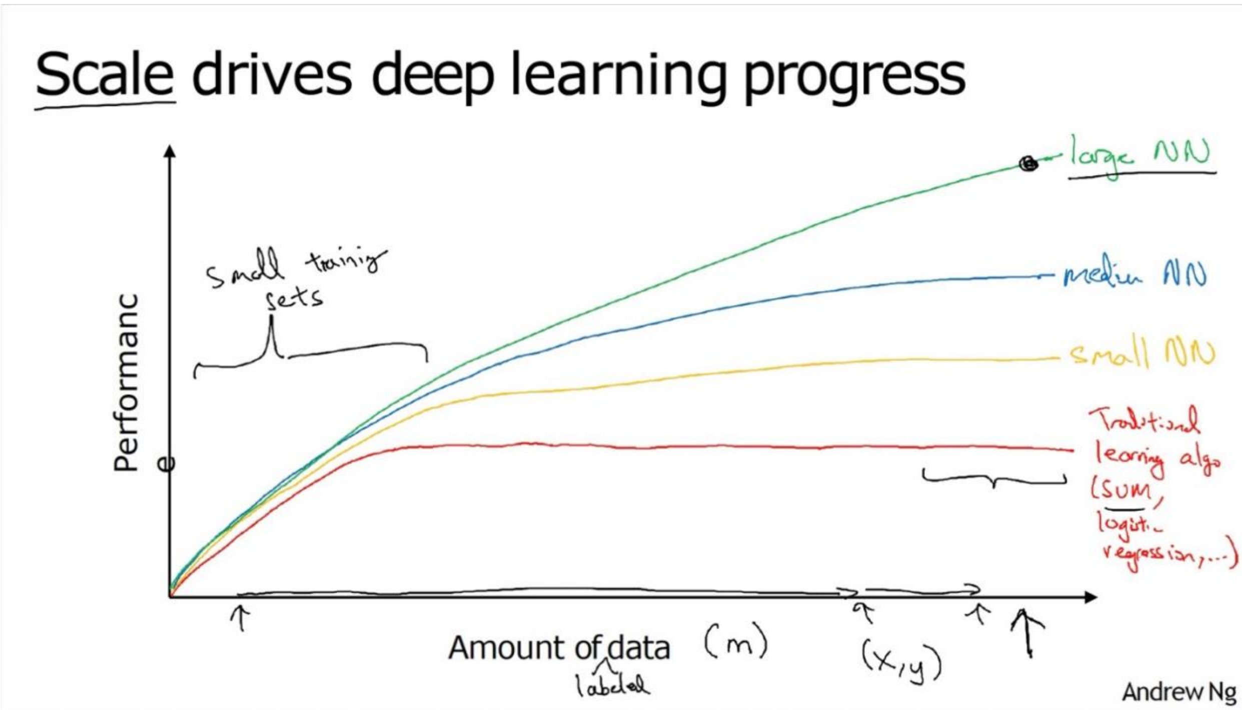
Deep Learning

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 ≤ 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: σ(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

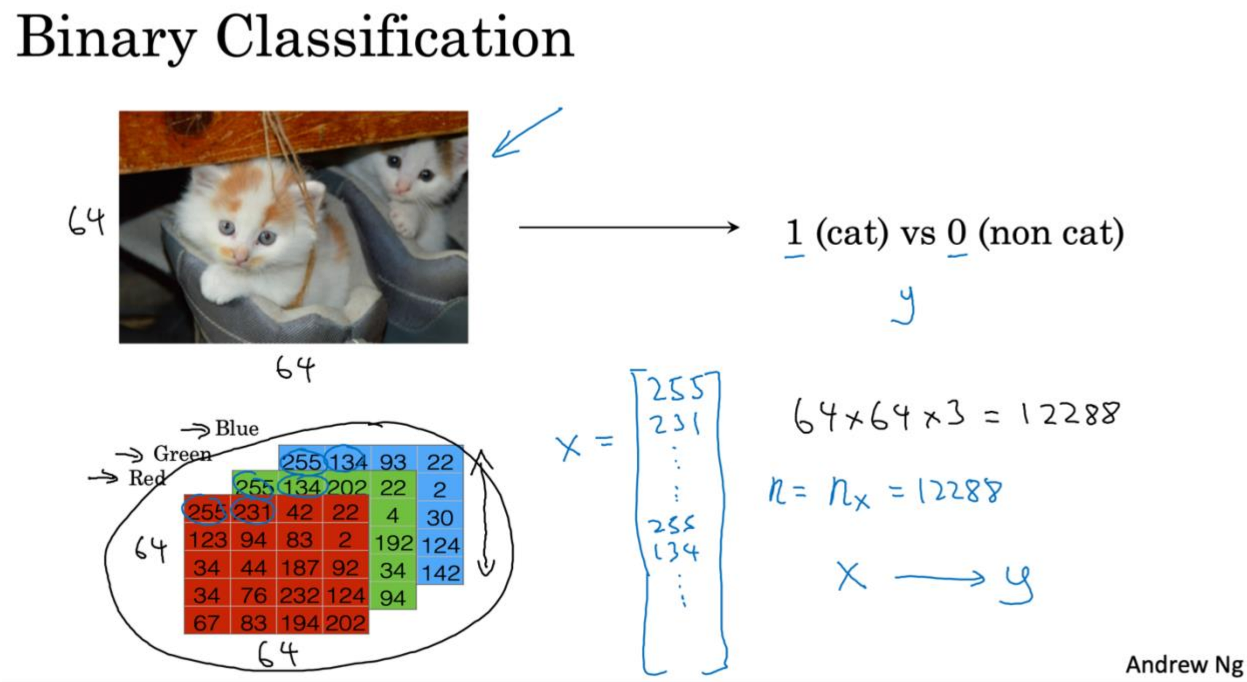
The Dying ReLU problem is a scenario where neurons using ReLU become permanently inactive due to consistently receiving negative inputs, leading to a zero gradient and preventing further learning. Leaky ReLU and other variants, along with careful initialization and learning rate strategies, can help 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	$\sigma(x)*(1-\sigma(x))$

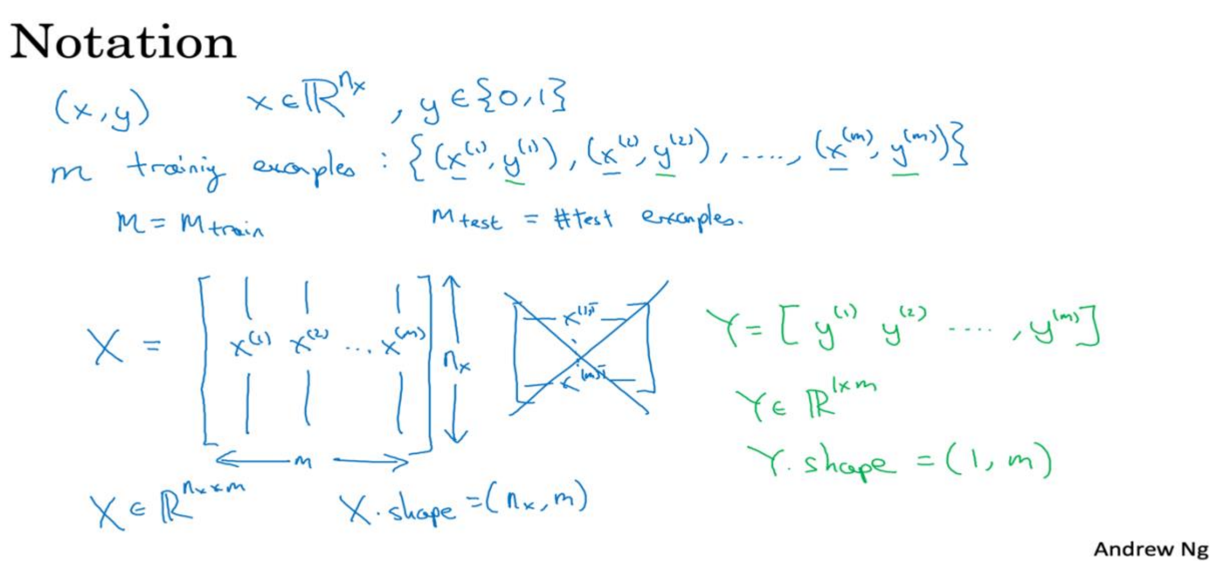
Logistic Regression for Binary Classification

- **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 ($\boldsymbol{X}$):**
  - 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 × 64 × 3 = 12,288
  - Notation: n_x = 12,288: Dimension of the input feature vector. n (lowercase) is sometimes used interchangeably with n_x.

Notation:

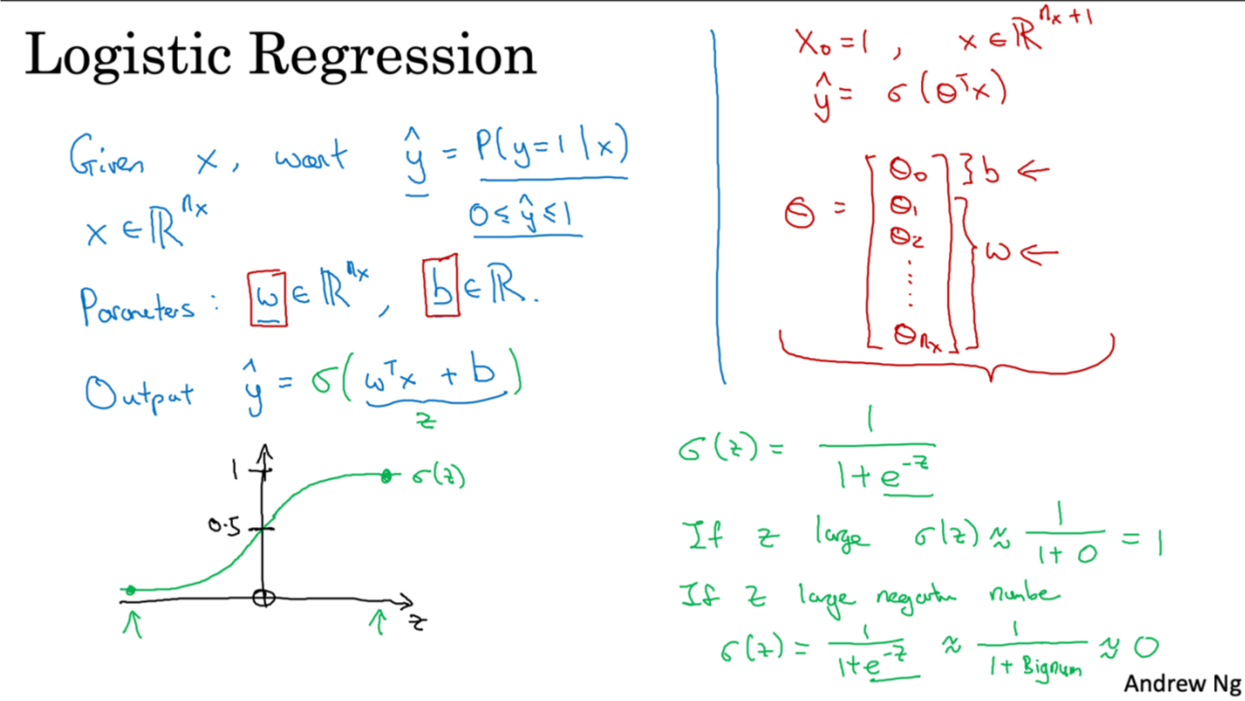


- **Training Example:**
  - A single training example is a pair (x, y), where:
  - $x \in \mathbb{R}^{n_x}$ is the feature vector.(i.e, n_x dimensional)

    $y \in \{0, 1\}$ is the label.
- **Training Set:**
  A training set consists of m training examples:
  $\{ (x^1, y^1), (x^2, y^2), \ldots, (x^m, y^m) \}$
  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 ($\boldsymbol{X}$):**
  The input feature vectors are stacked column-wise to form the matrix X:
  $X = [x^1 \ x^2 \ \ldots \ x^m]$
  Shape: $X \in \mathbb{R}^{n \times m}$
  Python Notation: X.shape = (n_x, m)
- **Output Matrix ($\boldsymbol{Y}$):**
  The labels are organized into a row vector Y:
  $Y = [y^1 \ y^2 \ \ldots \ y^m]$
  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: $\sigma(z) = 1 / (1 + e^{-z})$

