Deep Learning

VVIT@2025

November 10, 2025

Overview

Introduction

What is Deep Learning?

Artificial Intelligence

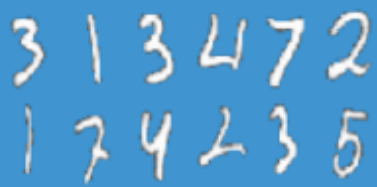
Any technique that enables computers to mimic human behavior

Machine Learning

Ability to learn without explicitly being programmed

Deep Learning

Extract patterns from data using neural networks



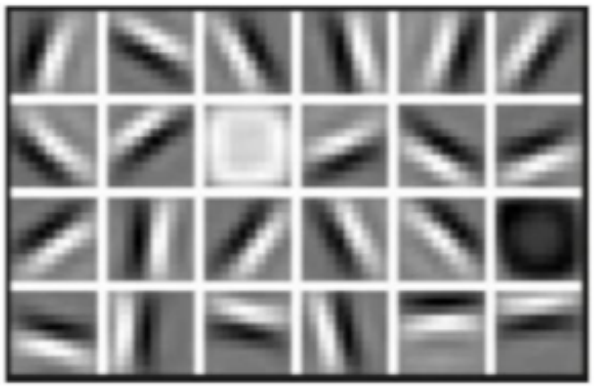
Why Deep Learning?

Why Deep Learning?

Hand engineered features are time consuming, brittle and not scalable in practice

Can we learn the **underlying features** directly from data?

Low Level Features



Lines & Edges

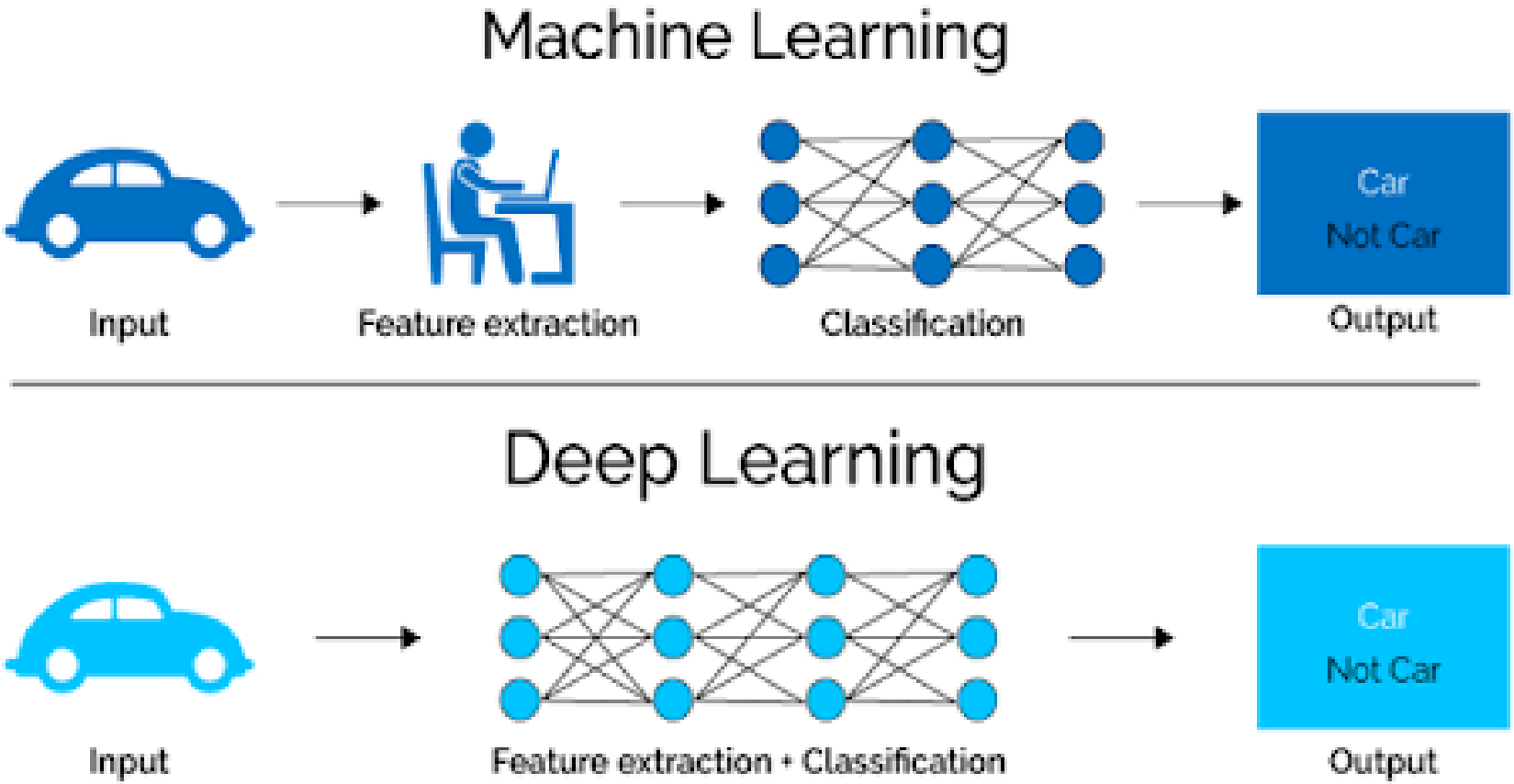
Mid Level Features

Eyes & Nose & Ears

High Level Features

Facial Structure

ML vs DL


Machine Learning
Input
Feature extraction
Classification
Car
Not Car
Output
Deep Learning
Input
Feature extraction + Classification
Car
Not Car
Output

Why Now?

1952 Stochastic Gradient Descent

1958 Perceptron
- Learnable Weights

1986 Backpropagation
- Multi-Layer Perceptron

1995 Deep Convolutional NN
- Digit Recognition

Neural Networks date back decades, so why the resurgence?

1. Big Data
- Larger Datasets
- Easier Collection & Storage

IMAGENET

WIKIPEDIA The Free Encyclopedia

2. Hardware
- Graphics Processing Units (GPUs)
- Massively Parallelizable

3. Software
- Improved Techniques
- New Models
- Toolboxes

TensorFlow

AI and Machine Learning

- John McCarthy, widely recognized as one of the godfathers of AI, defined it as the science and engineering of making intelligent machines.
- Machine learning is a subset of AI. That is, all machine learning counts as AI, but not all AI counts as machine learning.
- Machine learning methods learn from data. Learning from data is used in situations where we don't have an analytic solution, but we do have data that we can use to construct an empirical solution.
- A computer program is said to learn from experience E with respect to some class of task T and performance measure P, if its performance at tasks in T,as measured by P, improves with experience E.

Deep Learning

Deep learning is a subset of machine learning.

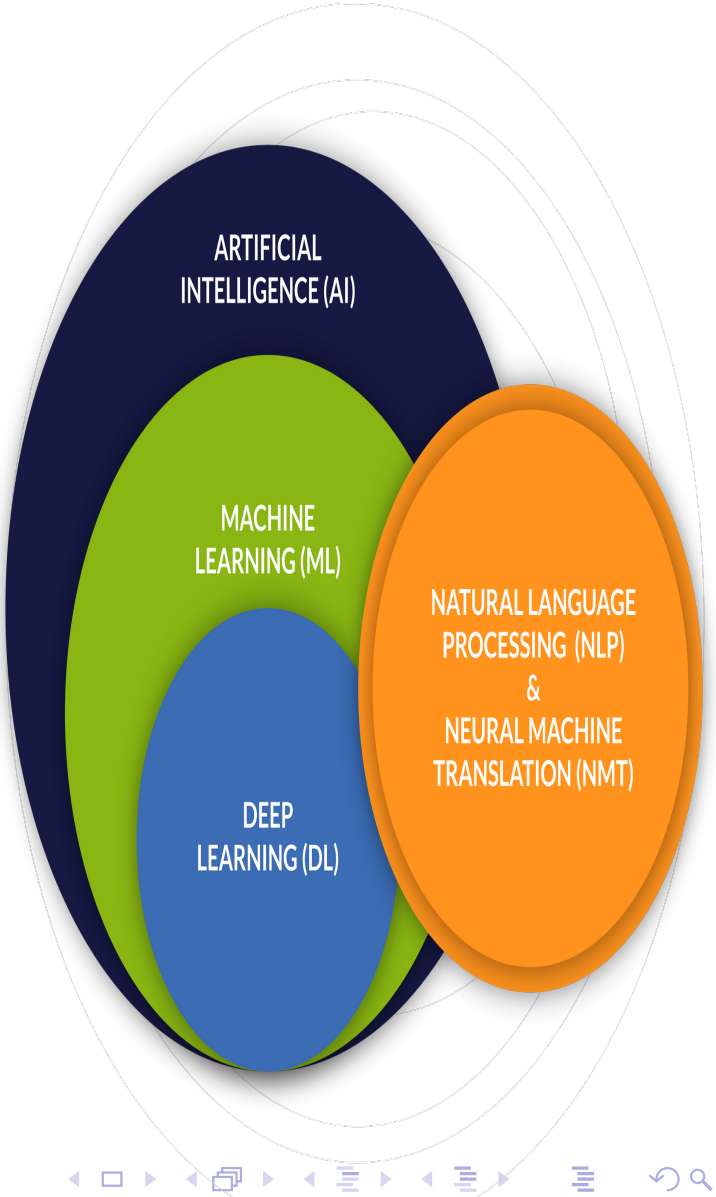
Deep Artificial Neural Network

The term "Deep learning" refers to a Deep artificial neural network.

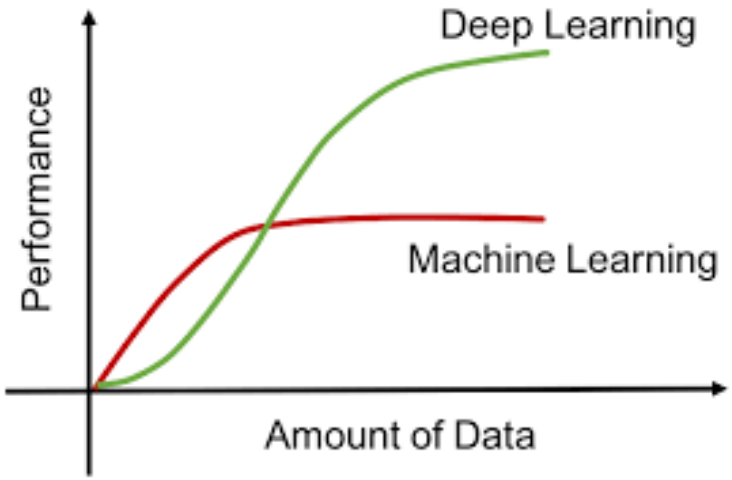
– inspired by the structure and function of the brain.

The term "Deep"

The term "Deep" in deep learning refers to the number of layers in the artificial neural network.



Deep Learning vs Machine Learning



Improvement in Performance

- Neural networks can continue to improve in performance with an increase in the amount of data
- Larger networks and more data lead to better results
- Contrary to other machine learning techniques that reach a performance plateau

Introduction

Deep Learning is a subset of Machine Learning, which in turn is a subset of Artificial Intelligence (AI). Deep Learning seeks to emulate the neural networks of the human brain in order to "learn" from vast amounts of data. While a neural network with a single layer can still make an approximate guess, additional hidden layers can help optimize the decision-making process.

Deep Learning

Definition of Deep Learning

Deep learning is a subfield of machine learning that aims to train artificial neural networks with multiple layers on extensive datasets. It enables the networks to learn representations and patterns directly from the data, without the need for explicit programming. The technique has achieved significant success in solving complex tasks such as image recognition, speech processing, and natural language understanding.

Fundamentals of Deep Learning

- Weights and Biases
- Forward Propagation
- Activation Function
- Loss Function
- Optimizer
- Backpropagation
- Overfitting and Underfitting
- Regularization

Fundamentals I

- **Weights and Biases**: The knowledge of a neural network is stored in the weights and biases that are learnt during the training process. These weights determine the influence that a given input (or neuron) has on an output.
- **Forward Propagation**: Forward propagation is the process in which the neural network makes its predictions. Starting from the input layer, it propagates the input through the network, layer by layer, until it reaches the output layer. At each neuron, it multiplies the input by the weights, adds the bias, and applies the activation function to generate the output.
- **Activation Function**: The activation function decides whether a neuron should be activated or not by transforming the weighted sum of the inputs and the bias. Common activation functions include the Sigmoid, Tanh, and ReLU.

Fundamentals II

- **Loss Function**: A loss function measures the difference between the predicted output of the neural network and the actual output during training. The goal is to minimize this difference.
- **Optimizer**: An optimizer is an algorithm used to adjust the parameters of your neural network, such as weights and learning rate, to reduce losses.
- **Backpropagation**: This is a method used to calculate the gradient of the loss function with respect to each weight in the neural network, which in turn is used to update the weights and reduce the loss.

Fundamentals III

- **Overfitting and Underfitting**: Overfitting occurs when a model learns the training data too well, including its noise and outliers, and performs poorly on unseen data. Underfitting is the opposite, where the model fails to learn the underlying patterns of the data.
- **Regularization**: Techniques like dropout, weight decay, early stopping, are used to prevent overfitting by adding a penalty to the loss function or by altering the architecture of the neural network.

Building Blocks of Neural Networks

- Neuron: The basic unit of computation in a neural network is the neuron, often called a node or unit. It receives input from some other nodes, or from an external source and computes an output.
- Layers of Neurons: Neurons are organized in layers. We have an input layer which directly receives the data, one or more hidden layers which process the inputs received from the previous layer, and an output layer which makes the final prediction.
- Perceptron: The simplest type of neural network is a single-layer perceptron network, which consists of a single layer of output nodes connected to a layer of input nodes.
- Multilayer Perceptron (MLP): An MLP has an input layer, an output layer, and one or more hidden layers between them. It uses a nonlinear activation function, typically the sigmoid function or the ReLU function.

Building Blocks of Neural Networks

- Convolutional Neural Networks (CNN): Designed for grid-structured input data such as images, CNNs use convolutional layers that filter inputs for useful information.
- Recurrent Neural Networks (RNN): An RNN is a type of neural network that has connections pointing backwards, making them good for modeling sequence data such as time series or natural language.
- Autoencoders (AE): An autoencoder is a neural network trained to attempt to copy its input to its output, typically by learning a compressed representation of the input data, which can then be used for dimensionality reduction or other tasks.

