[1]}$, $b^{[1]}$ & $a^{[0]}$ so.

$$w^{[1]} = \begin{bmatrix} — & w_1^{[1]T} & — \\ — & w_2^{[1]T} & — \\ — & w_3^{[1]T} & — \\ — & w_4^{[1]T} & — \end{bmatrix}$$

∴ You can think of it as we have 4 logistic regression units and each of the logistic regression unit has corresponding parameter vector w. By stacking those 4 vector together we get (4,3) matrix

Matrix : (4, 3)

4 activation function 3 input vector from $a^{[0]}$

Similarly, $a^{[0]} = x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$ (3, 1) 3 input features.

$$b^{[1]} = \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}$$

(4, 1) vector

So;

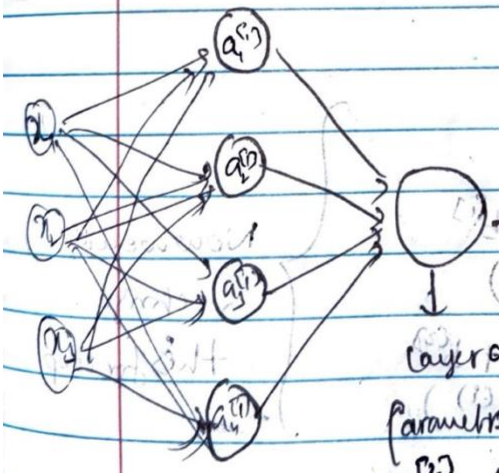
$$z^{[1]} = \begin{bmatrix} -w_1^{[1]T}- \\ -w_2^{[1]T}- \\ -w_3^{[1]T}- \\ -w_4^{[1]T}- \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T} a^{[0]} + b_1^{[1]} \\ w_2^{[1]T} a^{[0]} + b_2^{[1]} \\ w_3^{[1]T} a^{[0]} + b_3^{[1]} \\ w_4^{[1]T} a^{[0]} + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}$$

$w^{[1]}$ $a^{[0]}$ $b^{[1]}$ $w^{[1]T}a^{[0]}+b^{[1]}$ $z^{[1]}$

Similarly now;

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = \sigma(z^{[1]}) \text{ i.e } \sigma\left(\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}\right)$$

So, Now we have, For a given input x i.e $a^{[0]}$



For given input $a^{[0]}$

$\rightarrow z^{[1]} = w^{[1]} a^{[0]} + b^{[1]}$
$(4,1) \quad (4,3) \quad (3,1) \quad (4,1)$

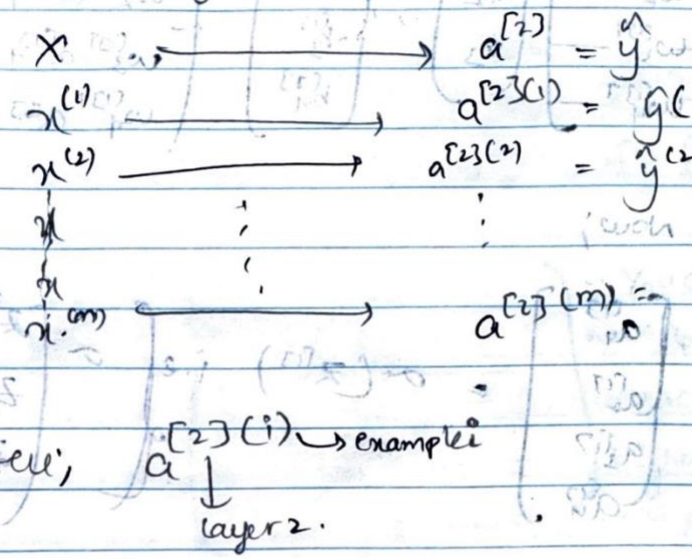
$\rightarrow a^{[1]} = \sigma(z^{[1]})$
$(4,1) \quad (4,1)$

$\rightarrow z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$
$(1,1) \quad (1,4) \quad (4,1) \quad (1,1)$

$\rightarrow a^{[2]} = \sigma(z^{[2]})$
$(1,1) \quad (1,1)$

$\Rightarrow a^{[2]} = \hat{y}$.

* Vectorizing across multiple examples: (m).



Here; $a^{[2](i)}$ → example i

↓ layer 2.

Now we compute 4 equations that we got previously for 1 traing example.

for i = 1 to m;

$z^{[1](i)} = w^{[1]} x^{(i)} + b^{[1]}$

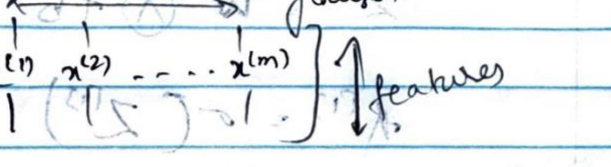
$a^{[1](i)} = \sigma(z^{[1](i)})$

$z^{[2](i)} = w^{[2]} a^{[1](i)} + b^{[2]}$

$a^{[2](i)} = \sigma(z^{[2](i)})$

Now we will vectorize this for loop.

Lets represent allour training examples with "X".
We know X canbe represnterdasa column vector:

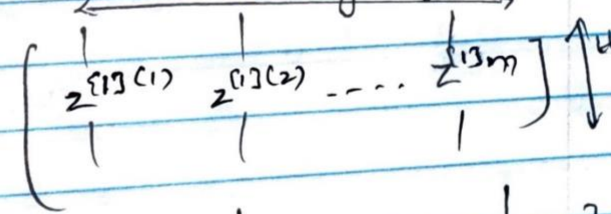


$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

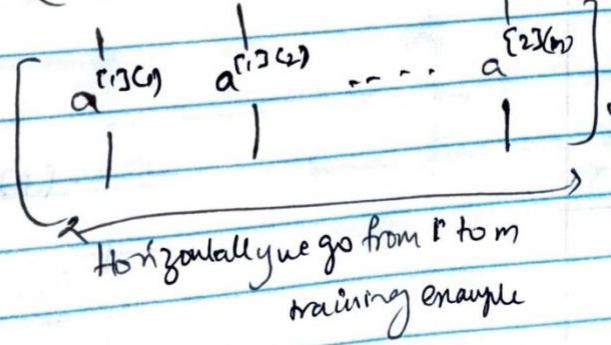
Now lets think of pixel value a cat example.

where; $x^{(1)}$ is picture of cat-1 and $x^{(1)}$ has all pixel values of cat-1 stacked vertically in it.
i.e $x^{(1)}$ has 12288 rows.
Same goes with rest of cat pictures i.e for m cat pictures.

Similarly; $z^{[1]}$ weighted sums inputs w and bias b) can be written as



$$Z^{[1]} = \begin{bmatrix} | & | & & | \\ z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1]m} \\ | & | & & | \end{bmatrix}$$



$$A^{[1]} = \begin{bmatrix} | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \cdots & a^{[2](m)} \\ | & | & & | \end{bmatrix}$$

* Formulas for computing derivatives.

→ Forward propogation.

$$z^{[1]} = W^{[1]} X + b^{[1]}$$

$$A^{[1]} = g^{[1]}(z^{[1]})$$

$$z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$

$$A^{[2]} = g^{[2]}(z^{[2]})$$

→ Backward Propogation:

$$dz^{[2]} = A^{[2]} - Y \quad \text{where } (Y = [y^{(1)}, y^{(2)} \ldots y^{(m)}])$$

$$dw^{[2]} = \frac{1}{m} dz^{[2]} \cdot A^{[1]T}$$

$$db^{[2]} = \frac{d(\quad)}{d(\quad)} db^{[2]} = \frac{1}{m} np.sum(dz^{[2]}, axis=1, keepdims=True)$$

∴ axis=1, operation performed horizontally across columns

keepdims=True, returns result in (m,1) shape instead of just (m,) shape

* Gradient Descent for NN:

Parameters: $w^{[1]}$, $b^{[1]}$, $w^{[2]}$, $b^{[2]}$

$n_x = n^{[0]}$; $n^{[1]}$; $n^{[2]} = 1$

↓ Input features ↓ Hidden units ↓ Output units.

So then; $w^{[1]} = (n^{[1]}, n^{[0]})$ dimension matrix

$b^{[1]} = (n^{[1]}, 1)$ vector (colum vector)

$w^{[2]} = (n^{[2]}, n^{[1]})$ dimension matrix

$b^{[2]} = (n^{[2]}, 1)$ colum vector.

Cost function: $J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^{n} \mathcal{L}(\hat{y}, y)$ ← $a^{[2]}$

Gradient Descent:

Repeat {

Computation predicts $(\hat{y}^{(i)}$; where $i = 1, 2, \cdots, m)$

do: $dw^{[1]} = \frac{d(J)}{d w^{[1]}}$, $db^{[1]} = \frac{\partial J}{\partial b^{[1]}}$, ...

update: $w^{[1]} = w^{[1]} - \alpha\, dw^{[1]}$,

$b^{[1]} = b^{[1]} - \alpha\, db^{[1]}$, ...

}

$$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})$$

↑ element wise product

where:

Dot product:

$$\text{say } c = x^T y \Rightarrow \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}^T \cdot \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \Rightarrow [x_1, x_2, \dots x_m] \cdot \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \sum_{i=1}^{m} x_i y_i$$

Element wise product:

$$\vec{z} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \odot \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} x_1 y_1 \\ x_2 y_2 \\ \vdots \\ x_m y_m \end{bmatrix}$$

$$dW^{[1]} = \frac{1}{m} dz^{[1]} X^T$$

$$db^{[1]} = \frac{1}{m} np.sum(dz^{[1]}, axis = 1, keepdims = True)$$

We will know discuss intuition behind these formulas.