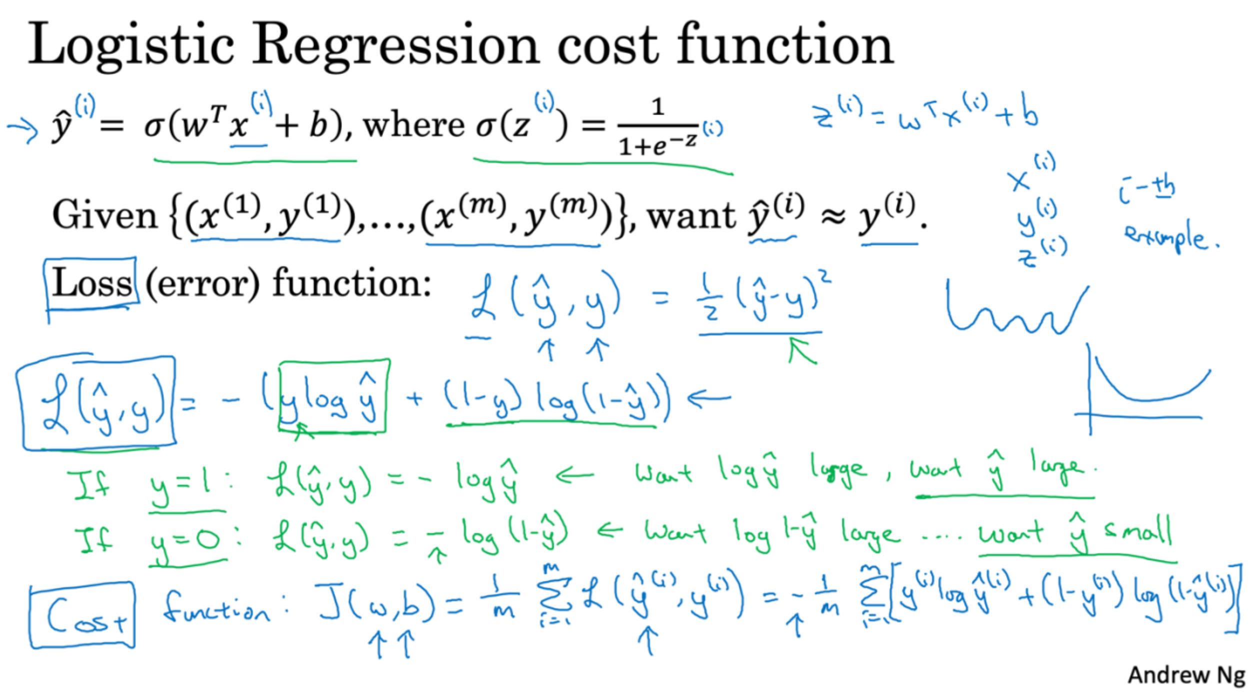
It maps any real number to a value between 0 and 1, making it suitable for representing probabilities.

- 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



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(ŷ, y) = -[y log(ŷ) + (1 y) log(1 ŷ)]

- 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.

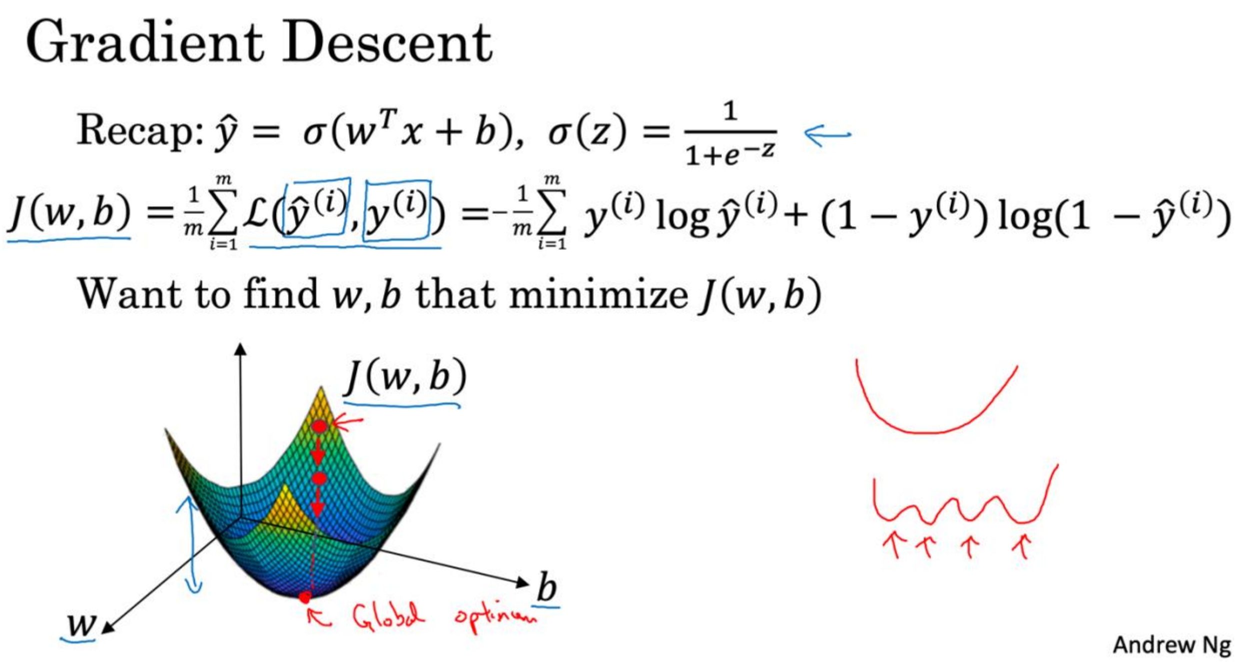
Cost Function: The cost function J(w, b) measures the overall performance of the algorithm on the entire training set. It is the average of the loss functions over all m training examples:

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

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

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.

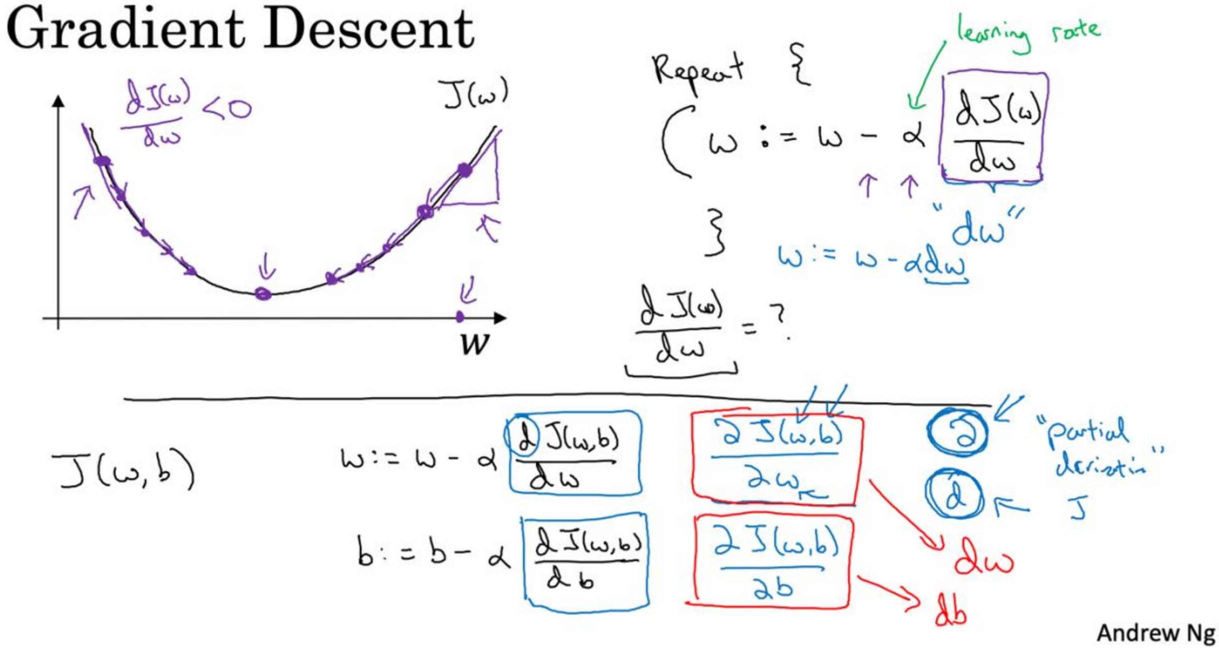
The loss function measures how well the algorithm's prediction $\hat{y}^{(i)}$ compares to the true label $y^{(i)}$ for a single training example.

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.

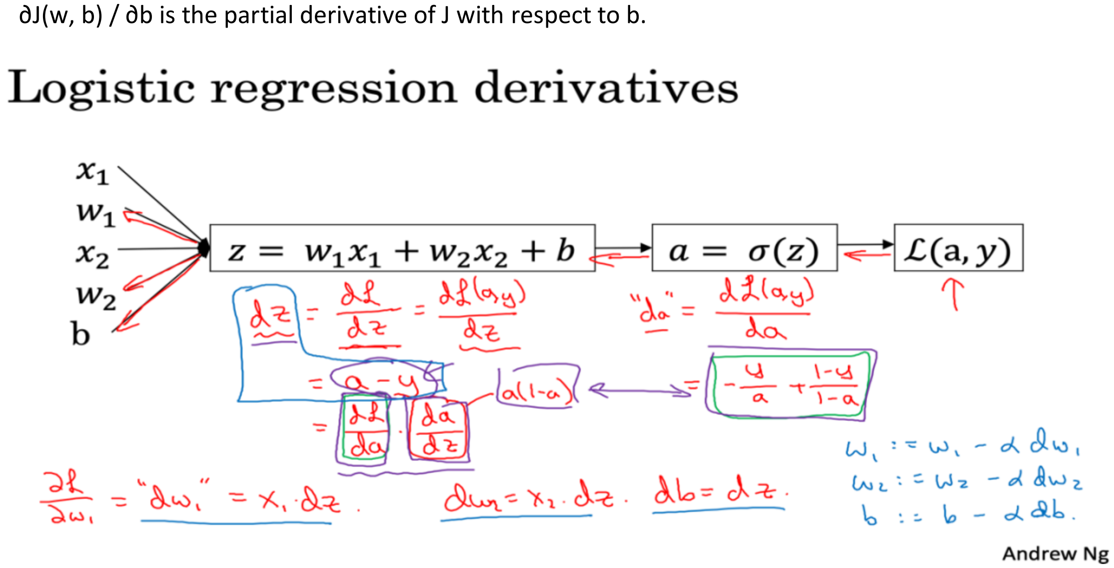


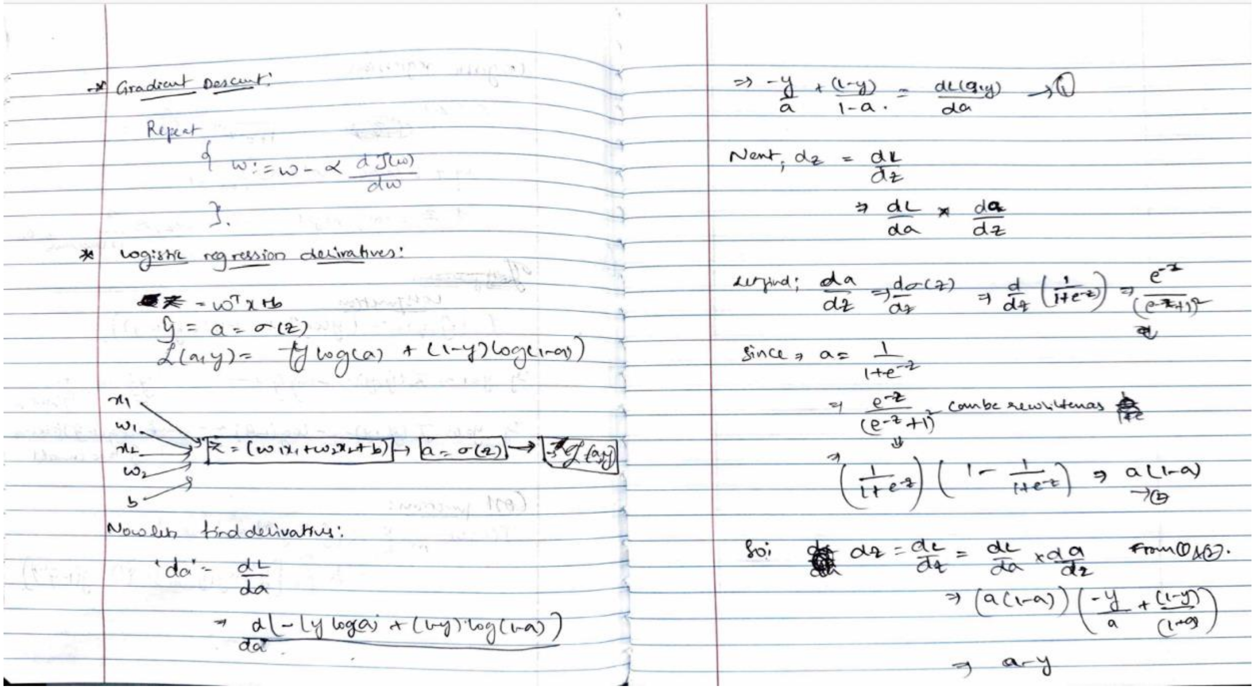
Each step moves the parameters closer to the global minimum of J.
Update Rule: The update rule for gradient descent is:

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.