Biological Neuron

November 10, 2025

cell differentiation and cell division

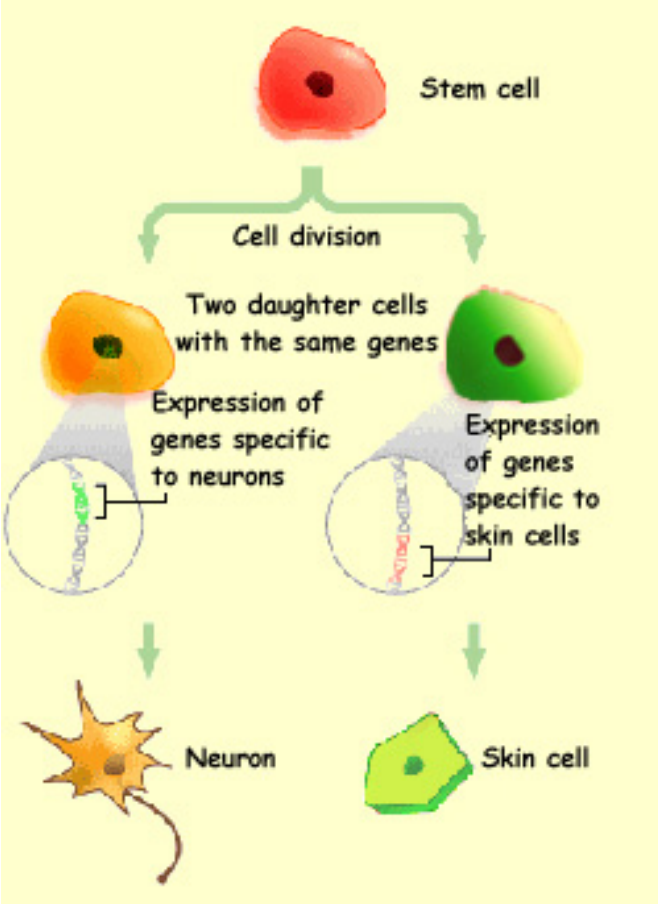


Figure: Cell Division

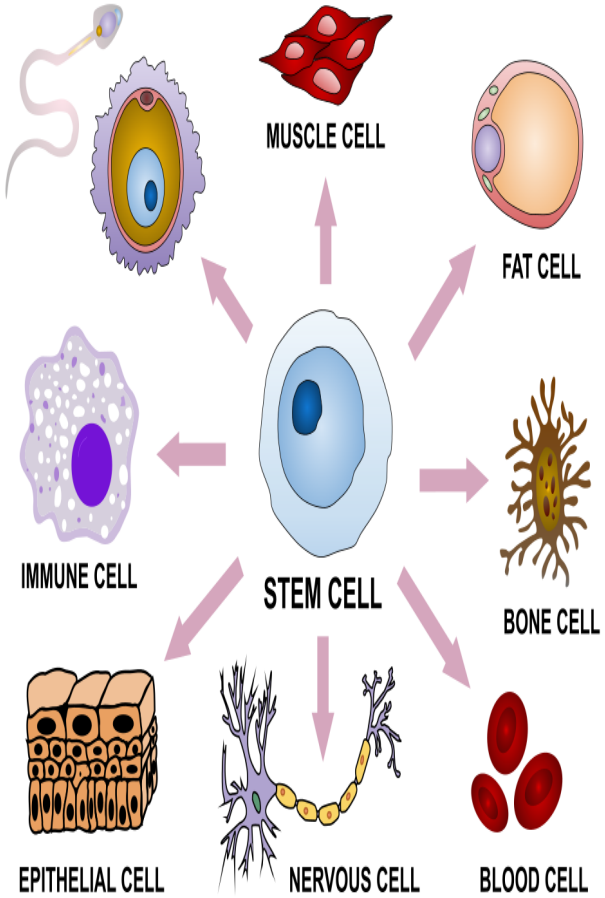


Figure: Stem Cell Differentiation into Various Tissue Types

Neuron

The human body is made up of trillions of cells. Cells of the nervous

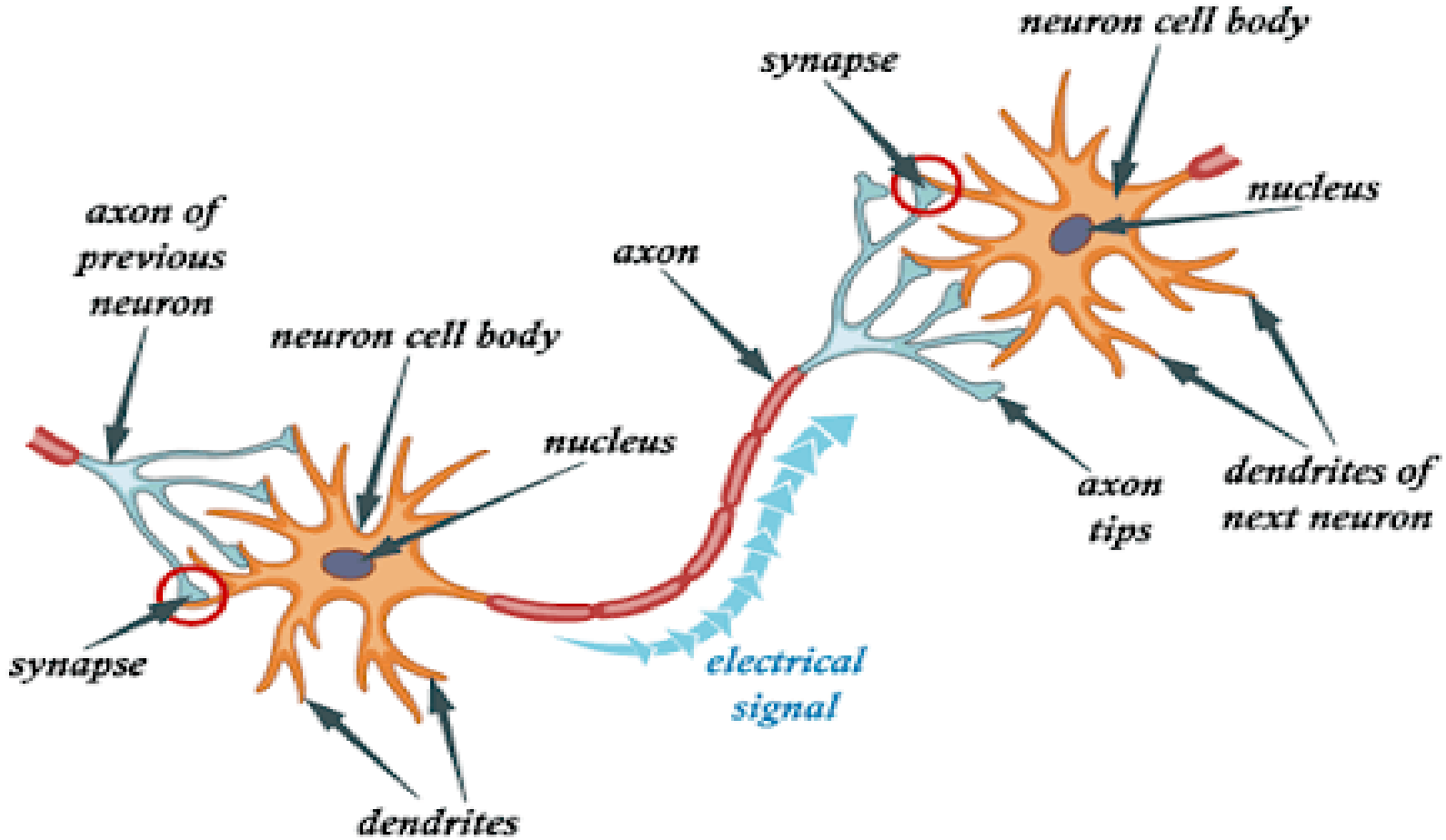
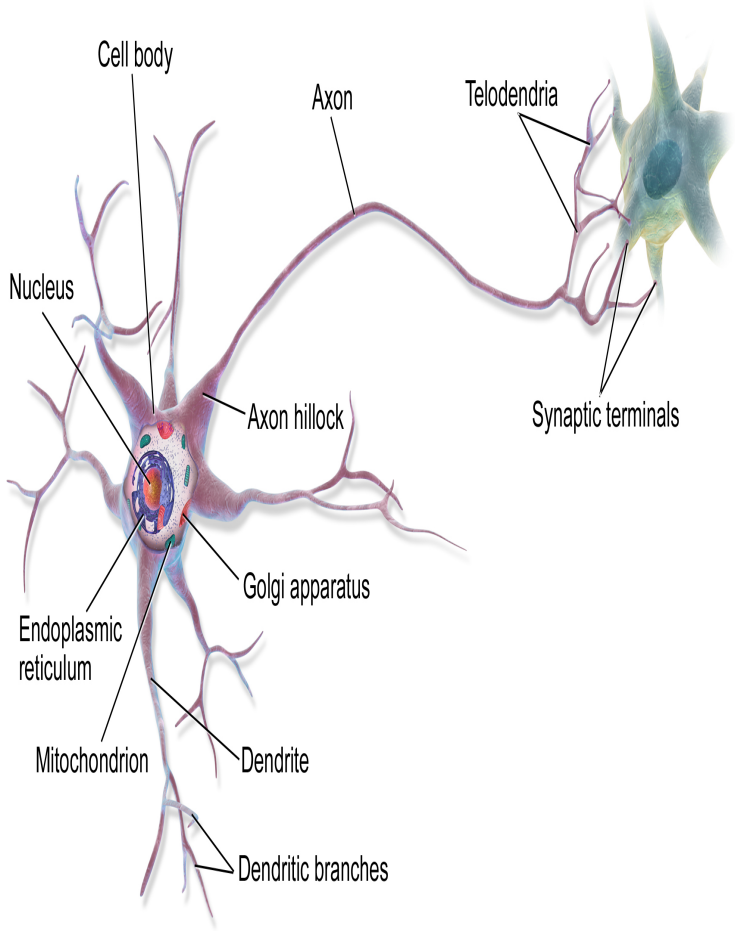


Figure: A model of a biological neural network

Anatomy of Biological Neuron-Neuroanatomy

The neuron consists of three important parts:

- Cell body: Directs all activities of the neuron.
- Dendrites: Short fibers that receive messages from other neurons and relay those messages to the cell body.
- Axon: A long single fiber that transmits messages from the cell body to dendrites of other neurons.



McCulloch-Pitts Neuron Model

November 10, 2025

Neuron

BULLETIN OF
MATHEMATICAL BIOPHYSICS
VOLUME 5, 1943

A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY

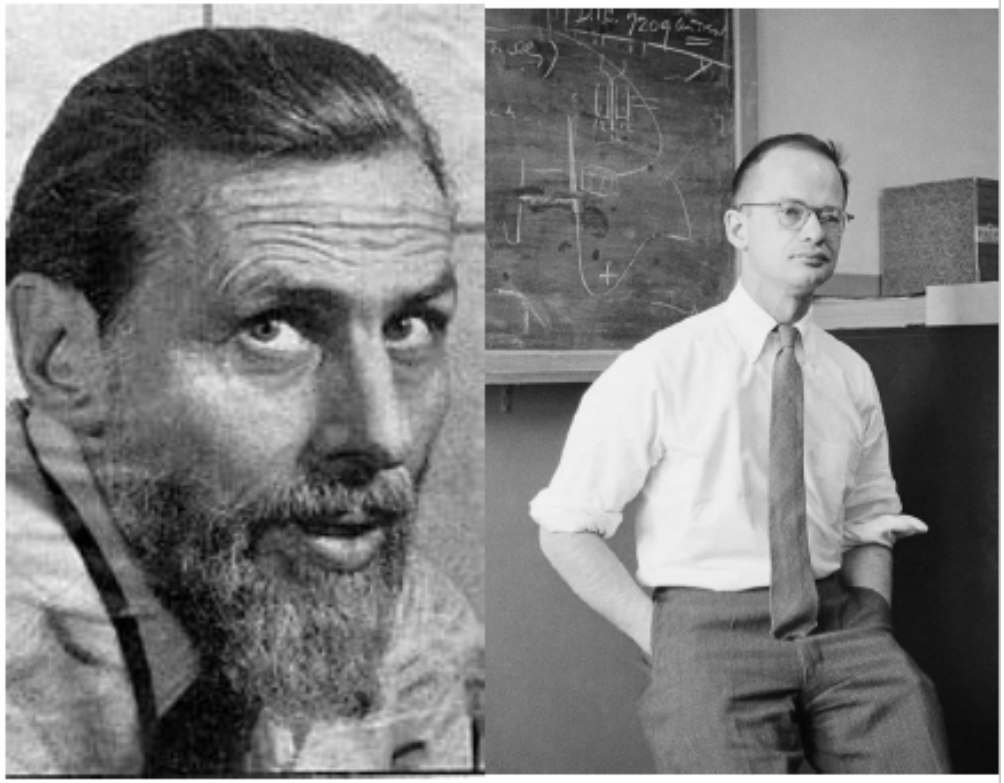
WARREN S. MCCULLOCH AND WALTER PITTS

FROM THE UNIVERSITY OF ILLINOIS, COLLEGE OF MEDICINE,
DEPARTMENT OF PSYCHIATRY AT THE ILLINOIS NEUROPSYCHIATRIC INSTITUTE,
AND THE UNIVERSITY OF CHICAGO

Because of the "all-or-none" character of nervous activity, neural events and the relations among them can be treated by means of propositional logic. It is found that the behavior of every net can be described in these terms, with the addition of more complicated logical means for nets containing circles; and that for any logical expression satisfying certain conditions, one can find a net behaving in the fashion it describes. It is shown that many particular choices among possible neurophysiological assumptions are equivalent, in the sense that for every net behaving under one assumption, there exists another net which behaves under the other and gives the same results, although perhaps not in the same time. Various applications of the calculus are discussed.

I. Introduction

Theoretical neurophysiology rests on certain cardinal assumptions. The nervous system is a net of neurons, each having a soma and an axon. Their adjunctions, or synapses, are always between the axon of one neuron and the soma of another. At any instant a neuron has some threshold, which excitation must exceed to initiate an impulse. This, except for the fact and the time of its occurrence, is determined by the neuron, not by the excitation. From the point of excitation the impulse is propagated to all parts of the neuron. The velocity along the axon varies directly with its diameter, from less than one meter per second in thin axons, which are usually short, to more than 150 meters per second in thick axons, which are usually long. The time for axonal conduction is consequently of little importance in determining the time of arrival of impulses at points unequally remote from the same source. Excitation across synapses occurs predominantly from axonal terminations to somata. It is still a moot point whether this depends upon irreciprocity of individual synapses or merely upon prevalent anatomical configurations. To suppose the latter requires no hypothesis *ad hoc* and explains known exceptions, but any assumption as to cause is compatible with the calculus to come. No case is known in which excitation through a single synapse has elicited a nervous impulse in any neuron, whereas any



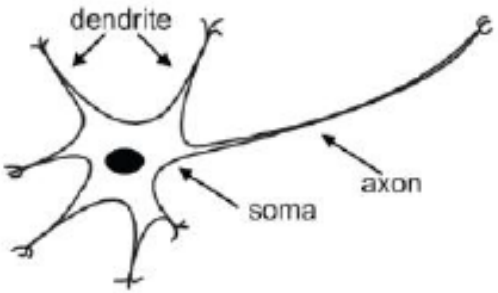
Warren McCulloch Walter Pitts

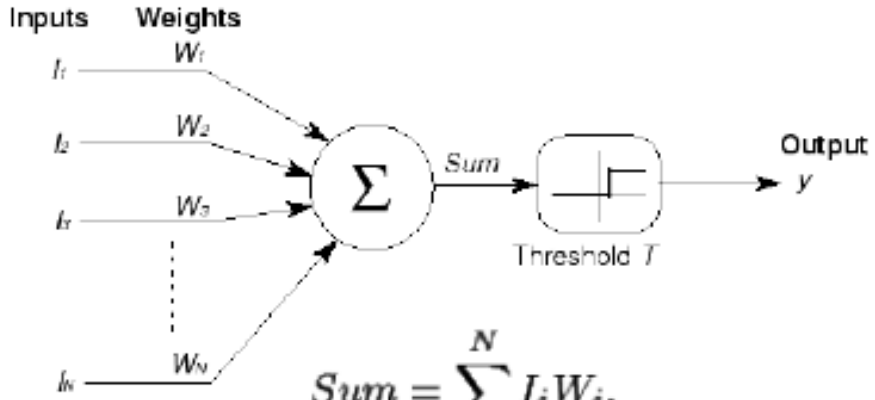
1943, "A Logical Calculus of the Ideas Immanent in Nervous Activity". With Walter Pitts. In: *Bulletin of Mathematical Biophysics* Vol 5, pp 115–133.

Neuron

The McCulloch-Pitts Model of Neuron (1942 model)

The early model of an artificial neuron is introduced by Warren McCulloch and Walter Pitts in 1943.





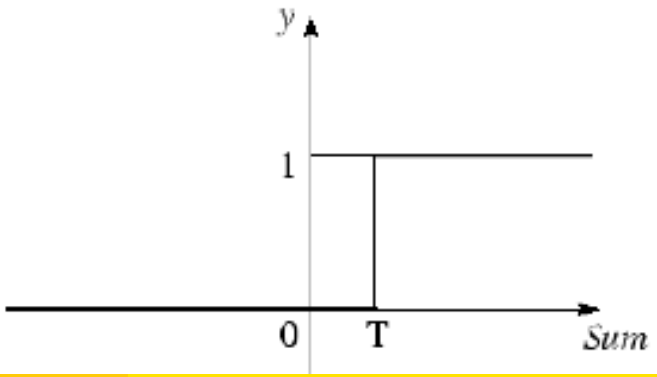
$$Sum = \sum_{i=1}^{N} I_i W_i,$$

The main feature of their neuron model is that a weighted sum of input signals is compared to a threshold to determine the neuron output.

$$y = f(Sum).$$

When the sum is greater than or equal to the threshold, the output is 1.

When the sum is less than the threshold, the output is 0.



McCulloch-Pitts Neuron Model

- They demonstrated that networks of these neurons could, in principle, compute any arithmetic or logical function.
- Unlike biological networks, the parameters of their networks had to be designed, as no training method was available.
- However, the perceived connection between biology and digital computers generated a great deal of interest in the McCulloch-Pitts model.

Table: Truth Table for the OR Logic Gate

Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	1

Table: Truth Table for the AND Logic Gate

Input A	Input B	Output
0	0	0
0	1	0
1	0	0
1	1	1

Implementation of basic logic gates with McCulloch Pitts Model

In each case, we have inputs in_i and outputs out and need to determine the appropriate weights and thresholds. It is easy to find solutions by inspection:

NOT

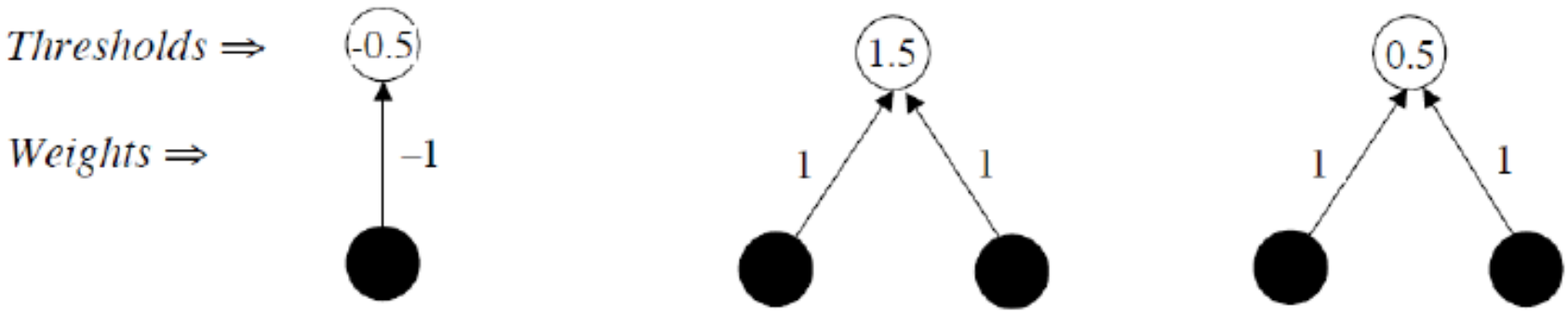
in	out
0	1
1	0

AND

in_1	in_2	out
0	0	0
0	1	0
1	0	0
1	1	1

OR

in_1	in_2	out
0	0	0
0	1	1
1	0	1
1	1	1



McCulloch-Pitts Neural Model

The McCulloch-Pitts neural model, depicted in the following Figure, is one of the earliest Artificial Neural Network (ANN) models.

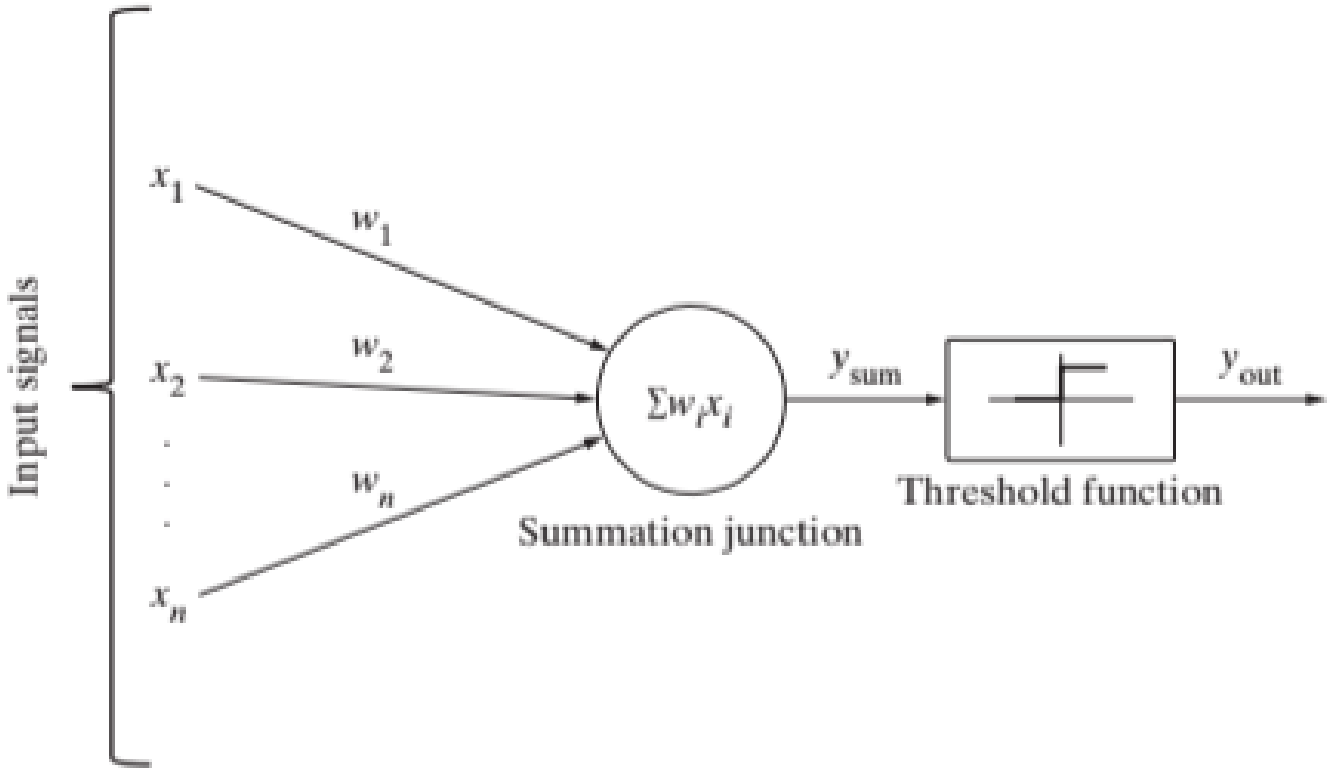


Figure: McCulloch-Pitts Neural Model

McCulloch-Pitts Neural Model

Key Features:

- Only two types of inputs: excitatory and inhibitory.
- Excitatory inputs have positive weights, while inhibitory inputs have negative weights.
- Inputs can be either 0 or 1.
- Activation function: Threshold function.
- Output signal (y_{out}) is 1 if the input sum (y_{sum}) is greater than or equal to a given threshold value, else 0.
- McCulloch-Pitts neurons can be used to design logical operations by correctly deciding the connection weights and threshold function.

