Deep Learning

Fundamentals and state of the art architectures

Jawad ALAOUI

Université Gustave Eiffel

NORMA

The Scope of Deeplearning module

- Introduction to Deep Learning and Neural Networks
- Standard Neural Networks
- Convolutional Neural Networks (CNNs)
- Recurrent Neural Networks (RNNs), LSTMs
- Transformers and Hugging Face
- Reinforcement Learning with Human Feedback (RLHF)
- Diffusion Models

The Problem

Image Classification for Autonomous Vehicles

Is this a bicycle ?

Why it matters: The classification of vehicles is essential for self-driving cars to navigate safely, as distinguishing between bicycles, motorcycles, and buses affects road decision-making.

Feature engineering vs feature learning: Classic machine learning models like linear and logistic regression struggle with complex image problems because they depend on handcrafted features, especially when processing unstructured data.
Handcrafted features cannot adequately handle the complexity of data due to variations in lighting, angles, and vehicle types in real-world situations, making manual extraction very complex.

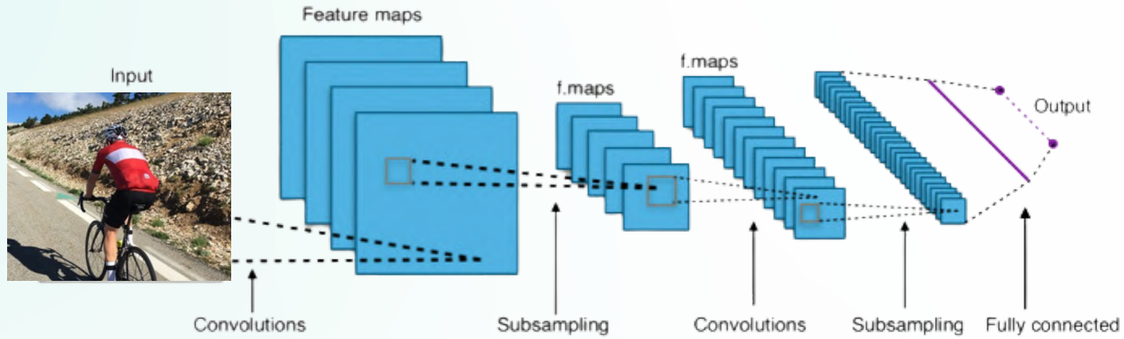
non-linearity: Moreover, many important characteristics of an image are non-linear. Recognizing the difference between a truck and a car requires complex interactions between pixel values that simple linear models can't capture.

The Solution

Introduction to Deep Learning

Deep learning demonstrates exceptional proficiency in complex tasks such as image classification by learning hierarchical representations from raw data. This capability enhances the understanding of high-dimensional data, exemplified in applications like vehicle classification in self-driving cars.

Its versatility extends across various domains, including natural language processing, speech recognition, and images segmentation, where it markedly surpasses traditional methodologies by automatically acquiring meaningful representations.



1943

Warren McCulloch and Walter Pitts introduced the concept of a simplified brain mode

1958

Frank Rosenblatt developed the Perceptron algorithm, one of the earliest models for neural networks capable of learning.

1965

The first working deep learning algorithm Alexey Ivakhnenko and Lapa in Ukraine

1967

The first deep learning multilayer perceptron trained by stochastic gradient descent by Shun'ichi Amari.

1989

Yann LeCun utilized backpropagation to train convolutional neural networks (CNNs) for tasks such as handwritten digit recognition.

2012

Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton won the ImageNet competition using their deep convolutional neural network, AlexNet.

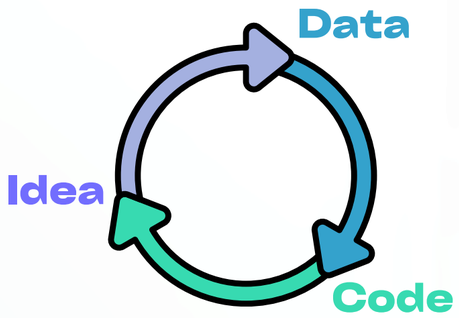
2017

Google researchers developed the Transformer model, transforming natural language processing and paving the way for large-scale models like BERT and GPT.

Why Deep Learning is Possible Today ?

The rise of GPUs and specialized hardware has greatly reduced the computational cost of training large neural networks, making deep learning more accessible and efficient.

The past decade has seen an explosion of data from mobile devices, sensors, and the Internet of Things (IoT), which traditional algorithms struggle to manage effectively.



Key innovations like Backpropagation, ReLU activation, and attention mechanisms have significantly enhanced the efficiency and performance of deep learning models, enabling faster training and improved results for large-scale networks.