* Gradient Descent:

Repeat {

$w := w - \alpha \frac{dJ(w)}{dw}$

}

* Logistic regression derivatives:

$z = w^T x + b$

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

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

$x_1, w_1, x_2, w_2, b \rightarrow \boxed{z = w_1x_1 + w_2x_2 + b} \rightarrow \boxed{a = \sigma(z)} \rightarrow \boxed{\mathcal{L}(a,y)}$

Now lets find derivatives:

'da' $= \frac{dL}{da}$

$\Rightarrow \frac{d}{da}\left(-(y\log a + (1-y)\log(1-a))\right)$

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

Next, $dz = \frac{dL}{dz}$

$\Rightarrow \frac{dL}{da} \times \frac{da}{dz}$

Second; $\frac{da}{dz} \Rightarrow \frac{d\sigma(z)}{dz} \Rightarrow \frac{d}{dz}\left(\frac{1}{1+e^{-z}}\right) \Rightarrow \frac{e^{-z}}{(e^{-z}+1)^2}$

Since, $a = \frac{1}{1+e^{-z}}$

$\Rightarrow \frac{e^{-z}}{(e^{-z}+1)^2}$ can be rewritten as

$\Rightarrow \left(\frac{1}{1+e^{-z}}\right)\left(1 - \frac{1}{1+e^{-z}}\right) \Rightarrow a(1-a) \rightarrow (2)$

So; $dz = \frac{dL}{dz} = \frac{dL}{da} \times \frac{da}{dz}$ From (1) & (2).

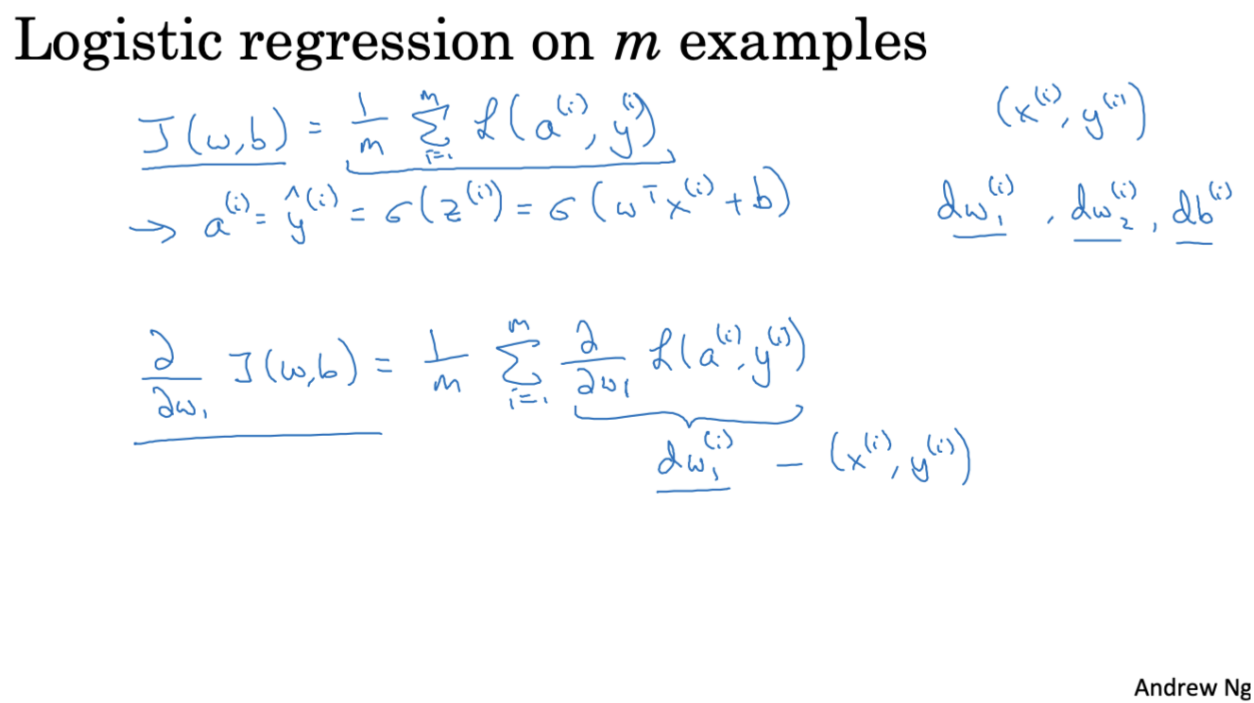
$\Rightarrow (a(1-a))\left(-\frac{y}{a} + \frac{(1-y)}{(1-a)}\right)$

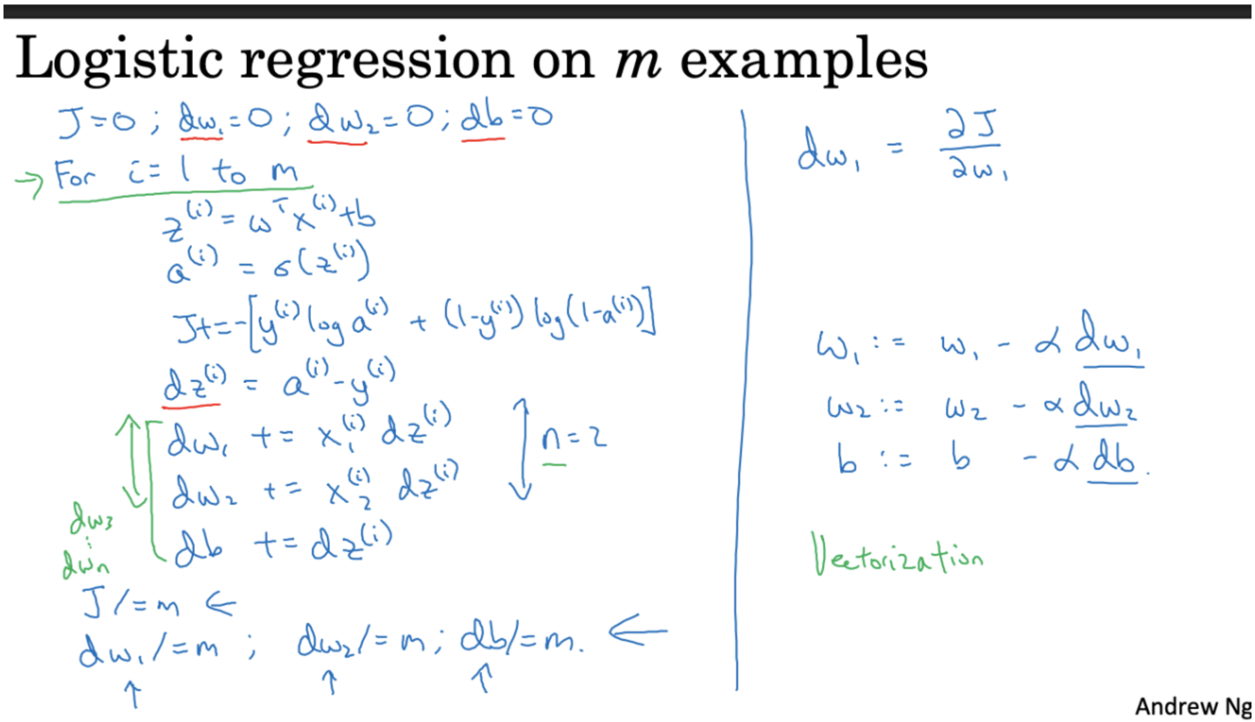
$\Rightarrow a - y$

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))
```





Vectorization

Definition:

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

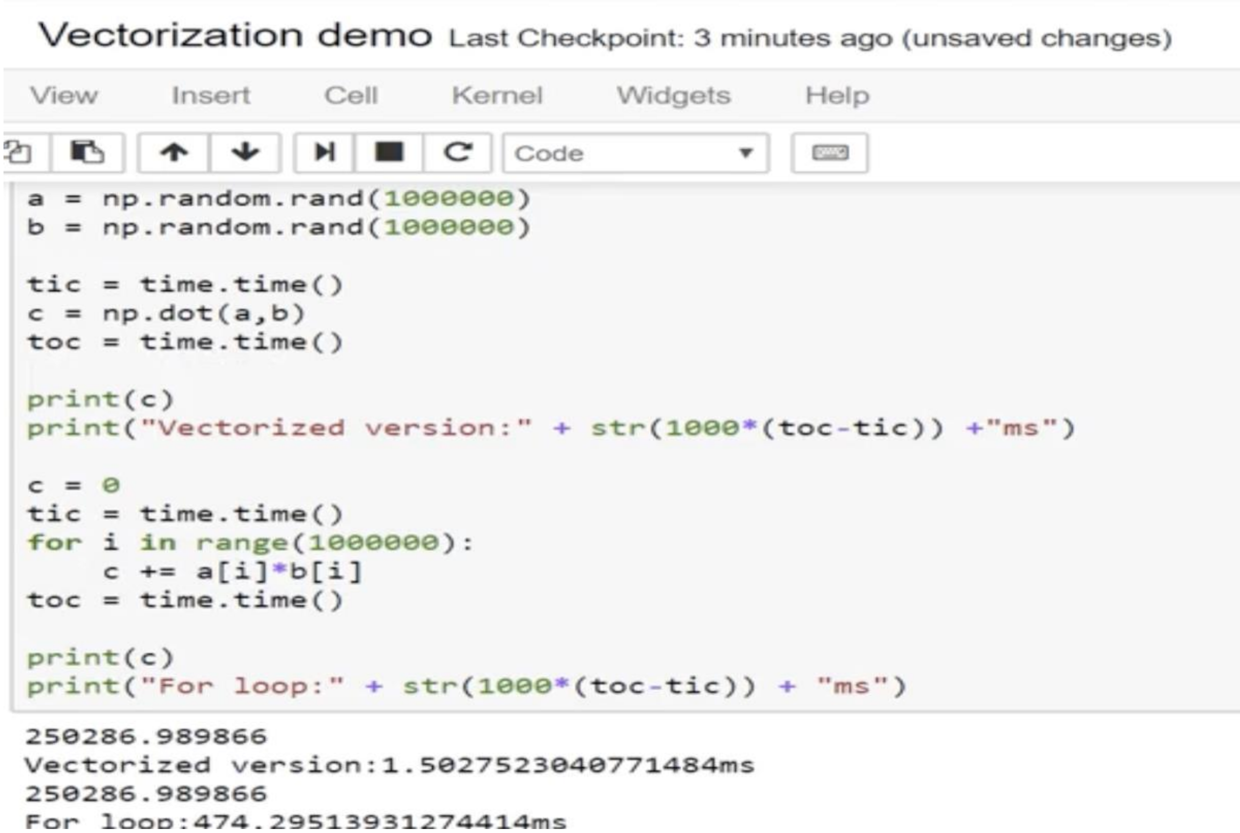
Importance: In deep learning, training is often performed on large datasets. Vectorization makes code run much faster, which is crucial for efficient training on these datasets.

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 ($n_x \times m$):
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
```

```
    db = (1 / m) * np.sum(dZ)
    w = w - alpha * dw
    b = b - alpha * db
    return dw, db, w, b, A
```

proteins, and fats in four

You want to calculate the percentage of calories from each source for each food
This involves dividing each column of the matrix by the total calories for that food

