

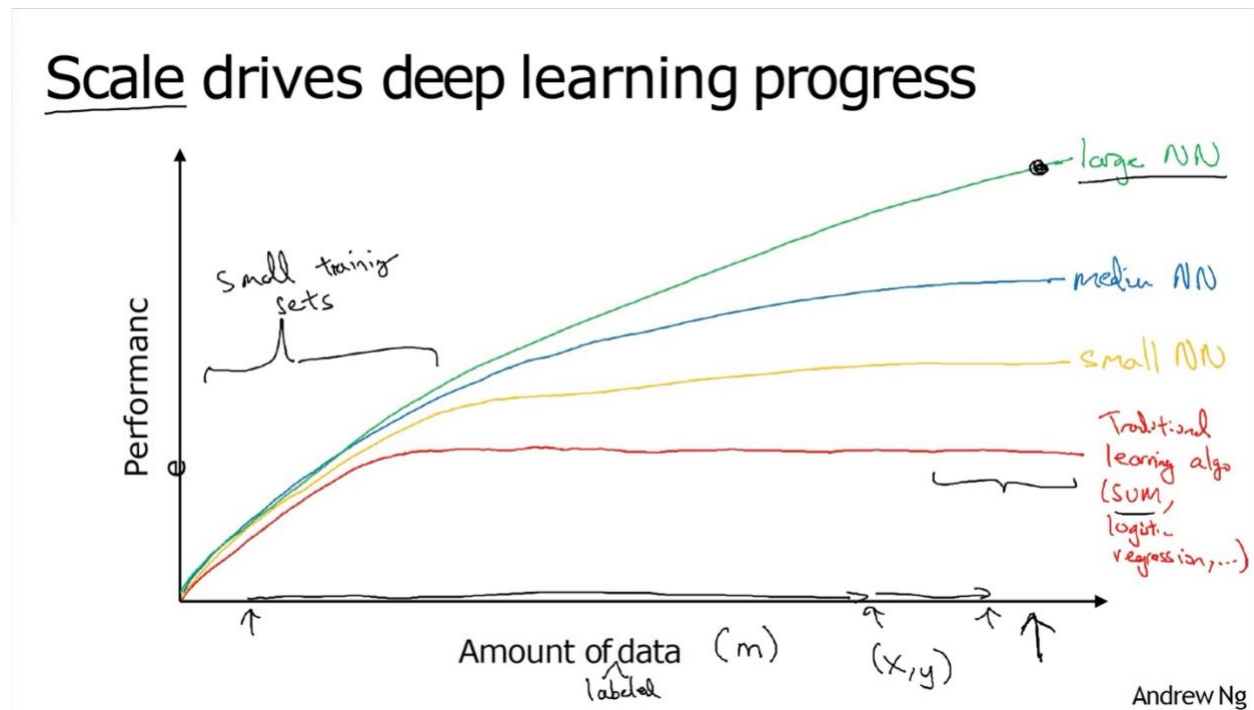
Deep Learning

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.

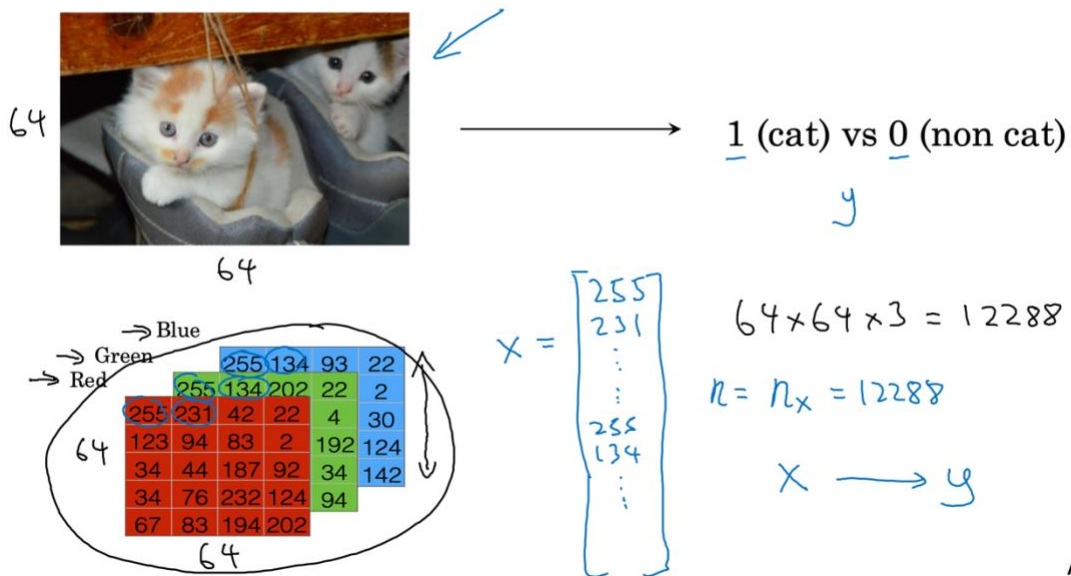
ReLU vs. Sigmoid:

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.

Binary Classification



Andrew Ng

- Feature Vector (\mathbf{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:

Notation

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

m training examples: $\{(\underline{x}^{(1)}, \underline{y}^{(1)}), (\underline{x}^{(2)}, \underline{y}^{(2)}), \dots, (\underline{x}^{(m)}, \underline{y}^{(m)})\}$

$M = M_{\text{train}} \quad M_{\text{test}} = \# \text{test examples.}$

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

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

$X_{\text{shape}} = (n_x, m)$

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

$Y \in \mathbb{R}^{1 \times m}$

$Y_{\text{shape}} = (1, m)$

Andrew Ng

- **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:

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

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 :

$$X = [x^1 \ x^2 \ \dots \ x^m]$$

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

Python Notation: $X.\text{shape} = (n_x, m)$

- **Output Matrix (Y):**

The labels are organized into a row vector Y :

$$Y = [y^1 \ y^2 \ \dots \ y^m]$$

Shape: $Y \in \mathbb{R}^{1 \times m}$

Python Notation: $Y.\text{shape} = (1, m)$

Logistic Regression:

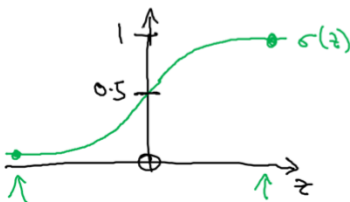
Logistic Regression

Given x , want $\hat{y} = \frac{P(y=1|x)}{0 \leq \hat{y} \leq 1}$

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

Parameters: $\boxed{w} \in \mathbb{R}^{n_x}$, $\boxed{b} \in \mathbb{R}$.

$$\text{Output } \hat{y} = \sigma\left(\frac{w^T x + b}{z}\right)$$



$$x_0 = 1, \quad x \in \mathbb{R}^{n_x+1}$$

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

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_{n_x} \end{bmatrix} \begin{matrix} \leftarrow b \\ \leftarrow w \end{matrix}$$

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

$$\text{If } z \text{ large } \sigma(z) \approx \frac{1}{1+0} = 1$$

If z large negative number

$$\sigma(z) = \frac{1}{1 + e^{-z}} \approx \frac{1}{1 + \text{Big num}} \approx 0$$

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 (\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.

$x \in \mathbb{R}^{n \times 1}$

Parameters:

The parameters of logistic regression consist of:

$w \in \mathbb{R}^{n \times 1}$: 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 \hat{y} : $\hat{y} = \sigma(z)$

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 $\Theta \in \mathbb{R}^{n \times 1}$ (where $\Theta = [b, w_1, w_2, \dots, w_n]^T$), we can write:

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

Cost Function in Logistic Regression

Logistic Regression cost function

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

Given $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$, want $\hat{y}^{(i)} \approx y^{(i)}$.

$x^{(i)}$
 $y^{(i)}$
 $z^{(i)}$ i -th example.

Loss (error) function: $\mathcal{L}(\hat{y}, y) = \frac{1}{2} (\hat{y} - y)^2$

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

If $y = 1$: $\mathcal{L}(\hat{y}, y) = -\log \hat{y} \leftarrow$ want $\log \hat{y}$ large, want \hat{y} large.