So, we have these equations in the end

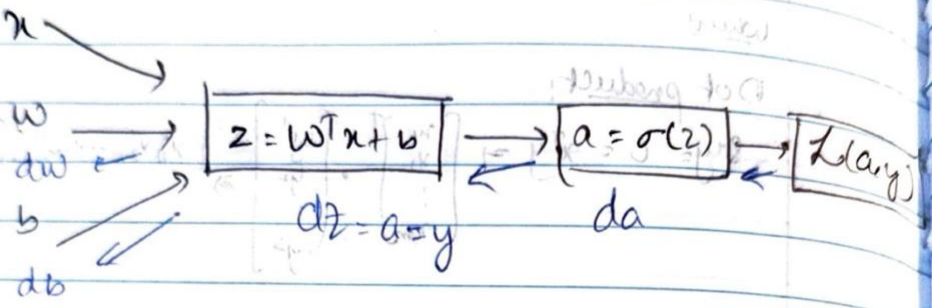
$$Z^{[1]} = W^{[1]} \overset{A^{[0]}}{\otimes} + b^{[1]}$$

$$A^{[1]} = \sigma(Z^{[1]})$$

$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$

$$A^{[2]} = \sigma(Z^{[2]})$$

* Initially, in logistic regression:



Since; $L(a,y) = -y \log a - (1-y) \log(1-a)$

$$da = \frac{dL(a,y)}{da} = \frac{y}{a} + \frac{1-y}{1-a} \rightarrow (1)$$

According to chain rule of calculus

$dz = da \cdot g'(z)$ where $g(z) = \sigma(z)$.

$$dz = \frac{dL}{dz} = \frac{dL}{da} \cdot \frac{da}{dz}$$

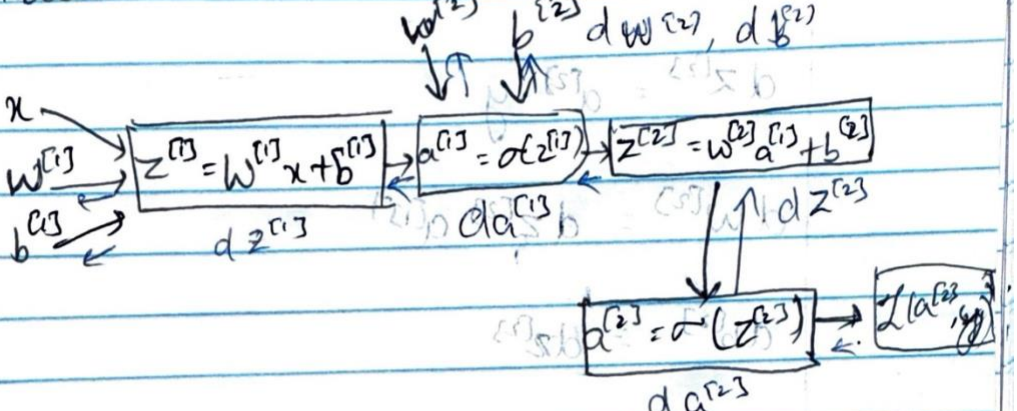
$$= \frac{dL}{da} \cdot \frac{d(\sigma(z))}{z}$$

$$= da \cdot g'(z)$$

Upon solving; $dz = a - y$.

Similarly; $dw = dz \cdot x$ $\quad db = dz$

* Now for a 2NN.



Now, $dz^{[2]} = a^{[2]} - y$

$$dW^{[2]} = dz^{[2]} \cdot a^{[1]T}$$

$$db^{[2]} = dz^{[2]}$$

$$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})$$

$$(n^{[1]}, 1) = (n^{[1]}, n^{[2]})(n^{[2]}, 1) * (n^{[1]}, 1)$$

$$dW^{[1]} = dz^{[1]} \cdot x^T$$

$$db^{[1]} = dz^{[1]}$$

* So now we have:

$$dz^{[2]} = a^{[2]} - y$$

$$dw^{[2]} = dz^{[2]} a^{[1]T}$$

$$db^{[2]} = dz^{[2]}$$

$$dz^{[1]} = w^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})$$

$$dw^{[1]} = dz^{[1]} x^T$$

$$db^{[1]} = dz^{[1]}$$

* Similarly, vectorized implementations are:

$$dZ^{[2]} = A^{[2]} - Y$$

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} \cdot A^{[1]T}$$

$$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis=1, keepdims=True)$$

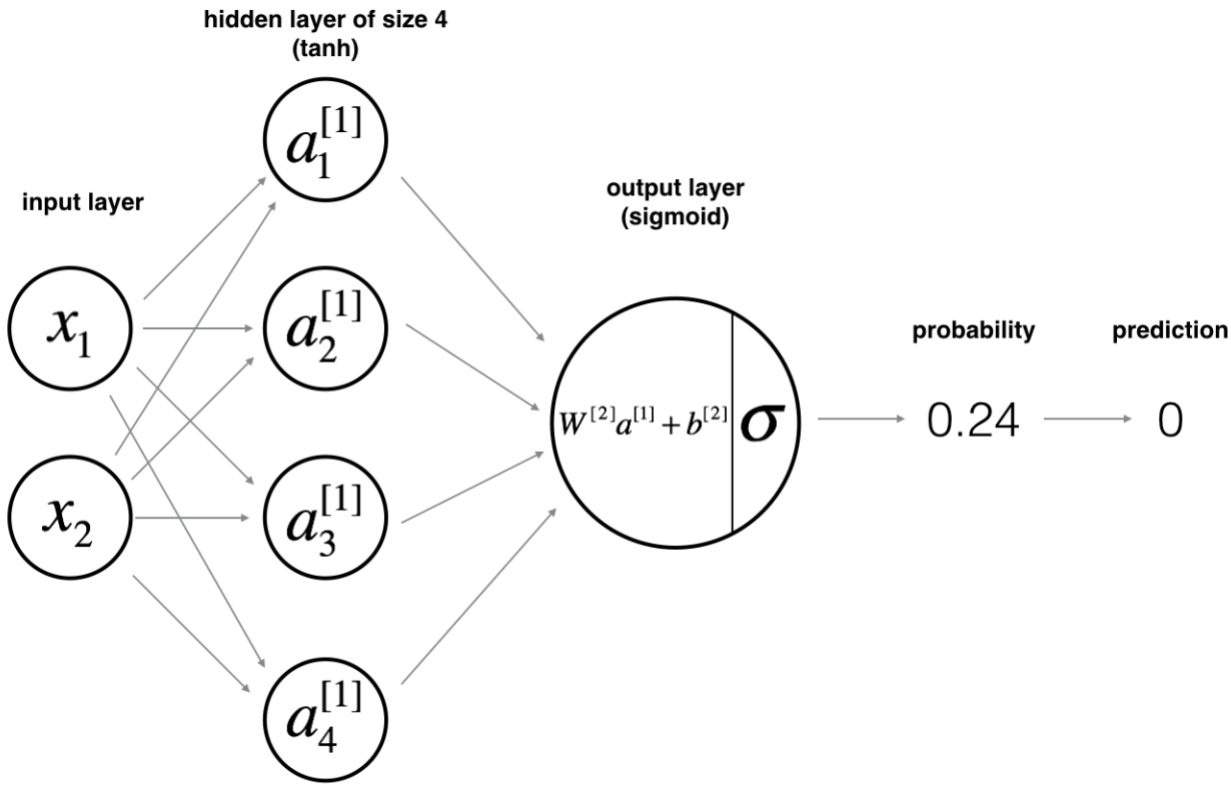
$$\underset{(n^{[1]}, m)}{dZ^{[1]}} = \underbrace{\underset{(n^{[1]}, m)}{W^{[2]T} dZ^{[2]}}}_{} * \underbrace{\underset{(n^{[1]}, m)}{g^{[1]'}(Z^{[1]})}}_{}$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T, \quad db = \frac{1}{m} np.sum(dZ^{[1]}, axis=1, keepdims=True)$$

Neural Network model

Logistic regression didn't work well on the flower dataset. Next, you're going to train a Neural Network with a single hidden layer and see how that handles the same problem.

The model:



Mathematically:

Mathematically:

For one example $x^{(i)}$:

$$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]} \tag{1}$$
$$a^{[1](i)} = \tanh(z^{[1](i)}) \tag{2}$$
$$z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2]} \tag{3}$$
$$\hat{y}^{(i)} = a^{[2](i)} = \sigma(z^{[2](i)}) \tag{4}$$
$$y^{(i)}_{prediction} = \begin{cases} 1 & \text{if } a^{[2](i)} > 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Given the predictions on all the examples, you can also compute the cost J as follows:

$$J = -\frac{1}{m} \sum_{i=0}^{m} \left(y^{(i)} \log\left(a^{[2](i)}\right) + (1 - y^{(i)}) \log\left(1 - a^{[2](i)}\right) \right) \tag{6}$$

Reminder: The general methodology to build a Neural Network is to:

1. Define the neural network structure (# of input units, # of hidden units, etc).

2. Initialize the model's parameters

3. Loop:

   Implement forward propagation

   Compute loss

   Implement backward propagation to get the gradients

   Update parameters (gradient descent)

In practice, you'll often build helper functions to compute steps 1-3, then merge them into one function called `nn_model()`. Once you've built `nn_model()` and learned the right parameters, you can make predictions on new data.

1. Defining the neural network structure

- n_x: the size of the input layer
- n_h: the size of the hidden layer (**set this to 4, as `n_h = 4`, but only for this Exercise 2**)
- n_y: the size of the output layer

```
n_x = X.shape[0]
n_h = 4
n_y = Y.shape[0]
```

2. Initialize the model's parameters

```
W1 = np.random.randn(n_h,n_x)*0.01
b1 = np.zeros((n_h,1))
W2 = np.random.randn(n_y, n_h)*0.01
b2 = np.zeros((n_y,1))
```

3. Loop:

- **Implement forward propagation**

```
Z1 = np.dot(W1,X) + b1
A1 = np.tanh(Z1)
Z2 = np.dot(W2,A1) + b2
A2 = sigmoid(Z2)
```

- **Compute loss**

```
m = Y.shape[1] # number of examples
# Compute the cross-entropy cost
logprobs = np.multiply(np.log(A2), Y) + np.multiply(np.log(1-A2), (1-Y))
cost = (-1/m)*np.sum(logprobs)
```

- **Implement backward propagation to get the gradients**

```
dZ2 = A2  Y
```

```
dW2 = (1/m)*np.dot(dZ2,A1.T)
db2 = (1/m)*np.sum(dZ2, axis = 1, keepdims = True)
dZ1 = np.dot(W2.T, dZ2)*(1  np.power(A1, 2))
dW1 = (1/m)*np.dot(dZ1, X.T)
db1 = (1/m)*np.sum(dZ1, axis = 1, keepdims = True)
```

- **Update parameters (gradient descent)**

```
W1 = W1-learning_rate*dW1
b1 = b1-learning_rate*db1
W2 = W2-learning_rate*dW2
b2 = b2-learning_rate*db2
```