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

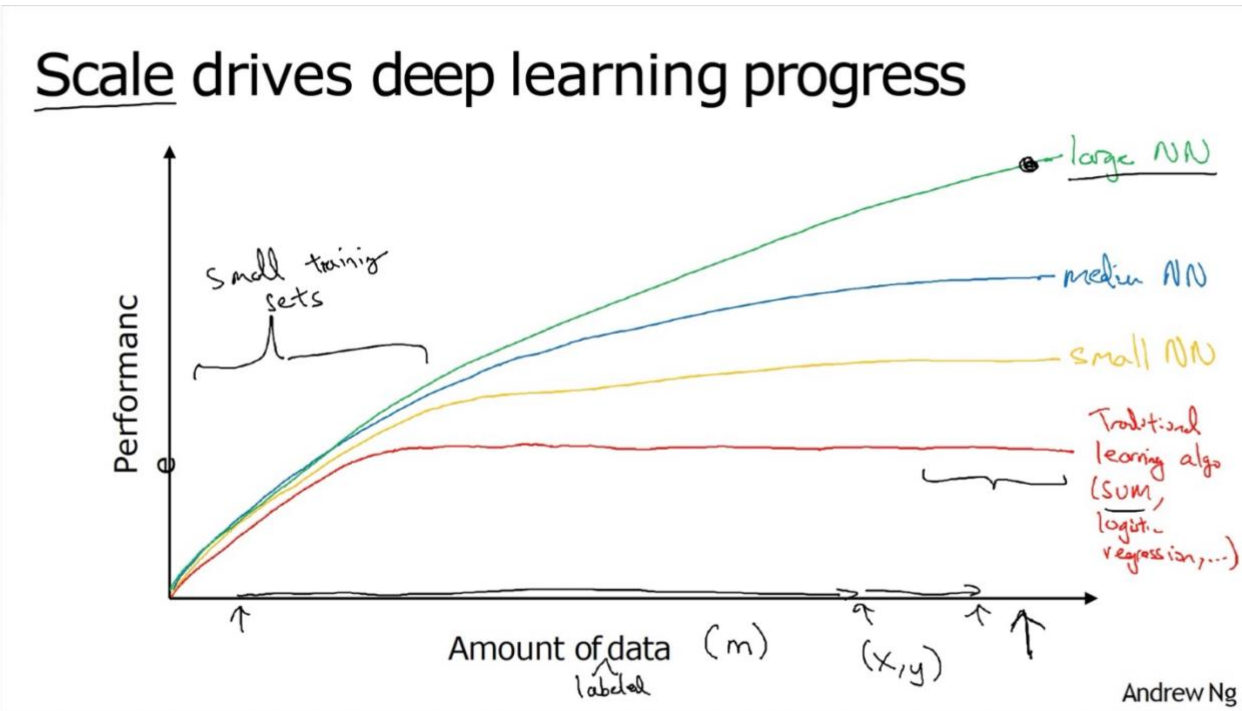
Deep Learning

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

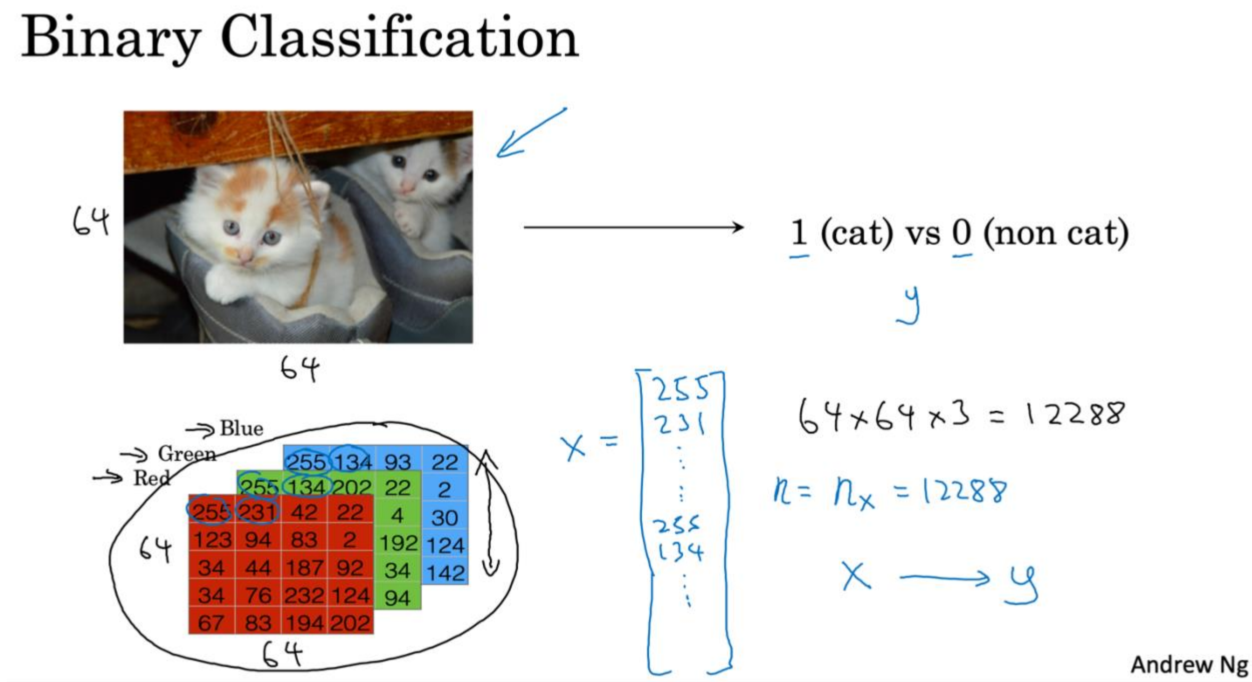
The Dying ReLU problem is a scenario where neurons using ReLU become permanently inactive due to consistently receiving negative inputs, leading to a zero gradient and preventing further learning. Leaky ReLU and other variants, along with careful initialization and learning rate strategies, can help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

ReLU vs. Sigmoid:

Feature	**ReLU (Rectified Linear Unit)**	**Sigmoid**
Mathematical Representation	f(x) = max(0, x)	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))

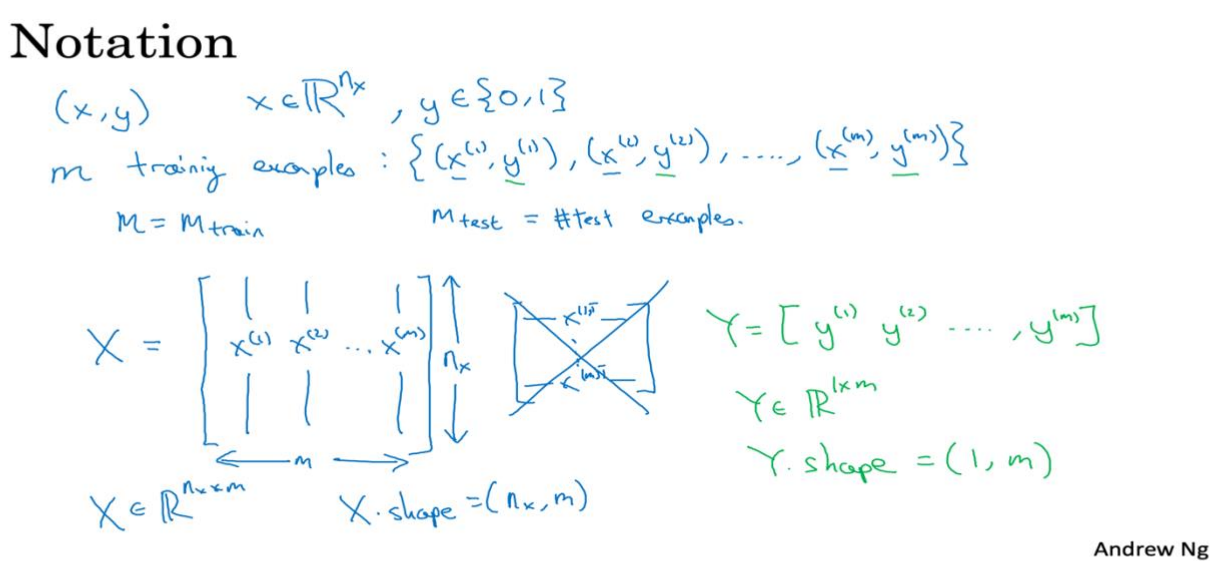
Logistic Regression for Binary Classification

- **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 × 64 × 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:

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

  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 ($\boldsymbol{X}$):**

  The input feature vectors are stacked column-wise to form the matrix X:

  $X = [x^1 \;\; x^2 \;\; \ldots \;\; x^m]$

  Shape: $X \in \mathbb{R}^{n_x \times m}$

  Python Notation: X.shape = (n_x, m)
- **Output Matrix ($\boldsymbol{Y}$):**

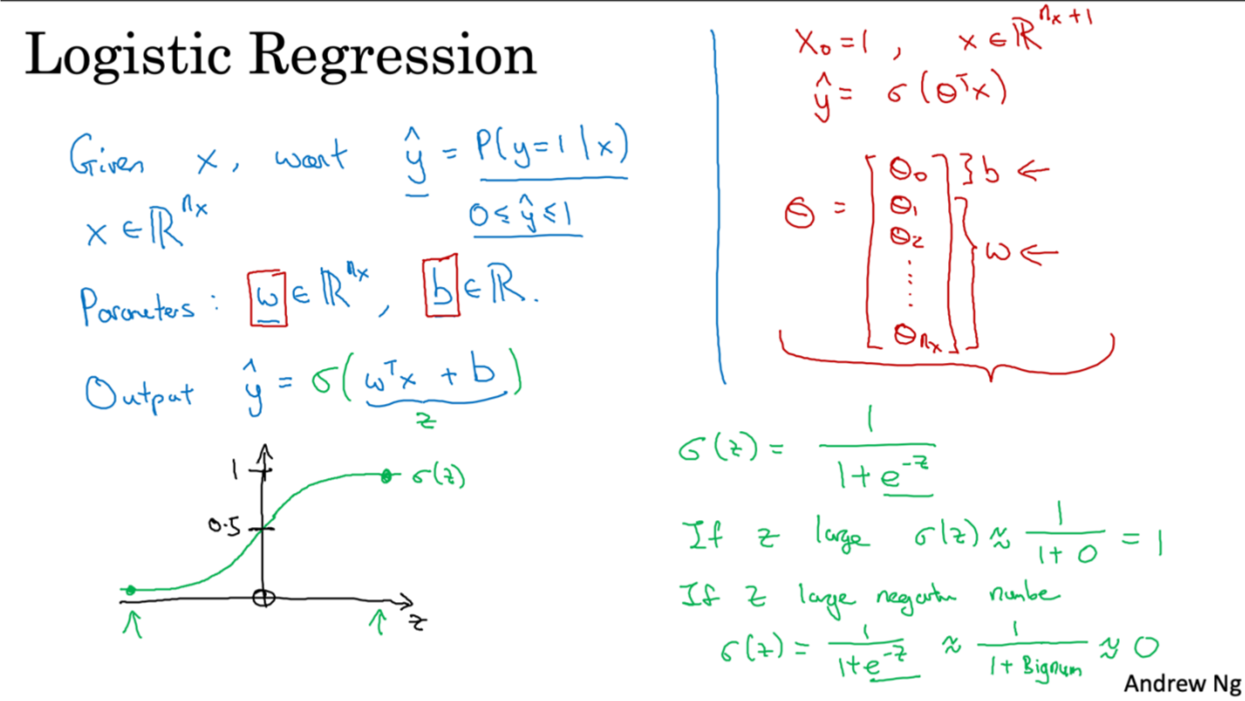
  The labels are organized into a row vector Y:

  $Y = [y^1 \;\; y^2 \;\; \ldots \;\; y^m]$

  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 ($\hat{y}$): The algorithm aims to output a prediction (denoted as $\hat{y}$), which represents the estimated probability that the output y is equal to 1, given the input features x.

Formally, $\hat{y} = P(y = 1 \mid x)$

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.

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

Parameters:

The parameters of logistic regression consist of:

$w \in \mathbb{R}^{nx}$: 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.