If $y = 0$: $\mathcal{L}(\hat{y}, y) = -\log (1 - \hat{y}) \leftarrow$ want $\log (1 - \hat{y})$ large \dots want \hat{y} small

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

Andrew Ng

The prediction \hat{y} 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 $L(\hat{y}, y)$ 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 \hat{y} as large as possible.

If $y = 0$, the loss function tries to make \hat{y} 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:

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

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

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

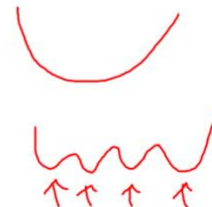
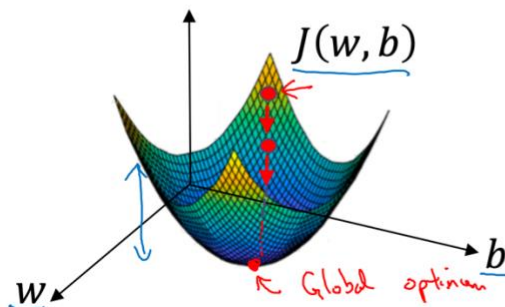
Gradient Descent

Gradient Descent

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

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

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



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.

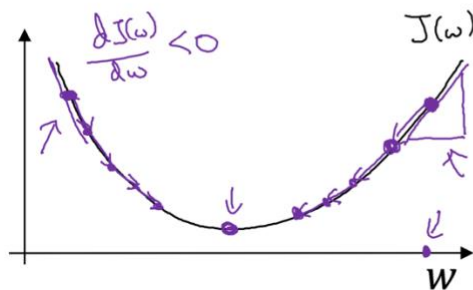
Visualization: The cost function $J(w, b)$ can be visualized as a surface where the horizontal axes represent the parameters w and b , and the vertical axis represents the value of J .

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.

Gradient Descent



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

learning rate (pointing to α)
"dw" (pointing to dw)

$J(w, b)$

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

$\frac{\partial J(w, b)}{\partial w}$
 $\frac{\partial J(w, b)}{\partial b}$

$\frac{\partial}{\partial}$ ← "partial derivative"
 J
 dw
 db

Andrew Ng

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

Update Rule: The update rule for gradient descent is:

$$w := w - \alpha (\partial J(w, b) / \partial w)$$

$$b := b - \alpha (\partial J(w, b) / \partial b)$$

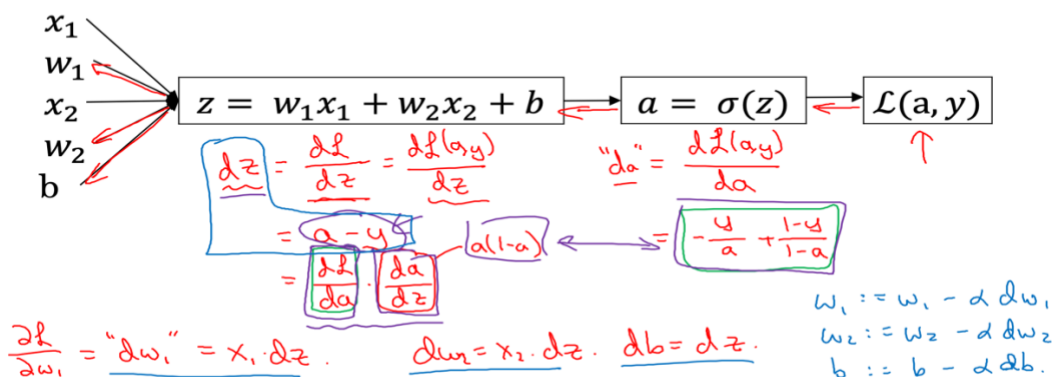
Where:

α is the learning rate, controlling the step size.

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

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

Logistic regression derivatives



Andrew Ng

Gradient Descent

Repeat

$$w := w - \alpha \frac{d\mathcal{L}(w)}{dw}$$

}

* logistic regression derivatives:

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

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

Now let's find derivatives:

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

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

Next, $\frac{dz}{dz} = \frac{d\mathcal{L}}{dz}$

$\Rightarrow \frac{d\mathcal{L}}{da} \times \frac{da}{dz}$

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

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

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

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

So: $\frac{d\mathcal{L}}{dz} = \frac{d\mathcal{L}}{da} \times \frac{da}{dz}$ from (1) & (2).