Deep Neural Network Overview

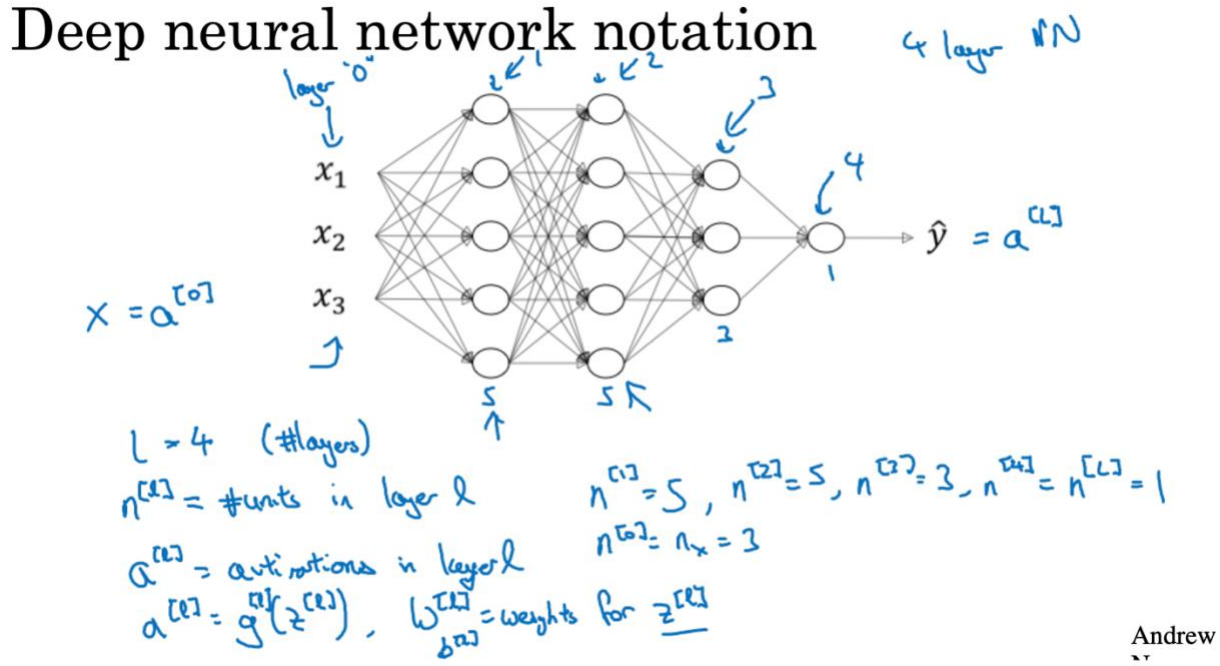
1. What is a Deep Neural Network?

A deep neural network is an extension of a neural network that includes multiple hidden layers between the input and output layers. The depth (number of layers) allows the model to learn more complex functions compared to shallow networks like logistic regression or single-layer neural networks.

2. Shallow vs. Deep Models

• Logistic Regression: A shallow model (1 layer).
• Neural Network with One Hidden Layer: Still considered shallow (2-layer network).
• Neural Network with Multiple Hidden Layers: Deep models. For example, a 5-hidden-layer network is considered deep.

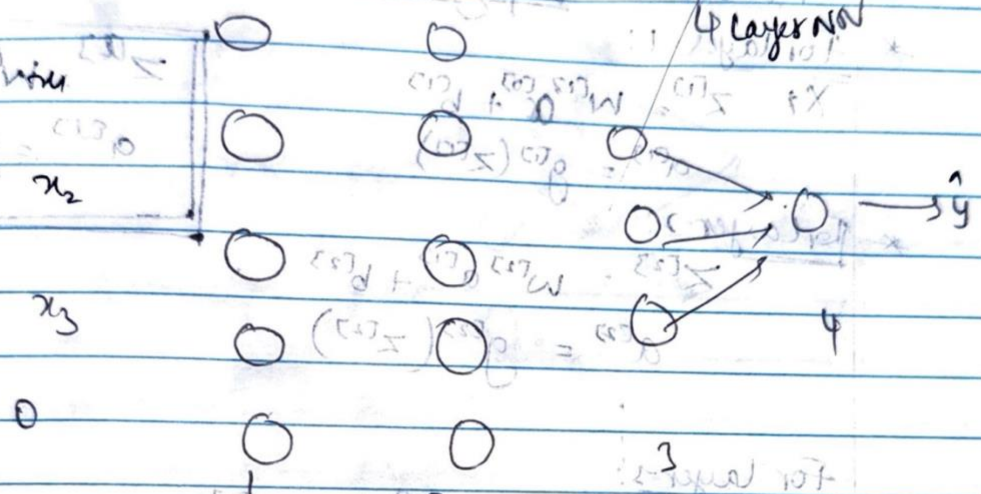
3. Deep Neural Network Notation



Let L denote the number of layers in the network.
$a^{[l]}$: Activations of layer "l"

$z^{[l]}$: Linear combination at layer “l”
$W^{[l]}$: Weight matrix at layer “l”
$b^{[l]}$: Bias vector at layer “l”
$a^{[0]} = x$: Input features
$a^{[L]} = \hat{y}$: Output of the network (prediction)
$n^{[l]}$: Number of units in layer “l”
For example, in a network with 4 layers (L = 4):
$n^{[0]} = 3$ (input features)
$n^{[1]} = 5$
$n^{[2]} = 5$
$n^{[3]} = 3$
$n^{[4]} = 1$ (output)

* Deep Nueral Network



L = 4 (# Layers)

$n^{[l]}$ = No. of units in layer "l".

$a^{[l]}$ = activations in layer "l".

$w^{[l]}$ = weights for $z^{[l]}$, $b^{[l]}$

So; $n^{[1]} = 5$, $n^{[2]} = 5$, $n^{[3]} = 3$, $n^{[4]} <=> n^{[L]} = 1$.

$n^{[0]} = n_x = 3$.

$X = a^{[0]}$

Similarly; $a^{[l]} = g^{[l]}(z^{[l]})$

* Forward Propogation

* For layer-1:

$$z^{[1]} = w^{[1]} a^{[0]} + b^{[1]}$$

$$a^{[1]} = g^{[1]}(z^{[1]})$$

$$\boxed{z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}}$$

$$\boxed{a^{[l]} = g^{[l]}(z^{[l]})}$$

* For layer-2:

$$z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$$

$$a^{[2]} = g^{[2]}(z^{[2]})$$

For layer-3:

$$z^{[3]} = w^{[3]} a^{[2]} + b^{[3]}$$

$$a^{[3]} = g^{[3]}(z^{[3]})$$

For layer 4:

$$z^{[4]} = w^{[4]} a^{[3]} + b^{[4]}$$

$$a^{[4]} = g^{[4]}(z^{[4]})$$

* Vectorization:

$$Z^{[1]} = w^{[1]} A^{[0]} + b^{[1]}$$

$$A^{[1]} = g^{[1]}(Z^{[1]})$$

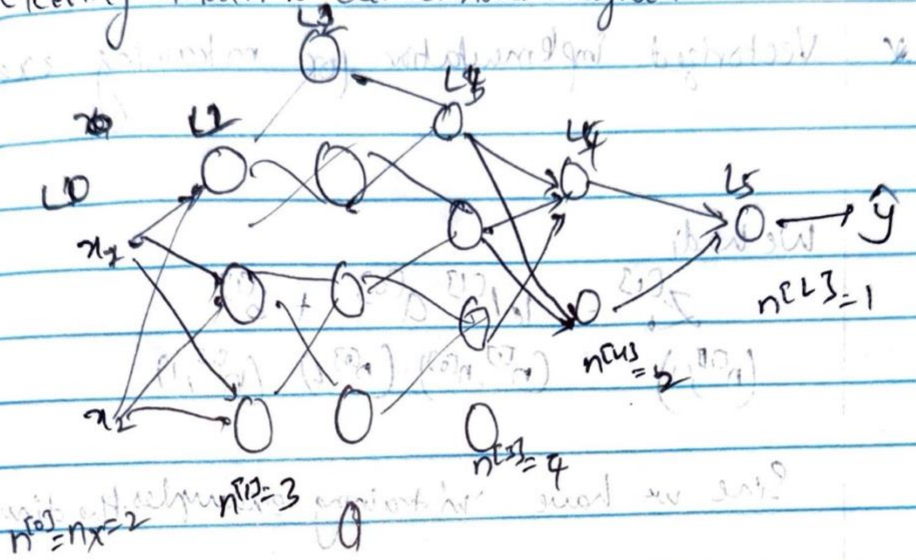
$$\hat{y} = g(Z^{[4]}) = A^{[4]}$$

Where:

$$Z^{[1]} = \begin{bmatrix} | & | & | & & | \\ z^{[1](1)} & z^{[1](2)} & z^{[1](3)} & \cdots & z^{[1](m)} \\ | & | & | & & | \end{bmatrix}$$

$$A^{[1]} = \begin{bmatrix} | & | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \cdots & & a^{[1](m)} \\ | & | & | & & | \end{bmatrix}$$

Getting matrix dimensions right:



So, $z^{[1]} = w^{[1]} \cdot x + b^{[1]}$

$(3,1) \quad (3,2) \quad (2,1) \quad (3,1)$

$(n^{[1]}, 1) \quad (n^{[1]}, n^{[0]}) \quad (n^{[0]}, 1) \quad (n^{[1]}, 1)$

Similarly, $w^{[2]} = (5,3) \quad (n^{[2]}, n^{[1]})$

$b^{[2]} = (3,1) \quad (n^{[2]}, 1)$

So, in general:

$$w^{[l]} = (n^{[l]}, n^{[l-1]})$$

$$b^{[l]} = (n^{[l]}, 1)$$

$$dw^{[l]} = (n^{[l]}, n^{[l-1]})$$

$$db^{[l]} = (n^{[l]}, 1)$$

* Vectorized implementation for m training examples:

We had;

$$z^{[1]} = W^{[1]} a^{[0]} + b^{[1]}$$
$$(n^{[1]}, 1) \quad (n^{[1]}, n^{[0]}) \; (n^{[0]}, 1) \quad (n^{[1]}, 1)$$

Since we have 'm' training examples the dimensions becom

$$Z^{[1]} = W^{[1]} A^{[0]} + b^{[1]}$$
$$(n^{[1]}, m) \quad (n^{[1]}, n^{[0]}) \; (n^{[0]}, m) \quad (n^{[1]}, 1)$$

due to broadcasting python will broadcast to $(n^{[1]}, m)$.