Numerous Architectures for Various Challenges

Supervised learning

Convolutional Neural Networks (CNNs):
Used for image classification and object detection tasks.

Recurrent Neural Networks (RNNs) & LSTMs:
Applied in time series forecasting and sequential data processing.

Transformers:
Solve text classification and sequence translation tasks using labeled data.

Unsupervised learning

Autoencoders:
Used for dimensionality reduction and anomaly detection in unlabeled d

Generative Adversarial Networks (GANs):
Generate synthetic data like images or videos using a two-network system.

Diffusion Models:
Generate high-quality images by learning to denoise noisy data.

Self-supervised learning

Transformers:
Learn from masked data for tasks like language representation.

Foundations: Introducing the Perceptron

The perceptron is a fundamental algorithm for binary classification and serves as the building block for more complex neural networks.

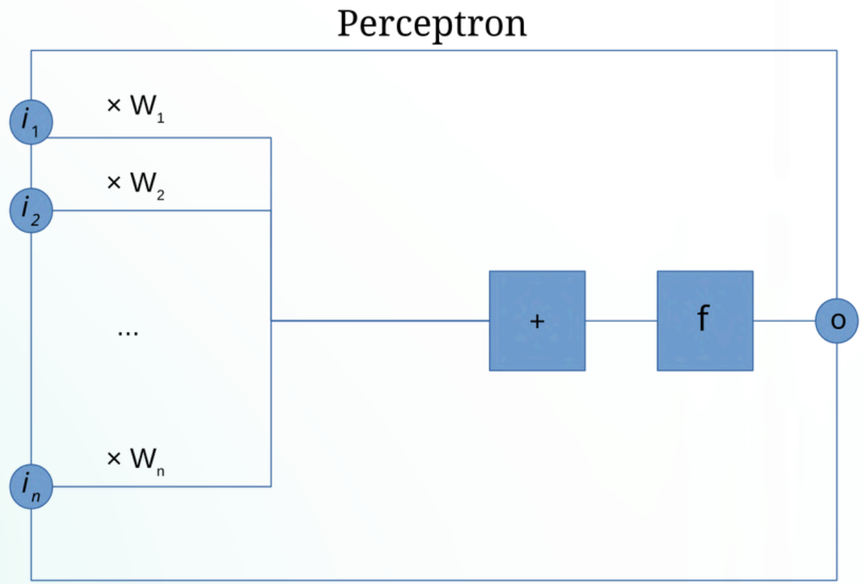
Input: $x = (x_1, x_2, \ldots, x_n)$

Weights: $w = (w_1, w_2, \ldots, w_n)$

Bias: b

Output: $o = f\left(\sum_{i=1}^{n} w_i \cdot x_i + b\right)$

Activation function*: $f(x) = \frac{1}{1 + e^{-x}}$ (Sigmoid function)



* Activation function is not always Sigmoid. many other options are available: Step function, ReLu, Thanh...

Perceptron vs logistic regression

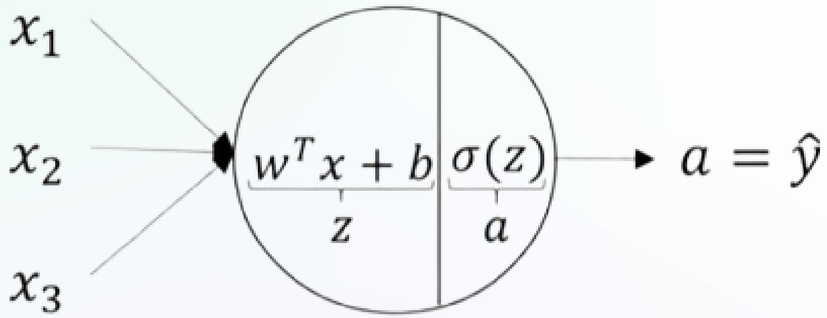
Logistic regression is A generalized linear model (GLM) for classification. It solves classification problems by outputting probabilities chosen to be the Bernoulli distribution.