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

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

$\Rightarrow -y(1-a) + (1-y)a$

Logistic regression on m examples

$$J(w, b) = \frac{1}{m} \sum_{i=1}^m \ell(a^{(i)}, y^{(i)}) \quad (x^{(i)}, y^{(i)})$$

$$\rightarrow a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b) \quad \underline{dw_1^{(i)}}, \underline{dw_2^{(i)}}, \underline{db^{(i)}}$$

$$\frac{\partial}{\partial w_1} J(w, b) = \frac{1}{m} \sum_{i=1}^m \underbrace{\frac{\partial}{\partial w_1} \ell(a^{(i)}, y^{(i)})}_{\underline{dw_1^{(i)}} - (x^{(i)}, y^{(i)})}$$

Andrew Ng

Logistic regression on m examples

$$J=0; \underline{dw_1}=0; \underline{dw_2}=0; \underline{db}=0$$

→ For $i=1$ to m

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

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

$$J += -[y^{(i)} \log a^{(i)} + (1-y^{(i)}) \log(1-a^{(i)})]$$

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

$$\begin{aligned} dw_1 &+= x_1^{(i)} dz^{(i)} \\ dw_2 &+= x_2^{(i)} dz^{(i)} \\ db &+= dz^{(i)} \end{aligned} \quad \begin{array}{l} \uparrow \\ n=2 \\ \downarrow \end{array}$$

$\frac{dw_1}{m} \leftarrow$
 $\frac{dw_2}{m} \leftarrow$
 $\frac{db}{m} \leftarrow$

$$dw_1 := w_1 - \alpha \underline{dw_1}$$

$$w_2 := w_2 - \alpha \underline{dw_2}$$

$$b := b - \alpha \underline{db}$$

Vectorization

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:

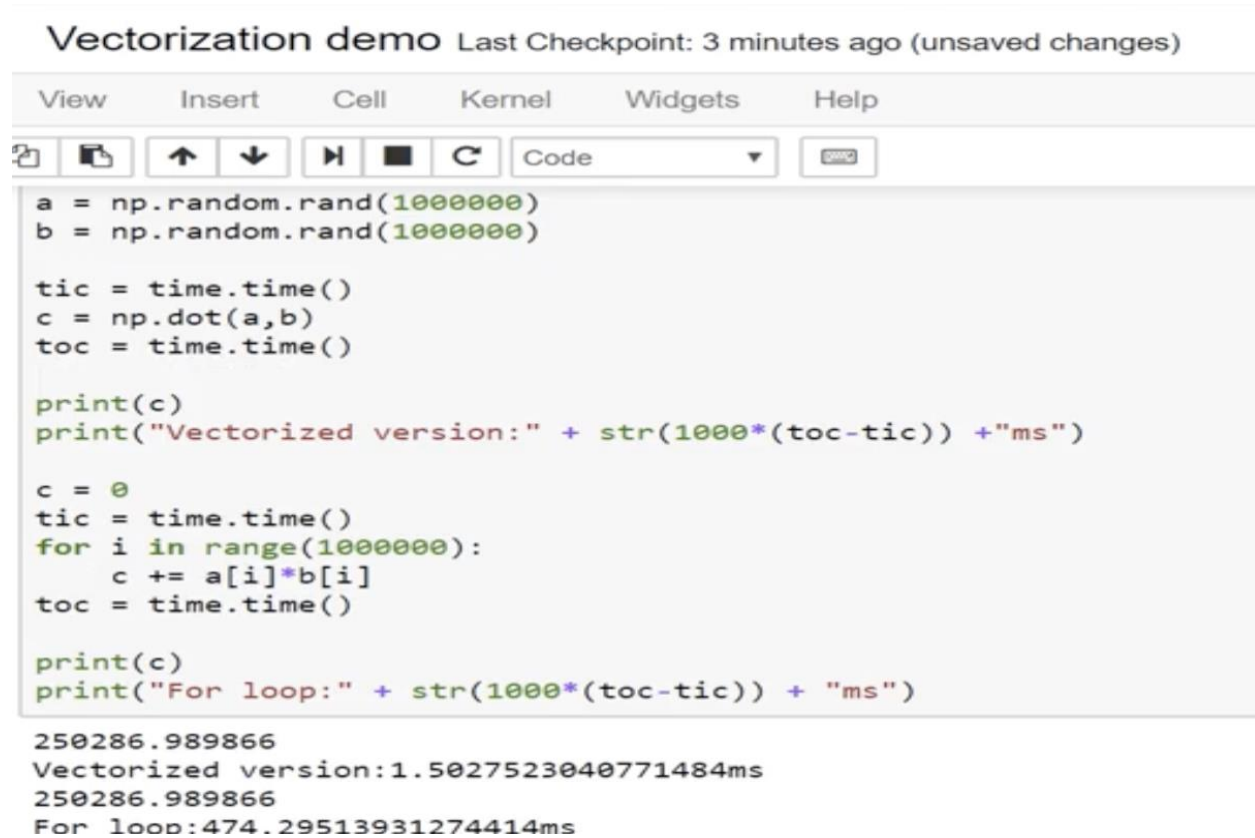
```
z = 0
for i in range(n_x):
    z += w_i * x_i
z += b
```