Analysis using McCulloch-Pitts Neural Model

Given Situations:

1. Situation 1 – It is not raining nor is it sunny.
2. Situation 2 – It is not raining, but it is sunny.
3. Situation 3 – It is raining, and it is not sunny.
4. Situation 4 – Wow, it is so strange! It is raining as well as it is sunny.

To analyze these situations using the McCulloch-Pitts neural model, we can consider the input signals as follows:

- x_1: Is it raining? (0 = No, 1 = Yes)
- x_2: Is it sunny? (0 = No, 1 = Yes)

John's Umbrella Decision:

- John carries an umbrella if it is sunny or if it is raining.
- We can use the McCulloch-Pitts neural model to determine when John will carry the umbrella.

McCulloch-Pitts Neural Model

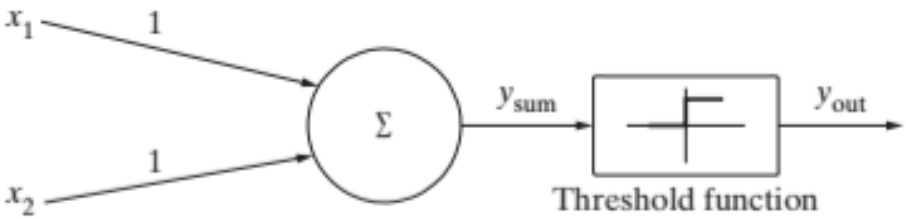
Situation	x_1	x_2	y_{sum}	y_{out}
1	0	0	0	0
2	0	1	1	1
3	1	0	1	1
4	1	1	2	1

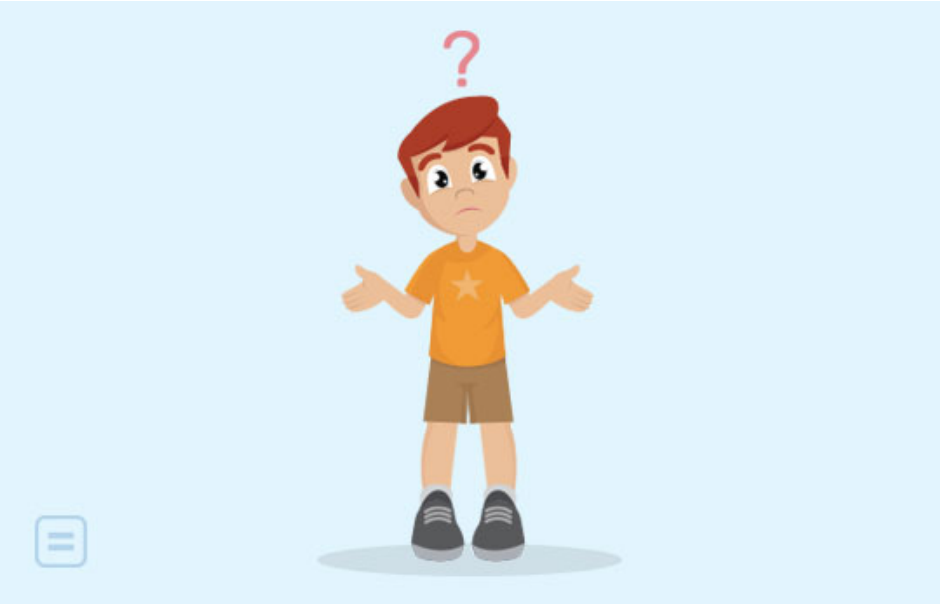
The McCulloch-Pitts neural model calculates the values as follows:

- $y_{\text{sum}} = w_1 \cdot x_1 + w_2 \cdot x_2$
- $y_{\text{out}} = \begin{cases} 1 & \text{if } y_{\text{sum}} \geq \text{threshold} \\ 0 & \text{otherwise} \end{cases}$

From the truth table, we can conclude that John needs to carry an umbrella in situations 2, 3, and 4. These situations correspond to when y_{out} is 1.The McCulloch-Pitts neural model successfully implements the logical OR function for determining John's umbrella decision.

John's Dilemma: To Umbrella or Not to Umbrella? Can You Help Him Out?





Question: What is the decision made by John regarding carrying an umbrella, based on the McCulloch-Pitts neural model and the given situations where x_1 represents raining (0 = No, 1 = Yes) and x_2 represents sunny (0 = No, 1 = Yes)?

Answer: Given:

- Threshold value: 1
- Weights: $w_1 = w_2 = 1$

Perceptron

The structural building block of deep learning

November 10, 2025

Perceptron

Psychological Review
Vol. 65, No. 6, 1958

THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN[1]

F. ROSENBLATT

Cornell Aeronautical Laboratory

If we are eventually to understand the capability of higher organisms for perceptual recognition, generalization, recall, and thinking, we must first have answers to three fundamental questions:

1. How is information about the physical world sensed, or detected, by the biological system?
2. In what form is information stored, or remembered?
3. How does information contained in storage, or in memory, influence recognition and behavior?

The first of these questions is in the province of sensory physiology, and is the only one for which appreciable understanding has been achieved. This article will be concerned primarily with the second and third questions, which are still subject to a vast amount of speculation, and where the few relevant facts currently supplied by neurophysiology have not yet been integrated into an acceptable theory.

With regard to the second question, two alternative positions have been maintained. The first suggests that storage of sensory information is in the form of coded representations or images, with some sort of one-to-one mapping between the sensory stimulus and the stored pattern. According to this hypothesis, if one understood the code or "wiring diagram" of the nervous system, one should, in principle, be able to discover exactly what an organism remembers by reconstructing the original sensory patterns from the "memory traces" which they have left, much as we might develop a photographic negative, or translate the pattern of electrical charges in the "memory" of a digital computer. This hypothesis is appealing in its simplicity and ready intelligibility, and a large family of theoretical brain models has been developed around the idea of a coded, representational memory (2, 3, 9, 14). The alternative approach, which stems from the tradition of British empiricism, hazards the guess that the images of stimuli may never really be recorded at all, and that the central nervous system simply acts as an intricate switching network, where retention takes the form of new connections, or pathways, between centers of activity. In many of the more recent developments of this position (Hebb's "cell assembly," and Hull's "cortical anticipatory goal response," for example) the "responses" which are associated to stimuli may be entirely contained within the CNS itself. In this case the response represents an "idea" rather than an action. The important feature of this approach is that there is never any simple mapping of the stimulus into memory, according to some code which would permit its later reconstruction. Whatever in-

[1] The development of this theory has been carried out at the Cornell Aeronautical Laboratory, Inc., under the sponsorship of the Office of Naval Research, Contract Nonr-2381(00). This article is primarily an adaptation of material reported in Ref. 15, which constitutes the first full report on the program.

Frank Rosenblatt

Rosenblatt, Frank (1958), The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain, Cornell Aeronautical Laboratory, Psychological Review, v65, No. 6, pp. 386–408.

Frank Rosenblatt's Perceptron (1958)

Frank Rosenblatt's Perceptron

"The model that sparked the neural revolution"

Frank Rosenblatt — Psychologist, Computer Scientist
Cornell Aeronautical Laboratory, 1958

What is the Perceptron?

Definition

A simple linear binary classifier that makes decisions by computing a weighted sum of inputs and applying a step function.

- Inspired by biological neurons
- Assigns weights to input features
- Produces binary output: **0 or 1**

Used to answer: "Does this input belong to Class A or B?"

How the Perceptron Works

Weighted Sum

$$z = \sum_{i=1}^{n} w_i x_i + b$$

Activation (Step Function)

$$y = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$

- **Weights (w_i):** importance of each input
- **Bias (b):** shifts the decision boundary
- **Output (y):** binary classification result

Perceptron Learning Algorithm

Weight Update Rule

$$w_i \leftarrow w_i + \eta(y_{\text{true}} - y_{\text{pred}})x_i$$

Training Steps:

1. Make a prediction
2. Calculate the error
3. Adjust weights if the prediction is incorrect

Only updates weights on error!

Limitations of the Perceptron

- **Linearly separable only** — cannot solve XOR
- **Single-layer only** — lacks hidden layers
- **No non-linearity** — limited real-world performance

The AI Winter

The failure to model non-linear problems led to a decline in neural network research in the 1970s.

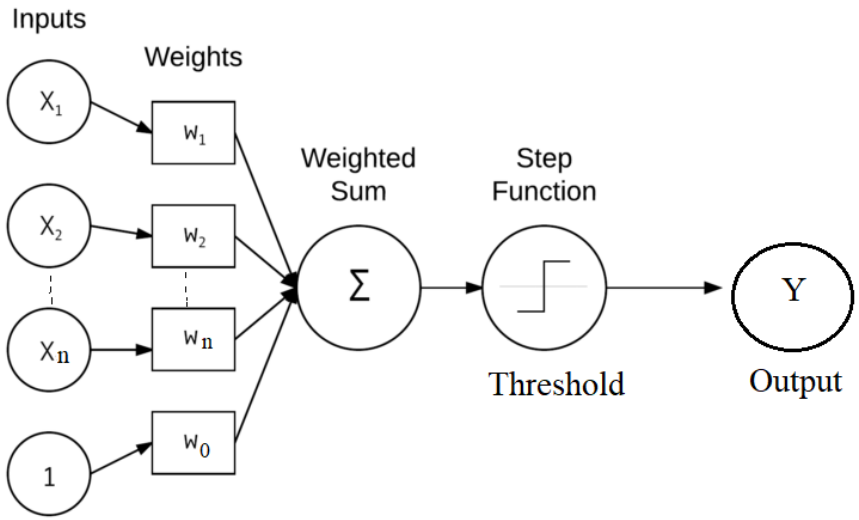
Legacy of Rosenblatt's Perceptron

Despite limitations...

- **Inspired MLPs and backpropagation**
- **Laid the foundation for modern deep learning**
- **Still the core of neural networks today**

The spark lives on!

Perceptron



Neurons in these networks were similar to those of McCulloch and Pitts

- Rosenblatt's key contribution was the introduction of a learning rule for training perceptron networks to solve pattern recognition problems.
- In addition to the variable weight values, the perceptron model added an extra input that represents bias. Thus, the modified equation is now as follows:
  $sum = \sum_{i=1}^{n} X_i W_i + W_0$
- Bias is a measure of how easy it is to get the perceptron to output 1.

Perceptron Learning Rule

- The weight update equation in the Perceptron learning rule is given by:

$$W_{i+1} = W_i + \Delta W$$

where $\Delta W = \eta \cdot E_{\text{error}} \cdot X$.

- The bias term b is updated separately using the equation:

$$b_{i+1} = b + \eta \cdot E_{\text{error}}$$

where η is the learning rate.

- The error term E_{error} is calculated as the difference between the target output, T, and the actual output, Y:

$$E_{\text{error}} = T - Y$$

Perceptrons

A perceptron is a type of artificial neuron that is used in artificial neural networks. It is a simple model of a biological neuron, and it can be used to perform simple tasks such as classification and regression.
A perceptron has three main components:

- Input: The input to a perceptron is a vector of numbers.
- Weights: The weights are a set of numbers that are used to determine how the input is processed.
- Bias: The bias is a number that is added to the weighted sum of the inputs.

The output of a perceptron is calculated using the following formula:

output=activation_function(weighted_sum+bias)

The activation function is a mathematical function that determines how the output of the perceptron is interpreted.

Note: The activation function used in a perceptron is the step function. The step function is a simple function that outputs 1 if the input is greater than or equal to 0, and 0 otherwise. This means that the perceptron can only output two possible values: 0 or 1.

Neuron's Output

The neuron's output, 0 or 1, is determined by whether the weighted sum $\sum_j w_j x_j$ is less than or greater than some threshold value.
Just like the weights, the threshold is a real number which is a parameter of the neuron.
To put it in more precise algebraic terms:

$$\text{output} = \begin{cases} 0, & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1, & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

Applications of Perceptrons

Perceptrons can be used to solve a variety of problems, such as:

- Classification: Given a set of input data, classify it into one of two or more categories.
- Regression: Given a set of input data, predict a continuous value.
- Pattern recognition: Identify patterns in data.

Perceptrons are a simple but powerful tool that can be used to solve a variety of problems. They are the building blocks of artificial neural networks, which are capable of solving even more complex problems.

Perceptron Learning Rule

1. Perceptron Learning Rule

The perceptron learning rule is a simple algorithm used to train a binary classifier, also known as a perceptron. It involves the following steps:

1. Initialize the weights and bias to small random values.
2. For each training example (input vector) and its corresponding target output:
   - Compute the predicted output of the perceptron by taking the dot product of the input vector and the weights, and adding the bias.
   - Apply the activation function (usually a step function) to the predicted output to obtain the actual output.
   - Calculate the error as the difference between the target output and the actual output.
   - Update the weights and bias using the following update rule:
     - For each weight, update it by adding the product of the learning rate (a small positive constant) and the error multiplied by the corresponding input value.
     - Update the bias by adding the product of the learning rate and the error.

Equations

2. Equations (Part 1)

The equations related to the perceptron learning rule are as follows:

1. **Prediction:**

$$\text{predicted\_output} = \text{activation\_function}(X \cdot W + b)$$

   - X: Input vector of size $(n \times 1)$, where n is the number of features.
   - W: Weight vector of size $(n \times 1)$, representing the weights for each feature.
   - b: Bias, a scalar value.
   - activation_function(): Activation function, which determines the interpretation of the output.

2. **Error Calculation:**

$$\text{error} = Y - \text{predicted\_output}$$

   - Y: Target output.

Equations (Part 2)

2. Equations (Part 2)

The equations related to the perceptron learning rule (continued) are as follows:

3. **Weight Update:**

$$\text{new\_W} = W + \text{learning\_rate} \times \text{error} \times X$$

$$\text{new\_b} = b + \text{learning\_rate} \times \text{error}$$

- learning_rate: Small positive constant that controls the step size of the weight update.

Perceptron Architecture (Cont'd)

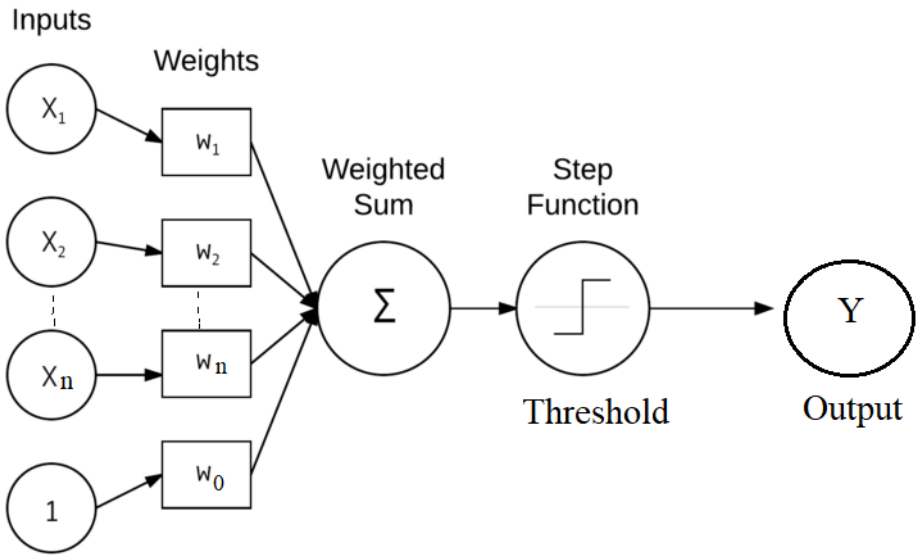


Figure: Perceptron architecture with N input nodes and a single output node.

There exist connections and their corresponding weights w_1, w_2, ..., w_i from the input x_i's to the single output node in the network. This node takes the weighted sum of inputs and applies a step function to determine the output class label.

Perceptron Architecture (Cont'd)

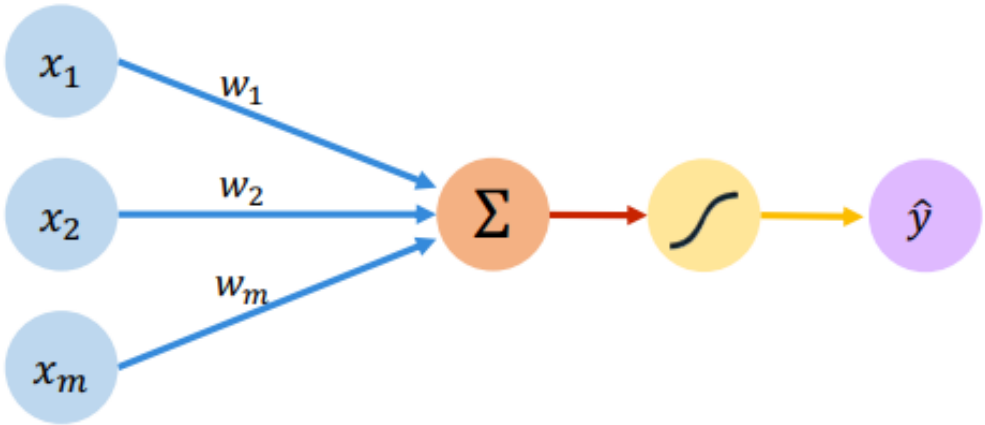
The Perceptron applies the following steps to compute the output:

1. Multiply each input x_i by its corresponding weight w_i.
2. Sum up the weighted inputs.
3. Apply an activation function, often a step function, to the weighted sum.
4. The output is determined by the activation function's result. In the case of a step function, the Perceptron outputs either 0 or 1 – 0 for class 1 and 1 for class 2.

In its original form, the Perceptron is simply a binary, two-class classifier. However, variations and extensions have been developed to handle multi-class classification and other complex tasks.