Similarly;

$$Z^{[1]}, A^{[1]} = (n^{[1]}, m)$$

$$dZ^{[1]}, dA^{[1]} = (n^{[1]}, m)$$

~~for~~ $l = 0$; $A^{[0]} = X = (n^{[0]}, m)$.

Mathematical Insight: Why Deep Networks Work

Deep neural networks perform exceptionally well on many tasks, especially due to their depth i.e., multiple hidden layers. This document outlines intuitive and mathematical insights into why depth matters, including a visualization from circuit theory.

1. Intuitive Composition in Deep Networks

A deep network computes complex functions by composing simpler functions layer by layer. For example:

- In image recognition, early layers detect edges.
- Middle layers combine edges to form parts (e.g., eyes, nose).
- Later layers compose these parts into full objects like faces.

Similarly, in speech recognition:

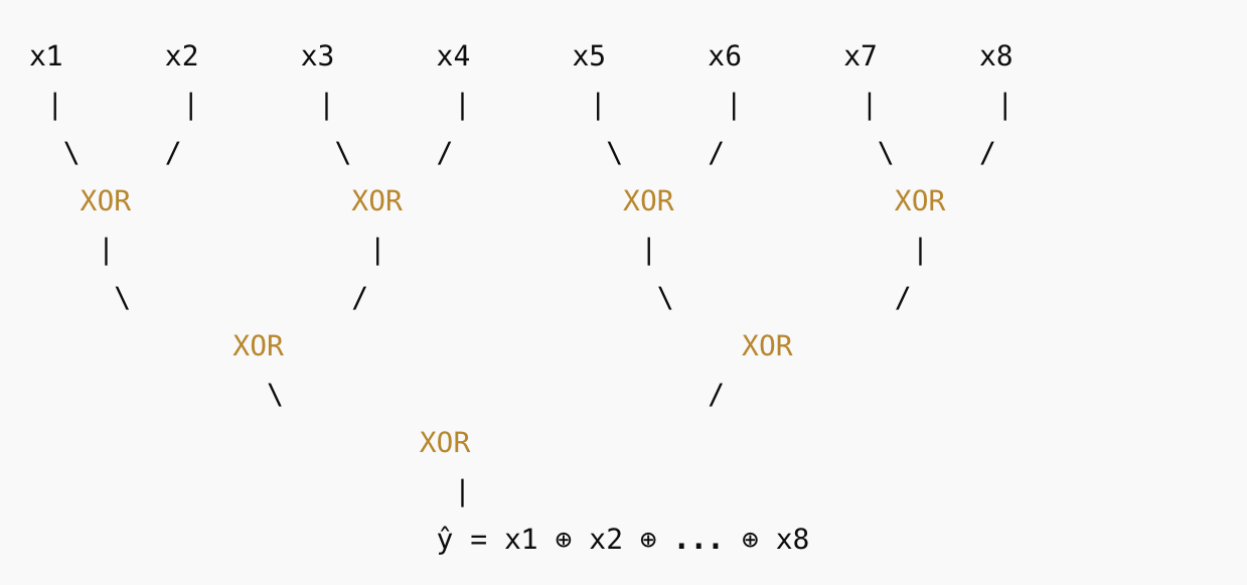
- Early layers detect waveform patterns.
- Intermediate layers detect phonemes.
- Final layers detect full words or phrases.

2. Mathematical Insight: Circuit Theory

Circuit theory provides a formal insight into the power of depth. Some functions, like XOR (exclusive OR), require exponentially more neurons in a shallow network compared to a deep one.

Here's a visual intuition:

In this XOR tree, each level combines inputs using XOR gates. The depth of the tree is $\log_2(N)$, where N is the number of inputs. This structure allows computing complex parity functions with relatively few neurons if multiple layers are allowed.



In contrast, if restricted to a single hidden layer (i.e., a shallow network), one must enumerate all 2^n input combinations. This requires exponentially many neurons, making shallow architectures computationally infeasible for certain tasks.

Forward & Backward Functions:

Layer l: $W^{[l]}, b^{[l]}$

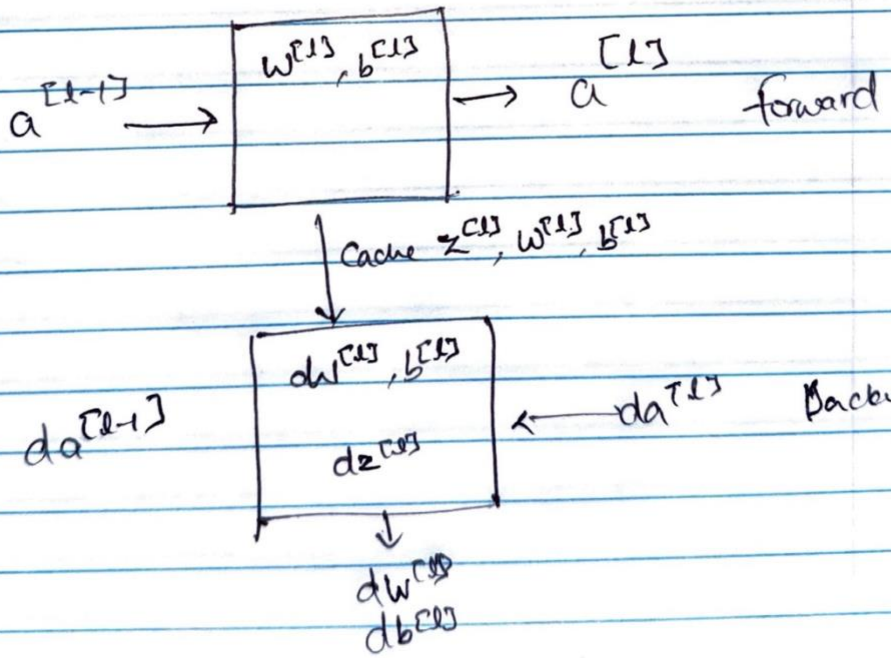
→ Forward: Input $a^{[l-1]}$ and Output $a^{[l]}$ & Cache $z^{[l]}$

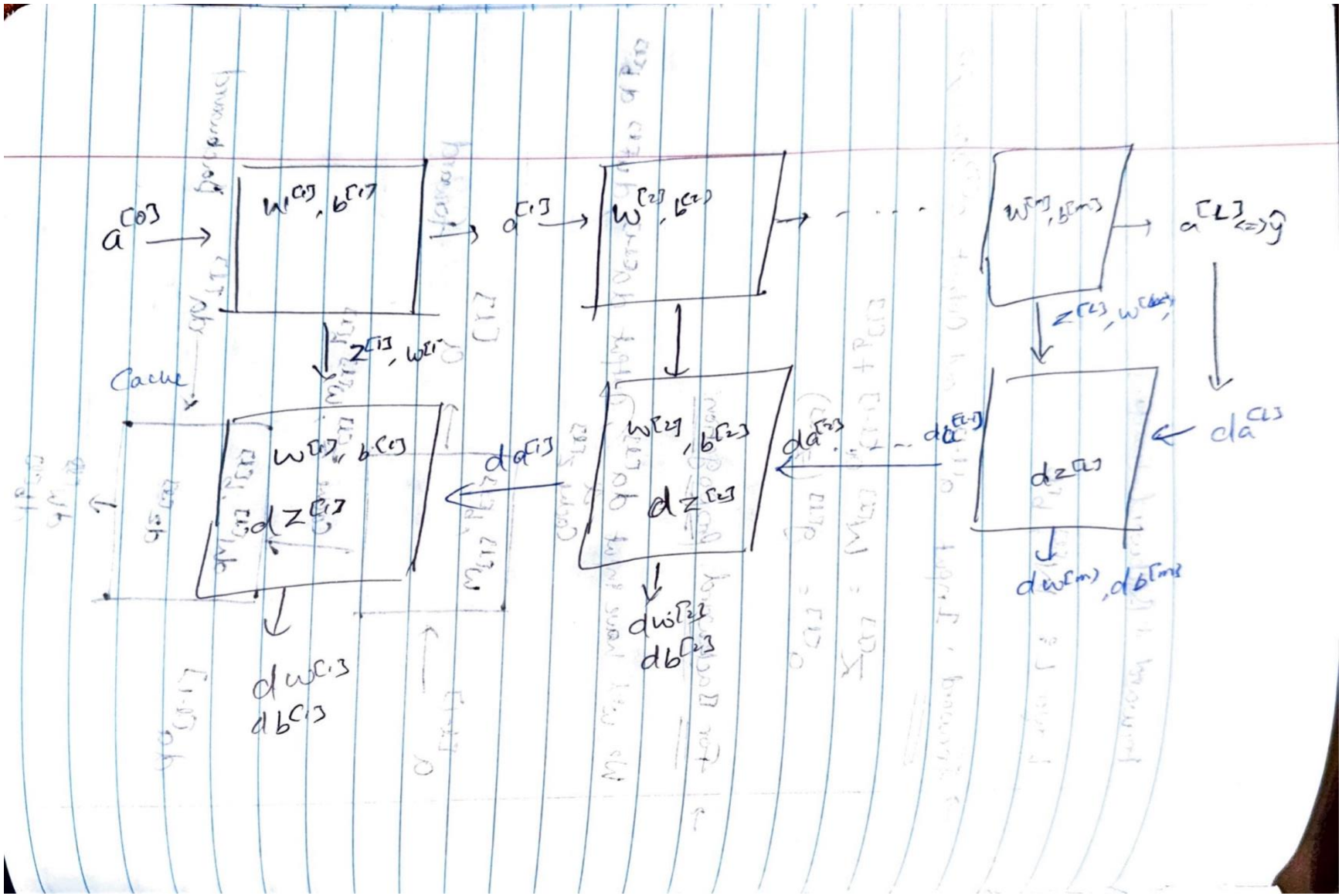
$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$$

$$a^{[l]} = g^{[l]}(z^{[l]})$$

→ For Backward propogation:

We will have input $da^{[l]}$ & Cache $z^{[l]}$, output $da^{[l-1]}, dw^{[l]}, db^{[l]}$.