The link function (logit) is: $g(\mu) = \log\left(\frac{\mu}{1-\mu}\right)$

$$P(o=1) = \mathrm{E}(o) = \mu = g^{-1}(z) = \frac{1}{1+e^{-z}}$$

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

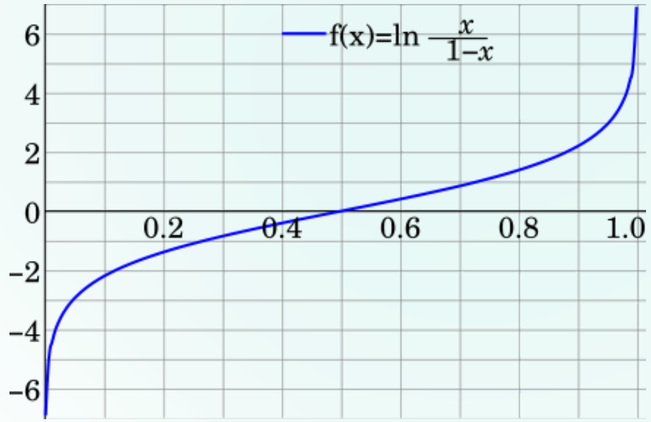
$$a = g^{-1}(z) = \sigma(z)$$



Let's begin by exploring what we can accomplish with a single neuron.

Logistic Regression – Link Function

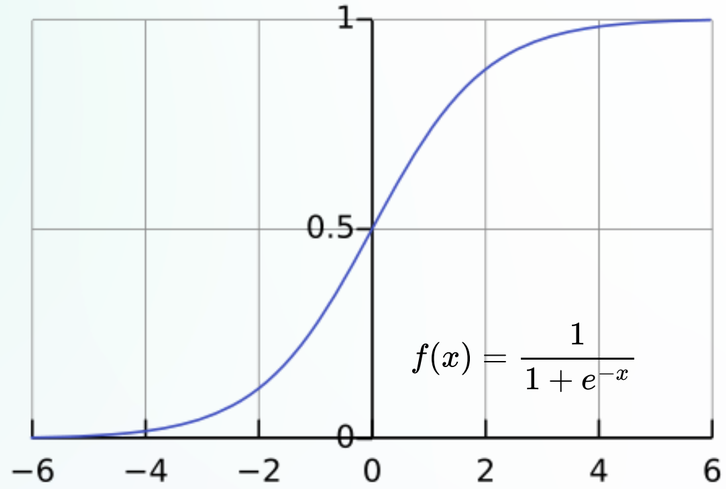
Logit Function



The logit function represents the log of the odds of the outcome, transforming probabilities from

$$(0, 1) \text{ to } (-\infty, +\infty)$$

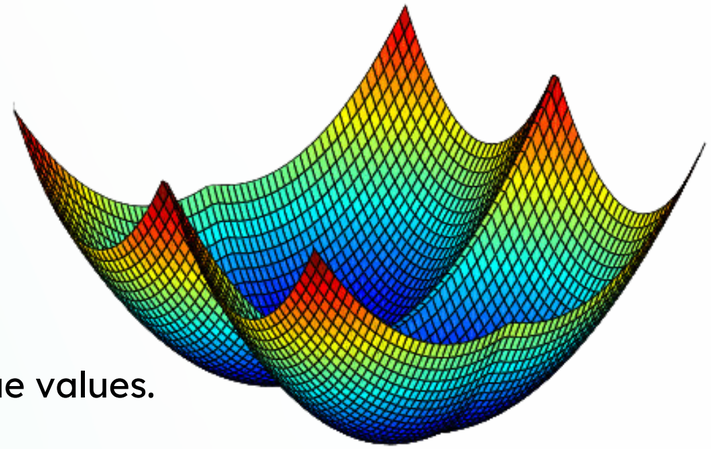
Sigmoid Function



The sigmoid function is the inverse of the logit function and maps values from

$$(-\infty, +\infty) \text{ to } (0, 1)$$

What is a Cost function



A cost function measures how well a model fits the data by quantifying the error between the predicted and actual outputs. It is a key component in the optimization process.

Multiple Ways to Define a Cost Function:

There are several ways to define a cost function, depending on the model and problem

Least Squares: Used in regression, it minimizes the squared differences between predicted and true values.

$$\text{Cost} = \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$$

Cross-Entropy (Log Loss): Applied in classification tasks, it measures the error between predicted probabilities and true classes.

$$\text{Cost} = E(-log(p(X))) = -\frac{1}{m} \sum_{i=1}^{m} \left(y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \right)$$

Maximum Likelihood: Maximizes the likelihood of observed data under the model, commonly used in probabilistic models.

$$\text{Cost} = -\text{Log Likelihood} = -\sum_{i=1}^{m} \log P(y_i|\theta)$$

Loss vs. Cost Function:

The loss function measures error for one training example, while the cost function aggregates this error across the entire dataset.

What is Gradient Descent?