The Perceptron

The Perceptron: Forward Propagation



Linear combination of inputs

Output

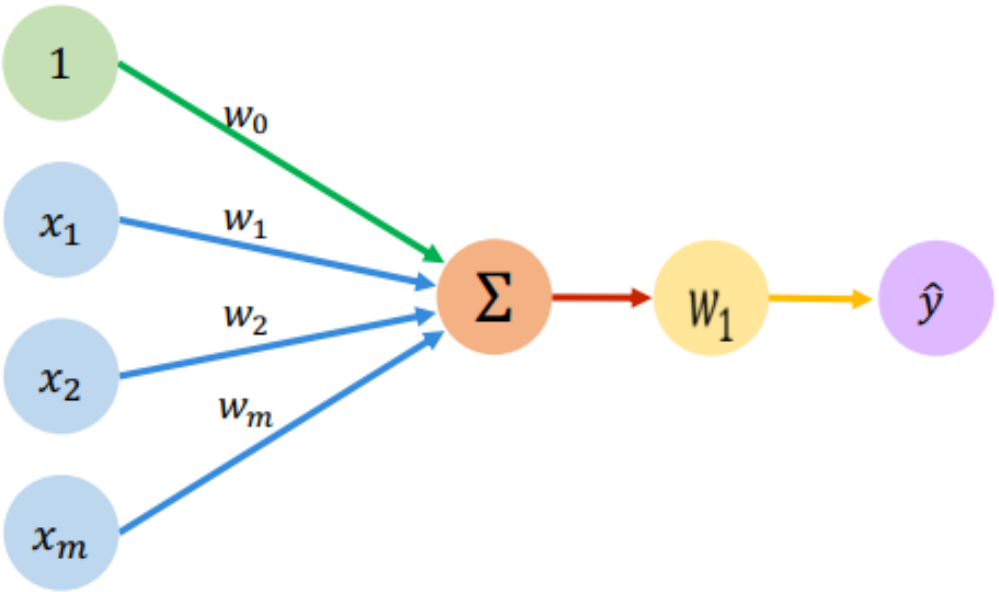
$$\hat{y} = g\left(\sum_{i=1}^{m} x_i \; w_i\right)$$

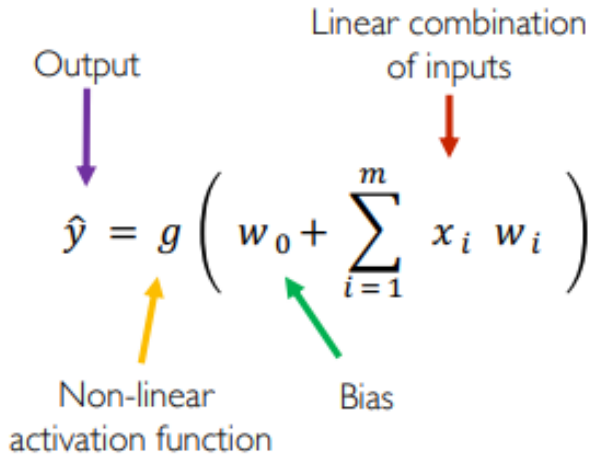
Non-linear activation function

Inputs Weights Sum Non-Linearity Output

The Perceptron

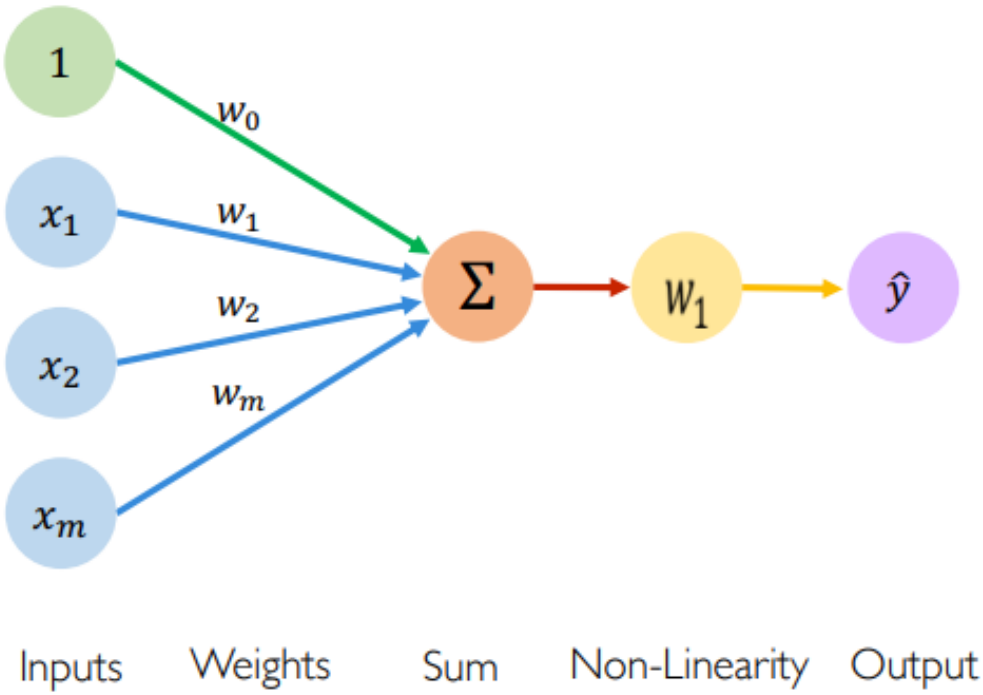
The Perceptron: Forward Propagation





The Perceptron

The Perceptron: Forward Propagation



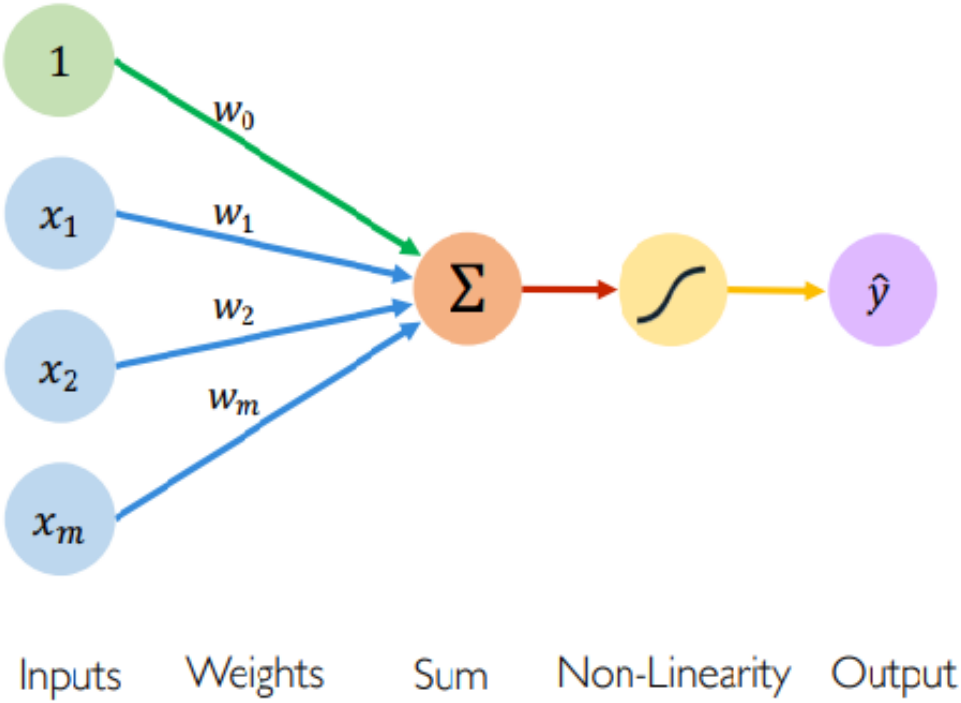
$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i\, w_i\right)$$

$$\hat{y} = g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W}\right)$$

where: $\boldsymbol{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ and $\boldsymbol{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$

The Perceptron

The Perceptron: Forward Propagation

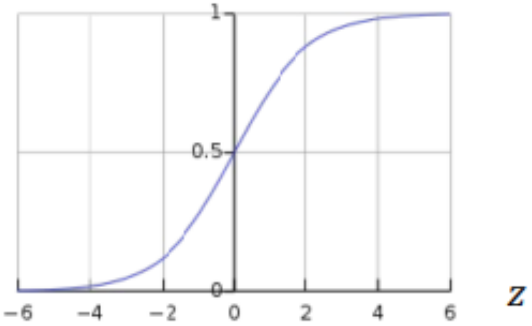


Activation Functions

$$\hat{y} = g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W} \right)$$

- Example: sigmoid function

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



Perceptron as classifier

Problem Description:

Consider a perceptron with two input features, x_1 and x_2, and a threshold activation function. The weights and bias for the perceptron are as follows:

$$w_1 = 0.6$$
$$w_2 = 0.6$$
$$\text{bias} = -1$$

The perceptron is used to simulate an AND gate, where the output should be 1 only when both inputs are 1; otherwise, the output should be 0.

Perceptron as classifier (Cont'd)

Input and Output Calculation:

Given:

$$\text{Input 1 } (x_1) = 0$$
$$\text{Input 2 } (x_2) = 1$$

The output of the perceptron can be computed as:

$$\text{output} = \text{activation\_function}(w_1 \cdot x_1 + w_2 \cdot x_2 + \text{bias})$$
$$\text{output} = \text{activation\_function}(0.6 \cdot 0 + 0.6 \cdot 1 - 1)$$

Here, the activation function is a step function that returns 1 if the input is greater than or equal to 0, and 0 otherwise.

Calculating the weighted sum and applying the activation function:

$$\text{output} = \text{activation\_function}(-0.4)$$

Since the weighted sum is negative, the step function will return 0.

Therefore, the output of the perceptron for the given inputs is 0.

Perceptron Classification Points

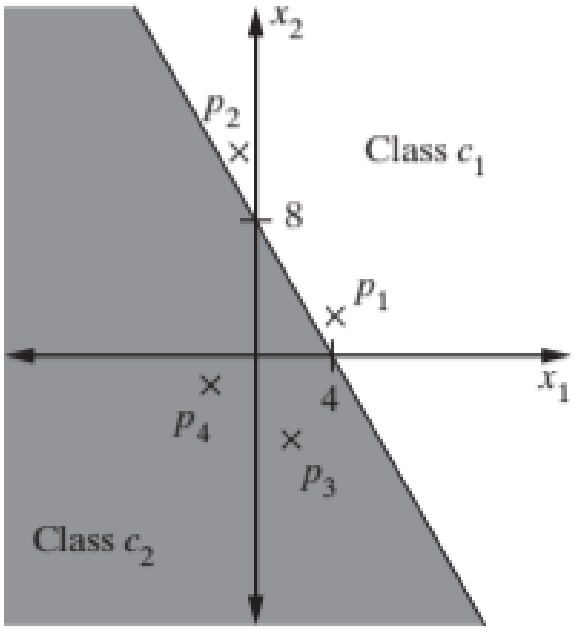
Let us examine if this perceptron is able to classify a set of points given below:

- $p1 = (5, 2)$ and $p2 = (-1, 12)$
- $p3 = (3, -5)$ and $p4 = (-2, -1)$

Assuming the perceptron has the following weights: $w_0 = -2$, $w_1 = \frac{1}{2}$, and $w_2 = \frac{1}{4}$.

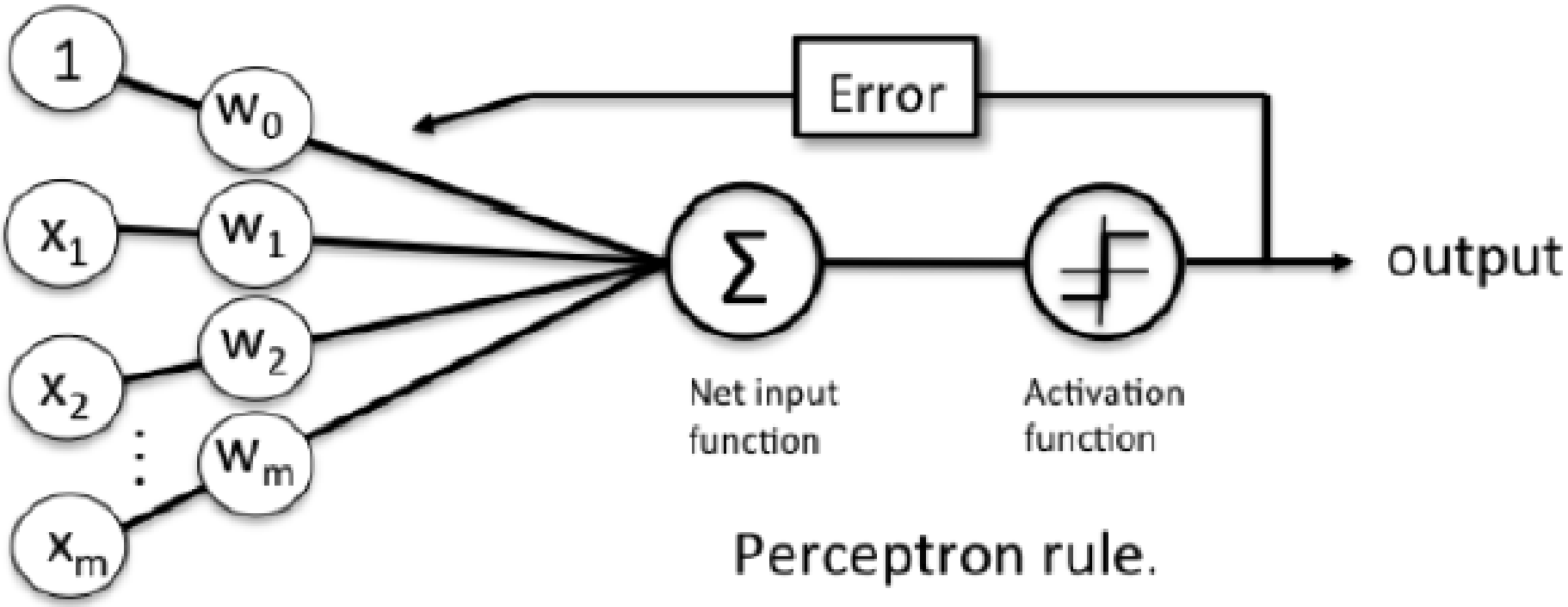
Perceptron Classification Results

Point	$v = \sum w_i x_i$	$y_{out} = f(v)$	Class
$p1$	$-2 + (1/2) * 5 + (1/4) * 2 = 1$	1	$c1$
$p2$	$-2 + (1/2) * (-1) + (1/4) * 12 = 0.5$	1	$c1$
$p3$	$-2 + (1/2) * 3 + (1/4) * (-5) = -1.75$	0	$c2$
$p4$	$-2 + (1/2) * (-2) + (1/4) * (-1) = -3.25$	0	$c2$



Classfication by decision boundary

Perceptron learning rule



Perceptron rule.

Limitations of Perceptrons

- One of the key assumptions for a perceptron to work properly is that the two classes should be *linearly separable*, i.e. the classes should be sufficiently separated from each other.
- If the classes are *non-linearly separable*, then the classification problem cannot be solved by a perceptron.

Multi-Layer Perceptron (MLP)

Multi-Layer Perceptron

The backbone of modern deep learning

Extends the single-layer perceptron with depth and power

What is a Multi-Layer Perceptron (MLP)?

Definition

An **MLP is a feedforward neural network** composed of input, one or more hidden layers, and an output layer.

- Each layer is made up of **neurons (nodes)**
- Each neuron applies: **weighted sum + non-linear activation**
- Enables modeling **non-linear** and complex decision boundaries

MLP Architecture

- **Input Layer:** Features $x_1, x_2, ..., x_n$
- **Hidden Layers:** Learn intermediate representations
- **Output Layer:** Final prediction

Each layer uses activation functions like ReLU, Sigmoid, Tanh

mlp_architecture.png

Forward Propagation

Data flows through layers as:

Layer Computation

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

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

- $W^{(l)}$: weights for layer l
- $b^{(l)}$: bias vector for layer l
- g: non-linear activation function

Backpropagation

Gradient Calculation

Backpropagation uses the chain rule to compute gradients of the loss with respect to weights:

$$\frac{\partial L}{\partial W^{(l)}} = \delta^{(l)} (a^{(l-1)})^T$$

- Updates weights using gradient descent
- Allows multi-layer networks to learn from errors

Activation Functions

- **Sigmoid:** $\sigma(x) = \frac{1}{1+e^{-x}}$
- **Tanh:** $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
- **ReLU:** $f(x) = \max(0, x)$

Choice of activation affects convergence, stability, and expressiveness.

Why MLPs Matter

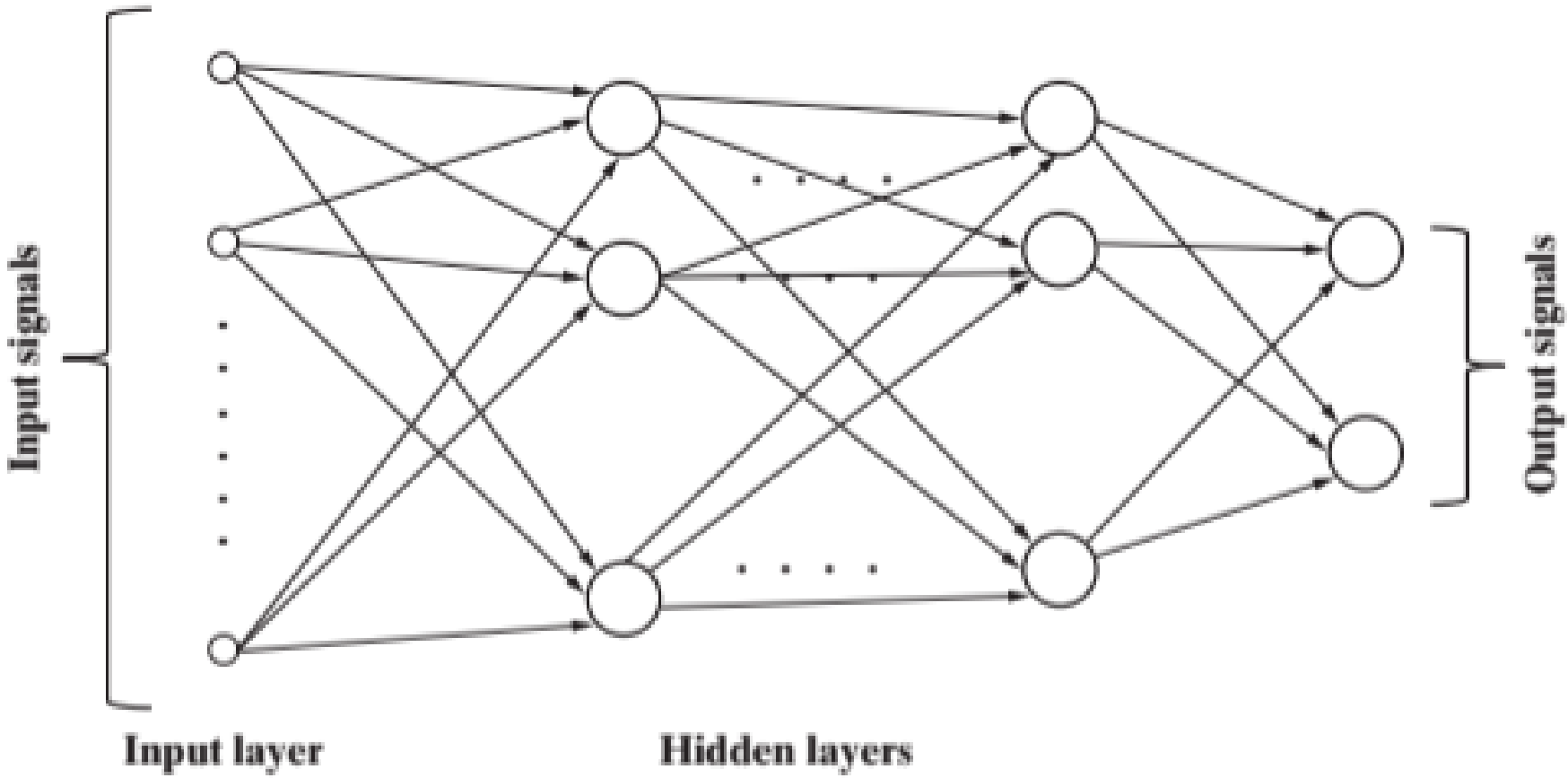
Powerful Universal Approximators

MLPs can approximate any continuous function with enough hidden units.