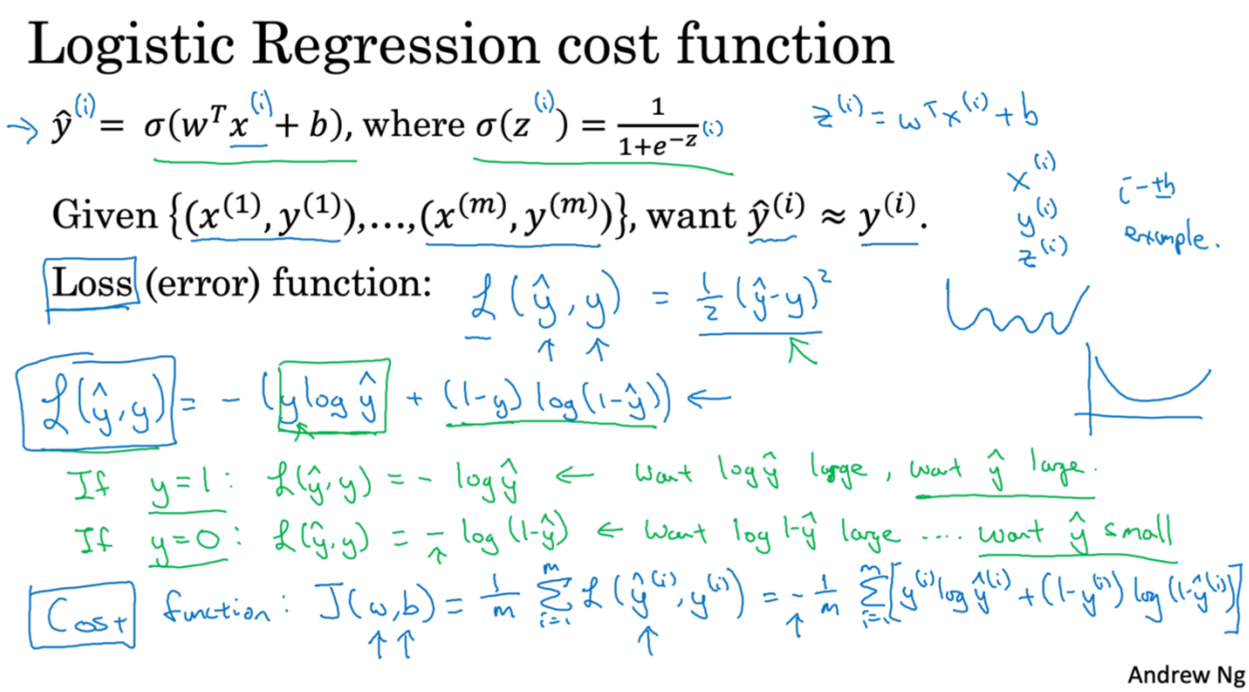
When z is very large, $\sigma(z) \approx 1$

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

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

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

Cost Function in Logistic Regression



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.

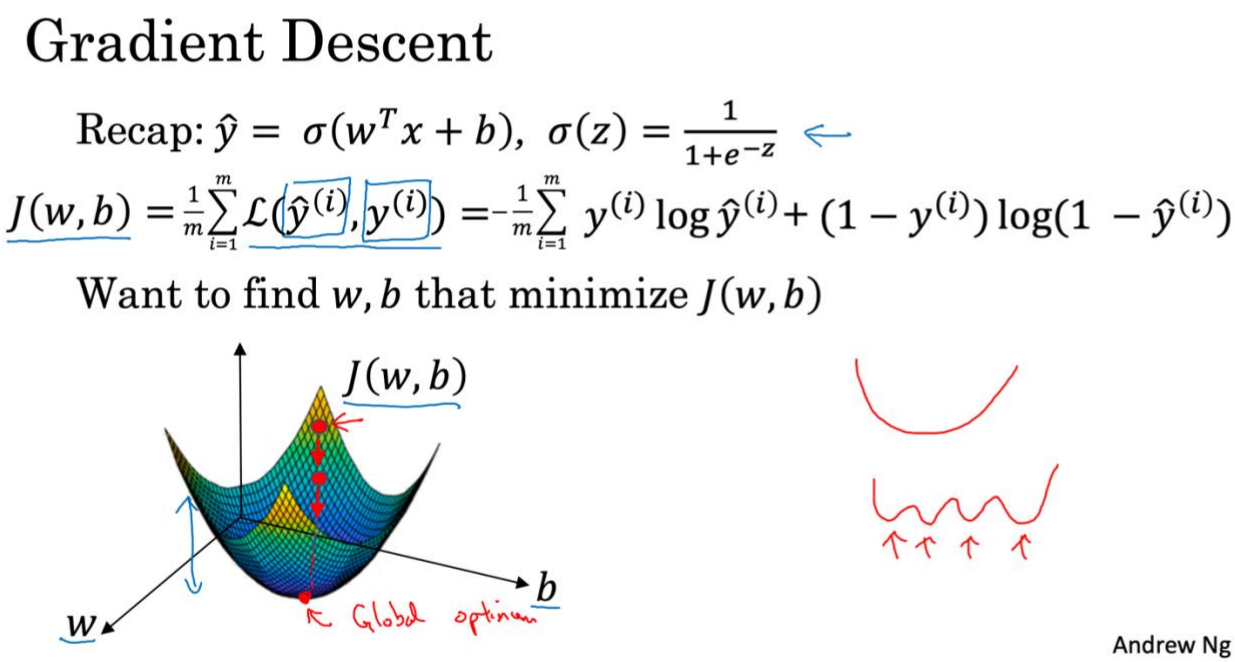
Cost Function: The cost function J(w, b) measures the overall performance of the algorithm on the entire training set. It is the average of the loss functions over all m training examples:

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

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

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.

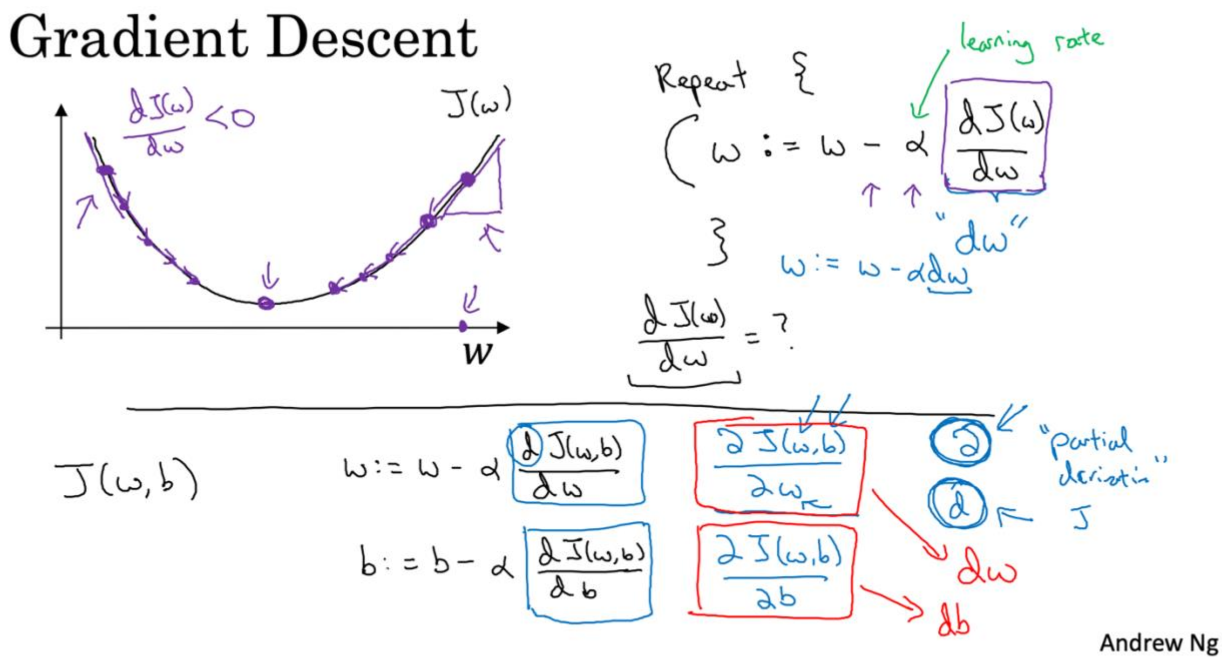
The loss function measures how well the algorithm's prediction $\hat{y}^{(i)}$ compares to the true label $y^{(i)}$ for a single training example.

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)

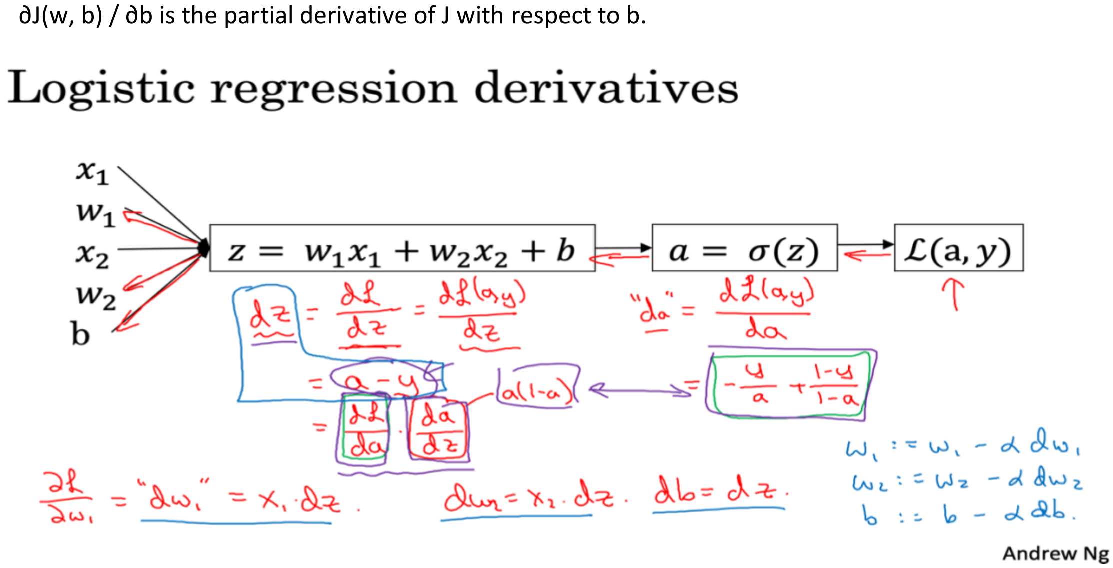
b := b   α (∂J(w, b) / ∂b)

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.



* Gradient Descent:

Repeat {

$$w := w - \alpha \frac{dJ(w)}{dw}$$

}.

* Logistic regression derivatives:

$$z = w^T x + b$$
$$\hat{y} = a = \sigma(z)$$
$$\mathcal{L}(a,y) = -(y \log(a) + (1-y)\log(1-a))$$

x_1, w_1, x_2, w_2, b $\rightarrow$ $\boxed{z = (w_1x_1 + w_2x_2 + b)}$ $\rightarrow$ $\boxed{a = \sigma(z)}$ $\rightarrow$ $\boxed{\mathcal{L}(a,y)}$

Now lets find derivatives:

$$\text{'da'} = \frac{dL}{da}$$
$$\Rightarrow \frac{d}{da}\left(-(y\log(a) + (1-y)\log(1-a)\right)$$

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

Next, $dz = \frac{dL}{dz}$

$$\Rightarrow \frac{dL}{da} \times \frac{da}{dz}$$

we find: $\frac{da}{dz} \Rightarrow \frac{d\sigma(z)}{dz} \Rightarrow \frac{d}{dz}\left(\frac{1}{1+e^{-z}}\right) \Rightarrow \frac{e^{-z}}{(e^{-z}+1)^2}$

Since, $a = \frac{1}{1+e^{-z}}$

$$\Rightarrow \frac{e^{-z}}{(e^{-z}+1)^2}$$ can be rewritten as

$$\Rightarrow \left(\frac{1}{1+e^{-z}}\right)\left(1 - \frac{1}{1+e^{-z}}\right) \Rightarrow a(1-a) \rightarrow (2)$$

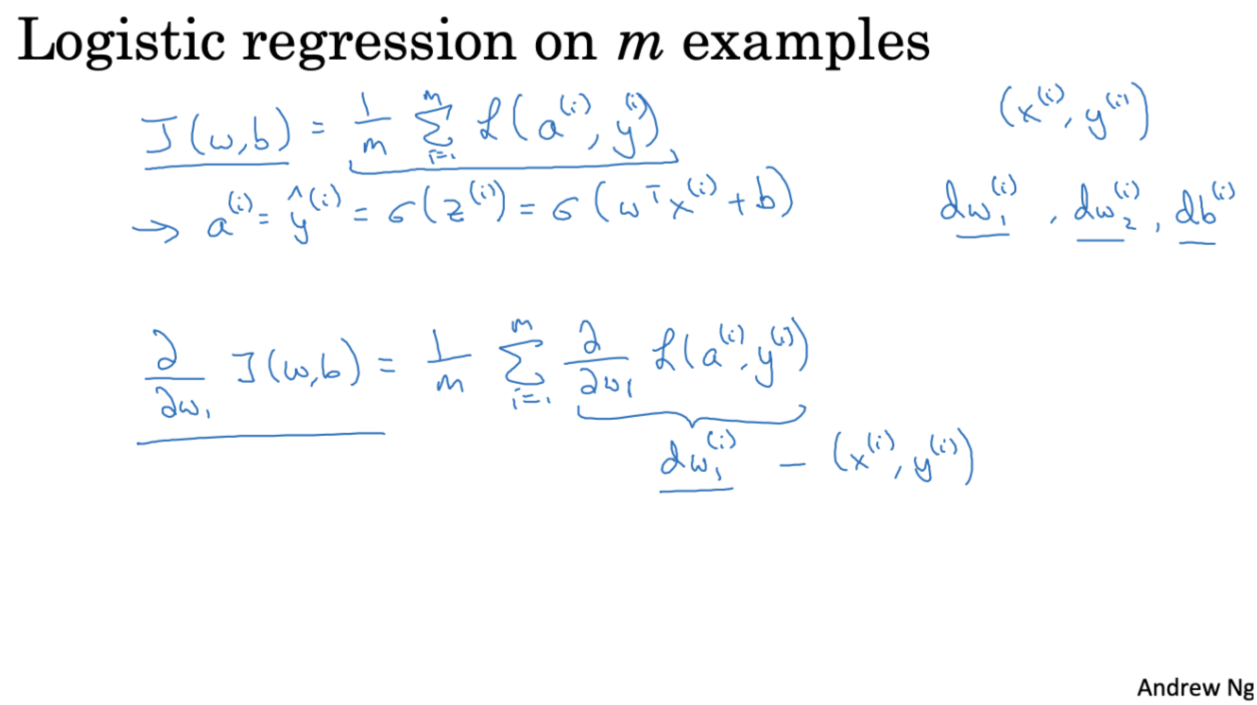
So: $dz = \frac{dL}{dz} = \frac{dL}{da} \times \frac{da}{dz}$ From (1)&(2).