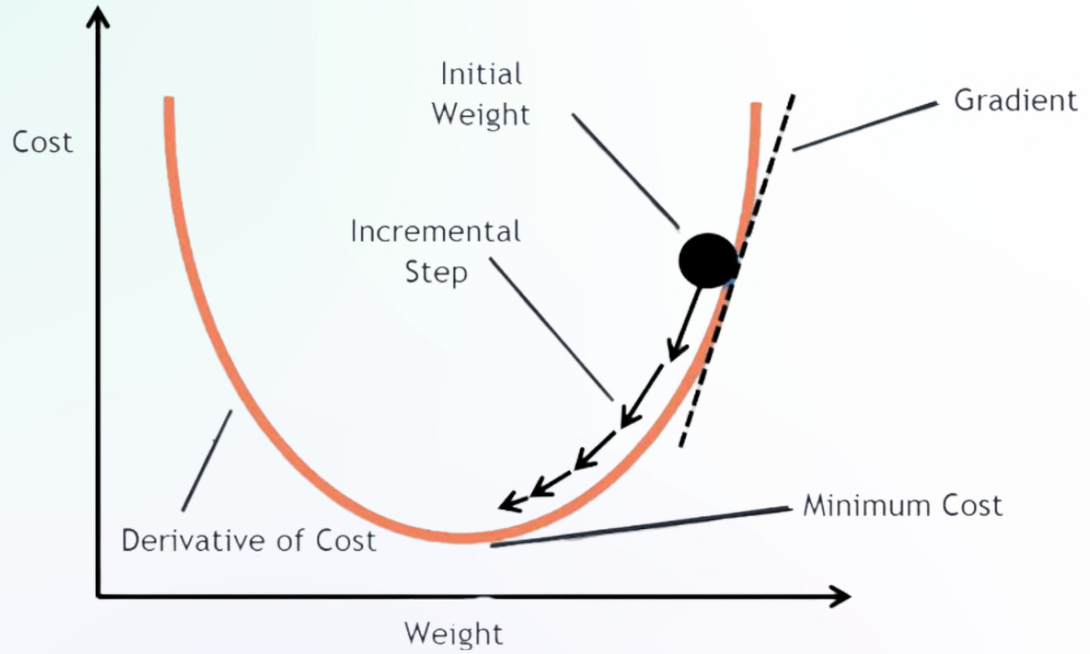
Gradient Descent, introduced by Augustin-Louis Cauchy in 1847, is an optimization method for minimizing functions. It is essential in machine learning for minimizing the cost function by iteratively adjusting parameters.

Learning rate

$$\theta := \theta - \alpha \frac{\partial}{\partial \theta} J(\theta)$$

Model parameters

Cost function



Cost function for Logistic Regression (1)

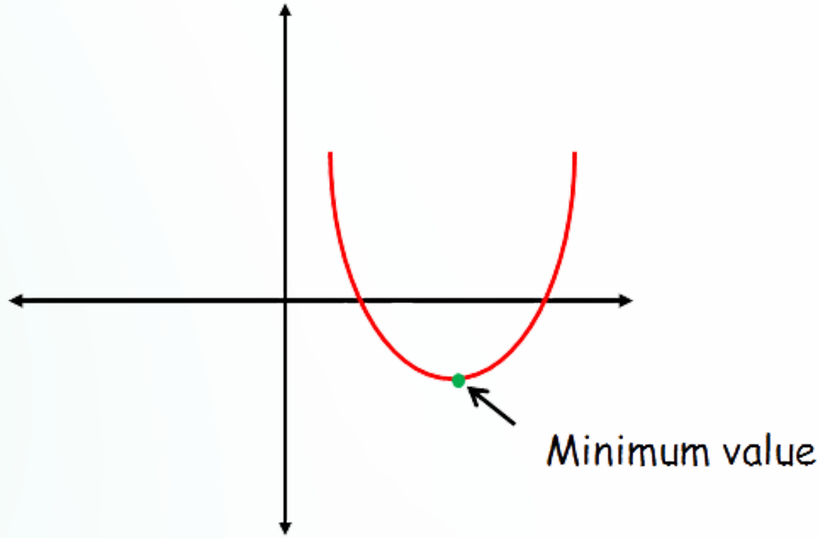
The cost function for logistic regression can be derived from both likelihood and cross-entropy, which lead to the same formulation.