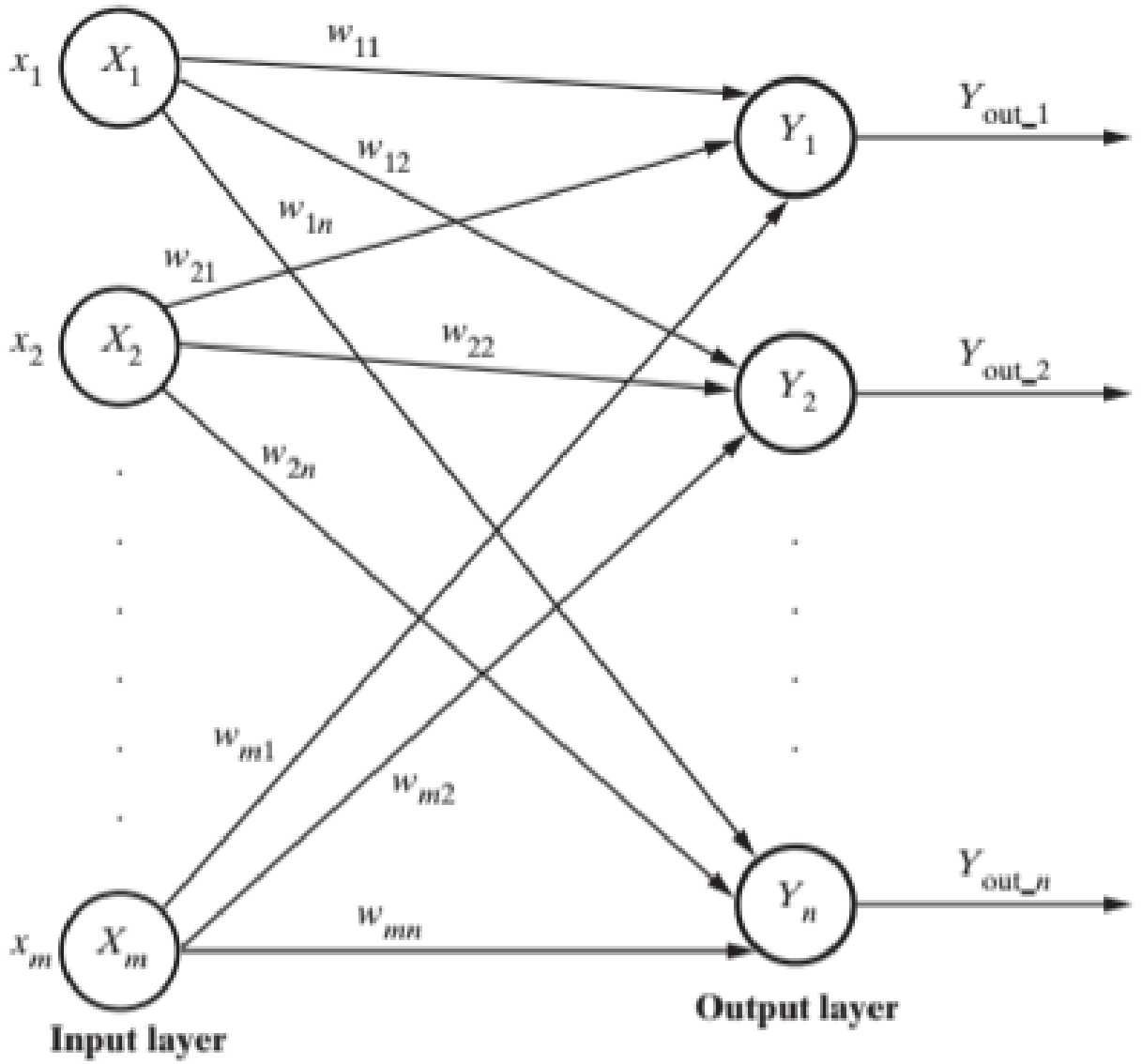
- Form the foundation of most deep learning models
- Adaptable to classification, regression, language, vision tasks
- Core of many architectures before CNNs, RNNs, Transformers

MLPs brought learning to the layers!

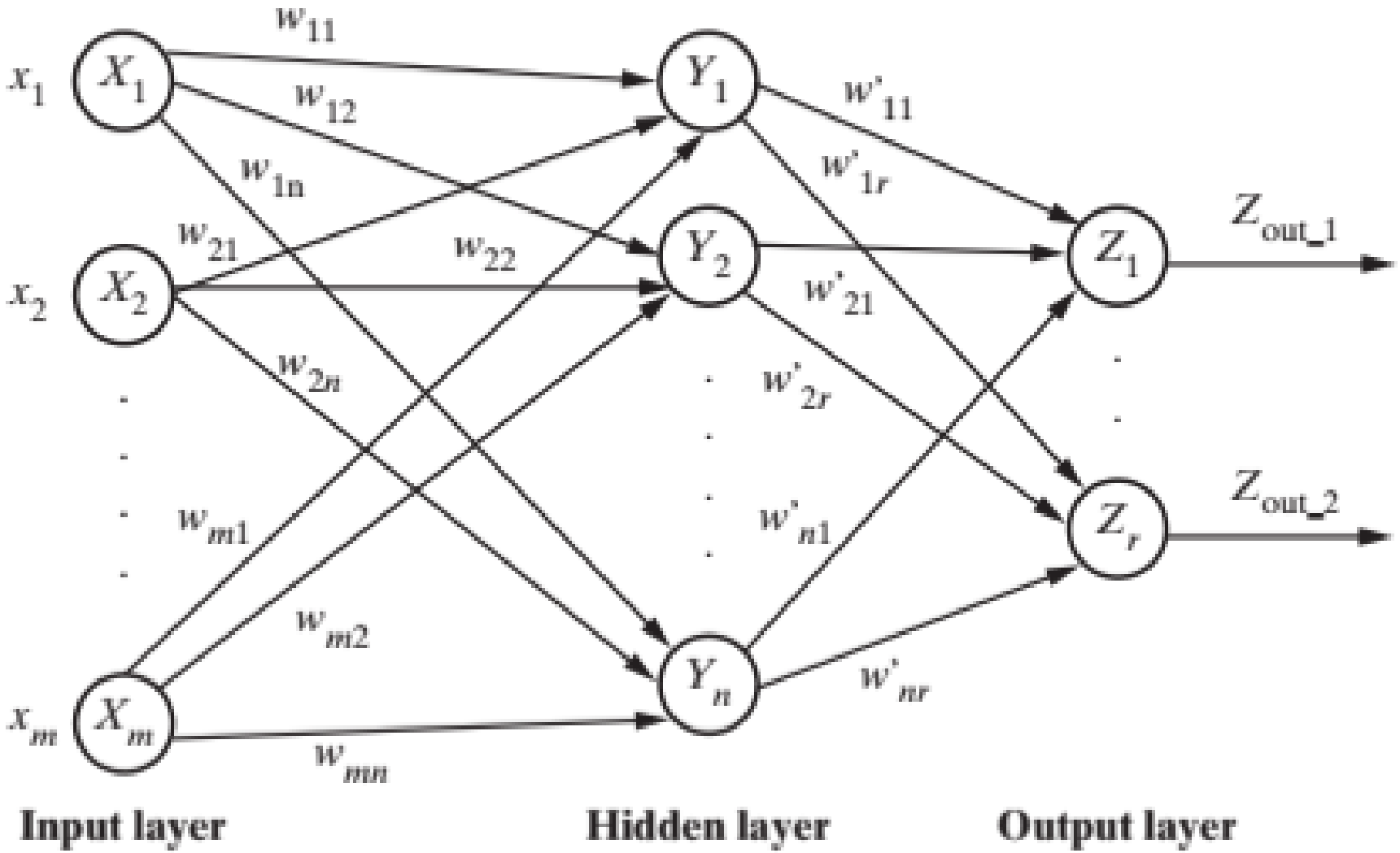
Multilayer Perceptron



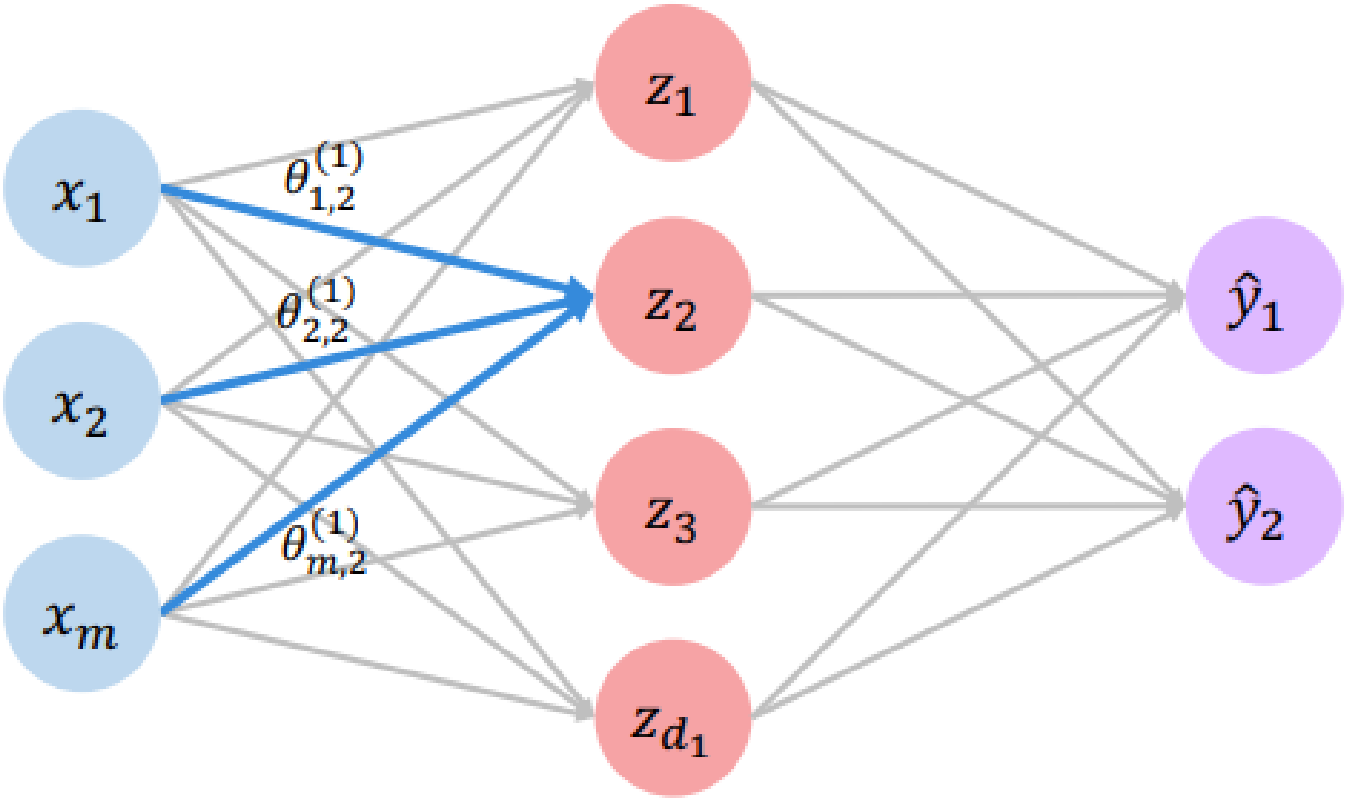
single-layer perceptron



Multilayer Perceptrons



Multilayer Perceptrons

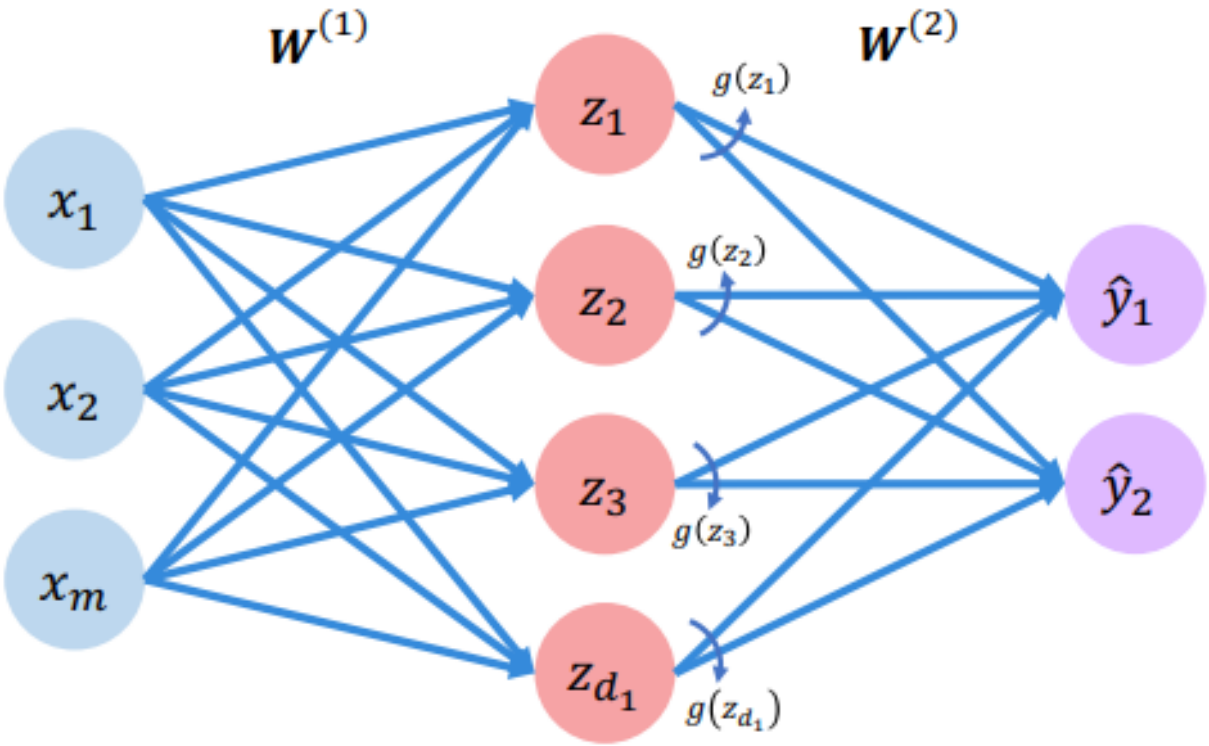


$$z_2 = w_{0,2}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,2}^{(1)}$$

$$= w_{0,2}^{(1)} + x_1 \, w_{1,2}^{(1)} + x_2 \, w_{2,2}^{(1)} + x_m \, w_{m,2}^{(1)}$$

Multilayer

Multilayer Perceptrons



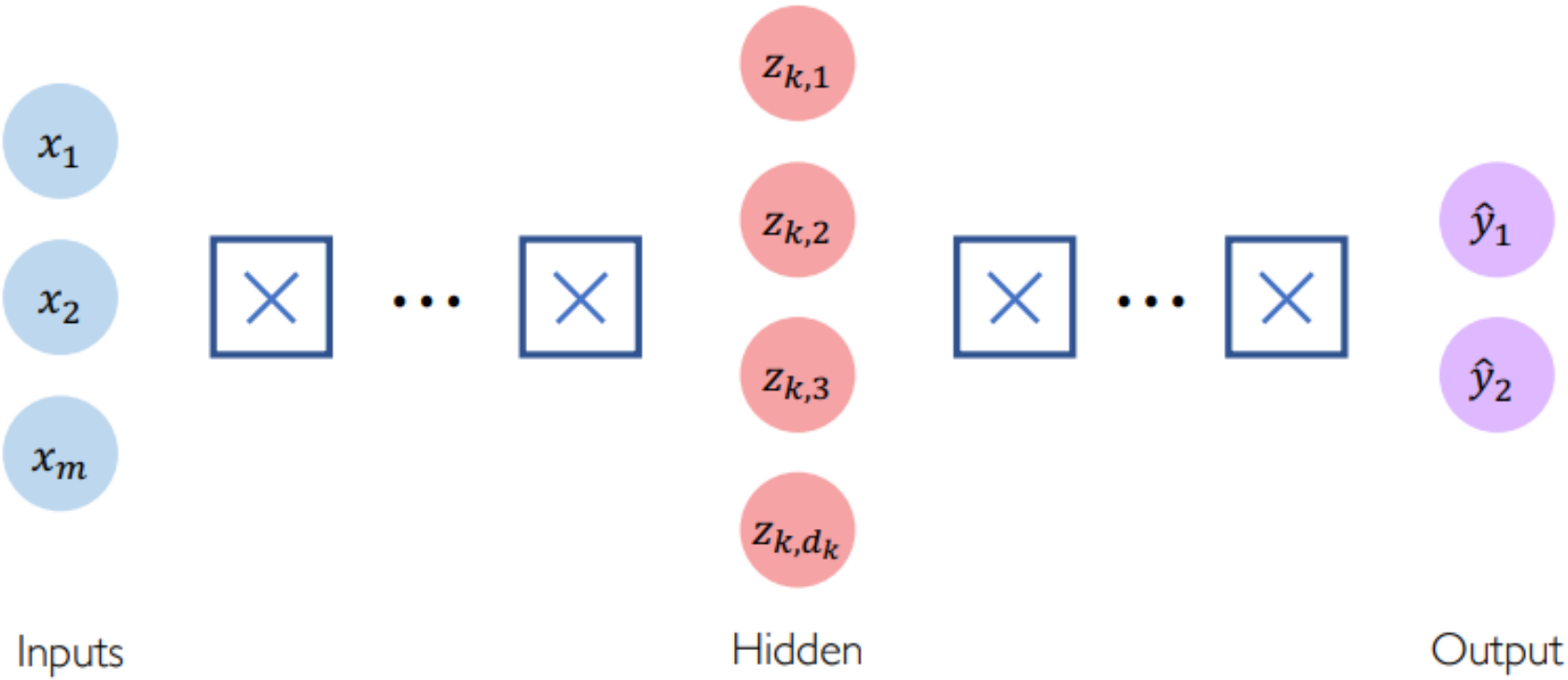
Inputs Hidden Final Output

$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j \, w_{j,i}^{(2)} \right)$$

Multilayer

Multilayer Perceptrons

Deep Neural Network



$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{d_{k-1}} g(z_{k-1,j})\, w_{j,i}^{(k)}$$

Applying Neural Networks: A Comprehensive Guide

The structural building block of deep learning

November 10, 2025

Problem Statement

Will I pass this class?

