DA6401:Introduction to Deep Learning
Module 1B-Modern Neuron & MLP

Ganapathy Krishnamurthi

Department of Data Science and Artificial Intelligence , IIT Madras

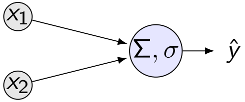
The Challenge: Beyond the Single Perceptron

- **Context:** Single perceptrons cannot solve non-linear problems (e.g., XOR).

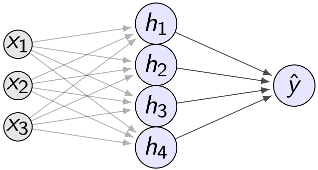
The Challenge: Beyond the Single Perceptron

- **Context:** Single perceptrons cannot solve non-linear problems (e.g., XOR).
- **Solution:** Networks of neurons (Multi-Layer Perceptrons) can represent non-linear decision boundaries.

Single Perceptron

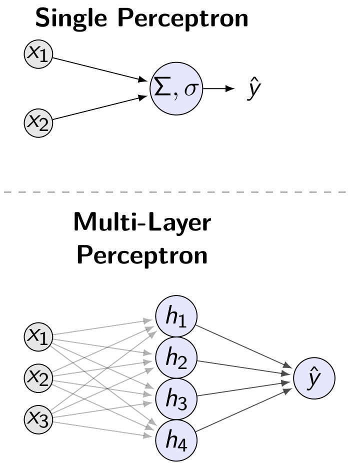


Multi-Layer Perceptron



The Challenge: Beyond the Single Perceptron

- **Context:** Single perceptrons cannot solve non-linear problems (e.g., XOR).
- **Solution:** Networks of neurons (Multi-Layer Perceptrons) can represent non-linear decision boundaries.



Learning Problem

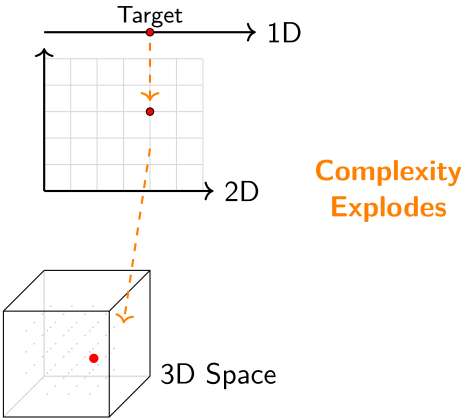
We have a learning rule for a single perceptron. But how do we adjust weights deep inside the network?

The Parameter Space Explosion

The Concept:

- In 1 dimension (1 weight), you just search a line. Easy.
- In 2 dimensions, you search a square. Harder.
- In 3 dimensions, you search a cube.
- **Neural Networks** often have 10^3 to 10^9 dimensions (weights).

The volume of the space where the "correct" configuration lives becomes infinitesimally small relative to the total volume.



Approach 1: Stochastic Weight Guessing

- **Concept:** Treat the network as a black box. Randomly sample weight vectors $\mathbf{w}$ from a high-dimensional space until performance is satisfactory.

Approach 1: Stochastic Weight Guessing

- **Concept:** Treat the network as a black box. Randomly sample weight vectors $\mathbf{w}$ from a high-dimensional space until performance is satisfactory.
- **Formal Failure Analysis (Curse of Dimensionality):**
  - Consider a small network with $N = 1000$ weights.
  - Assume each weight is quantized to just two values, $w_i \in \{-1, 1\}$.
  - The search space size is $|\mathcal{W}| = 2^{1000} \approx 10^{301}$.
  - For context, the number of atoms in the observable universe is $\approx 10^{80}$.

Approach 1: Stochastic Weight Guessing

- **Concept:** Treat the network as a black box. Randomly sample weight vectors $\mathbf{w}$ from a high-dimensional space until performance is satisfactory.
- **Formal Failure Analysis (Curse of Dimensionality):**
  - Consider a small network with $N = 1000$ weights.
  - Assume each weight is quantized to just two values, $w_i \in \{-1, 1\}$.
  - The search space size is $|\mathcal{W}| = 2^{1000} \approx 10^{301}$.
  - For context, the number of atoms in the observable universe is $\approx 10^{80}$.
- **Conclusion:** The volume of the parameter space grows exponentially with the number of parameters. brute-force or undirected random search is theoretically non-viable for practical networks.
- We need a systematic method to navigate this space.

Approach 2: Parameterization

Core Idea: We define our network as a function with a set of learnable parameters (weights and biases), denoted by w. The goal is to find the optimal values for θ.

Approach 2: Parameterization

Core Idea: We define our network as a function with a set of learnable parameters (weights and biases), denoted by w. The goal is to find the optimal values for θ.

The Network as a Function

We express the model's prediction as

$$\hat{y} = f(x; w),$$

where:

- x is the numerical input data.
- f is the network architecture (e.g., layers, activation functions).
- w represents all the weights and biases in the network.
- $\hat{y}$ is the model's prediction.

Quantifying Performance: The Loss Function

- To improve, we must quantify "badness." We define a **Loss Function** (or Cost/Objective Function), $\mathcal{L}$.

Quantifying Performance: The Loss Function

- To improve, we must quantify "badness." We define a **Loss Function** (or Cost/Objective Function), $\mathcal{L}$.