The likelihood function maximizes the probability of observing the data

$$L(\beta) = \prod_{i=1}^{m} p(x_i)^{y_i} \cdot (1 - p(x_i))^{1-y_i}$$

Log-Likelihood simplifies the computation

$$\log L(\beta) = \sum_{i=1}^{m} y_i \log(p(x_i)) + (1 - y_i) \log(1 - p(x_i))$$



Negative Log-Likelihood (Cross-Entropy): Minimizing the negative log-likelihood is equivalent to minimizing cross-entropy

$$-\log L(\beta) = -\sum_{i=1}^{m} y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)$$

Cost function for Logistic Regression (2)

The general cost function for logistic regression is derived from the negative log-likelihood and, in this context, is divided by m (the number of training examples) to normalize the loss over the dataset.

$$J = -\frac{1}{m} \sum_{i=1}^{m} (y_i \log(a_i) + (1 - y_i) \log(1 - a_i))$$

Where:

$$a_i = \sigma(z_i)$$

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

$$z_i = \sum_{j=1}^{n} w_j \cdot x_{i,j} + b$$

and y are the real values of the output (taget value).

Define Partial Derivative

The general cost function for logistic regression is derived from the negative log-likelihood and, in this context, is divided by mmm (the number of training examples) to normalize the loss over the dataset.

Chain Rule to Derive the Partial Derivative:

$$\frac{\partial J}{\partial w_j} = \frac{1}{m} \sum_{i=1}^{m} \left(\frac{\partial J_i}{\partial a_i} \cdot \frac{\partial a_i}{\partial z_i} \cdot \frac{\partial z_i}{\partial w_j} \right)$$

$$\begin{cases} \dfrac{\partial J_i}{\partial a_i} = -\dfrac{y_i}{a_i} + \dfrac{1 - y_i}{1 - a_i} = a_i - y_i \\ \dfrac{\partial a_i}{\partial z_i} = a_i(1 - a_i) \\ \dfrac{\partial z_i}{\partial w_j} = x_{ij} \end{cases} \longrightarrow \begin{cases} \dfrac{\partial J}{\partial w_j} = \dfrac{1}{m} \sum_{i=1}^{m} (a_i - y_i) x_{ij} \\ \dfrac{\partial J}{\partial b} = \dfrac{1}{m} \sum_{i=1}^{m} (a_i - y_i) \end{cases}$$

Forward propagation algorithm

In forward propagation, we initialize the parameters W and b, then compute the linear combination of the input features:

Linear Combination vectorize for W:

$$z_i := w^T x_i + b$$

Activation (Sigmoid):

$$a_i := \frac{1}{1 + e^{-z_i}}$$

Cost Function (Logistic Regression):

$$J := -\frac{1}{m} \sum_{i=1}^{m} (y_i \log(a_i) + (1 - y_i) \log(1 - a_i))$$

Finally, the cost function measures the error between the predicted and actual values:

Back propagation

After the forward propagation step, we use backpropagation to update the model's parameters. The update rules are based on the gradient of the cost function with respect to each parameter.

Weights Update:

$$w_j := w_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (a_i - y_i) x_{ij}$$

Bias Update:

$$b := b - \alpha \frac{1}{m} \sum_{i=1}^{m} (a_i - y_i)$$

Once these updates are made, forward propagation is repeated to improve model predictions.

Stopping Condition:

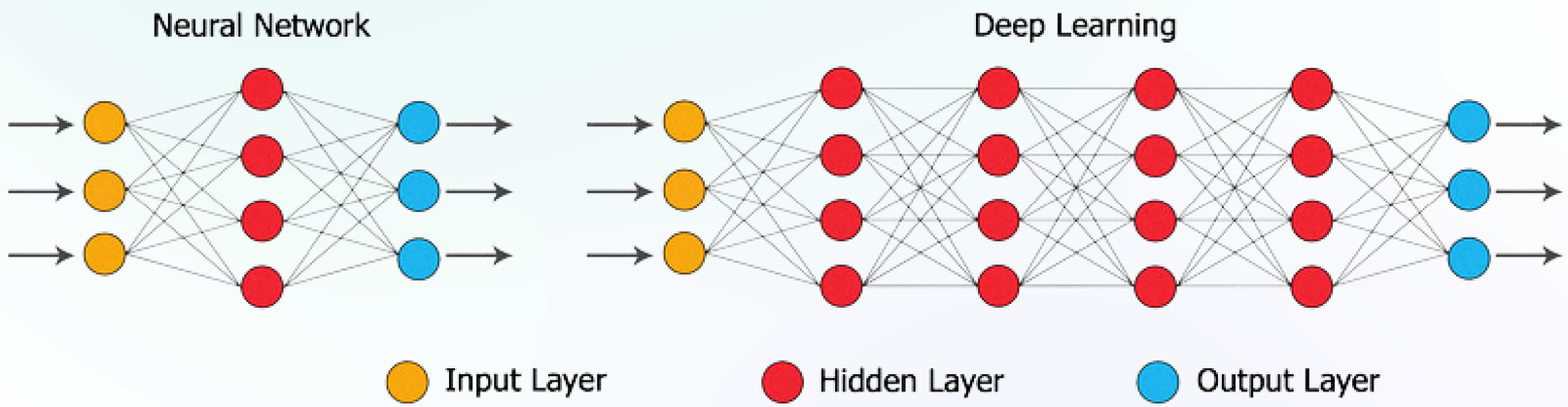
We stop when the cost function change is minimal (convergence) or when it reaches a set threshold, indicating the model has learned sufficiently for accurate predictions.