Let's start with a simple two feature model:

- x_1 = Number of lectures you attend
- x_2 = Hours spent on the final project

Example problem-Will I pass this class?

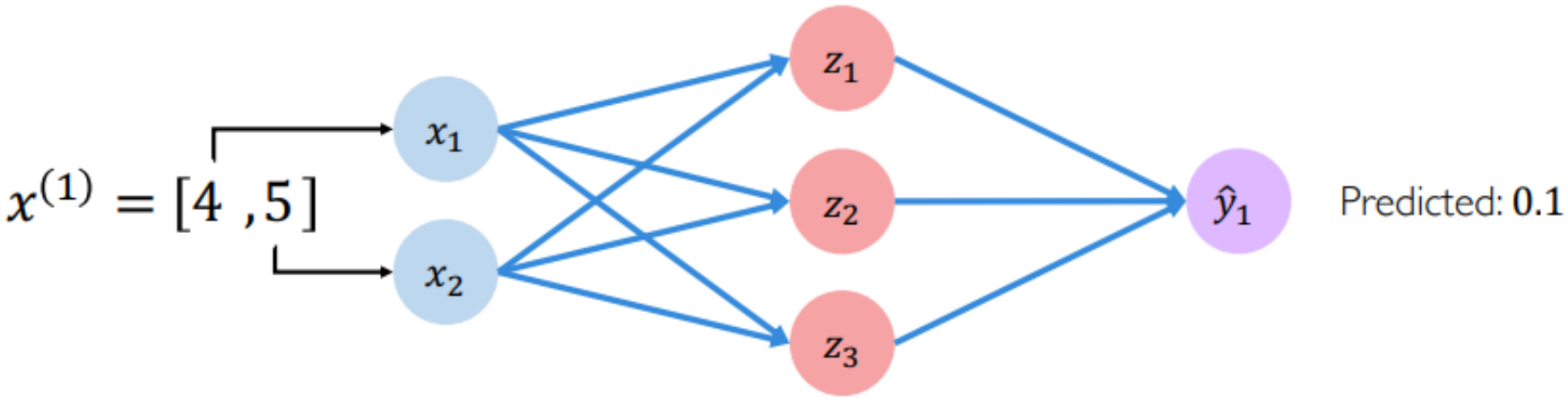


Figure: Example problem

Example problem-Will I pass this class?

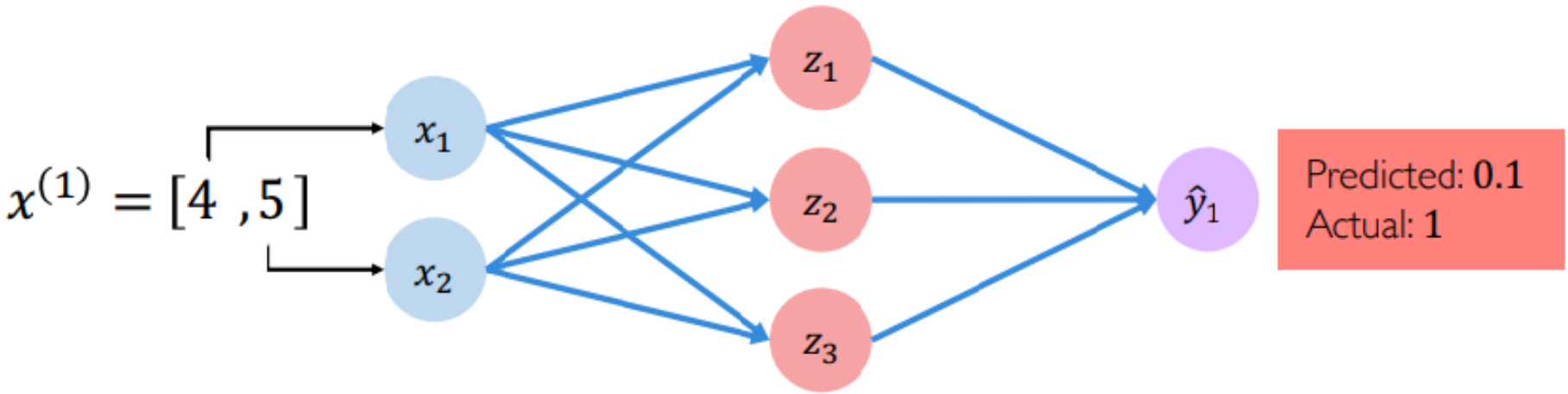
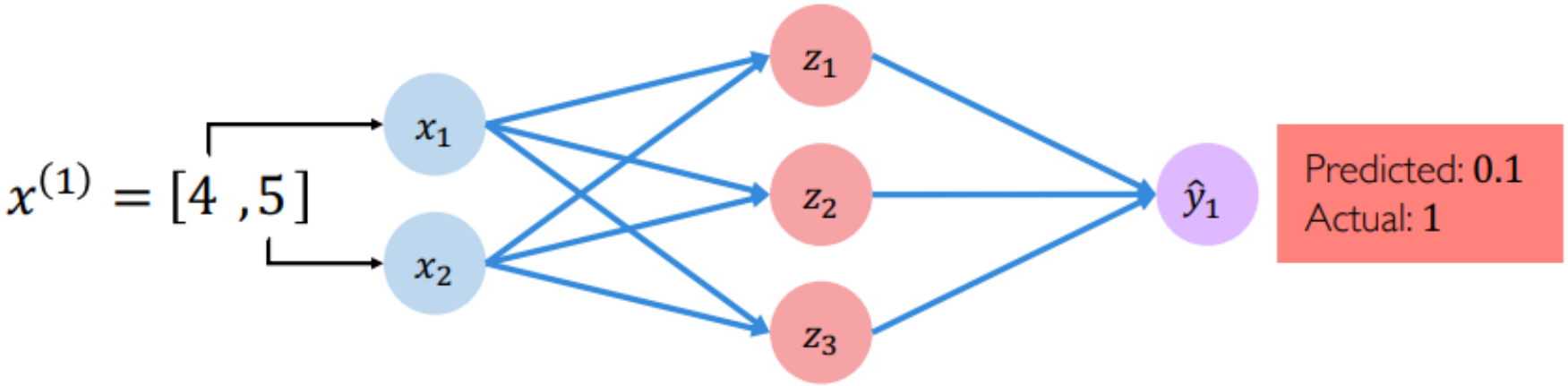


Figure: Example problem with actual and predicted output

Quantifying Loss

The loss of our network measures the cost incurred from incorrect predictions



$$\mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

Predicted Actual

Cost Function vs. Loss Function (Part 1)

In neural networks, the terms **cost function** and **loss function** are commonly used, but they can have different interpretations based on context.

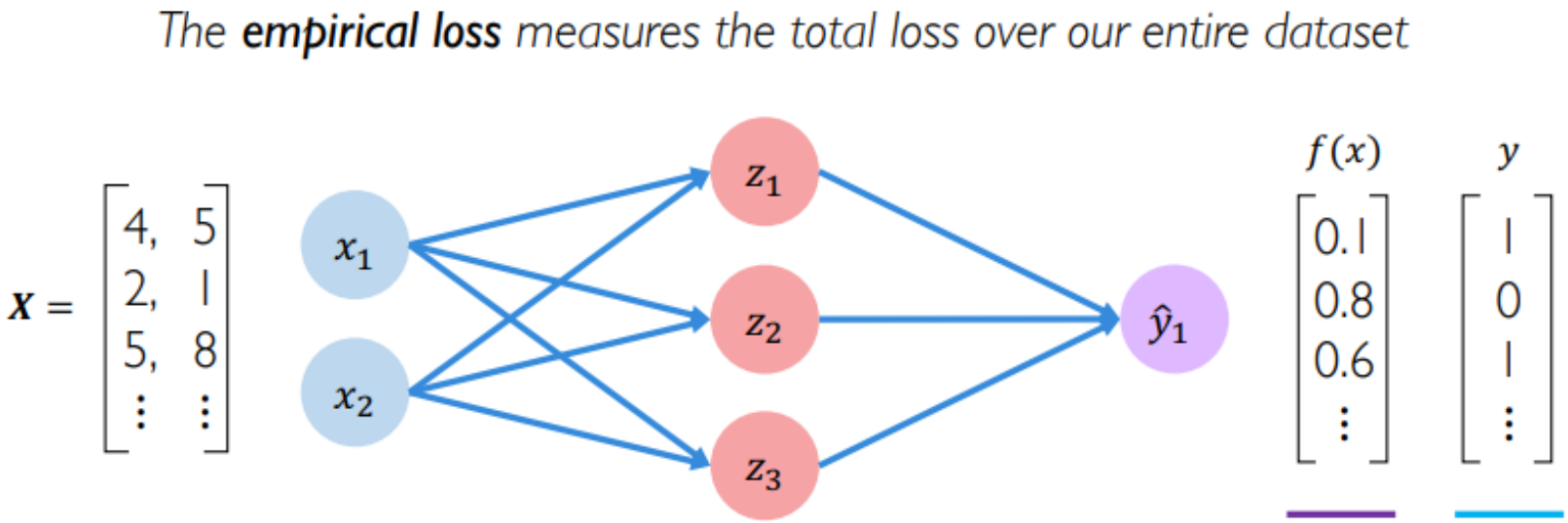
- The **loss function** measures the cost or error on a *single training example*. It calculates the discrepancy between the predicted output and the true output for that particular example.
- The **cost function** measures the overall cost or error of the network across the *entire training dataset*. It takes into account the predictions made by the network on multiple training examples and computes an aggregated measure of the network's performance.

Cost Function vs. Loss Function (Part 2)

- The cost function can be obtained by averaging the losses over the training examples or using other aggregation techniques.
- In many cases, the terms "cost function" and "loss function" are used interchangeably, especially when the network is trained using batch-based optimization algorithms, where the cost function aggregates the losses of multiple training examples.

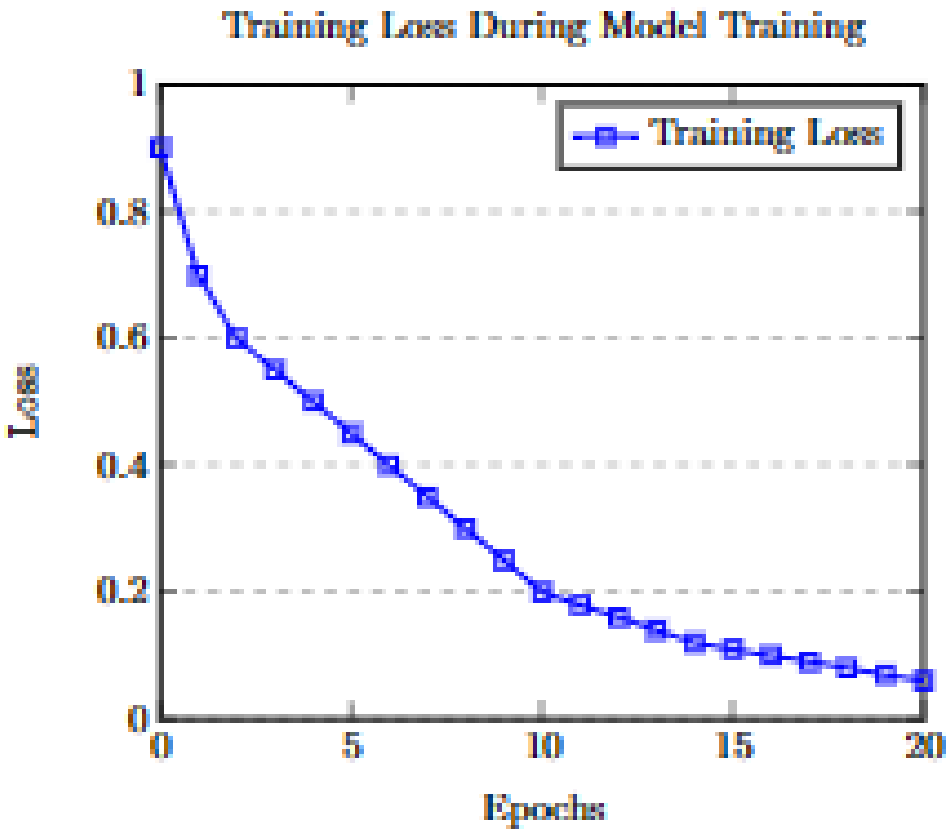
Empirical Loss

The empirical loss measures the total loss over our entire dataset



Loss in Neural Networks

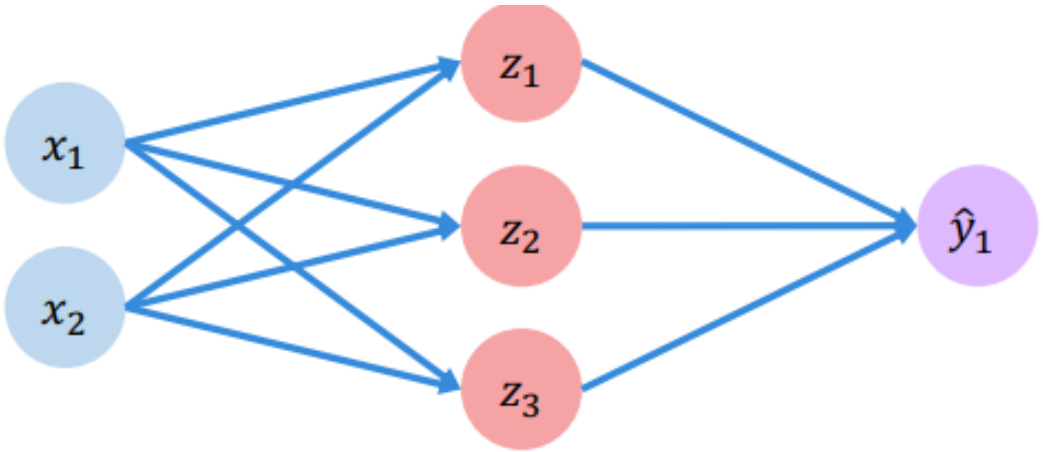
In neural networks, the **loss** measures the cost incurred from incorrect predictions. It quantifies the discrepancy between the predicted output and the true output values.



Binary Cross Entropy Loss

Cross entropy loss can be used with models that output a probability between 0 and 1

$$\boldsymbol{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



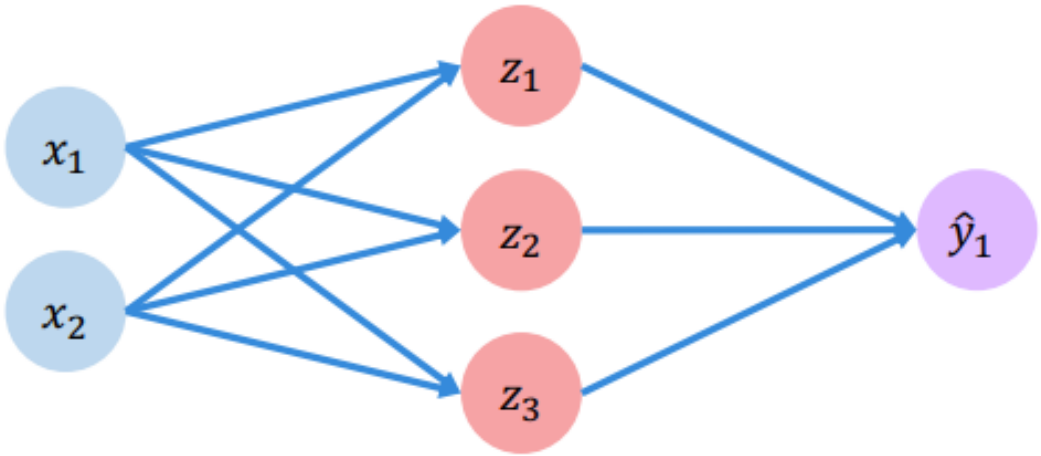
$$f(x) = \begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \qquad y = \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

$$\boldsymbol{J}(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log\Big(\underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big) + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log\Big(1 - \underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big)$$

Mean Squared Error Loss

Mean squared error loss can be used with regression models that output continuous real numbers

$$\boldsymbol{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



$$\boldsymbol{J}(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - f\left(x^{(i)}; \boldsymbol{W}\right)\right)^2$$

Actual Predicted

$f(x)$ y

$$\begin{bmatrix} 30 \\ 80 \\ 85 \\ \vdots \end{bmatrix} \quad \begin{bmatrix} 90 \\ 20 \\ 95 \\ \vdots \end{bmatrix}$$

Final Grades (percentage)

Training Neural Networks

The structural building block of deep learning

November 10, 2025

Loss Optimization

We want to find the network weights that achieve the lowest loss

We want to find the network weights that ***achieve the lowest loss***

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} J(\boldsymbol{W})$$

Remember:

$$\boldsymbol{W} = \left\{\boldsymbol{W}^{(0)}, \boldsymbol{W}^{(1)}, \dots\right\}$$