- Given a dataset $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$, we measure the discrepancy between predictions $\hat{y}^{(i)} = f(x^{(i)}; \mathbf{w})$ and targets $y^{(i)}$.

Quantifying Performance: The Loss Function

- To improve, we must quantify "badness." We define a **Loss Function** (or Cost/Objective Function), $\mathcal{L}$.
- Given a dataset $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$, we measure the discrepancy between predictions $\hat{y}^{(i)} = f(x^{(i)}; \mathbf{w})$ and targets $y^{(i)}$.
- **Example: Mean Squared Error (MSE)** used for regression:

$$J(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - \hat{y}^{(i)})^2$$

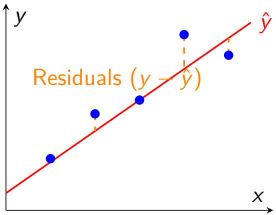


Figure: Visualizing MSE: The loss function minimizes the sum of squared vertical residuals (orange lines).

The Parameter Space & Loss Landscape

- **Shift in Perspective:** During training, the data $\mathcal{D}$ is fixed. The network's parameters $\mathbf{w}$ are the variables.

The Parameter Space & Loss Landscape

- **Shift in Perspective:** During training, the data $\mathcal{D}$ is fixed. The network's parameters **w** are the variables.
- We view the loss as a function solely of the weights: $J(\mathbf{w}) : \mathbb{R}^d \to \mathbb{R}$.

The Parameter Space & Loss Landscape

- **Shift in Perspective:** During training, the data $\mathcal{D}$ is fixed. The network's parameters $\mathbf{w}$ are the variables.
- We view the loss as a function solely of the weights: $J(\mathbf{w}) : \mathbb{R}^d \to \mathbb{R}$.
- This defines a high-dimensional "**loss landscape**."
  - **Coordinates:** The values of the weights $(w_1, w_2, \ldots, w_d)$.
  - **Altitude:** The loss value $J(\mathbf{w})$.

Non-Convex Loss Landscape

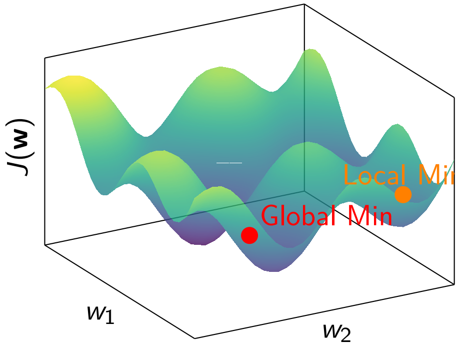


Figure: A visualization of a complex 2D weight space resulting in a rugged loss surface.

Optimization via Calculus

- Assuming $J(\mathbf{w})$ is differentiable, calculus provides the direction of steepest ascent.

Optimization via Calculus

- Assuming $J(\mathbf{w})$ is differentiable, calculus provides the direction of steepest ascent.
- **The Gradient:** The vector of partial derivatives with respect to all weights:

$$\nabla_{\mathbf{w}} J = \left[\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, \ldots, \frac{\partial J}{\partial w_d} \right]^T$$

Optimization via Calculus

- Assuming $J(\mathbf{w})$ is differentiable, calculus provides the direction of steepest ascent.
- **The Gradient:** The vector of partial derivatives with respect to all weights:

$$\nabla_{\mathbf{w}} J = \left[\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, \ldots, \frac{\partial J}{\partial w_d} \right]^T$$

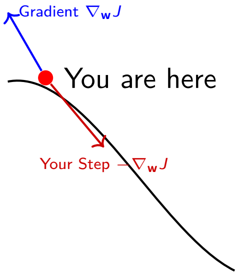
- The negative gradient, $-\nabla_{\mathbf{w}} J$, points in the direction of steepest *descent*.

Optimization via Calculus

- Assuming $J(\mathbf{w})$ is differentiable, calculus provides the direction of steepest ascent.
- **The Gradient:** The vector of partial derivatives with respect to all weights:

$$\nabla_{\mathbf{w}} J = \left[\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, \ldots, \frac{\partial J}{\partial w_d} \right]^T$$

- The negative gradient, $-\nabla_{\mathbf{w}} J$, points in the direction of steepest *descent*.
- We move opposite to the direction of gradient to get to the minimum of the loss function. This is called **Gradient Descent**



The Supervised Learning Setup

A typical supervised machine learning system consists of five key components:

1. **Data:** A collection of n labeled examples, denoted as $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$.

A typical supervised machine learning system consists of five key components:

1. **Data:** A collection of n labeled examples, denoted as $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$.
2. **Model:** An approximation function that maps input x to output y.

The Supervised Learning Setup

A typical supervised machine learning system consists of five key components:

1. **Data:** A collection of n labeled examples, denoted as $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$.
2. **Model:** An approximation function that maps input x to output y.
3. **Parameters:** The unknown variables (e.g., weights $\mathbf{w}$) within the model that determine its behavior. These must be learned.

The Supervised Learning Setup

A typical supervised machine learning system consists of five key components:

1. **Data:** A collection of n labeled examples, denoted as $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$.
2. **Model:** An approximation function that maps input x to output y.
3. **Parameters:** The unknown variables (e.g., weights $\mathbf{w}$) within the model that determine its behavior. These must be learned.
4. **Objective / Loss Function:** A metric $J(\mathbf{w})$ (e.g., Squared Error) that quantifies the difference between the model's prediction $\hat{y}$ and the true label y.