$a^{[0]}$
$w^{[1]}, b^{[1]}$
$a^{[1]}$
$w^{[2]}, b^{[2]}$
$w^{[L]}, b^{[L]}$
$a^{[L]} = \hat{y}$
$z^{[1]}, w^{[1]}$
$z^{[2]}, w^{[2]}$
$z^{[L]}, w^{[L]}$
Cache
$da^{[1]}$
$da^{[2]}$
$da^{[L-1]}$
$da^{[L]}$
$dz^{[1]}$
$dz^{[2]}$
$dz^{[L]}$
$dw^{[1]}$
$db^{[1]}$
$dw^{[2]}$
$db^{[2]}$
$dw^{[L]}, db^{[L]}$

So; For Farward propogation:

Input = $a^{[l-1]}$ Where; $z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$

$a^{[l]} = g^{[l]}(z^{[l]})$.

Output = $a^{[l]}$

Cache = $z^{[l]}$ When, vectorized:

$Z^{[l]} = W^{[l]} \cdot A^{[l-1]} + b^{[l]}$

$A^{[l]} = g^{[l]}(Z^{[l]})$.

*

For Backward propogation:

Input = $da^{[l]}$

Output: $da^{[l-1]}, dW^{[l]}, db^{[l]}$

where; $dz^{[l]} = da^{[l]} * g^{[l]'}(z^{[l]})$

$dw^{[l]} = dz^{[l]} \cdot a^{[l-1]T}$

$db^{[l]} = dz^{[l]}$

$da^{[l-1]} = w^{[l]T} \cdot dz^{[l]}$

vectorized:

$dZ^{[l]} = dA^{[l]} * g^{[l]'}(Z^{[l]})$

$dW^{[l]} = \frac{1}{m} dZ^{[l]} \cdot A^{[l-1]T}$

$db^{[l]} = \frac{1}{m} np.sum(dZ^{[l]}, axis=1, keepdims=True)$

$dA^{[l-1]} = W^{[l]T} \cdot dZ^{[l]}$

Hyperparameters and Empirical Nature of Deep Learning

1. What Are Hyperparameters?

In deep learning, hyperparameters are values set before the learning process begins. They guide how the model learns the true parameters (weights W and biases b). Examples of hyperparameters include:
Learning rate (α): Controls how large the steps of gradient descent are.
Number of iterations: How many passes to make over the training data.
Number of hidden layers (L): Defines the depth of the network.
Number of hidden units per layer.
Choice of activation functions (ReLU, tanh, sigmoid).
Hyperparameters are different from model parameters. They are not learned from data but instead chosen manually or via search strategies.

2. Why Are Hyperparameters Important?

Hyperparameters determine how effectively and efficiently your model learns. They control how W and b evolve during training. Thus, they have a major impact on final model performance.

3. Empirical Tuning: A Trial and Error Process

Choosing good hyperparameters is often an empirical process. Typical workflow:
1. Choose initial values (e.g., $\alpha = 0.01$).
2. Train the model and observe performance (e.g., does cost function J decrease?).
3. Adjust if needed:
If learning is too slow → Increase α.
If cost blows up → Decrease α.
If cost decreases nicely → Keep α.
This cycle of trying values, training, and adjusting is repeated many times.

4. Challenges Across Different Domains

Deep learning is applied in fields such as computer vision, speech recognition, natural language processing, online advertising, web search, and product recommendations. Each domain might require different hyperparameter settings.
Even when switching between tasks, intuition about hyperparameters may not always carry over. Thus, it is recommended to always try a range of values.

5. Hyperparameter Drift Over Time

Even within the same task (e.g., online advertising), the best hyperparameters may change over time. Possible reasons:
Changes in data distribution.
Improvements in computational infrastructure (CPUs, GPUs).
Updates in the training pipeline.
It is advisable to periodically re-tune hyperparameters (every few months) to maintain optimal performance.

6. Summary and Future Outlook

Currently, tuning hyperparameters remains somewhat empirical, requiring trial-and-error. Future deep learning research may eventually lead to better guidance for choosing hyperparameters. For now, practitioners should:
Try multiple hyperparameter settings.
Evaluate them on a validation set.
Select the best-performing configuration.

Note that the formulas shown in the next video have a few typos. Here is the correct set of formulas.

$$dZ^{[L]} = A^{[L]} - Y$$

$$dW^{[L]} = \frac{1}{m} dZ^{[L]} A^{[L-1]^T}$$

$$db^{[L]} = \frac{1}{m} np.sum(dZ^{[L]}, axis = 1, keepdims = True)$$

$$dZ^{[L-1]} = W^{[L]^T} dZ^{[L]} * g'^{[L-1]}(Z^{[L-1]})$$

Note that * denotes element-wise multiplication)

$$\vdots$$

$$dZ^{[1]} = W^{[2]^T} dZ^{[2]} * g'^{[1]}(Z^{[1]})$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} A^{[0]^T}$$

Note that $A^{[0]^T}$ is another way to denote the input features, which is also written as X^T

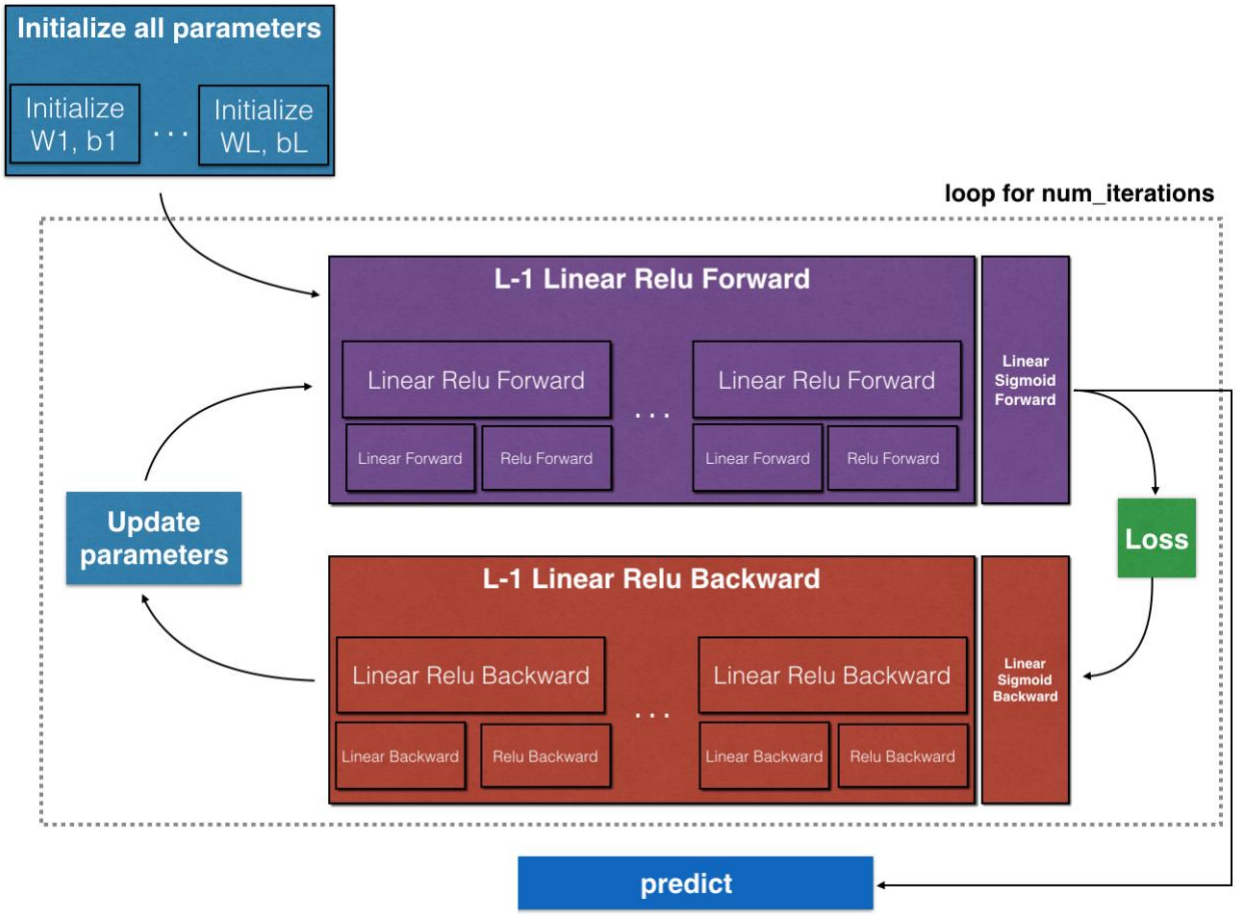
$$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$$

Outline¶

To build your neural network, you'll be implementing several "helper functions." These helper functions will be used in the next assignment to build a two-layer neural network and an L-layer neural network.

Each small helper function will have detailed instructions to walk you through the necessary steps. Here's an outline of the steps in this assignment:

- Initialize the parameters for a two-layer network and for an LL-layer neural network
- Implement the forward propagation module (shown in purple in the figure below)
  - Complete the LINEAR part of a layer's forward propagation step (resulting in Z[l]Z[l]).
  - The ACTIVATION function is provided for you (relu/sigmoid)
  - Combine the previous two steps into a new [LINEAR->ACTIVATION] forward function.
  - Stack the [LINEAR->RELU] forward function L-1 time (for layers 1 through L-1) and add a [LINEAR->SIGMOID] at the end (for the final layer LL). This gives you a new L_model_forward function.
- Compute the loss
- Implement the backward propagation module (denoted in red in the figure below)
  - Complete the LINEAR part of a layer's backward propagation step
  - The gradient of the ACTIVATION function is provided for you(relu_backward/sigmoid_backward)
  - Combine the previous two steps into a new [LINEAR->ACTIVATION] backward function
  - Stack [LINEAR->RELU] backward L-1 times and add [LINEAR->SIGMOID] backward in a new L_model_backward function
- Finally, update the parameters



Note:

For every forward function, there is a corresponding backward function. This is why at every step of your forward module you will be storing some values in a cache. These cached values are useful for computing gradients.

In the backpropagation module, you can then use the cache to calculate the gradients. Don't worry, this assignment will show you exactly how to carry out each of these steps!