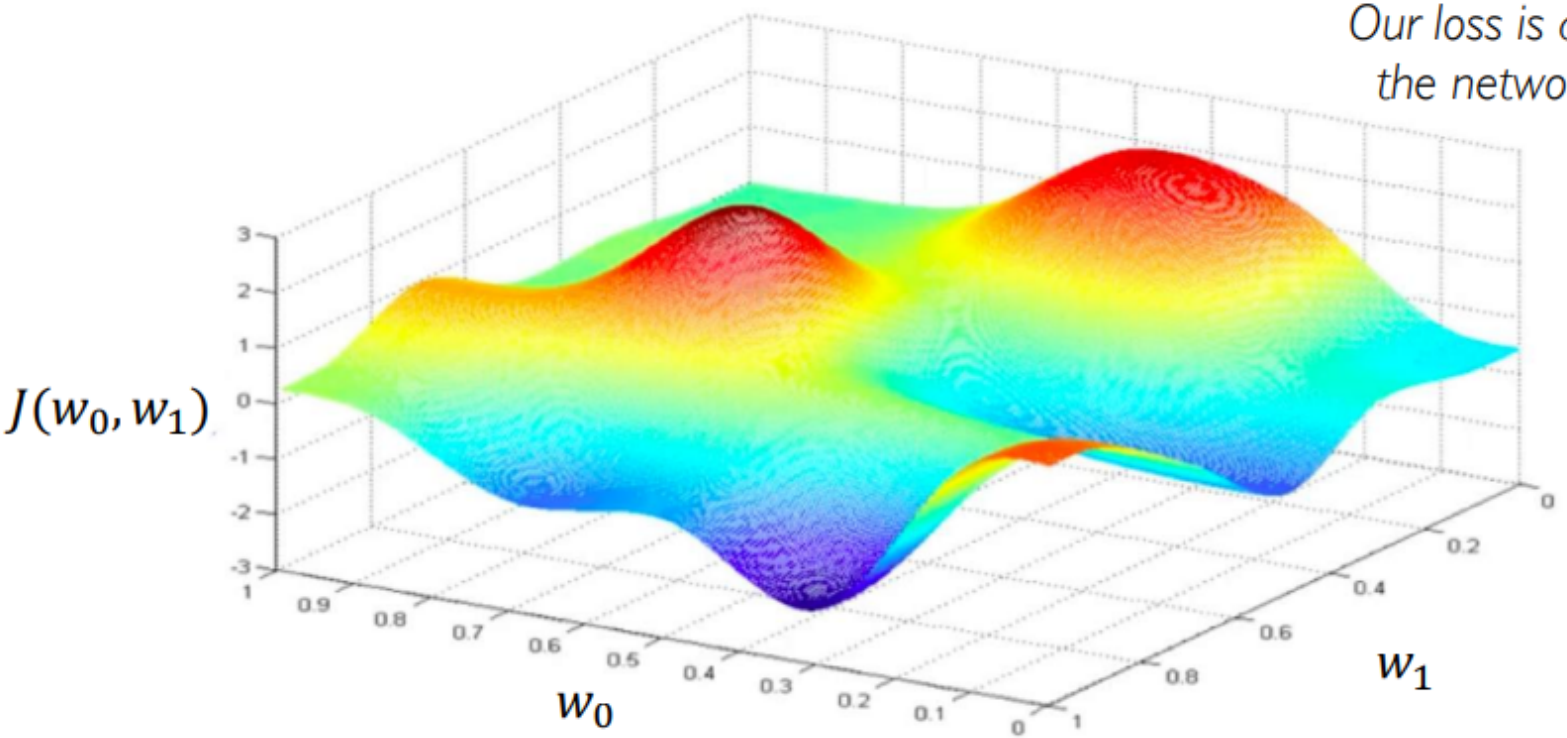
Loss Optimization

We want to find the network weights that achieve the lowest loss

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} J(\boldsymbol{W})$$

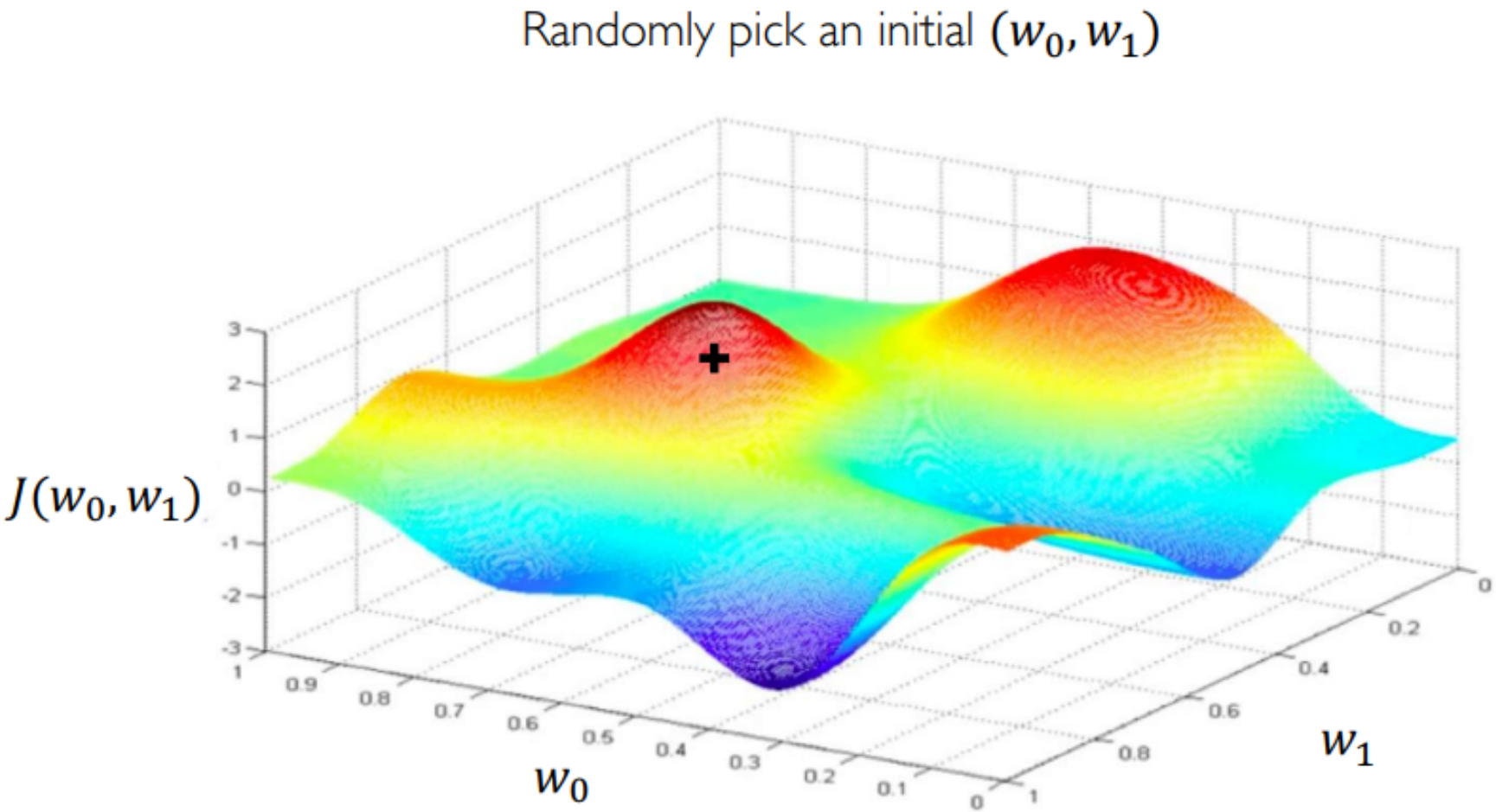
Remember:

Our loss is a function of the network weights!



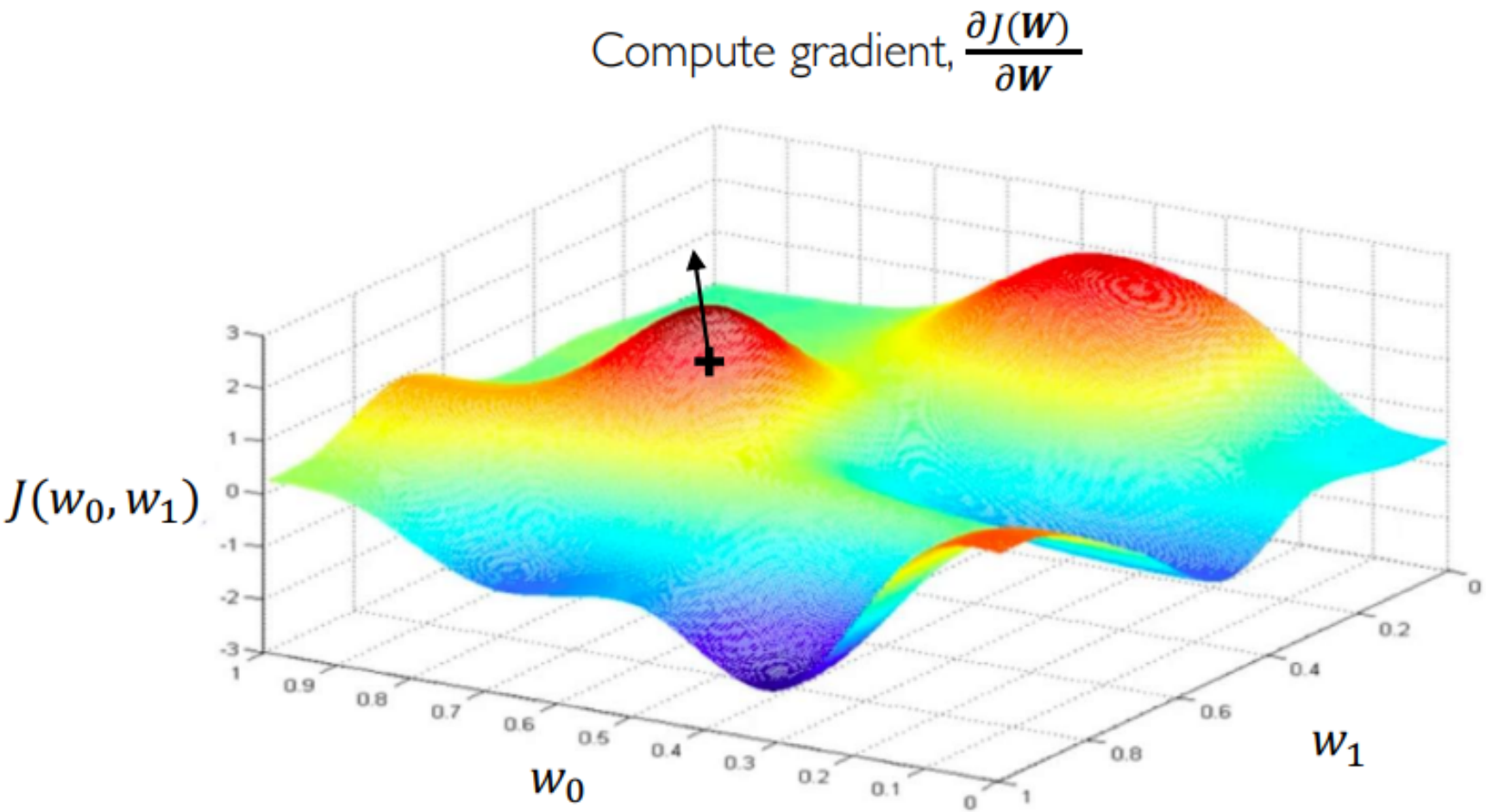
Loss Optimization

We want to find the network weights that achieve the lowest loss



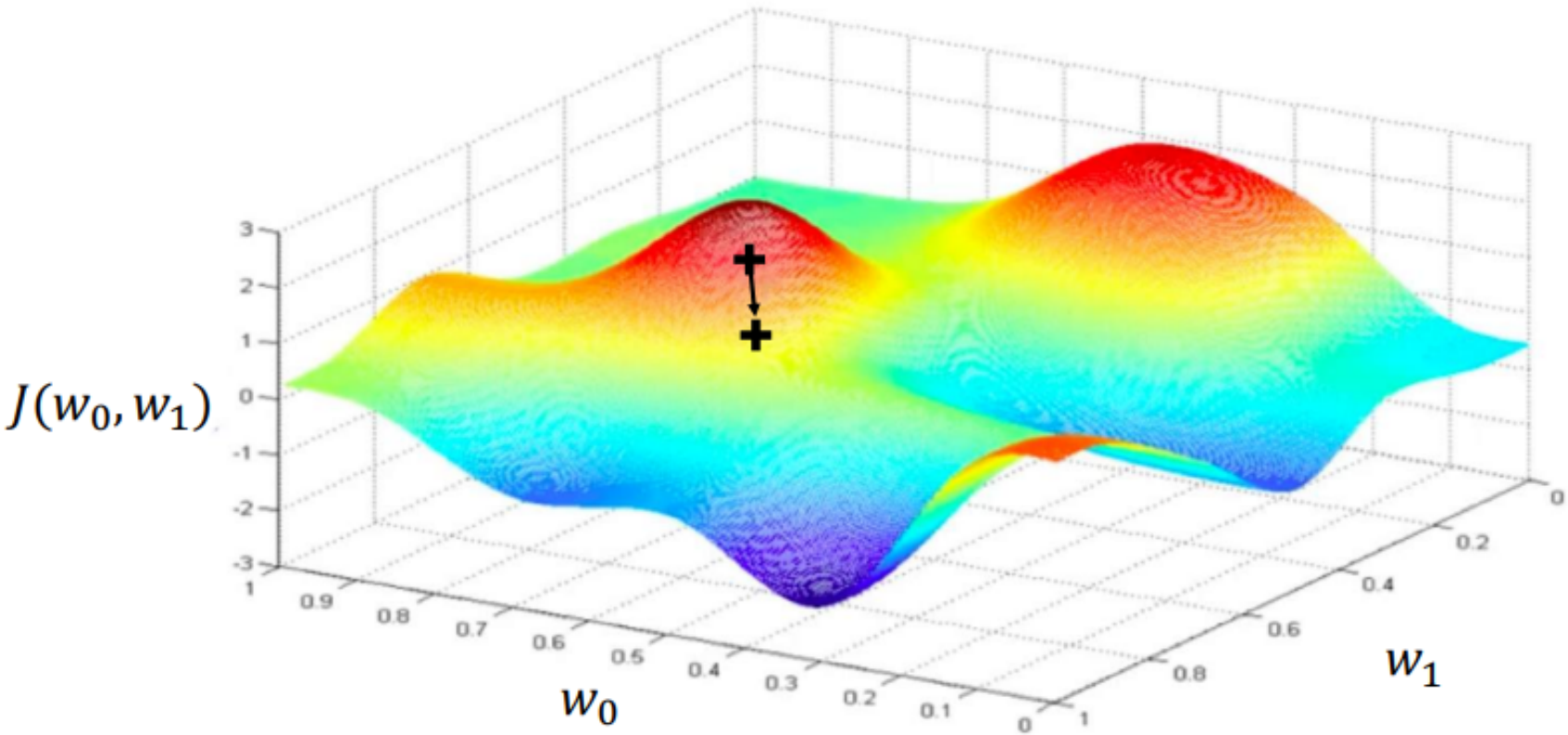
Loss Optimization

We want to find the network weights that achieve the lowest loss



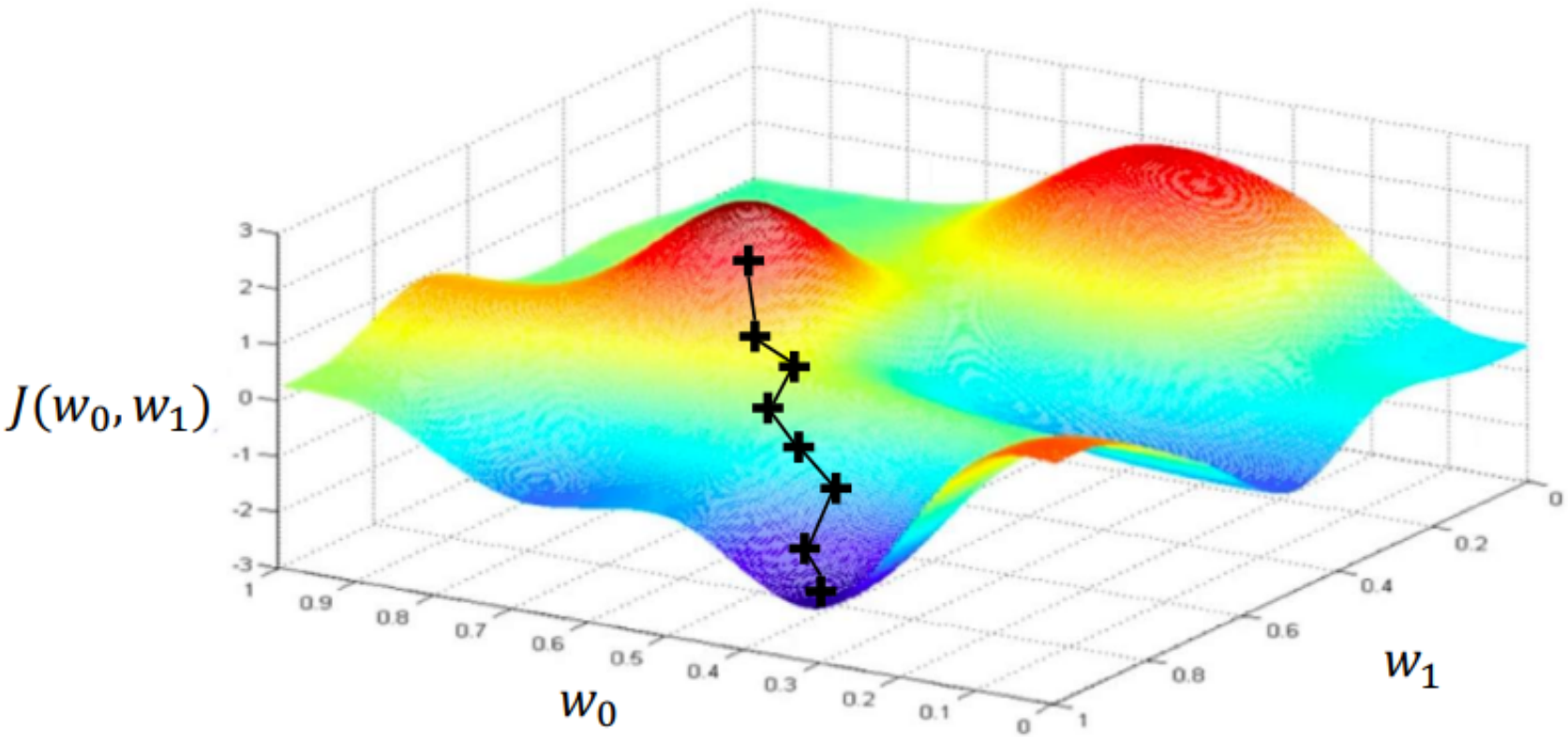
Loss Optimization

Take small step in opposite direction of gradient



Gradient Descent

Repeat until convergence



Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

```
weights = tf.random_normal(shape, stddev=sigma)
```

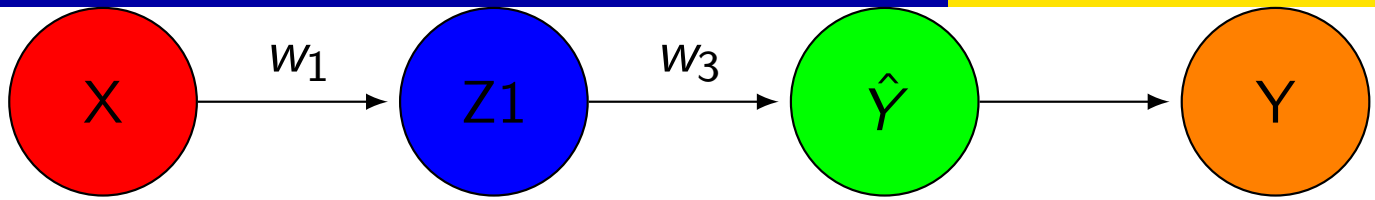
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

```
grads = tf.gradients(ys=loss, xs=weights)
```

4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

```
weights_new = weights.assign(weights - lr * grads)
```