The Supervised Learning Setup

A typical supervised machine learning system consists of five key components:

1. **Data:** A collection of n labeled examples, denoted as $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$.
2. **Model:** An approximation function that maps input x to output y.
3. **Parameters:** The unknown variables (e.g., weights $\mathbf{w}$) within the model that determine its behavior. These must be learned.
4. **Objective / Loss Function:** A metric $J(\mathbf{w})$ (e.g., Squared Error) that quantifies the difference between the model's prediction $\hat{y}$ and the true label y.
5. **Learning Algorithm:** The optimization method (e.g., Gradient Descent) used to adjust the parameters $\mathbf{w}$ to minimize the Loss Function.

Recap: The Perceptron

Formal Definition

A Perceptron is a linear classifier that maps a real-valued input vector $\mathbf{x}$ to a binary output y.

$$y = \phi(\mathbf{w} \cdot \mathbf{x} + b)$$

Where $\phi(z)$ is the **Step Function**:

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

Recap: The Perceptron

Formal Definition

A Perceptron is a linear classifier that maps a real-valued input vector $\mathbf{x}$ to a binary output y.

$$y = \phi(\mathbf{w} \cdot \mathbf{x} + b)$$

Where $\phi(z)$ is the **Step Function**:

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

Two Critical Limitations

1. Capacity

It can only solve linearly separable problems.

Recap: The Perceptron

Formal Definition

A Perceptron is a linear classifier that maps a real-valued input vector $\mathbf{x}$ to a binary output y.

$$y = \phi(\mathbf{w} \cdot \mathbf{x} + b)$$

Where $\phi(z)$ is the **Step Function**:

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

Two Critical Limitations

1. Capacity

It can only solve linearly separable problems.

*We addressed this by stacking Perceptrons into a **Neural Network** (MLP).*

Recap: The Perceptron

Formal Definition

A Perceptron is a linear classifier that maps a real-valued input vector $\mathbf{x}$ to a binary output y.

$$y = \phi(\mathbf{w} \cdot \mathbf{x} + b)$$

Where $\phi(z)$ is the **Step Function**:

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

Two Critical Limitations

1. Capacity

It can only solve linearly separable problems.

*We addressed this by stacking Perceptrons into a **Neural Network** (MLP).*

2. Threshold

The Step function (ϕ) has a "Harsh Threshold." Its derivative is either 0 or undefined.

Recap: The Perceptron

Formal Definition

A Perceptron is a linear classifier that maps a real-valued input vector $\mathbf{x}$ to a binary output y.

$$y = \phi(\mathbf{w} \cdot \mathbf{x} + b)$$

Where $\phi(z)$ is the **Step Function**:

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

Two Critical Limitations

1. Capacity

It can only solve linearly separable problems.

We addressed this by stacking Perceptrons into a ***Neural Network*** *(MLP).*

2. Threshold

The Step function (ϕ) has a "Harsh Threshold." Its derivative is either 0 or undefined.

How do we get a Gradient if the derivative is 0?

The Modern Neuron

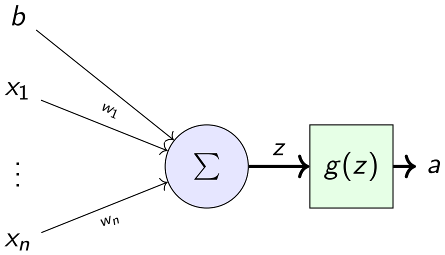


Figure: A neuron with differentiable activation function

The Modern Neuron

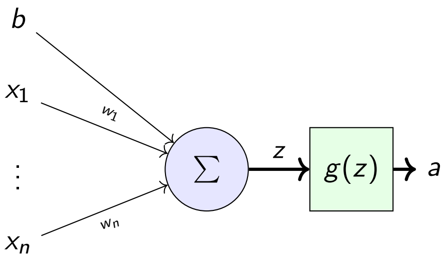


Figure: A neuron with differentiable activation function

Step 1: Linear Combination

- Calculates a weighted sum of inputs plus a bias.
- Output is $z = \mathbf{w} \cdot \mathbf{x} + b$.

The Modern Neuron

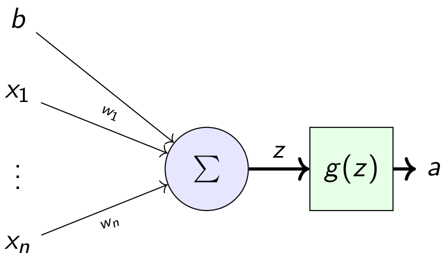


Figure: A neuron with differentiable activation function

Step 1: Linear Combination

- Calculates a weighted sum of inputs plus a bias.
- Output is $z = \mathbf{w} \cdot \mathbf{x} + b$.

Step 2: Differentiable Activation

- Passes the linear output z through an differentiable activation function $a = g(z)$.

The Modern Neuron

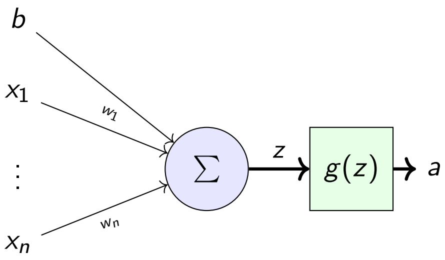


Figure: A neuron with differentiable activation function

Step 1: Linear Combination

- Calculates a weighted sum of inputs plus a bias.
- Output is $z = \mathbf{w} \cdot \mathbf{x} + b$.