Notation:

- Superscript $[l]$ denotes a quantity associated with the l^{th} layer.
  - Example: $a^{[L]}$ is the L^{th} layer activation. $W^{[L]}$ and $b^{[L]}$ are the L^{th} layer parameters.
- Superscript (i) denotes a quantity associated with the i^{th} example.
  - Example: $x^{(i)}$ is the i^{th} training example.
- Lowerscript i denotes the i^{th} entry of a vector.
  - Example: $a_i^{[l]}$ denotes the i^{th} entry of the l^{th} layer's activations).

```
W1 = np.random.randn(n_h,n_x)*0.01
b1 = np.zeros((n_h, 1))
W2 = np.random.randn(n_y, n_h) *0.01
b2 = np.zeros((n_y, 1))
```

```
    W1 = np.random.randn(n_h,n_x)*0.01
    b1 = np.zeros((n_h, 1))
    W2 = np.random.randn(n_y, n_h) *0.01
    b2 = np.zeros((n_y, 1))
    W1 = np.random.randn(n_h,n_x)*0.01
    b1 = np.zeros((n_h, 1))
    W2 = np.random.randn(n_y, n_h) *0.01
    b2 = np.zeros((n_y, 1))
```

1. Initialize parameters:

```
    W1 = np.random.randn(n_h,n_x)*0.01
    b1 = np.zeros((n_h, 1))
    W2 = np.random.randn(n_y, n_h) *0.01
    b2 = np.zeros((n_y, 1))
```

2. initialize_parameters_deep NN

3.2 - L-layer Neural Network

The initialization for a deeper L-layer neural network is more complicated because there are many more weight matrices and bias vectors. When completing the `initialize_parameters_deep` function, you should make sure that your dimensions match between each layer. Recall that $n^{[l]}$ is the number of units in layer l. For example, if the size of your input X is $(12288, 209)$ (with $m = 209$ examples) then:

	Shape of W	Shape of b	Activation	Shape of Activation
Layer 1	$(n^{[1]}, 12288)$	$(n^{[1]}, 1)$	$Z^{[1]} = W^{[1]} X + b^{[1]}$	$(n^{[1]}, 209)$
Layer 2	$(n^{[2]}, n^{[1]})$	$(n^{[2]}, 1)$	$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$	$(n^{[2]}, 209)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Layer L-1	$(n^{[L-1]}, n^{[L-2]})$	$(n^{[L-1]}, 1)$	$Z^{[L-1]} = W^{[L-1]} A^{[L-2]} + b^{[L-1]}$	$(n^{[L-1]}, 209)$
Layer L	$(n^{[L]}, n^{[L-1]})$	$(n^{[L]}, 1)$	$Z^{[L]} = W^{[L]} A^{[L-1]} + b^{[L]}$	$(n^{[L]}, 209)$

Remember that when you compute $WX + b$ in python, it carries out broadcasting. For example, if:

$$W = \begin{bmatrix} w_{00} & w_{01} & w_{02} \\ w_{10} & w_{11} & w_{12} \\ w_{20} & w_{21} & w_{22} \end{bmatrix} \quad X = \begin{bmatrix} x_{00} & x_{01} & x_{02} \\ x_{10} & x_{11} & x_{12} \\ x_{20} & x_{21} & x_{22} \end{bmatrix} \quad b = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} \tag{2}$$

Then $WX + b$ will be:

$$WX + b = \begin{bmatrix} (w_{00}x_{00} + w_{01}x_{10} + w_{02}x_{20}) + b_0 & (w_{00}x_{01} + w_{01}x_{11} + w_{02}x_{21}) + b_0 & \cdots \\ (w_{10}x_{00} + w_{11}x_{10} + w_{12}x_{20}) + b_1 & (w_{10}x_{01} + w_{11}x_{11} + w_{12}x_{21}) + b_1 & \cdots \\ (w_{20}x_{00} + w_{21}x_{10} + w_{22}x_{20}) + b_2 & (w_{20}x_{01} + w_{21}x_{11} + w_{22}x_{21}) + b_2 & \cdots \end{bmatrix} \tag{3}$$

```
    parameters['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims[l-1])*0.01
    parameters['b' + str(l)] = np.zeros((layer_dims[l],1))
```

3. linear_forward

```
    Z = np.dot(W, A) + b
    # YOUR CODE ENDS HERE
    cache = (A, W, b)
```

4. linear_activation_forward

4 - Forward Propagation Module

4.1 - Linear Forward ¶

Now that you have initialized your parameters, you can do the forward propagation module. Start by implementing some basic functions that you can use again later when implementing the model. Now, you'll complete three functions in this order:

- LINEAR
- LINEAR -> ACTIVATION where ACTIVATION will be either ReLU or Sigmoid.
- [LINEAR -> RELU] × (L-1) -> LINEAR -> SIGMOID (whole model)

The linear forward module (vectorized over all the examples) computes the following equations:

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]} \tag{4}$$

where $A^{[0]} = X$.

Exercise 3 - linear_forward

Build the linear part of forward propagation.

Reminder: The mathematical representation of this unit is $Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$. You may also find `np.dot()` useful. If your dimensions don't match, printing `W.shape` may help.

4.2 - Linear-Activation Forward

In this notebook, you will use two activation functions:

- **Sigmoid**: $\sigma(Z) = \sigma(WA + b) = \frac{1}{1+e^{-(WA+b)}}$. You've been provided with the `sigmoid` function which returns **two** items: the activation value " `a` " and a " `cache` " that contains " `Z` " (it's what we will feed in to the corresponding backward function). To use it you could just call:

```
A, activation_cache = sigmoid(Z)
```

- **ReLU**: The mathematical formula for ReLu is $A = RELU(Z) = max(0, Z)$. You've been provided with the `relu` function. This function returns **two** items: the activation value " `A` " and a " `cache` " that contains " `Z` " (it's what you'll feed in to the corresponding backward function). To use it you could just call:

```
A, activation_cache = relu(Z)
```

For added convenience, you're going to group two functions (Linear and Activation) into one function (LINEAR->ACTIVATION). Hence, you'll implement a function that does the LINEAR forward step, followed by an ACTIVATION forward step.

Exercise 4 - linear_activation_forward

Implement the forward propagation of the *LINEAR->ACTIVATION* layer. Mathematical relation is: $A^{[l]} = g(Z^{[l]}) = g(W^{[l]}A^{[l-1]} + b^{[l]})$ where the activation "g" can be sigmoid() or relu(). Use `linear_forward()` and the correct activation function.

```
if activation == "sigmoid":
    #(≈ 2 lines of code)
    # Z, linear_cache = ...
    # A, activation_cache = ...
    # YOUR CODE STARTS HERE
    Z, linear_cache = linear_forward(A_prev, W, b)
    A, activation_cache = sigmoid(Z)

    # YOUR CODE ENDS HERE

elif activation == "relu":
    #(≈ 2 lines of code)
    # Z, linear_cache = ...
    # A, activation_cache = ...
    # YOUR CODE STARTS HERE
    Z, linear_cache = linear_forward(A_prev, W, b)
```

```
    A, activation_cache = relu(Z)

    # YOUR CODE ENDS HERE
cache = (linear_cache, activation_cache)
```

4.3 - L-Layer Model

For even *more* convenience when implementing the L-layer Neural Net, you will need a function that replicates the previous one (`linear_activation_forward` with RELU) $L-1$ times, then follows that with one `linear_activation_forward` with SIGMOID.

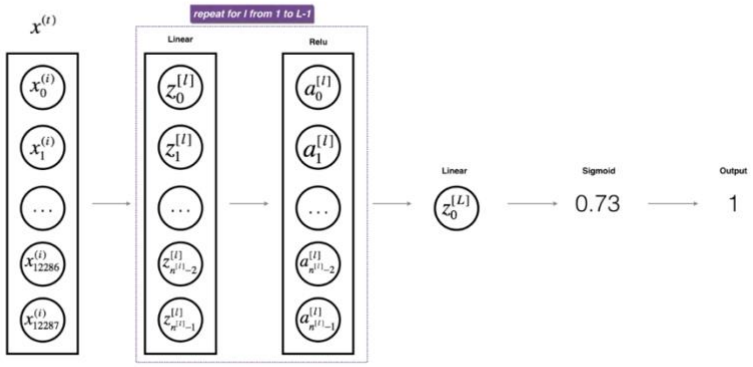


Figure 2 : *[LINEAR -> RELU] × (L-1) -> LINEAR -> SIGMOID* model

5. L_model_forward

Exercise 5 - L_model_forward

Implement the forward propagation of the above model.

Instructions: In the code below, the variable `AL` will denote $A^{[L]} = \sigma(Z^{[L]}) = \sigma(W^{[L]} A^{[L-1]} + b^{[L]})$. (This is sometimes also called `Yhat`, i.e., this is $\hat{Y}$.)

Hints:

- Use the functions you've previously written
- Use a for loop to replicate [LINEAR->RELU] (L-1) times
- Don't forget to keep track of the caches in the "caches" list. To add a new value `c` to a `list`, you can use `list.append(c)`.

```
caches = []
A = X
L = len(parameters) // 2                  # number of layers in the neural network

# Implement [LINEAR -> RELU]*(L-1). Add "cache" to the "caches" list.
# The for loop starts at 1 because layer 0 is the input
for l in range(1, L):
    A_prev = A
    #(≈ 2 lines of code)
    # A, cache = ...
    # caches ...
    # YOUR CODE STARTS HERE
    A, cache = linear_activation_forward(A_prev, parameters["W" + str(l)],parameters["b" + str(l)], "relu")
    caches.append(cache)
    # YOUR CODE ENDS HERE

# Implement LINEAR -> SIGMOID. Add "cache" to the "caches" list.
#(≈ 2 lines of code)
```

```
# AL, cache = ...
# caches ...
# YOUR CODE STARTS HERE
AL, cache = linear_activation_forward(A, parameters["W" + str(L)],parameters["b" + str(L)], "sigmoid")
caches.append(cache)

# YOUR CODE ENDS HERE
```

Exercise 6 compute_cost**:**

5 - Cost Function

Now you can implement forward and backward propagation! You need to compute the cost, in order to check whether your model is actually learning.