Neural Network representation

A shallow neural network consists of only one or two hidden layers, while a deep neural network has multiple hidden layers, allowing it to learn more complex patterns in the data.
Logistic regression can be viewed as a single-layer neural network where the output is the probability of a class.

Layers:

- Input Layer: Takes the input features
- Hidden Layers: Perform computations and feature transformations.
- Output Layer: Produces the final predictions.



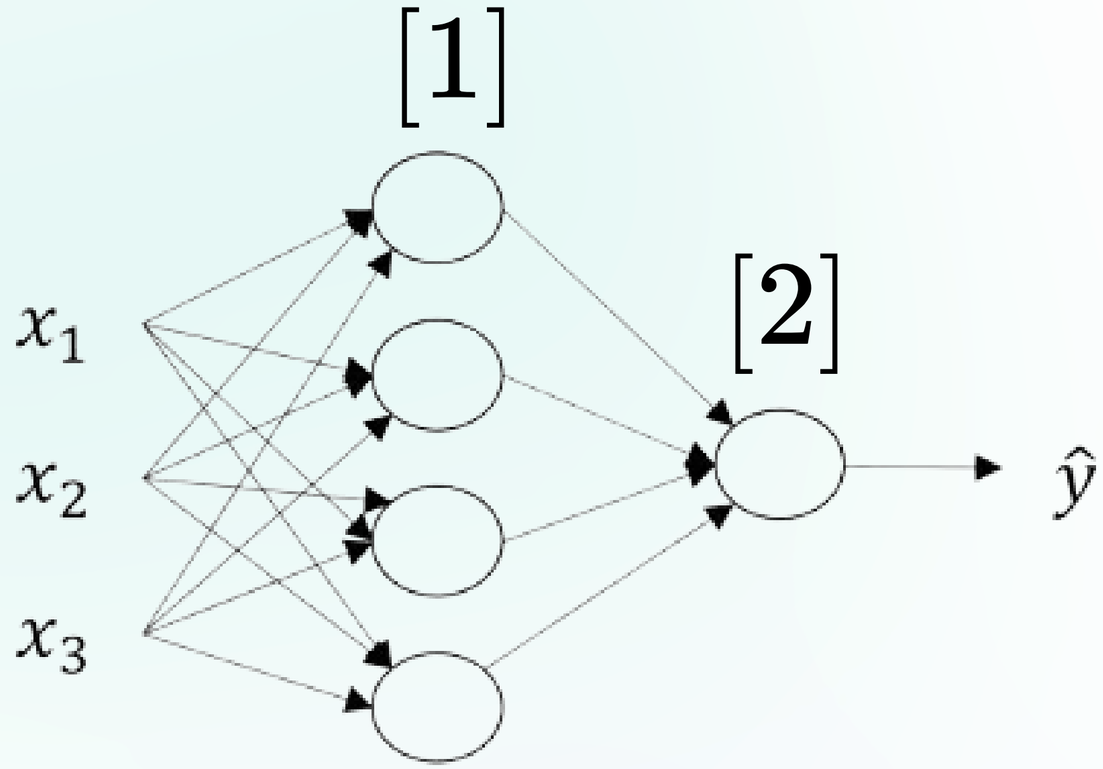
Shallow Neural Network - 2 Layers

Each neuron in the hidden layer performs two main operations:

$$\begin{cases} z_j^{[1]} = w_j^{[1]T} x + b_j^{[1]}, \\ a_j^{[1]} = \sigma(z_j^{[1]}) \end{cases} \quad [1]$$

Same for the last output layer:

$$\begin{cases} z_j^{[2]} = w^{[2]T} a_j^{[1]} + b_j^{[2]}, \\ \hat{y}_j = \sigma(z_j^{[2]}) \end{cases} \quad [2]$$



Shallow Neural Network - Vectorization

Vectorization occurs in two aspects: input features and weight matrices are processed simultaneously, allowing parallel computation for all neurons and training examples.

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

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

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

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

X is now a matrix where each column is a training example.

W is the a matrix where each line j is the transpose of wj.