5. Return weights

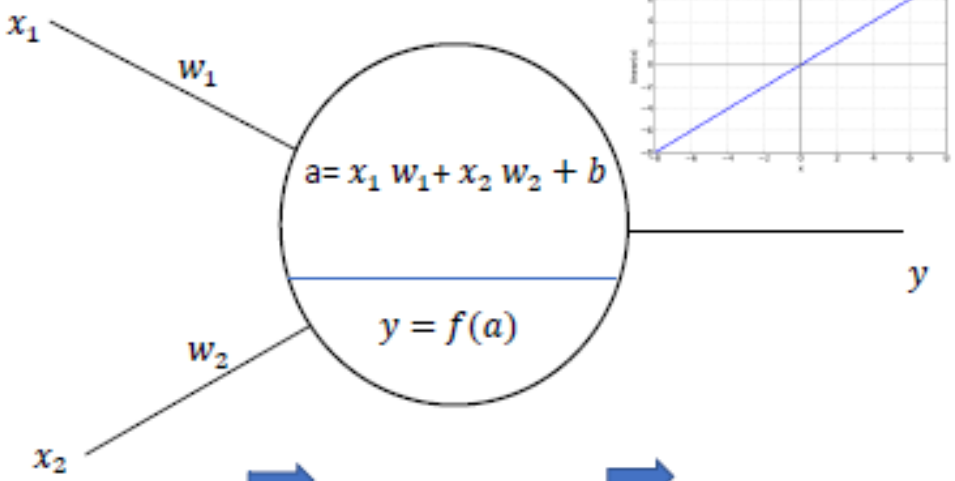

X
w_1
Z1
w_3
$\hat{Y}$
Y

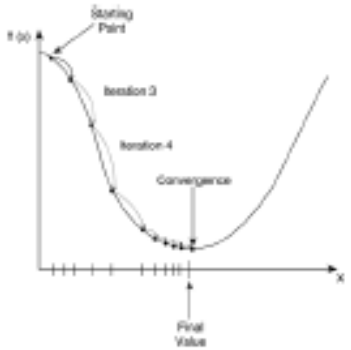
Computing Gradients: Backpropagation

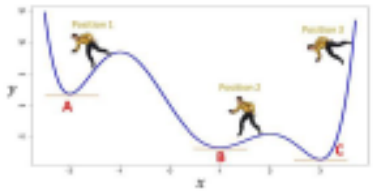
gradient_figure.png

Single neuron with linear activation function

Single neuron with linear activation function







$$E = \frac{1}{2}(t-y)^2$$

$$y = a$$

$t =$ target output

$$w_1 = w_1 - \alpha \frac{\partial E}{\partial w1}$$

$$\frac{\partial E}{\partial w1} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial w1}$$

$$\frac{\partial E}{\partial w1} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial a}\frac{\partial a}{\partial w1}$$

$$\frac{\partial E}{\partial y} = \frac{\partial(\frac{1}{2}(t-y)^2)}{\partial y} = -(t-y)$$

$$\frac{\partial y}{\partial a} = 1$$

$$\frac{\partial a}{\partial w1} = \frac{\partial(x_1 w_1 + x_2 w_2 + b)}{\partial w1} = x_1$$

$$\frac{\partial E}{\partial w1} = -(t-y)\, x_1$$

$$w_1 = w_1 + \alpha(t\text{-}y)\, x_1$$

$$w_2 = w_2 + \alpha(t\text{-}y)\, x_2$$

$$b = b + \alpha(t\text{-}y)$$

$$w_{j,i} = w_{j,i} + \alpha\,(t_j - y_j)\, x_i$$

$w_{j,i}$ = weight of i^{th} input of j^{th} neuron

$$W = W + \alpha\,(t\text{-}y)\,X$$

$$a = XW^T$$

$$X = [x_1, x_2]$$

$$W = [w_1, w_2]$$

Activation Functions in Neural Networks

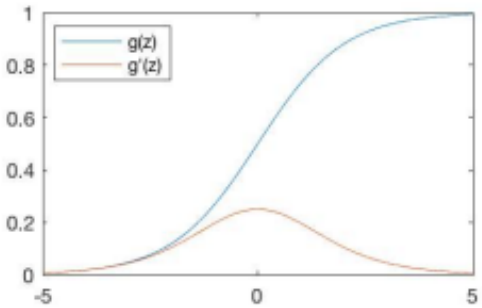
The structural building block of deep learning

November 10, 2025

Activation Functions

Common Activation Functions

Sigmoid Function

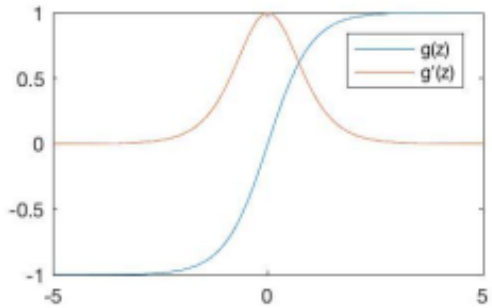


$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

tf.nn.sigmoid(z)

Hyperbolic Tangent

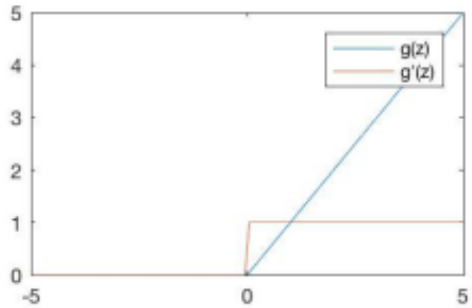


$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

tf.nn.tanh(z)

Rectified Linear Unit (ReLU)



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

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

tf.nn.relu(z)

Activation Function

- Most neural networks transform the latent variables via an activation function, which is typically a nonlinear function.
- The need for such nonlinearity comes from the goal of the network to approximate complicated and highly varying functions.
- Without nonlinearity, deep models would become simple linear regression models, unable to learn complex mappings from inputs to outputs.
- The activation functions are applied to the neuron's weighted inputs (or simply the neuron's activation) to generate the neuron's output.

Sigmoid

The sigmoid activation function squashes its input to range between 0 and 1, which can be useful in creating probabilistic models. However, it suffers from saturation for inputs with large magnitude, causing gradients to vanish and learning to slow down or even stop.

$$f(x) = \frac{1}{1 + e^{-x}}$$

Tanh

The tanh function is similar to the sigmoid but squashes inputs to range between -1 and 1. It is zero centered, making it easier to model inputs that have strongly negative, neutral, and strongly positive values. However, like the sigmoid function, it also suffers from the saturation problem.

$$f(x) = \tanh(x)$$

ReLU

The rectified linear unit (ReLU) is one of the most popular activation functions for deep learning models. It is computationally efficient and does not suffer as much from the saturation problem. The ReLU function is defined as follows:

$$f(x) = \max(0, x)$$

Its main drawback is that it is not differentiable at zero, and its output is unbounded.

Leaky ReLU and PReLU

Variations of the ReLU such as the leaky ReLU and parametric ReLU (PReLU), which have small negative slopes for negative inputs, are proposed to solve the issue of dead neurons (neurons which are not activated) found in ReLU.

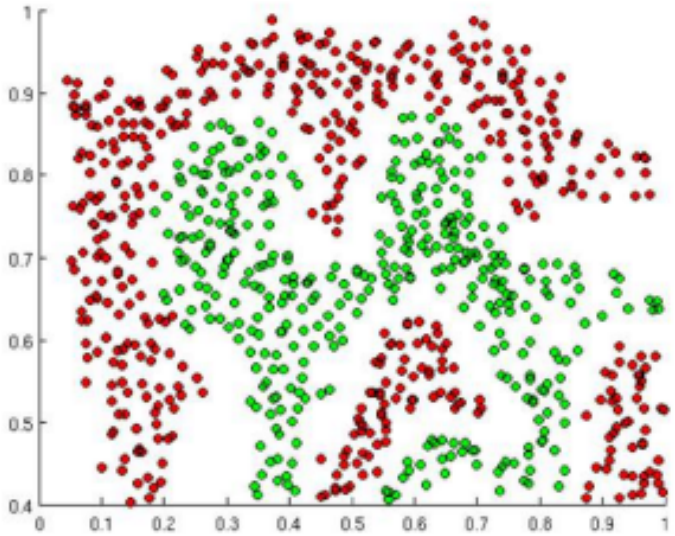
$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{if } x < 0 \end{cases}$$

where α is a small constant in leaky ReLU, and a learned parameter in PReLU.

The Perceptron

Importance of Activation Functions

The purpose of activation functions is to ***introduce non-linearities*** *into the network*

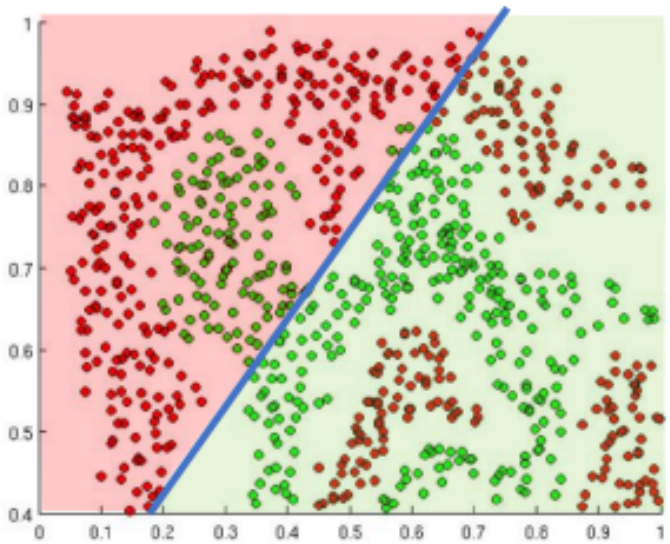


What if we wanted to build a Neural Network to distinguish green vs red points?

The Perceptron

Importance of Activation Functions

The purpose of activation functions is to ***introduce non-linearities*** *into the network*

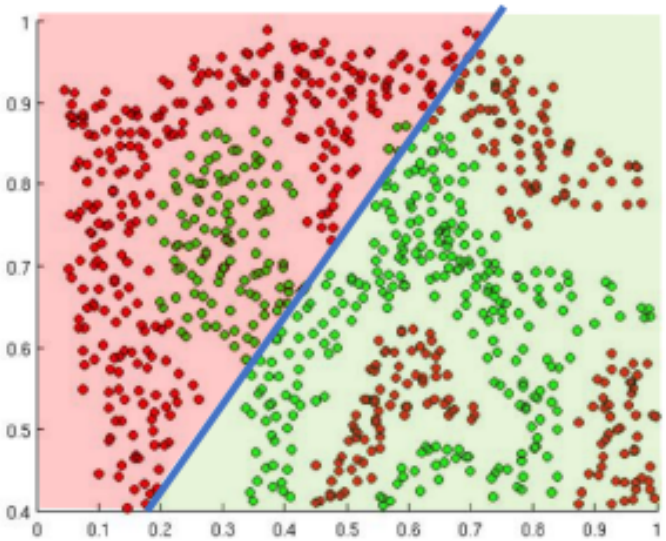


Linear Activation functions produce linear decisions no matter the network size

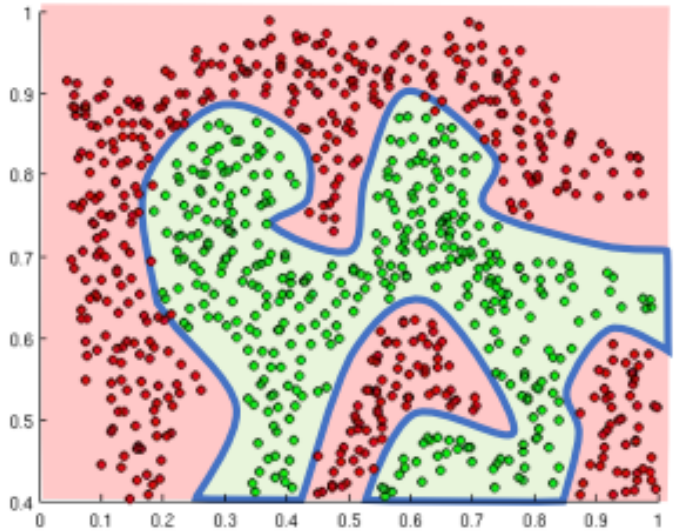
The Perceptron

Importance of Activation Functions

The purpose of activation functions is to ***introduce non-linearities*** *into the network*



Linear Activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

SGD with Momentum

- SGD with Momentum is an optimization algorithm used in training neural networks.
- It improves convergence and accelerates learning compared to standard SGD.

Basic Idea of SGD

- The standard SGD algorithm updates weights directly based on gradients from mini-batches.
- It can have slow convergence, especially in the presence of noisy gradients.

Introduction to Momentum

- Momentum is an enhancement to the basic SGD algorithm.
- It accumulates past gradients to smooth out the optimization process.

The Momentum Term

- A "momentum" term is introduced to gradient updates.
- The momentum term is a moving average of past gradients.

SGD with Momentum Algorithm

- Update step for weights and biases:

$$
\begin{aligned}
v_{dW} &= \beta v_{dW} + (1-\beta) dW \\
v_{db} &= \beta v_{db} + (1-\beta) db \\
W &= W - \text{learning\_rate} \cdot v_{dW} \\
b &= b - \text{learning\_rate} \cdot v_{db}
\end{aligned}
$$

- β is the momentum coefficient (commonly around 0.9).

SGD with Momentum Algorithm

- Update step for weights and biases:

$$v_{dW} = \beta v_{dW} + (1 - \beta) dW$$
$$v_{db} = \beta v_{db} + (1 - \beta) db$$
$$W = W - \text{learning\_rate} \cdot v_{dW}$$
$$b = b - \text{learning\_rate} \cdot v_{db}$$

- β is the momentum coefficient (commonly around 0.9).
- **Explanation of variables:**
  - v_{dW}: The velocity term for the weight gradients. It accumulates past gradients of the weights over time and helps to smoothen the optimization process.
  - dW: The gradient of the cost function with respect to the weights. It represents the rate of change of the cost function concerning the model's weights and is computed using backpropagation.
  - v_{db}: The velocity term for the bias gradients. Similar to v_{dW}, it accumulates past gradients of the biases.
  - db: The gradient of the cost function with respect to the biases. It

Advantages of SGD with Momentum

- Faster convergence compared to standard SGD.
- Better handling of noisy gradients.

Comparison with Standard SGD

- SGD with Momentum provides faster optimization and better efficiency.
- It helps overcome the issues of standard SGD in deep learning.

Conclusion

- SGD with Momentum is a powerful optimization algorithm for training neural networks.
- It enhances convergence and accelerates the learning process.

Sigmoid Function

- Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$
- Derivative: $\sigma'(x) = \sigma(x)(1 - \sigma(x))$

Problem Setup

- Input: $x = 2$, Weight: $w = 0.5$, Bias: $b = 0$
- Target output: $y_{\text{true}} = 1$
- Activation function: Sigmoid

Forward Pass

$$z = wx + b = 0.5 \cdot 2 + 0 = 1$$

$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-1}} \approx 0.731$$

Loss Calculation

$$L = \frac{1}{2}(\hat{y} - y_{\text{true}})^2$$
$$= \frac{1}{2}(0.731 - 1)^2 = \frac{1}{2}(0.0724) \approx 0.0362$$

Backpropagation: Chain Rule

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w}$$

$$\frac{\partial L}{\partial \hat{y}} = \hat{y} - y_{\text{true}} = -0.269$$

$$\frac{\partial \hat{y}}{\partial z} = \hat{y}(1 - \hat{y}) = 0.731 \cdot 0.269 \approx 0.196$$

$$\frac{\partial z}{\partial w} = x = 2$$

$$\frac{\partial L}{\partial w} = -0.269 \cdot 0.196 \cdot 2 \approx -0.1055$$

Gradient for Bias

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial b}$$
$$= -0.269 \cdot 0.196 \cdot 1 \approx -0.0527$$

Weight Update

Learning rate $\eta = 0.1$

$$w_{\text{new}} = w - \eta \cdot \frac{\partial L}{\partial w} = 0.5 + 0.01055 = 0.5105$$

$$b_{\text{new}} = b - \eta \cdot \frac{\partial L}{\partial b} = 0 + 0.00527 = 0.00527$$

Summary

- Output: $\hat{y} \approx 0.731$
- Loss: ≈ 0.0362
- Gradients: $\frac{\partial L}{\partial w} \approx -0.1055$, $\frac{\partial L}{\partial b} \approx -0.0527$
- Updated Parameters: $w = 0.5105$, $b = 0.00527$

Network Architecture

- Input layer: 1 neuron ($x = 1.0$)
- Hidden layer: 2 neurons with sigmoid activation
- Output layer: 1 neuron with sigmoid activation
- Weights and biases (example values):
  - Hidden: $w_1 = 0.4, b_1 = 0.1,\ w_2 = 0.7, b_2 = -0.2$
  - Output: $v_1 = 0.6, v_2 = 0.9, b_3 = 0.05$

Forward Pass: Hidden Layer

$$z_1 = w_1 x + b_1 = 0.4 \cdot 1 + 0.1 = 0.5$$

$$z_2 = w_2 x + b_2 = 0.7 \cdot 1 - 0.2 = 0.5$$

$$h_1 = \sigma(z_1) = \frac{1}{1 + e^{-0.5}} \approx 0.622$$

$$h_2 = \sigma(z_2) \approx 0.622$$

Forward Pass: Output Layer

$$
\begin{aligned}
z_3 &= v_1 h_1 + v_2 h_2 + b_3 \\
&= 0.6 \cdot 0.622 + 0.9 \cdot 0.622 + 0.05 \\
&= 0.3732 + 0.5598 + 0.05 = 0.983 \\
\hat{y} &= \sigma(z_3) = \frac{1}{1 + e^{-0.983}} \approx 0.728
\end{aligned}
$$

Loss Calculation

$$y_{\text{true}} = 1.0$$

$$L = \frac{1}{2}(\hat{y} - y_{\text{true}})^2 = \frac{1}{2}(0.728 - 1)^2$$

$$= \frac{1}{2}(-0.272)^2 = \frac{1}{2}(0.074) \approx 0.037$$

Backprop: Output Layer

$$\frac{\partial L}{\partial \hat{y}} = \hat{y} - y = -0.272$$

$$\frac{\partial \hat{y}}{\partial z_3} = \hat{y}(1-\hat{y}) \approx 0.728(1-0.728) = 0.198$$

$$\frac{\partial L}{\partial z_3} = -0.272 \cdot 0.198 = -0.0538$$

$$\frac{\partial L}{\partial v_1} = \frac{\partial L}{\partial z_3} \cdot h_1 = -0.0538 \cdot 0.622 \approx -0.0335$$

$$\frac{\partial L}{\partial v_2} = -0.0335, \quad \frac{\partial L}{\partial b_3} = -0.0538$$

Backprop: Hidden Layer

$$\frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial z_3} \cdot v_1 = -0.0538 \cdot 0.6 = -0.0323$$

$$\frac{\partial h_1}{\partial z_1} = h_1(1 - h_1) = 0.622(1 - 0.622) = 0.235$$

$$\frac{\partial L}{\partial z_1} = -0.0323 \cdot 0.235 \approx -0.0076$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial z_1} \cdot x = -0.0076$$

$$\frac{\partial L}{\partial b_1} = -0.0076$$

(Same for neuron 2)

Summary

- Output: $\hat{y} \approx 0.728$, Target: $y = 1.0$
- Loss: $L \approx 0.037$
- Gradients:
  - Output weights: $\partial L/\partial v_1 \approx -0.0335$, $\partial L/\partial v_2 \approx -0.0335$
  - Hidden weights: $\partial L/\partial w_1 \approx -0.0076$, etc.

Q&A

- Thank you for your attention! Questions and comments are welcome.