Step 2: Differentiable Activation

- Passes the linear output z through an differentiable activation function $a = g(z)$.

Why Differentiability is the Superpower?

Differentiable $\rightarrow$ **gradient**. This gradient is used to adjust weights to fix errors. It is the core requirement for **backpropagation**.

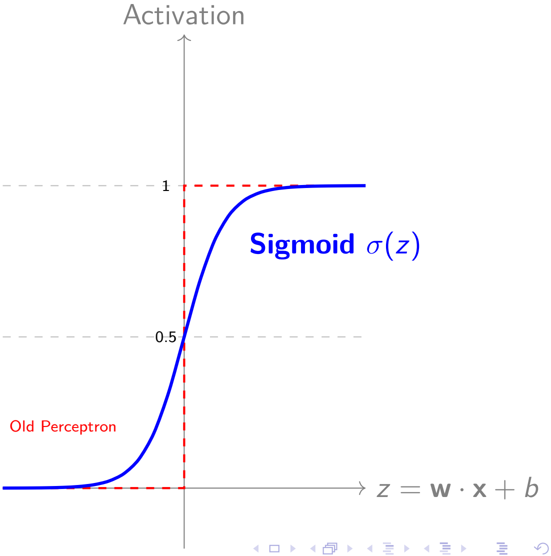
The Definition

We replace the sharp Step Function with the smooth **Sigmoid Function** (σ):

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

Key Properties:

- **Continuous Output:** Returns a value between 0 and 1 (e.g., 0.73).
- **Probabilistic Interpretation:** Can be seen as $P(y = 1|\mathbf{x})$.
- **Differentiable:** The slope exists everywhere.
  - $\sigma'(z) \neq 0$ (mostly).



Evolution of the Neuron: Side-by-Side Comparison

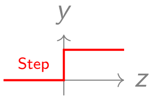
1. The Perceptron

Equation:

$$y = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \geq 0 \\ 0 & \text{if } \mathbf{w} \cdot \mathbf{x} + b < 0 \end{cases}$$

Characteristics:

- **Output:** Binary $\{0, 1\}$.
- **Nature:** Hard Threshold.
- **Derivative:** 0 or Undefined.



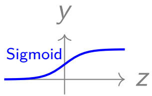
2. The Sigmoid Neuron

Equation:

$$y = \sigma(z) = \frac{1}{1 + e^{-(\mathbf{w} \cdot \mathbf{x} + b)}}$$

Characteristics:

- **Output:** Real value $[0, 1]$.
- **Nature:** Smooth (S-Curve).
- **Derivative:** Non-zero $(y(1-y))$.



Let's compute the output of a hidden layer with 3 neurons, receiving input from 2 neurons, using **Sigmoid Activation**.

Let's compute the output of a hidden layer with 3 neurons, receiving input from 2 neurons, using **Sigmoid Activation**.

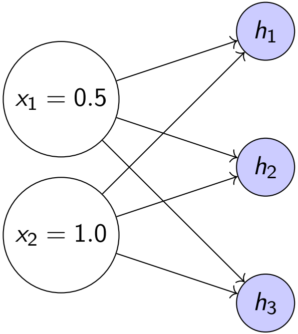


Figure: Input vector **x** has 2 features.
Hidden layer has 3 neurons.

Let's compute the output of a hidden layer with 3 neurons, receiving input from 2 neurons, using **Sigmoid Activation**.

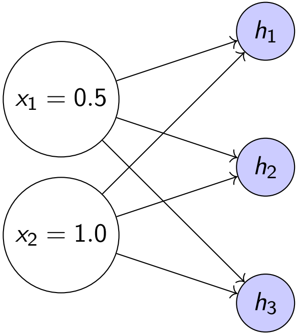


Figure: Input vector **x** has 2 features. Hidden layer has 3 neurons.

1. Define inputs, weights, and biases:

$$\mathbf{x} = \begin{bmatrix} 0.5 \\ 1.0 \end{bmatrix} \quad \mathbf{W} = \begin{bmatrix} 0.2 & 0.7 \\ -0.4 & 0.1 \\ 0.9 & -0.3 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 0.1 \\ 0.2 \\ -0.5 \end{bmatrix}$$

Let's compute the output of a hidden layer with 3 neurons, receiving input from 2 neurons, using **Sigmoid Activation**.

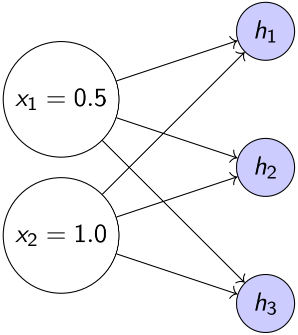


Figure: Input vector $\mathbf{x}$ has 2 features. Hidden layer has 3 neurons.