Activation functions

Activation functions introduce non-linearity, allowing neural networks to model complex relationships.

Sigmoid :

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

- Range: (0, 1)
- Use Case: Output layer in binary classification.
- Limitation: Prone to vanishing gradients with large or small inputs.

Rectified Linear Unit (ReLU):

$$\max(0, z)$$

- Range: [0, ∞)
- Use Case: Widely used in hidden layers.
- Advantage: Faster learning by mitigating vanishing gradients for positive inputs.

Hyperbolic Tangent:

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

- Range: (-1, 1)
- Advantage: Centers data around zero, often better than sigmoid for hidden layers.
- Limitation: Also suffers from vanishing gradients with extreme inputs.

Leaky ReLU:

$$a = \max(\alpha z, z)$$

- α is a small constant (e.g., 0.01).
- Range: (-∞, ∞)
- Advantage: Prevents “dead neurons” by allowing small gradient when z is negative.

Derivatives of activation functions

Sigmoid :

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

Small gradients for large or small z, causing slow learning.

Hyperbolic Tangent:

$$\tanh'(z) = 1 - \tanh^2(z)$$

Reduces vanishing gradient issues compared to sigmoid but still susceptible at extreme values.

Rectified Linear Unit (ReLU):

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

Provides a consistent gradient for positive inputs, promoting faster learning.

Leaky ReLU:

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

Small gradient for negative z values, preventing dead neurons.

Activation function choice and best practices

- Experimentation Encouraged: Test different activation functions across layers to find the best fit.
- Validation for Selection: Evaluate model performance on a validation or development set to determine the most effective activation function.
- Common Practices:
  - Hidden Layers: ReLU or Leaky ReLU for faster convergence and better gradients.
  - Output Layer: Sigmoid for binary classification, softmax for multi-class classification.
- Key Takeaway: ReLU is generally preferred in hidden layers, but consider task specifics and be open to alternative functions.

Gradient descent - Forward propagation

First Layer (Hidden Layer):

$$Z^{[1]} = W^{[1]}X + b^{[1]} \qquad A^{[1]} = g^{[1]}(Z^{[1]})$$

Second Layer (Output Layer):

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]} \qquad A^{[2]} = g^{[2]}(Z^{[2]}) = \sigma(Z^{[2]})$$

Gradient descent - back propagation

Backpropagation calculates the gradients of the loss function for each neural network parameter, enabling updates to weights and biases to minimize loss.

Derivatives for Output Layer

$$\begin{cases} dZ^{[2]} = A^{[2]} - Y \\ dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T} \\ db^{[2]} = \frac{1}{m} \sum dZ^{[2]} \end{cases}$$

Derivatives for Hidden Layer

$$\begin{cases} dZ^{[1]} = W^{[2]T} dZ^{[2]} \circ g^{[1]'}(Z^{[1]}) \\ dW^{[1]} = \frac{1}{m} dZ^{[1]} X^{T} \\ db^{[1]} = \frac{1}{m} \sum dZ^{[1]} \end{cases}$$

Gradient matrices has same dimensions as their original matrices

Weight Initialization

To avoid neurons learning the same function, we initialize weights randomly. If weights are zero, all neurons in a layer will compute the same function, limiting learning.

$$\begin{cases} W^{[1]} \sim \mathcal{N}(0, 0.01^2), & b^{[1]} = 0 \\ W^{[2]} \sim \mathcal{N}(0, 0.01^2), & b^{[2]} = 0 \end{cases}$$

Large initial weights can slow learning due to small slopes in activation functions.

Generalized L-Layer Neural Network

Deep learning leverages layered structures to model complex functions more compactly than shallow networks.

L-layer deep neural network model structure is:

$$[\text{LINEAR} \rightarrow \text{g}]^{L-1} \rightarrow \text{LINEAR} \rightarrow \text{SIGMOID}$$

- The first L–1 layers use the tanh or ReLu activation function.
- The output layer uses the sigmoid activation function

Equations for an L-Layer Network:

$$Z^{[i]} = W^{[i]} A^{[i-1]} + b^{[i]}$$
$$A^{[i]} = g(Z^{[i]})$$

Dimensions of the weights and bias matrices.

The input layer is of the size (x, m) where m is the number of images.