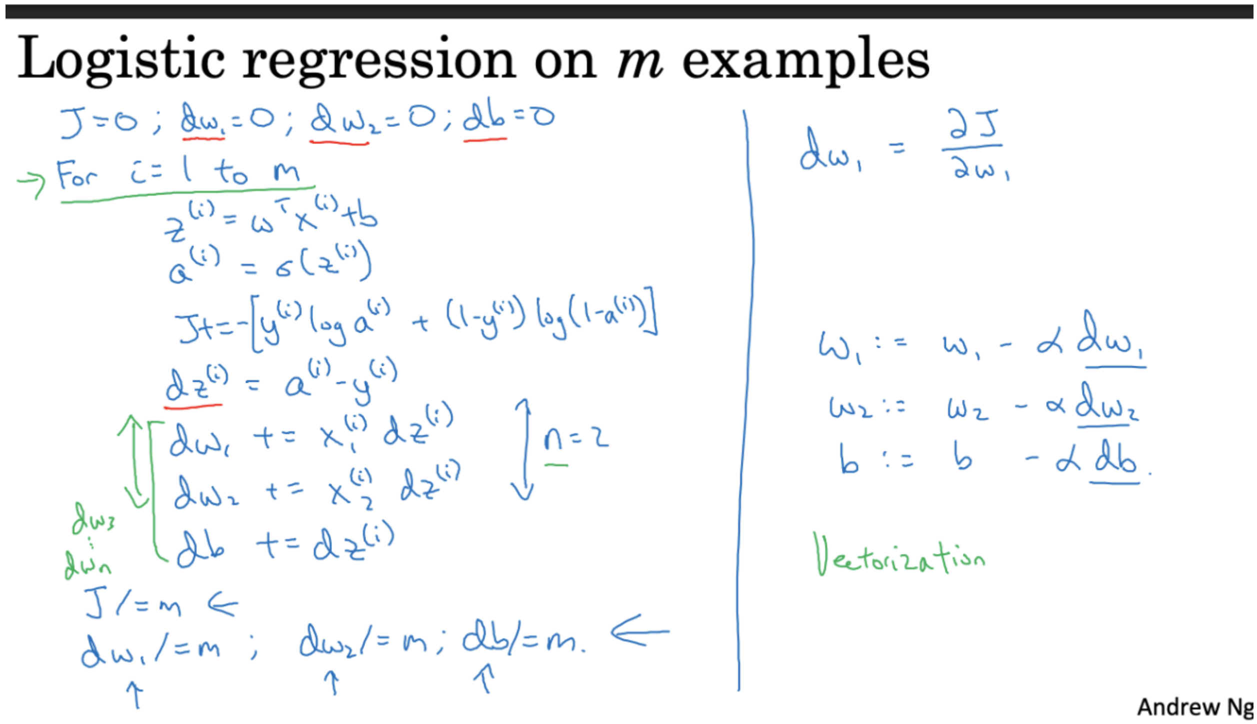
$$\Rightarrow (a(1-a))\left(-\frac{y}{a} + \frac{(1-y)}{(1-a)}\right)$$
$$\Rightarrow a - y$$

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))
```





Vectorization

Definition:

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

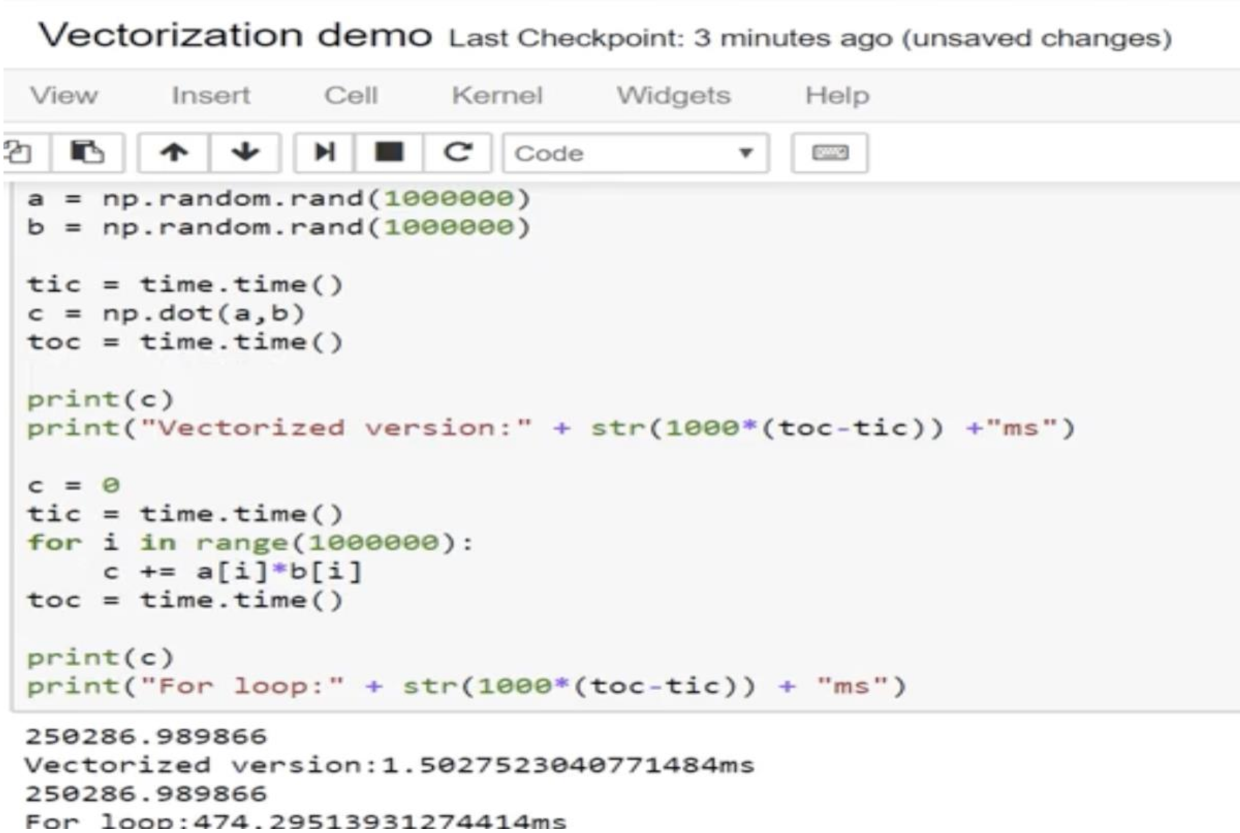
Importance: In deep learning, training is often performed on large datasets. Vectorization makes code run much faster, which is crucial for efficient training on these datasets.

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(nx):
        z += wi * xi
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 ($n_x \times m$):
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
```

```
    db = (1 / m) * np.sum(dZ)
    w = w - alpha * dw
    b = b - alpha * db
    return dw, db, w, b, A
```

proteins, and fats in four

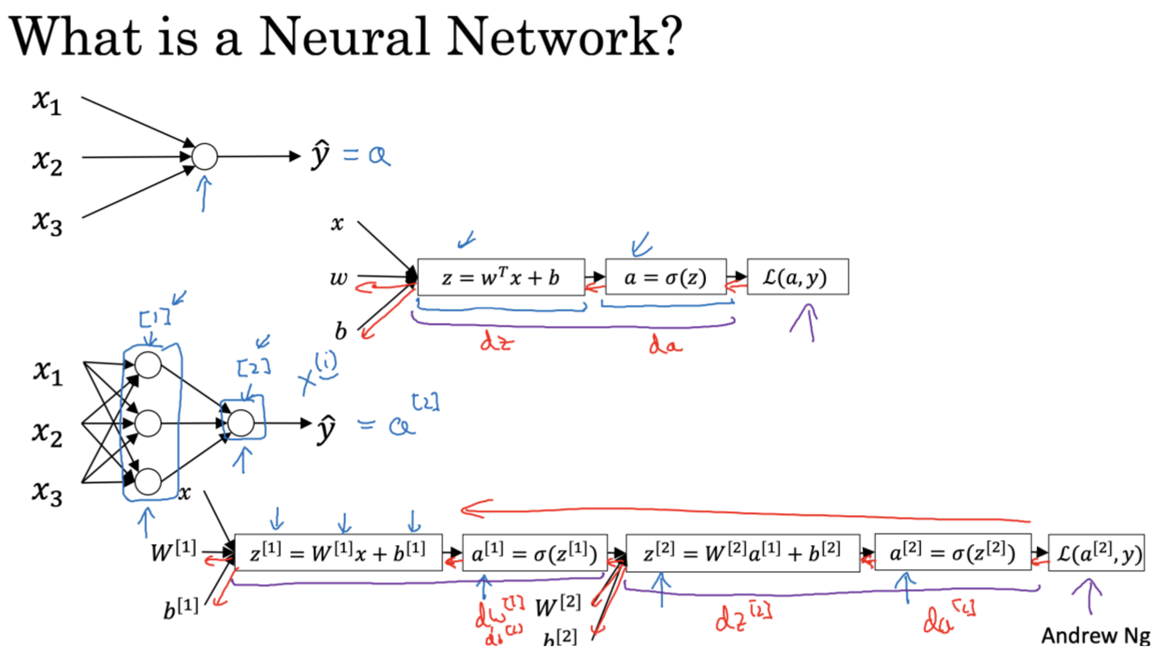
You want to calculate the percentage of calories from each source for each food
This involves dividing each column of the matrix by the total calories for that food

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 ($\hat{y}$) 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.