1. **Define inputs, weights, and biases:**

$$\mathbf{x} = \begin{bmatrix} 0.5 \\ 1.0 \end{bmatrix} \quad \mathbf{W} = \begin{bmatrix} 0.2 & 0.7 \\ -0.4 & 0.1 \\ 0.9 & -0.3 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 0.1 \\ 0.2 \\ -0.5 \end{bmatrix}$$

2. **Calculate the weighted sum $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$:**

$$\mathbf{z} = \begin{bmatrix} 0.2 & 0.7 \\ -0.4 & 0.1 \\ 0.9 & -0.3 \end{bmatrix} \begin{bmatrix} 0.5 \\ 1.0 \end{bmatrix} + \begin{bmatrix} 0.1 \\ 0.2 \\ -0.5 \end{bmatrix} = \begin{bmatrix} 0.9 \\ 0.1 \\ -0.35 \end{bmatrix}$$

Let's compute the output of a hidden layer with 3 neurons, receiving input from 2 neurons, using **Sigmoid Activation**.

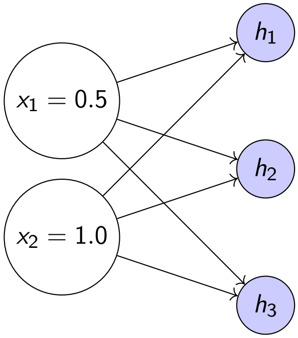


Figure: Input vector **x** has 2 features. Hidden layer has 3 neurons.

The activation function is Sigmoid:

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

Let's compute the output of a hidden layer with 3 neurons, receiving input from 2 neurons, using **Sigmoid Activation**.

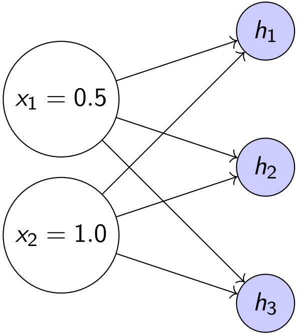


Figure: Input vector **x** has 2 features. Hidden layer has 3 neurons.

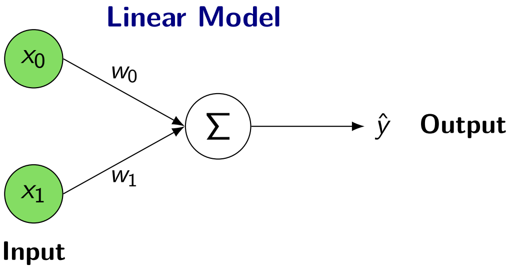
The activation function is Sigmoid:

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

3. Apply Sigmoid activation $\mathbf{a} = \sigma(\mathbf{z})$:

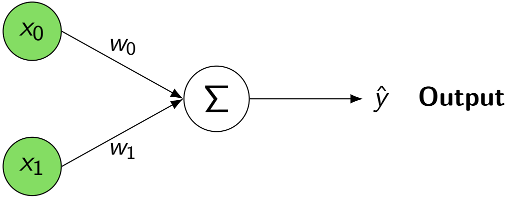
$$\mathbf{a} = \begin{bmatrix} \frac{1}{1+e^{-0.9}} \\ \frac{1}{1+e^{-0.1}} \\ \frac{1}{1+e^{0.35}} \end{bmatrix} \approx \begin{bmatrix} 0.71 \\ 0.52 \\ 0.41 \end{bmatrix}$$

This vector **a** is the output of this layer.


Linear Model
x_0
w_0
x_1
w_1
Σ
$\hat{y}$
Output
Input

Linear Model vs Artificial Neural Networks

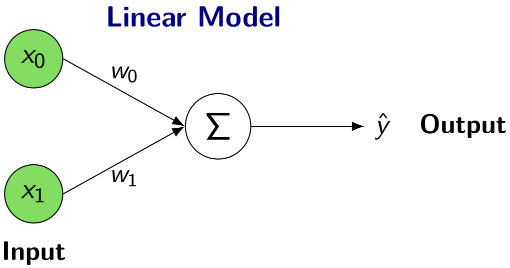
Linear Model



Input

$$\hat{y} = w_0 x_0 + w_1 x_1 = \sum w_i x_i$$

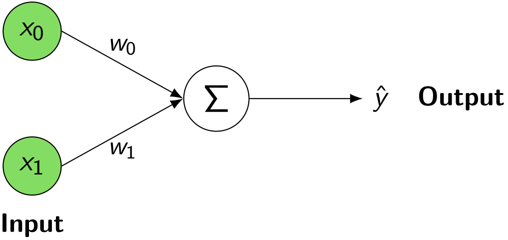
Linear Model vs Artificial Neural Networks



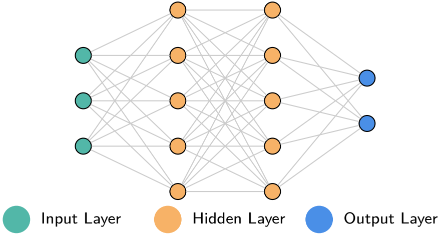
$$\hat{y} = w_0 x_0 + w_1 x_1 = \sum w_i x_i$$

A Linear Model is weighted sum of input features.

Linear Model



Neural Network Model

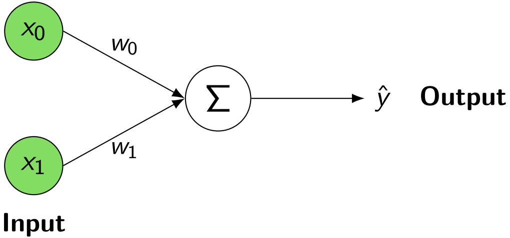


$$\hat{y} = w_0 x_0 + w_1 x_1 = \sum w_i x_i$$

A Linear Model is weighted sum of input features.