Exercise 6 - compute_cost

Compute the cross-entropy cost J, using the following formula:

$$-\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}\log(a^{[L](i)}) + (1-y^{(i)})\log(1-a^{[L](i)})) \tag{7}$$

cost = (-1/m) * np.sum(Y * np.log(AL) + (1 Y) * np.log(1 AL))
let's analyze the cost function implementations for a neural network designed for binary classification.

1. Incorrect Implementation
Python
```
cost = (-1/m) * np.sum((Y * np.log(AL)), ((1-Y) * np.log(1-AL)))
```
- Why Incorrect:
  - np.sum() is intended to sum the elements of a single array.
  - In this incorrect version, np.sum() is given two separate arrays as arguments: (Y * np.log(AL)) and ((1-Y) * np.log(1-AL)).
  - This does not perform the element-wise summation needed for the cross-entropy loss calculation. It tries to sum a "tuple" of arrays, leading to errors or an unintended result.

2. Correct but Not Optimal
Python
```
cost = (-1/m) * (np.dot(Y, np.log(AL).T) + np.dot((1-Y), np.log(1-AL).T))
```
- Why Correct (but not optimal):
  - This version uses np.dot(), which performs matrix multiplication.
  - By transposing np.log(AL) and np.log(1-AL) (using .T), the code adjusts the matrix dimensions to be compatible for np.dot(). If AL and Y are both (1, m), then np.log(AL).T is (m, 1), and the dot product results in a (1, 1) matrix (a scalar).
  - It calculates the cost, but it's less efficient than the optimal method because matrix multiplication is computationally more expensive than element-wise multiplication and summation.
- Why Transpose:
  - The transpose (.T) is necessary to align the dimensions of the matrices for the np.dot() operation.
  - np.dot() requires specific matrix dimensions to perform multiplication, and the transpose rearranges the rows and columns to satisfy those requirements.

3. Optimal Implementation
Python
```
cost = (-1/m) * np.sum(Y * np.log(AL) + (1  Y) * np.log(1  AL))
```
- Why Optimal:
  - This is the most efficient and direct way to calculate the cross-entropy cost.

- Y * np.log(AL) and (1-Y) * np.log(1 AL) perform element-wise multiplication. This directly calculates the loss for each training example.
- The + operator then adds the two components of the loss *for each example*.
- np.sum() sums the losses across all training examples.
- No transpose is needed because the operations are element-wise, so the arrays don't need to be reshaped for matrix multiplication. Element-wise operations are highly optimized in NumPy, making this implementation very efficient.

In summary, the optimal solution leverages NumPy's efficient element-wise operations, avoiding the computational overhead of matrix multiplication and unnecessary transposes.

6 - Backward Propagation Module

Just as you did for the forward propagation, you'll implement helper functions for backpropagation. Remember that backpropagation is used to calculate the gradient of the loss function with respect to the parameters.

Reminder:

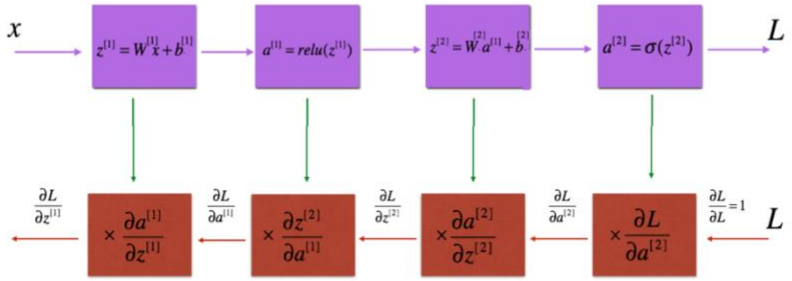


Figure 3: Forward and Backward propagation for LINEAR->RELU->LINEAR->SIGMOID
The purple blocks represent the forward propagation, and the red blocks represent the backward propagation.

Now, similarly to forward propagation, you're going to build the backward propagation in three steps:

1. LINEAR backward
2. LINEAR -> ACTIVATION backward where ACTIVATION computes the derivative of either the ReLU or sigmoid activation
3. [LINEAR -> RELU] × (L-1) -> LINEAR -> SIGMOID backward (whole model)

7. linear_backward

6.1 - Linear Backward

For layer l, the linear part is: $Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$ (followed by an activation).

Suppose you have already calculated the derivative $dZ^{[l]} = \frac{\partial \mathcal{L}}{\partial Z^{[l]}}$. You want to get $(dW^{[l]}, db^{[l]}, dA^{[l-1]})$.

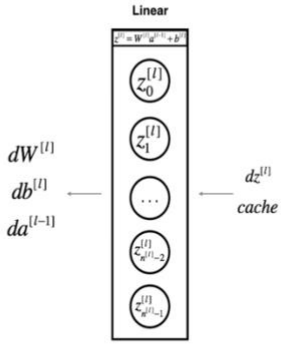


Figure 4

The three outputs $(dW^{[l]}, db^{[l]}, dA^{[l-1]})$ are computed using the input $dZ^{[l]}$.

Here are the formulas you need:

$$dW^{[l]} = \frac{\partial \mathcal{J}}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T} \tag{8}$$

$$db^{[l]} = \frac{\partial \mathcal{J}}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[l](i)} \tag{9}$$

$$dA^{[l-1]} = \frac{\partial \mathcal{L}}{\partial A^{[l-1]}} = W^{[l]T} dZ^{[l]} \tag{10}$$

$A^{[l-1]T}$ is the transpose of $A^{[l-1]}$.

```
A_prev, W, b = cache
m = A_prev.shape[1]
START CODE HERE (≈ 3 lines of code)
# dW = ...
# db = ... sum by the rows of dZ with keepdims=True
# dA_prev = ...
# YOUR CODE STARTS HERE
dW = (1/m)*np.dot(dZ,A_prev.T)
db = (1/m)*np.sum(dZ, axis = 1, keepdims=True)
dA_prev = np.dot(W.T, dZ)
```

8. linear_activation_backward

6.2 - Linear-Activation Backward

Next, you will create a function that merges the two helper functions: **`linear_backward`** and the backward step for the activation **`linear_activation_backward`**.

To help you implement `linear_activation_backward`, two backward functions have been provided:

- **`sigmoid_backward`**: Implements the backward propagation for SIGMOID unit. You can call it as follows:

```
dZ = sigmoid_backward(dA, activation_cache)
```

- **`relu_backward`**: Implements the backward propagation for RELU unit. You can call it as follows:

```
dZ = relu_backward(dA, activation_cache)
```

If $g(.)$ is the activation function, `sigmoid_backward` and `relu_backward` compute

$$dZ^{[l]} = dA^{[l]} * g'(Z^{[l]}). \tag{11}$$

Exercise 8 - linear_activation_backward

Implement the backpropagation for the *LINEAR->ACTIVATION* layer.

◈ linear_cache

Stores values from the linear computation Z = W @ A_prev + b.
It includes:

- A_prev: activation from the previous layer
- W: weights of the current layer
- b: biases of the current layer

Used to compute gradients during linear_backward.

◈ activation_cache

Stores Z, the input to the activation function (e.g., sigmoid(Z) or relu(Z)).
Needed to compute the gradient of the activation during activation_backward.

Together, they are stored as:

cache = (linear_cache, activation_cache)

Used in back propagation to compute:

- dA_prev, dW, db via linear_backward
- dZ via sigmoid_backward or relu_backward

```
    linear_cache, activation_cache = cache

    if activation == "relu":
        #(≈ 2 lines of code)
        # dZ =  ...
        # dA_prev, dW, db =  ...
        # YOUR CODE STARTS HERE
        dZ = relu_backward(dA, activation_cache)
        dA_prev, dW, db =  linear_backward(dZ, linear_cache)

        # YOUR CODE ENDS HERE

    elif activation == "sigmoid":
        #(≈ 2 lines of code)
        # dZ =  ...
        # dA_prev, dW, db =  ...
        # YOUR CODE STARTS HERE
        dZ = sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
        # YOUR CODE ENDS HERE
```

9. L_model_backward

6.3 - L-Model Backward

Now you will implement the backward function for the whole network!

Recall that when you implemented the `L_model_forward` function, at each iteration, you stored a cache which contains (X,W,b, and z). In the back propagation module, you'll use those variables to compute the gradients. Therefore, in the `L_model_backward` function, you'll iterate through all the hidden layers backward, starting from layer L. On each step, you will use the cached values for layer l to backpropagate through layer l. Figure 5 below shows the backward pass.

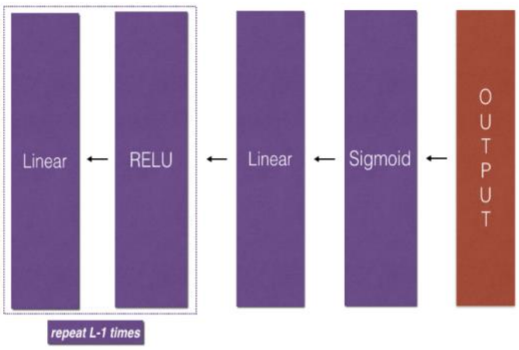


Figure 5: Backward pass

Initializing backpropagation:

To backpropagate through this network, you know that the output is: $A^{[L]} = \sigma(Z^{[L]})$. Your code thus needs to compute `dAL` $= \frac{\partial \mathcal{L}}{\partial A^{[L]}}$. To do so, use this formula (derived using calculus which, again, you don't need in-depth knowledge of!):

```
dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL)) # derivative of cost with respect to AL
```

You can then use this post-activation gradient `dAL` to keep going backward. As seen in Figure 5, you can now feed in `dAL` into the LINEAR->SIGMOID backward function you implemented (which will use the cached values stored by the L_model_forward function).

After that, you will have to use a `for` loop to iterate through all the other layers using the LINEAR->RELU backward function. You should store each dA, dW, and db in the grads dictionary. To do so, use this formula :

$$grads[" dW " + str(l)] = dW^{[l]} \tag{15}$$

For example, for $l = 3$ this would store $dW^{[l]}$ in `grads["dW3"]`.