Vectorized implementation:

In Python (using numpy):

```
z = np.dot(w, x) + b
```

This avoids loops and is much faster.

Demonstration:

The screenshot shows a Jupyter Notebook interface with a title bar 'Vectorization demo' and a subtitle 'Last Checkpoint: 3 minutes ago (unsaved changes)'. The notebook has a menu bar with 'View', 'Insert', 'Cell', 'Kernel', 'Widgets', and 'Help'. Below the menu is a toolbar with icons for file operations, navigation, and execution. The code cell contains the following Python code:

```
a = np.random.rand(1000000)
b = np.random.rand(1000000)

tic = time.time()
c = np.dot(a,b)
toc = time.time()

print(c)
print("Vectorized version:" + str(1000*(toc-tic)) + "ms")

c = 0
tic = time.time()
for i in range(1000000):
    c += a[i]*b[i]
toc = time.time()

print(c)
print("For loop:" + str(1000*(toc-tic)) + "ms")
```

The output of the code is displayed below the cell:

```
250286.989866
Vectorized version:1.5027523040771484ms
250286.989866
For loop:474.29513931274414ms
```

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: A Recap

- 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
```

- **Calculate db:**

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

- **Calculate dw:**

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

4. Vectorized Gradient Descent

- **Update Parameters:**

```
w = w - alpha * dw
```

```
b = b - alpha * db
```

5. Complete Vectorized Implementation (Single Iteration)

```
def vectorized_logistic_regression(X, Y, w, b,
alpha):
    m = X.shape[1]
    Z = np.dot(w.T, X) + b
    A = sigmoid(Z)
    dZ = A - Y
    dw = (1 / m) * np.dot(X, dZ.T)
    db = (1 / m) * np.sum(dZ)
    w = w - alpha * dw
    b = b - alpha * db
    return dw, db, w, b, A
```

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
```

Broadcasting

Definition: Broadcasting is a feature in Python (specifically with NumPy) that allows operations on arrays of different shapes without explicit for loops

Purpose: It enables vectorized operations, making code more efficient and concise

Example:

Imagine you have a 3x4 matrix (A) representing the calories from carbohydrates, proteins, and fats in four different foods.

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

Week 2 Assignment: Logistic_Regression_with_a_Neural_Network_mindset

Exercise 1

Understanding the Shape of Image Data

In many image classification tasks, the dataset is often structured as a NumPy array with the following dimensions:

```
(number_of_examples, height, width, color_channels)
```

For example, if you have 50 training images that are 64x64 pixels in size and have 3 color channels (Red, Green, Blue), the `train_set_x_orig` array would likely have a shape of `(50, 64, 64, 3)`.

Exercise 2:

You start with 50 images, each a 64x64 pixel image with Red, Green, and Blue color channels (shape: `(50, 64, 64, 3)`).

`train_set_x_orig.shape[0]` gives you the number of images, which is 10.

`.reshape(10, -1)` transforms your data into a 2D array with 10 rows. Each original 64x64x3 image is flattened into a single row of $64 \times 64 \times 3 = 12288$ -pixel values.