Layer number	Shape of W	Shape of b	Linear Output	Shape of Activation
Layer 1	$(n[1], x)$	$(n[1], 1)$	$Z[1] = W[1]X + b[1]$	$(n[1], m)$
Layer 2	$(n[2], n[1])$	$(n[2], 1)$	$Z[2] = W[2]A[1] + b[2]$	$(n[2], m)$
:	:	:	:	:
Layer L – 1	$(n[L–1], n[L–2])$	$(n[L–1], 1)$	$Z[L–1] = W[L–1]A[L–2] + b[L–1]$	$(n[L–1], m)$
Layer L	$(n[L], n[L–1])$	$(n[L], 1)$	$Z[L] = W[L]A[L–1] + b[L]$	$(n[L], m)$

Backpropagation for L-Layer Neural Network

General Structure: Backpropagation computes the gradients required for updating the parameters W and b in each layer of a neural network. The goal is to minimize the loss by adjusting the weights and biases based on the computed gradients.

Compute the Gradient at the Output Layer

$$\begin{cases} dZ^{[L]} = A^{[L]} - Y \\ dW^{[L]} = \frac{1}{m} dZ^{[L]} A^{[L-1]T} \\ db^{[L]} = \frac{1}{m} \sum dZ^{[L]} \end{cases}$$

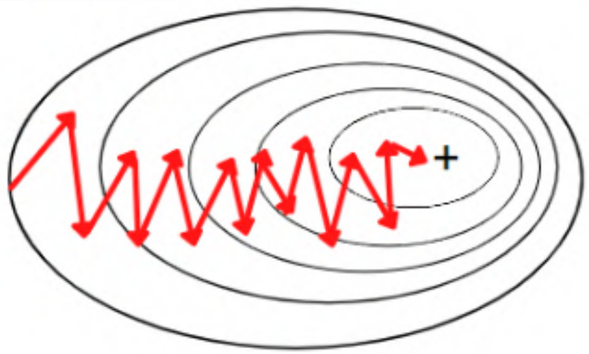
Iterate Backward Through Each Hidden Layer:

$$\begin{cases} dZ^{[i]} = (W^{[i+1]})^T dZ^{[i+1]} \circ g'^{[i]}(Z^{[i]}) \\ dW^{[i]} = \frac{1}{m} dZ^{[i]} A^{[i-1]T} \\ db^{[i]} = \frac{1}{m} \sum dZ^{[i]} \end{cases}$$

Stochastic gradient descent

Stochastic Gradient Descent (SGD) is a variant of gradient descent that updates the model parameters using a single training example at each iteration. Instead of computing the gradient over the entire dataset, it approximates it with just one example:

$$\nabla F_{\mathrm{MB}}(\mathbf{x}) = \nabla f_i(\mathbf{x})$$

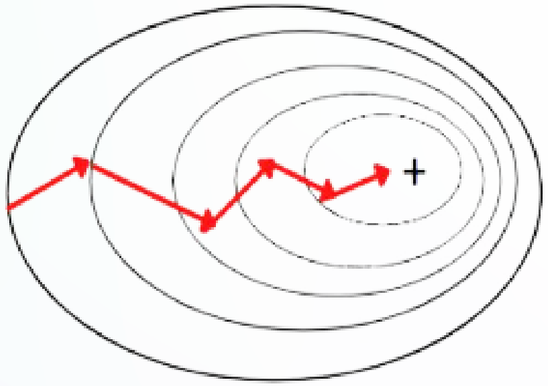


- High Variance: Updates based on individual examples can create a noisy path to the minimum.
- Frequent Updates: Enables faster updates than batch gradient descent, potentially accelerating convergence.
- Improved Generalization: Noisiness may help escape local minima, resulting in better generalization.

Mini-batch gradient descent

Mini-batch gradient descent is a compromise between batch gradient descent and stochastic gradient descent. Instead of using the entire dataset or a single example, it computes the gradient using a small batch of b training examples:

$$\nabla F_{\mathrm{MB}}(\mathbf{x}) = \frac{1}{b} \sum_{i=1}^{b} \nabla f_i(\mathbf{x})$$



- Epoch: One complete pass through the training dataset; multiple epochs may be needed for convergence in mini-batch gradient descent.
- Batch Size: Number of samples used for each gradient update.
- Efficiency and Stability: Balances computational efficiency of stochastic gradient descent with the stability of batch gradient descent.

Hyperparameters in an L-Layer Network

What are Hyperparameters?

Hyperparameters are the configurations set before training a model. They control the learning process but are not learned from data.

List of Hyperparameters:

- Learning Rate α: Controls the step size in gradient descent.
- Number of Layers : The depth of the network, affecting its ability to capture complex patterns.
- Number of Neurons per Layer: Determines the width of each layer.
- Batch Size: Number of training examples in each mini-batch.
- Number of Epochs: The total number of passes through the entire training dataset.
- Activation Functions: Defines the type of non-linearity introduced in each layer .
- Initialization Method: Strategy for initializing weights, influencing convergence.