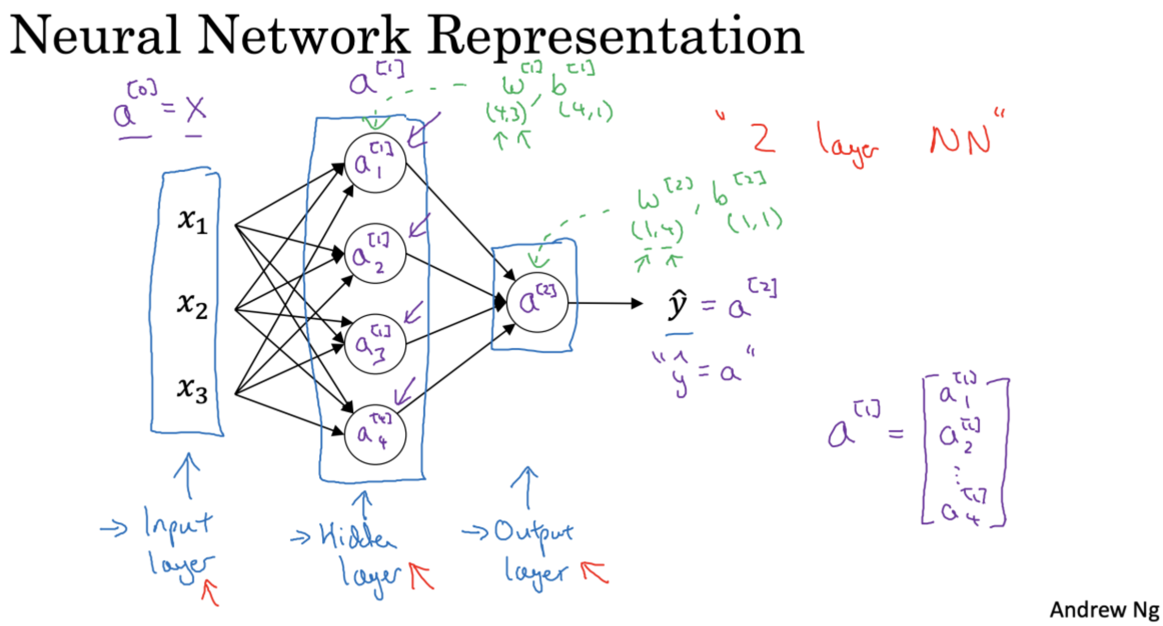
4. Key Differences from Logistic Regression

- **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)

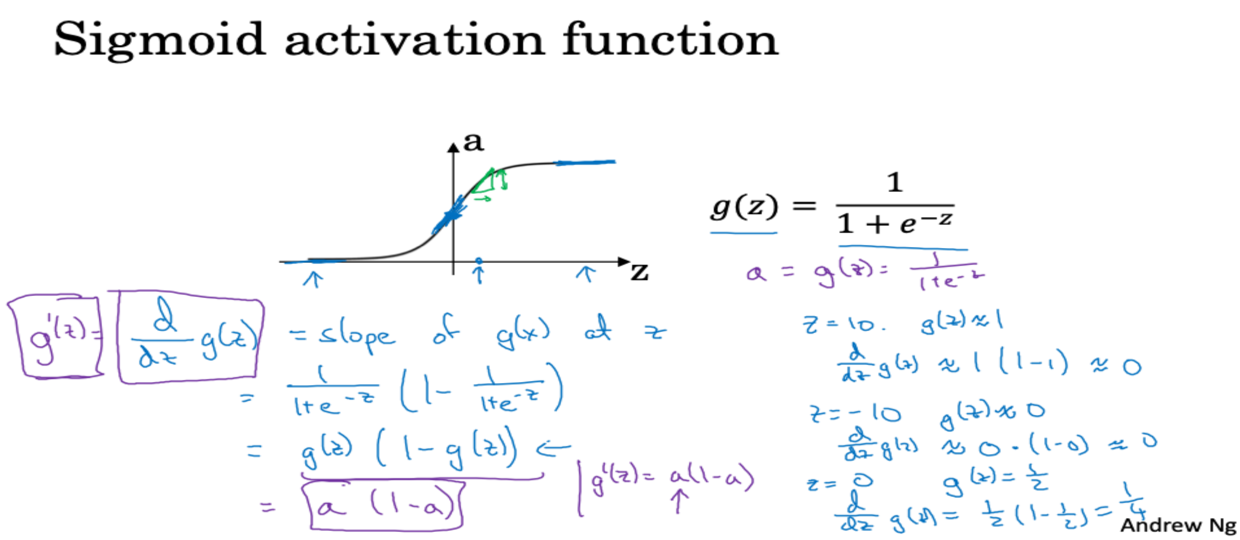
Range: (0, 1)

Use Case: Output layer for binary classification

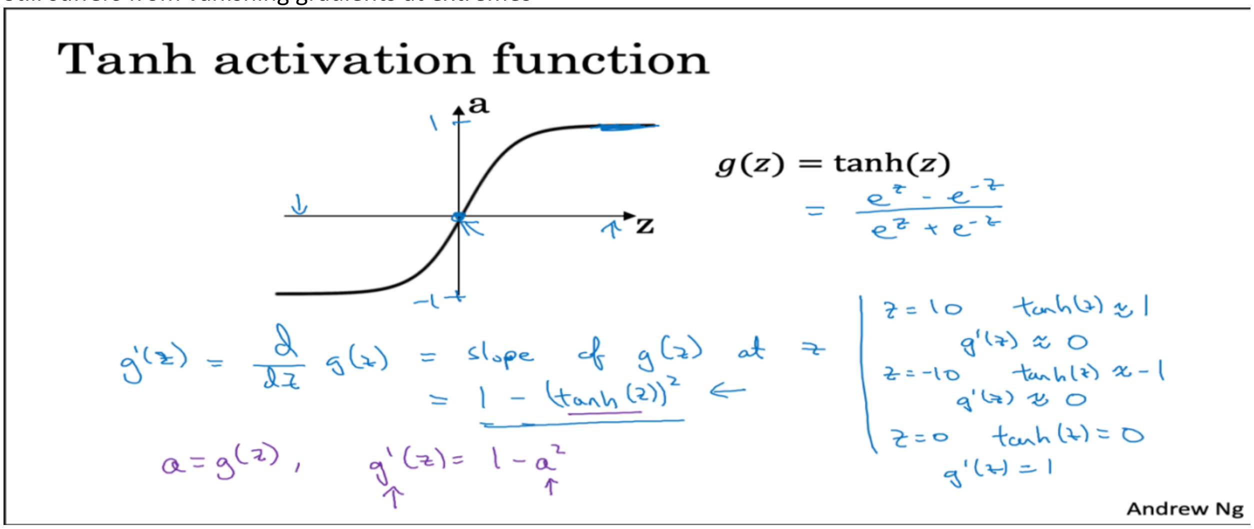
Pros: Outputs a probability

Cons: Vanishing gradient for large |z|

Not zero-centered (mean ≈ 0.5)



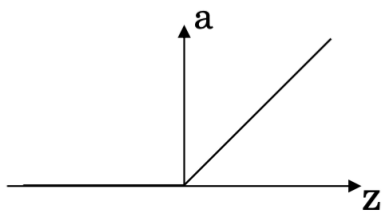
Tanh
Formula: a = (e^z e^-z) / (e^z + e^-z)
Range: (-1, 1)
Use Case: Hidden layers
Pros:
Zero-centered
Better learning than sigmoid
Cons:
Still suffers from vanishing gradients at extremes



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

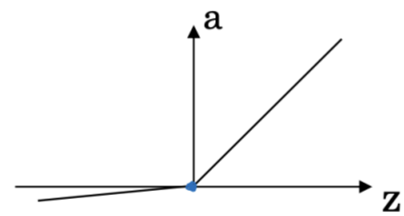


ReLU

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

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

$z = 0.0000 \ldots 0$



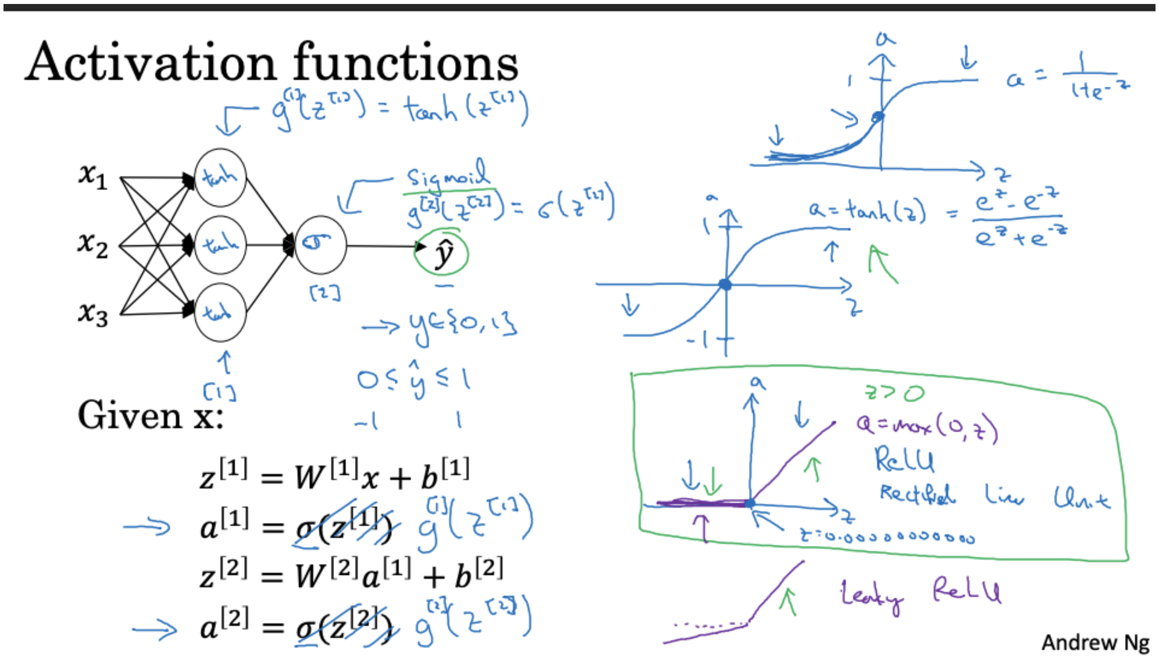
Leaky ReLU

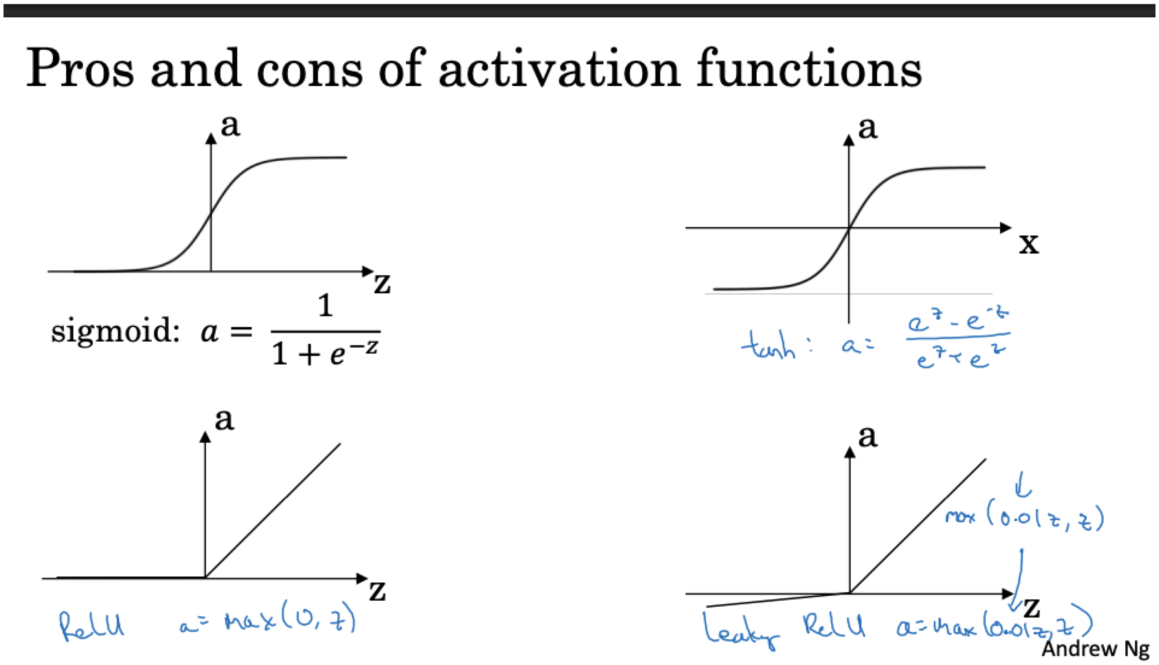
$g(z) = \max(0.01z, z)$

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

Andrew Ng

Graphical Representation





Best Practices

Use sigmoid for output in binary classification.

Use ReLU for hidden layers as default.

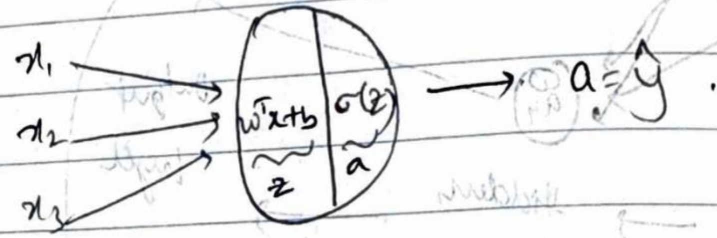
Tanh can be used in hidden layers for zero-centered output.

Leaky ReLU helps avoid dead neurons from ReLU.