So, you go from having 10 images represented as 3D arrays to having a table with 10 rows, where each row is a 1D list of all the pixel information for that image

Note: `-1` in `reshape(10, -1)` tells NumPy to automatically calculate the size of the second dimension. It does this by ensuring the total number of elements in the reshaped array is the same as the original. Since the original had $10 \times 64 \times 64 \times 3 = 122880$ elements and the first dimension is fixed at 10, the second dimension becomes $122880/10 = 12288$. So, the array is reshaped into a `(10, 12288)` array.

Why Flatten Images?

Imagine your image as a grid of colored squares (pixels). For a computer to process this image using certain simple machine learning models (like basic neural networks or logistic regression directly on pixels), it's easier if we present this grid as a single, long list of numbers.

Think of it like this:

1. Original Image (as the computer sees it): A multi-layered structure. For a color image, it's like having multiple grids stacked on top of each other (Red, Green, Blue), and each grid has rows and columns of pixel intensity values. For a grayscale image, it's just one grid.
2. Problem for Simple Models: Simple models often expect a single list of features for each input. If we feed in the 2D (or 3D) image structure directly, these models wouldn't know how to interpret the spatial relationships between the pixels.
3. Solution: Flattening: We "flatten" the image by taking all the pixel values and arranging them in a single column (or row). This turns the 2D/3D grid into a 1D list (a vector). Now, each pixel value becomes a single "feature" that the model can work with.

How `train_set_x_orig.reshape(train_set_x_orig.shape[0], -1).T` Flattens the Image (Chronological Breakdown):

Let's assume `train_set_x_orig` has a shape of (150, 64, 64, 3) as in your notes. This means we have 150 images, each 64 pixels high, 64 pixels wide, with 3 color channels.

1. Initial State: `train_set_x_orig` is a 4-dimensional array: (number_of_images, height, width, color_channels). In our example: (150, 64, 64, 3).
2. `train_set_x_orig.shape[0]`: This part extracts the number of images, which is 150. We want to keep this as the number of rows in our reshaped array, where each row will represent one image.
3. `-1` in `reshape`: The `-1` is a clever trick in NumPy. It tells NumPy to automatically calculate the size of that dimension so that the total number of elements in the array remains the same. In our case, the total number of pixels in one image is $64 \times 64 \times 3 = 12288$. So, `reshape(150, -1)` transforms the array into a 2D array with a shape of (150, 12288).
 - o Result after `reshape`: We now have a 2D array where each of the 150 rows is a 1D array containing all the 12288 pixel values of a single image. The image's 2D structure (and color channels) has been laid out sequentially into a single row.
4. `.T` (Transpose): The `.T` at the end performs the transpose operation. This swaps the rows and columns of the array. If our array was (150, 12288), after the transpose, it becomes (12288, 150).
 - o Final State after `.T`: Now, we have a 2D array where:
 - The number of rows is 12288, representing each individual pixel position across all the images.
 - The number of columns is 150, with each column being a 1D array of length 12288 containing the flattened pixel values of a single image.
5. **Why the Transpose?**

In the context of building a simple neural network for image classification (like the logistic regression as a neural network example often shown), it's often convenient to have each *column* of the input matrix represent a single training example. This aligns well with how matrix multiplications are performed in the forward and backward propagation steps of the neural network.

6. Vector in This Context

In this context, when we talk about a "flattened image," we are essentially referring to representing the 2D (or 3D with color channels) image as a **1D vector**. This vector is a sequence of pixel values arranged in a specific order (e.g., row by row, then channel by channel).

For example, if you have a 2x2 grayscale image:

```
[[10, 20],  
 [30, 40]]
```

When flattened into a vector (row by row), it becomes:

```
[10, 20, 30, 40]
```

If it were a 2x2 RGB image, the vector would interleave the color channels:

```
[[[255, 0, 0], [0, 255, 0]],  
 [[0, 0, 255], [255, 255, 255]]]
```

Flattened (assuming row-major order and then channels):

```
[255, 0, 0, 0, 255, 0, 0, 0, 255, 255, 255, 255]
```

So, in the context of image processing for these types of machine learning models, a **vector** is a 1-dimensional array that contains all the pixel values of an image arranged in a linear sequence. Each flattened image becomes a single data point represented by this vector, and these vectors form the columns (after the transpose) of the input matrix to the model.