Linear Model vs Artificial Neural Networks

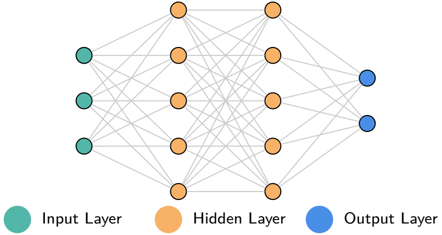
Linear Model



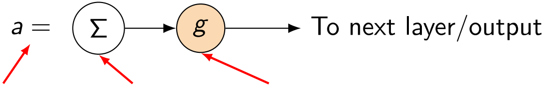
$$\hat{y} = w_0 x_0 + w_1 x_1 = \sum w_i x_i$$

A Linear Model is weighted sum of input features.

Neural Network Model



$$a = g\left(\sum w_i x_i\right)$$



A **Neuron** has a **linear** and a **nonlinear** operation

Why Do We Need Activation Functions?

The Problem: Stacking Linear Layers is Useless

Without a non-linear activation function, a deep network simply collapses into a single linear model, no matter how many layers it has.

Why Do We Need Activation Functions?

The Problem: Stacking Linear Layers is Useless

Without a non-linear activation function, a deep network simply collapses into a single linear model, no matter how many layers it has.

Without Non-Linearity:

- A layer is a linear operation: $\mathbf{W}\mathbf{x} + \mathbf{b}$.

Why Do We Need Activation Functions?

The Problem: Stacking Linear Layers is Useless

Without a non-linear activation function, a deep network simply collapses into a single linear model, no matter how many layers it has.

Without Non-Linearity:

- A layer is a linear operation: $\mathbf{W}\mathbf{x} + \mathbf{b}$.
- Stacking them:
  $L_2(L_1(\mathbf{x})) = \mathbf{W}_2(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$.

Why Do We Need Activation Functions?

The Problem: Stacking Linear Layers is Useless

Without a non-linear activation function, a deep network simply collapses into a single linear model, no matter how many layers it has.

Without Non-Linearity:

- A layer is a linear operation: $\mathbf{W}\mathbf{x} + \mathbf{b}$.
- Stacking them:
  $L_2(L_1(\mathbf{x})) = \mathbf{W}_2(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$.
- This simplifies to: $(\mathbf{W}_2\mathbf{W}_1)\mathbf{x} + (\mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2)$.

The Problem: Stacking Linear Layers is Useless

Without a non-linear activation function, a deep network simply collapses into a single linear model, no matter how many layers it has.

Without Non-Linearity:

- A layer is a linear operation: $\mathbf{W}\mathbf{x} + \mathbf{b}$.
- Stacking them:
  $L_2(L_1(\mathbf{x})) = \mathbf{W}_2(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$.
- This simplifies to: $(\mathbf{W}_2\mathbf{W}_1)\mathbf{x} + (\mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2)$.
- This is just another linear function: $\mathbf{W}'\mathbf{x} + \mathbf{b}'$.

Why Do We Need Activation Functions?

The Problem: Stacking Linear Layers is Useless

Without a non-linear activation function, a deep network simply collapses into a single linear model, no matter how many layers it has.

Without Non-Linearity:

- A layer is a linear operation: $\mathbf{W}\mathbf{x} + \mathbf{b}$.
- Stacking them:
  $L_2(L_1(\mathbf{x})) = \mathbf{W}_2(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$.
- This simplifies to: $(\mathbf{W}_2\mathbf{W}_1)\mathbf{x} + (\mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2)$.
- This is just another linear function: $\mathbf{W}'\mathbf{x} + \mathbf{b}'$.

A 100-layer linear network has the same power as a 1-layer network.

Why Do We Need Activation Functions?

The Problem: Stacking Linear Layers is Useless

Without a non-linear activation function, a deep network simply collapses into a single linear model, no matter how many layers it has.

Without Non-Linearity:

- A layer is a linear operation: $\mathbf{W}\mathbf{x} + \mathbf{b}$.
- Stacking them: $L_2(L_1(\mathbf{x})) = \mathbf{W}_2(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$.
- This simplifies to: $(\mathbf{W}_2\mathbf{W}_1)\mathbf{x} + (\mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2)$.
- This is just another linear function: $\mathbf{W}'\mathbf{x} + \mathbf{b}'$.

A 100-layer linear network has the same power as a 1-layer network.

With Non-Linearity (g):

Why Do We Need Activation Functions?

The Problem: Stacking Linear Layers is Useless

Without a non-linear activation function, a deep network simply collapses into a single linear model, no matter how many layers it has.

Without Non-Linearity:

- A layer is a linear operation: $\mathbf{W}\mathbf{x} + \mathbf{b}$.
- Stacking them: $L_2(L_1(\mathbf{x})) = \mathbf{W}_2(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$.
- This simplifies to: $(\mathbf{W}_2\mathbf{W}_1)\mathbf{x} + (\mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2)$.
- This is just another linear function: $\mathbf{W}'\mathbf{x} + \mathbf{b}'$.

A 100-layer linear network has the same power as a 1-layer network.