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

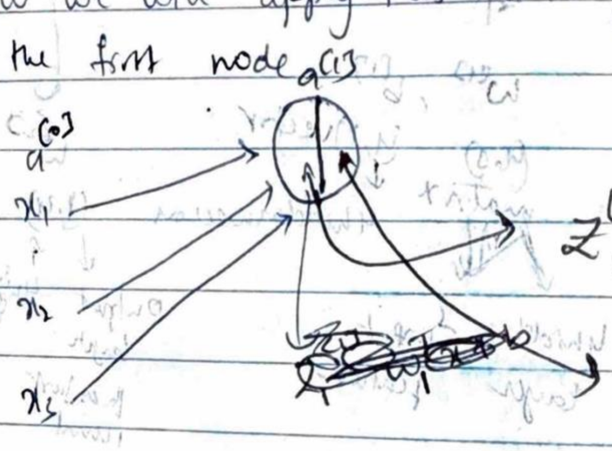
① $z = w^T x + b$

② $a = \sigma(z)$



* Now we will apply this analogy to our 2NN. For the first node $a^{[1]}$

* For Hidden layer Node 1



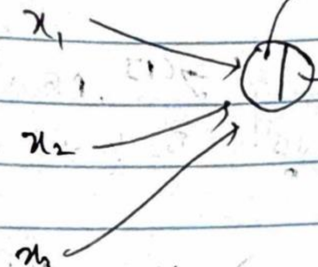
$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}$$

$$a_1^{[1]} = \sigma(z_1^{[1]})$$

<u>Notation:</u>

$a_i^{[l]}$ — $[l]$ ← layer number, i ← node number in that layer

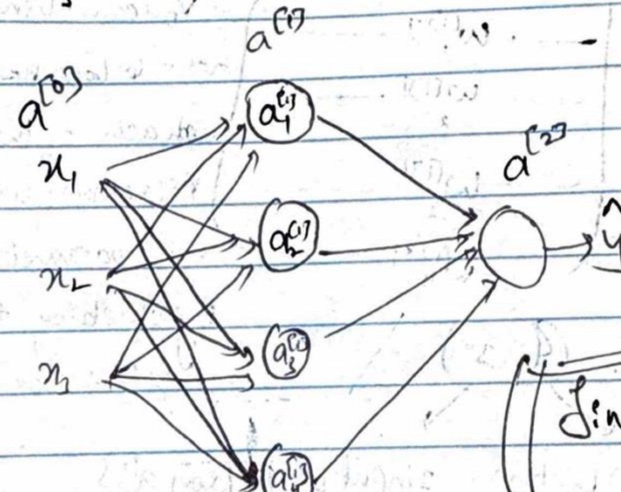
* For Hidden layer Node 2:



$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}$$

$$a_2^{[1]} = \sigma(z_2^{[1]})$$

=>



$\boxed{\text{Since } x = a^{[0]}}$

$$w_1^{[1]T} . a^{[0]} = (w_1^{[1]})^T . a^{[0]}$$

Now we have:

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

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

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

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

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

→ Lets start with vectorizing $Z^{[1]}$, Now to vectorize it we need to vectorize $w^{[1]}$, $b^{[1]}$ & $a^{[0]}$ 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 correspond-ing 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;

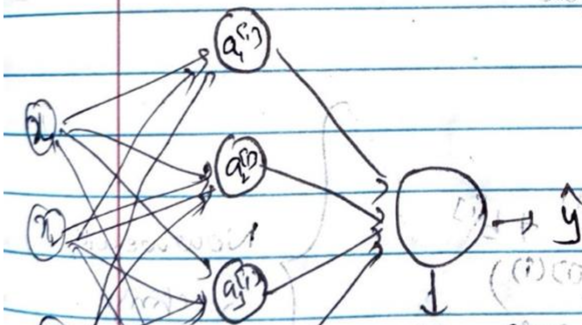
$$\begin{array}{ccccc} w^{[1]} & a^{[0]} & b^{[1]} & w^{[1]T}a^{[0]}+b^{[1]} & z^{[1]} \end{array}$$

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

Similarly now;

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

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



layer $a^{[2]}$

parameters:

$w^{[2]}$, $b^{[2]}$

(1,4) (1,1)

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

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

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

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

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

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

* Vectorizing across multiple examples: (m).

$X \longrightarrow a^{[2]} = \hat{y}$

$x^{(1)} \longrightarrow a^{[2](1)} = \hat{y}^{(1)}$

$x^{(2)} \longrightarrow a^{[2](2)} = \hat{y}^{(2)}$

$\vdots$

$x^{(m)} \longrightarrow a^{[2](m)} = \hat{y}^{(m)}$

Here; $a^{[2](i)}$ → example i; [2] → layer 2.

Now we compute 4 equations that we got previously for 1 training 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.

* ~~the~~ Lets represent all our training example with "X".
we know X can be represented as a column vector:

$$X = \begin{array}{c} \xleftrightarrow{\text{Training example}} \\ \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \end{array} \updownarrow \text{features}$$

Now lets think of pixel value of a cat example.
~~[illegible] say all the~~
where; $x^{(1)}$ is picture of cat-1 and $x^{(1)}$ has all ~~the~~ 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 (weighted sum of inputs w and bias b) can be written as

$$Z^{[1]} = \begin{array}{c} \xleftrightarrow{\text{Training examples}} \\ \begin{bmatrix} | & | & & | \\ z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1](m)} \\ | & | & & | \end{bmatrix} \end{array} \updownarrow \text{Hidden units}$$

$$A^{[1]} = \underbrace{\begin{bmatrix} | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \cdots & a^{[2](m)} \\ | & | & & | \end{bmatrix}}_{\text{Horizontally we go from 1st to m training example}} \updownarrow$$

From 1 hidden layer activation function to next

* Formulas for computing derivatives.

→ Forward propogation.

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

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

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

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

→ <u>Backward</u> <u>Propogation:</u>

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

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

$db^{[2]} = \frac{1}{m}$ 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: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$

$n_x = n^{[0]}$ (Input features), $n^{[1]}$ (Hidden units), $n^{[2]} = 1$ (Output units).

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

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

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

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

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

Gradient Descent:

Repeat {

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

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

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:

say: $c = x^T y \Rightarrow \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}^T \cdot \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_3 \end{bmatrix} \Rightarrow [x_1, x_2, \dots x_m] \cdot \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \sum_{i=1}^{m} x_i y_i$

Element wise product:

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

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

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

We will know discuss intuition behind these formulas.

So, we have these equation in the end

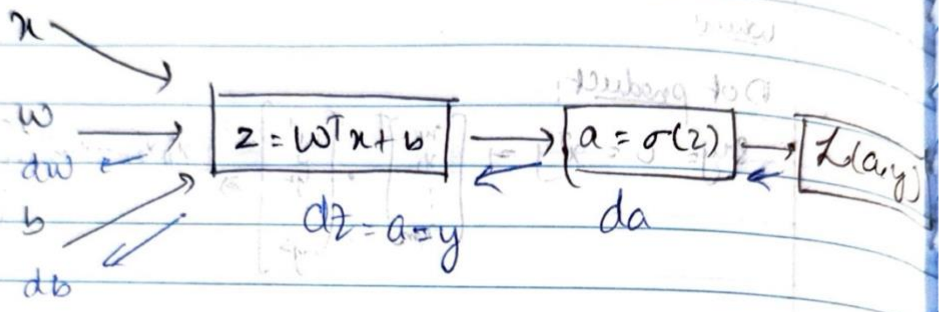
$$Z^{[1]} = W^{[1]} \overset{A^{[0]}}{X} + b^{[1]}$$

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

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

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

* Initially in logistic regression:



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

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

According to chain rule of calculus

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

$$(\because)\ dz = \frac{dL}{dz} = \frac{dL}{da} \cdot \frac{da}{dz}$$

So;

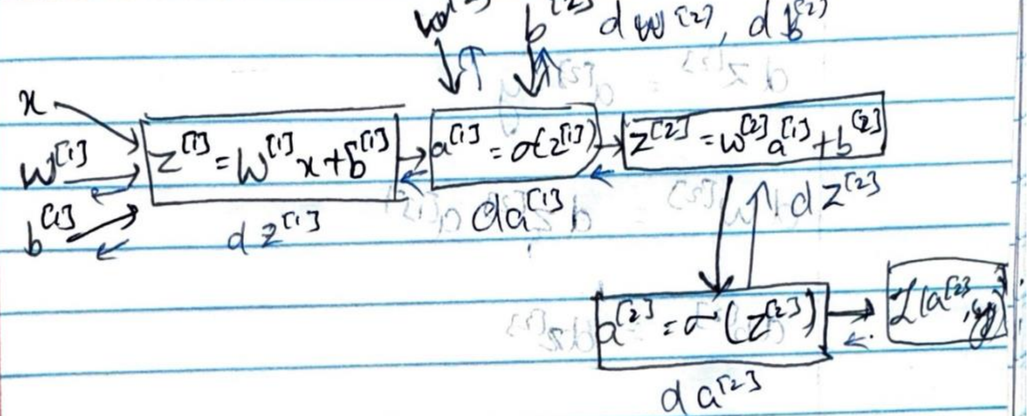
$$\Rightarrow \frac{dL}{da} \cdot \frac{d(\sigma(z))}{z}$$

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

Upon solving: $dz = a - y$.

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

* 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:

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

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

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

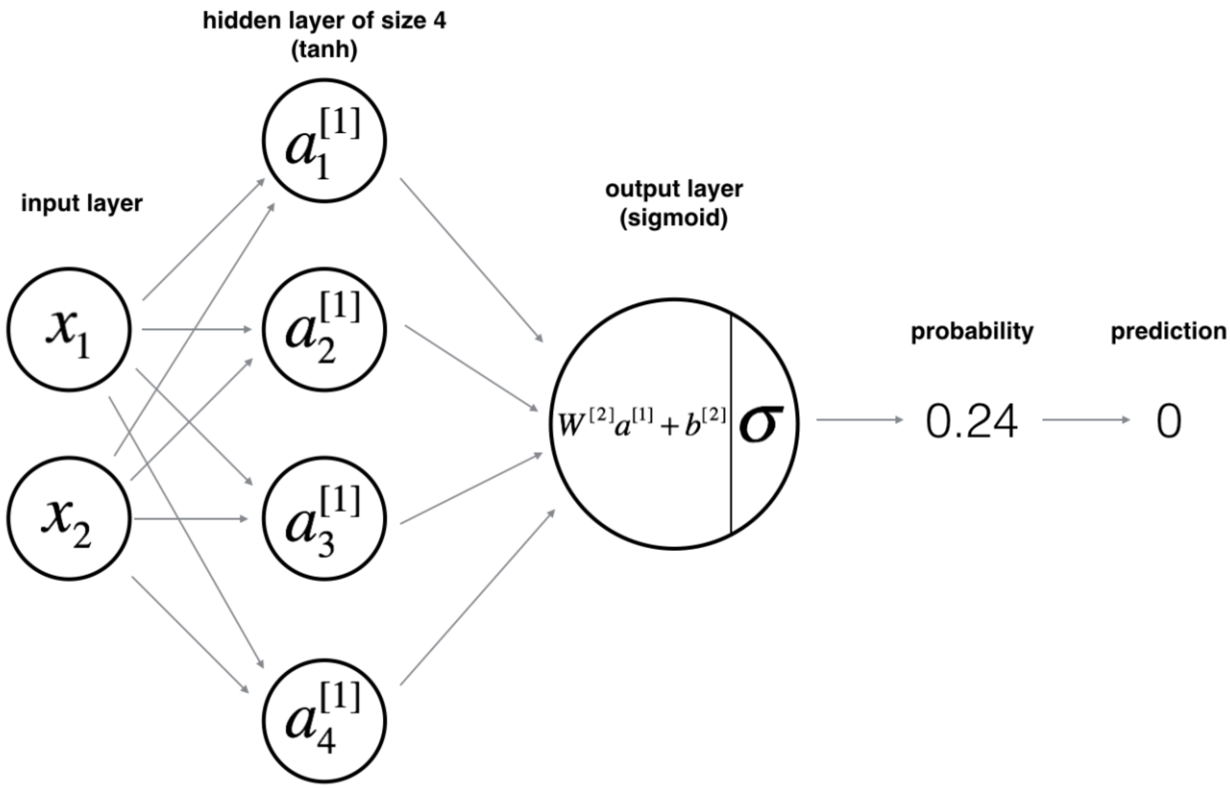
$$\underset{(n^{[1]},m)}{dZ^{[1]}} = \underbrace{W^{[2]T} dZ^{[2]}}_{(n^{[1]},m)} \odot \underset{(n^{[1]},m)}{g^{[1]'}(Z^{[1]})}$$

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

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
```