Exercise 9 - L_model_backward

Implement backpropagation for the *[LINEAR->RELU] × (L-1) -> LINEAR -> SIGMOID* model.

```
grads = {}
    L = len(caches) # the number of layers
    m = AL.shape[1]
    Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL

    # Initializing the backpropagation
    dAL =  (np.divide(Y, AL)  np.divide(1  Y, 1  AL))


    current_cache = caches[L-1]
    dA_prev_temp, dW_temp, db_temp = linear_activation_backward(dAL, current_cache, activation = "sigmoid")
    grads["dA" + str(L-1)] = dA_prev_temp
    grads["dW" + str(L)] = dW_temp
    grads["db" + str(L)] = db_temp

    # YOUR CODE ENDS HERE
```

```
    # Loop from l=L-2 to l=0
    for l in reversed(range(L-1)):
        # lth layer: (RELU -> LINEAR) gradients.
        # Inputs: "grads["dA" + str(l + 1)], current_cache". Outputs: "grads["dA" + str(l)] , grads["dW" + str(l + 1)] , grads["db" + str(l + 1)]
        #(approx. 5 lines)
        # current_cache = ...
        # dA_prev_temp, dW_temp, db_temp = ...
        # grads["dA" + str(l)] = ...
        # grads["dW" + str(l + 1)] = ...
        # grads["db" + str(l + 1)] = ...
        # YOUR CODE STARTS HERE
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linear_activation_backward(grads["dA" + str(l + 1)], current_cache, activation = "relu")
        grads["dA" + str(l)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp
```

10. update_parameters

6.4 - Update Parameters

In this section, you'll update the parameters of the model, using gradient descent:

$$W^{[l]} = W^{[l]} - \alpha \, dW^{[l]} \tag{16}$$
$$b^{[l]} = b^{[l]} - \alpha \, db^{[l]} \tag{17}$$

where α is the learning rate.

After computing the updated parameters, store them in the parameters dictionary.

Exercise 10 - update_parameters

Implement `update_parameters()` to update your parameters using gradient descent.

Instructions: Update parameters using gradient descent on every $W^{[l]}$ and $b^{[l]}$ for $l = 1, 2, \ldots, L$.

```
    """
    parameters = copy.deepcopy(params)
    L = len(parameters) // 2 # number of layers in the neural network
    # Update rule for each parameter. Use a for loop.
    #(≈ 2 lines of code)
    for l in range(L):
        # parameters["W" + str(l+1)] = ...
        # parameters["b" + str(l+1)] = ...
        # YOUR CODE STARTS HERE
        parameters["W" + str(l+1)] = parameters["W" + str(l+1)]  learning_rate * grads["dW" + str(l+1)]
        parameters["b" + str(l+1)] = parameters["b" + str(l+1)]  learning_rate * grads["db" + str(l+1)]
```

3 - Model Architecture

3.1 - 2-layer Neural Network

Now that you're familiar with the dataset, it's time to build a deep neural network to distinguish cat images from non-cat images!

You're going to build two different models:

- A 2-layer neural network
- An L-layer deep neural network

Then, you'll compare the performance of these models, and try out some different values for L.

Let's look at the two architectures:

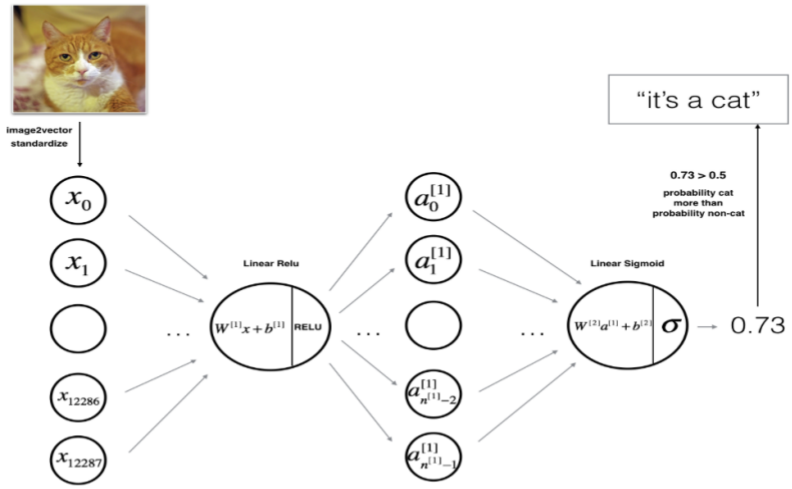


Figure 2: 2-layer neural network.
The model can be summarized as: INPUT -> LINEAR -> RELU -> LINEAR -> SIGMOID -> OUTPUT.

Detailed Architecture of Figure 2:

- The input is a (64,64,3) image which is flattened to a vector of size $(12288, 1)$.
- The corresponding vector: $[x_0, x_1, \ldots, x_{12287}]^T$ is then multiplied by the weight matrix $W^{[1]}$ of size $(n^{[1]}, 12288)$.
- Then, add a bias term and take its relu to get the following vector: $[a_0^{[1]}, a_1^{[1]}, \ldots, a_{n^{[1]}-1}^{[1]}]^T$.
- Multiply the resulting vector by $W^{[2]}$ and add the intercept (bias).
- Finally, take the sigmoid of the result. If it's greater than 0.5, classify it as a cat.

3.2 - L-layer Deep Neural Network

It's pretty difficult to represent an L-layer deep neural network using the above representation. However, here is a simplified network representation:

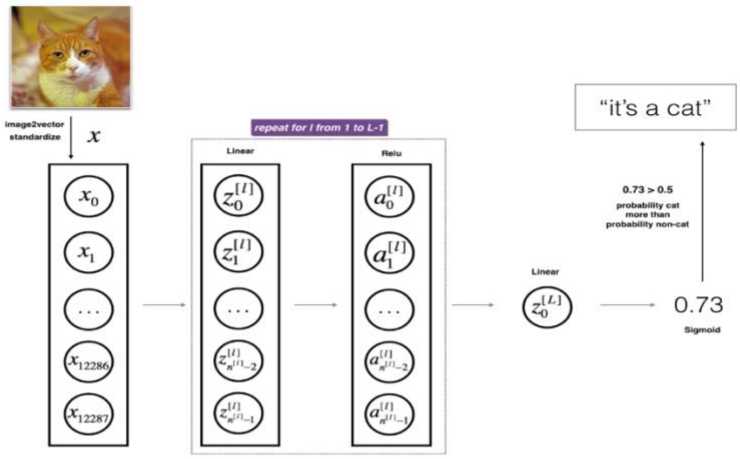


Figure 3: L-layer neural network.
The model can be summarized as: [LINEAR -> RELU] × (L-1) -> LINEAR -> SIGMOID

Detailed Architecture of Figure 3:

- The input is a (64,64,3) image which is flattened to a vector of size (12288,1).
- The corresponding vector: $[x_0, x_1, \ldots, x_{12287}]^T$ is then multiplied by the weight matrix $W^{[1]}$ and then you add the intercept $b^{[1]}$. The result is called the linear unit.
- Next, take the relu of the linear unit. This process could be repeated several times for each $(W^{[l]}, b^{[l]})$ depending on the model architecture.
- Finally, take the sigmoid of the final linear unit. If it is greater than 0.5, classify it as a cat.

3.3 - General Methodology

As usual, you'll follow the Deep Learning methodology to build the model:

1. Initialize parameters / Define hyperparameters
2. Loop for num_iterations: a. Forward propagation b. Compute cost function c. Backward propagation d. Update parameters (using parameters, and grads from backprop)
3. Use trained parameters to predict labels

Now go ahead and implement those two models!

4 - Two-layer Neural Network

Exercise 1 - two_layer_model

Use the helper functions you have implemented in the previous assignment to build a 2-layer neural network with the following structure: *LINEAR -> RELU -> LINEAR -> SIGMOID*. The functions and their inputs are:

```
def initialize_parameters(n_x, n_h, n_y):
    ...
    return parameters
def linear_activation_forward(A_prev, W, b, activation):
    ...
    return A, cache
def compute_cost(AL, Y):
    ...
    return cost
def linear_activation_backward(dA, cache, activation):
    ...
    return dA_prev, dW, db
def update_parameters(parameters, grads, learning_rate):
    ...
    return parameters
```

5 - L-layer Neural Network

Exercise 2 - L_layer_model

Use the helper functions you implemented previously to build an L-layer neural network with the following structure: *[LINEAR -> RELU]×(L-1) -> LINEAR -> SIGMOID*. The functions and their inputs are:

```
def initialize_parameters_deep(layers_dims):
    ...
    return parameters
def L_model_forward(X, parameters):
    ...
    return AL, caches
def compute_cost(AL, Y):
    ...
    return cost
def L_model_backward(AL, Y, caches):
    ...
    return grads
def update_parameters(parameters, grads, learning_rate):
    ...
    return parameters
```

Course 2: Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization

Practical Aspects of Deep Learning:

1. Overview of Practical Deep Learning

Goal: Learn how to make a neural network work well in practice.
Topics covered:
Hyperparameter tuning
Data set setup (train/dev/test)
Efficient optimization and debugging

2. Iterative Process in Applied Machine Learning

When starting a project:
Choose initial settings: number of layers, number of hidden units, learning rate, activation functions.
Important: It is almost impossible to guess the best settings initially.
Typical workflow:
1. Build a neural network (choice of layers, units, etc.).
2. Train and run an experiment.
3. Evaluate the result.
4. Refine the design and repeat.

3. Train/Dev/Test Set Setup

Traditional Setup

Split datasets:
70% training / 30% testing (older practice)
60% training / 20% dev / 20% test

Modern Big Data Setup

If millions of examples:
Use very small dev/test sets (e.g., 1% dev, 1% test).
Example: 1 million examples → 10,000 for dev, 10,000 for test.
Reason: You only need enough samples to choose between model options, not a full large dev/test set.

5. Train and Test Set Distribution Matching

Important rule of thumb:
Dev and test sets should come from the same distribution.
The training set can come from a different distribution (e.g., scraped web images vs. user-uploaded mobile photos).
Reason:
You tune hyperparameters based on dev set performance.
You want the dev set to reflect the test conditions closely.

6. When It's Okay Not to Have a Separate Test Set

If you don't need an unbiased final evaluation:
Use only the train and dev sets.
Technically, the "test set" becomes a "dev set" (hold-out validation).
Caution:
Many teams misuse the term "test set" when they really mean "dev set" (since they tune hyperparameters on it).
If no separate test set: Final reported performance may be slightly optimistic.

Key Takeaways

Deep learning development is highly empirical and iterative.
Efficient progress depends on:
Smart data setup.
Iterative tuning and evaluation.
Rule:
Train and dev/test can have different distributions, but dev and test must match.