With Non-Linearity (g):

- The equation becomes: $g(\mathbf{W}_2 g(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2)$.

Why Do We Need Activation Functions?

The Problem: Stacking Linear Layers is Useless

Without a non-linear activation function, a deep network simply collapses into a single linear model, no matter how many layers it has.

Without Non-Linearity:

- A layer is a linear operation: $\mathbf{W}\mathbf{x} + \mathbf{b}$.
- Stacking them: $L_2(L_1(\mathbf{x})) = \mathbf{W}_2(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$.
- This simplifies to: $(\mathbf{W}_2\mathbf{W}_1)\mathbf{x} + (\mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2)$.
- This is just another linear function: $\mathbf{W}'\mathbf{x} + \mathbf{b}'$.

A 100-layer linear network has the same power as a 1-layer network.

With Non-Linearity (g):

- The equation becomes: $g(\mathbf{W}_2 g(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2)$.
- This function **cannot** be simplified.

The Problem: Stacking Linear Layers is Useless

Without a non-linear activation function, a deep network simply collapses into a single linear model, no matter how many layers it has.

Without Non-Linearity:

- A layer is a linear operation: $\mathbf{Wx} + \mathbf{b}$.
- Stacking them: $L_2(L_1(\mathbf{x})) = \mathbf{W}_2(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$.
- This simplifies to: $(\mathbf{W}_2\mathbf{W}_1)\mathbf{x} + (\mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2)$.
- This is just another linear function: $\mathbf{W}'\mathbf{x} + \mathbf{b}'$.

A 100-layer linear network has the same power as a 1-layer network.

With Non-Linearity (g):

- The equation becomes: $g(\mathbf{W}_2 g(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2)$.
- This function **cannot** be simplified.
- This allows the network to learn arbitrarily complex, "wiggly" functions.

Representation Power of Multi-Layer Perceptron

The Universal Approximation Theorem

Theorem (Cybenko 1989, Hornik 1991)

Let $\sigma(\cdot)$ be a non-constant, bounded, and continuous activation function (e.g., Sigmoid, Tanh, ReLU).
Then, for any continuous function $f(x)$ defined on a compact subset of $\mathbb{R}^n$ and for any error tolerance $\epsilon > 0$, there exists a Neural Network with **one hidden layer** containing a finite number of neurons that can approximate $f(x)$ such that:

$$|F(x) - f(x)| < \epsilon \quad \forall x$$

The Universal Approximation Theorem

Theorem (Cybenko 1989, Hornik 1991)

Let $\sigma(\cdot)$ be a non-constant, bounded, and continuous activation function (e.g., Sigmoid, Tanh, ReLU).
Then, for any continuous function $f(x)$ defined on a compact subset of $\mathbb{R}^n$ and for any error tolerance $\epsilon > 0$, there exists a Neural Network with **one hidden layer** containing a finite number of neurons that can approximate $f(x)$ such that:

$$|F(x) - f(x)| < \epsilon \quad \forall x$$

Implication:

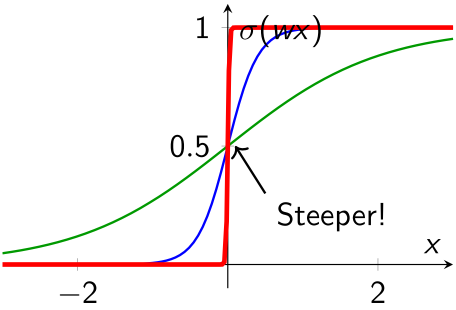
- Neural Networks are *universal function approximators*.
- In theory, a simple 2-layer network (1 hidden layer) can solve *any* problem, given enough neurons.

Controlling Steepness with Weight (w)

The Mechanism:

- Consider the sigmoid: $\sigma(w \cdot x)$.
- The weight w acts as a "gain" factor.
- **Small** w**:** The function is lazy and linear near the origin.
- **Large** w**:** The function transitions rapidly from 0 to 1.
- As $w \to \infty$, the sigmoid converges to a hard step.

Effect of Weight magnitude

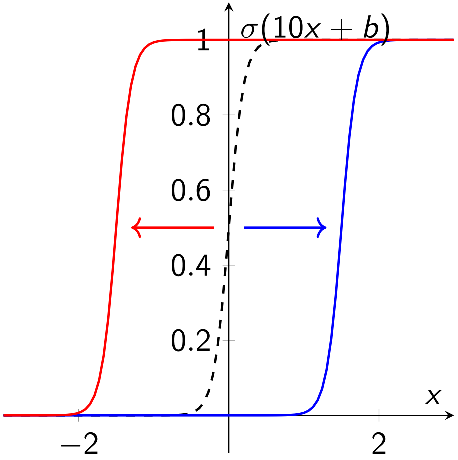


Controlling Position with Bias (b)

The Mechanism:

- Now consider: $\sigma(w \cdot x + b)$.
- The "step" occurs when the input to the sigmoid is 0.
- equation: $wx + b = 0 \implies x = -b/w$.
- **Interpretation:** The center of the step is shifted to position $s = -b/w$.
- This allows us to place the "switch" anywhere on the x-axis.

Shifting the Step ($w = 10$)



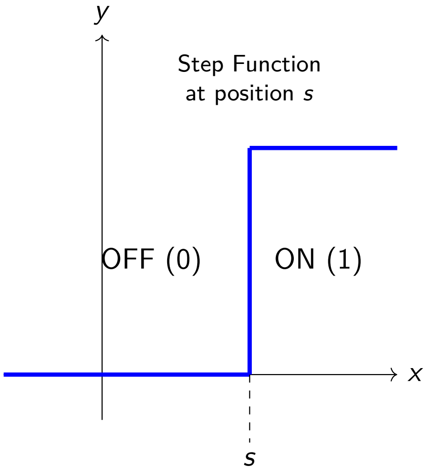
Creating the "Hard" Step

The Limit Definition:

- By combining a very large weight ($w \to \infty$) and a specific bias, we can simulate a Heaviside Step Function $H(x)$.
- We define the step position as s.
- We set weights such that $b = -w \cdot s$.

$$\lim_{w \to \infty} \sigma(w(x - s)) = \begin{cases} 0 & \text{if } x < s \\ 1 & \text{if } x > s \end{cases}$$

Why this matters: This creates a "switch" that turns ON at exactly position $x = s$. We will use pairs of these switches to build "bumps."



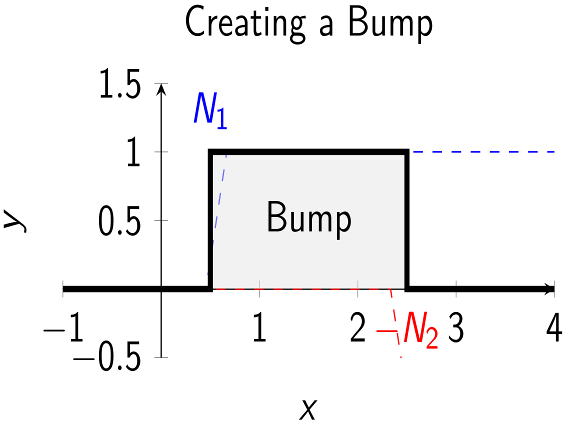
Constructing a "Bump"

The Tower Construction:

1. Take Neuron 1 with step at s_1.
2. Take Neuron 2 with step at s_2.
3. Subtract Neuron 2 from Neuron 1.

$$h(x) = \sigma(w(x - s_1)) - \sigma(w(x - s_2))$$

Result: A rectangular function (a "bump") that is non-zero only between s_1 and s_2.

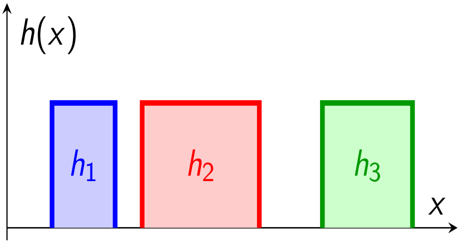


The Building Blocks (Bumps)

From Step to Bump:

- Recall: Two neurons create one "bump" $h_j(x)$.
- We can create N such bumps, scattered across the input space.
- **Key Idea:** Each bump is local. It is zero everywhere except for a specific region.
- We can independently control:
  - **Position:** Where the bump sits (via biases b).
  - **Width:** How wide the bump is (via weights w).

Independent "Hidden" Units



Scaling and Summing to Fit

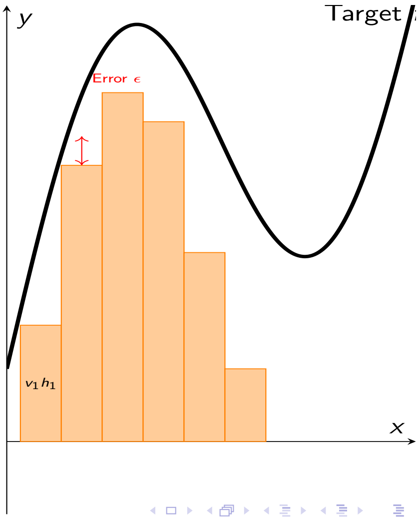
Fitting the Function:

- We now have a set of bumps $h_j(x)$.
- The final output neuron computes a weighted sum:

$$F(x) = \sum_{j=1}^{N} v_j \cdot h_j(x)$$

- **Role of Output Weights (v_j):** They scale the height of each bump to match the target function $f(x)$ at that location.
- **Result:** A "histogram-like" approximation (Riemann Sum) of the curve.
- As $N \to \infty$ (more bumps), the approximation error $\to 0$.

Weighted Sum $\approx f(x)$



Caveats and Practical Reality

Theory vs. Practice

The Universal Approximation Theorem is an **existence theorem**, not a constructive one.

- **Existence:** It says a network *exists*. It does not tell us how to find the weights.
- **Efficiency:** The theorem requires a potentially **infinite** number of neurons in the hidden layer as the function becomes more complex.
- **Optimization:** Finding the optimal parameters using Gradient Descent is non-trivial (local minima, saddle points).
- **Why Deep Learning?** Instead of one massive wide layer, we use *deep* layers (many layers). This allows us to represent complex functions more efficiently (with fewer total parameters).

Looking Ahead: From Existence to Discovery

Module 2: Optimization & Learning

In the next section, we will learn how to efficiently navigate the loss landscape to find these parameters using calculus and adaptive algorithms:

- **Backpropagation:** The engine for computing gradients.
- **Gradient Descent:** The fundamental update rule.
- **Advanced Optimizers:** Adam, AdaDelta, and RMSProp.