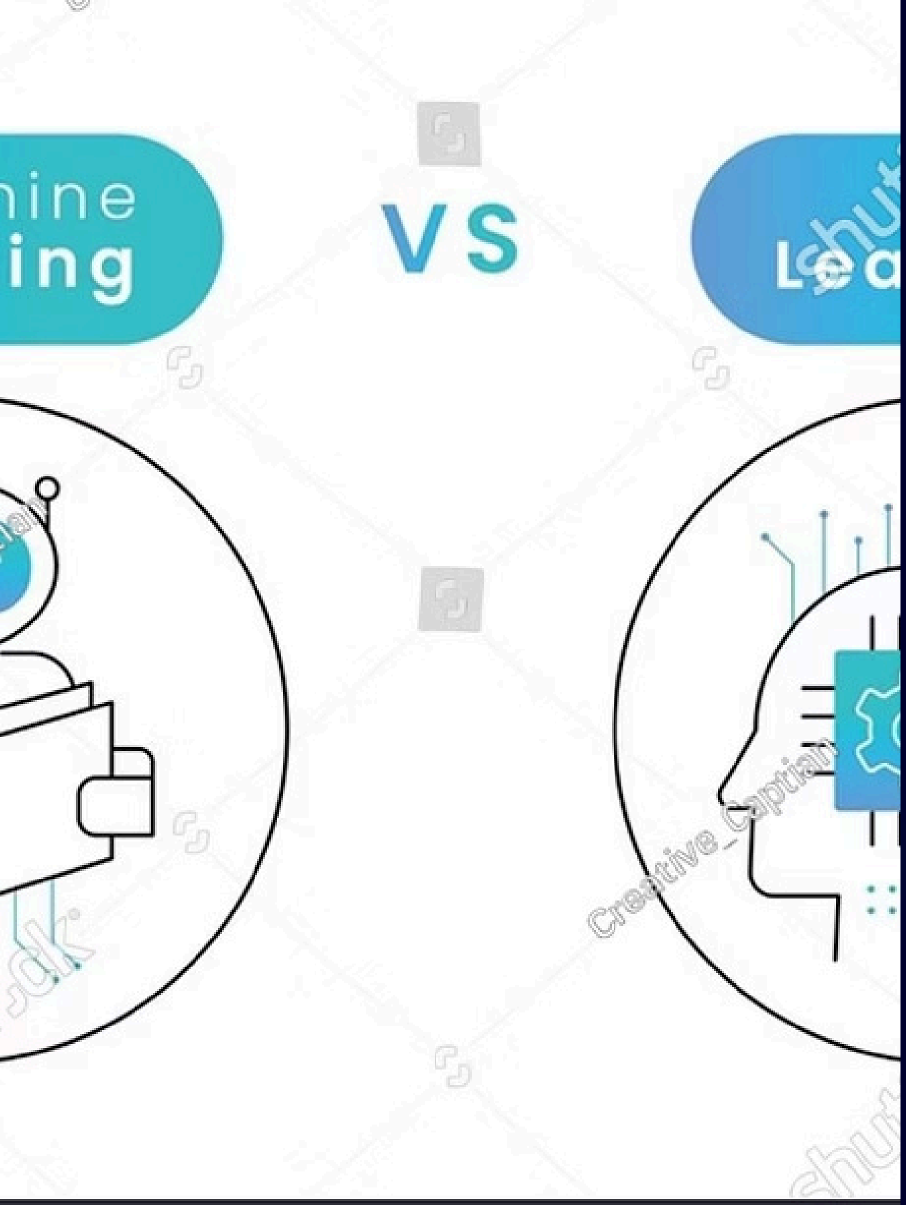
# Introduction to Neural Networks and Deep Learning

Machine Learning and Data Analysis

**Instructor:** Hernán Andrés Morales-Navarrete



## Why Neural Networks Matter

### The Challenge

Traditional machine learning models like linear regression and support vector machines often struggle when confronted with complex, high-dimensional data. These classical approaches rely on handcrafted features and linear or simple non-linear decision boundaries, which limit their capacity to capture intricate patterns in real-world datasets.

Neural networks revolutionize this paradigm by learning hierarchical, non-linear representations directly from raw data, eliminating the need for manual feature engineering.

### Physics Applications

#### Particle Physics

Classifying particle tracks at CERN's Large Hadron Collider

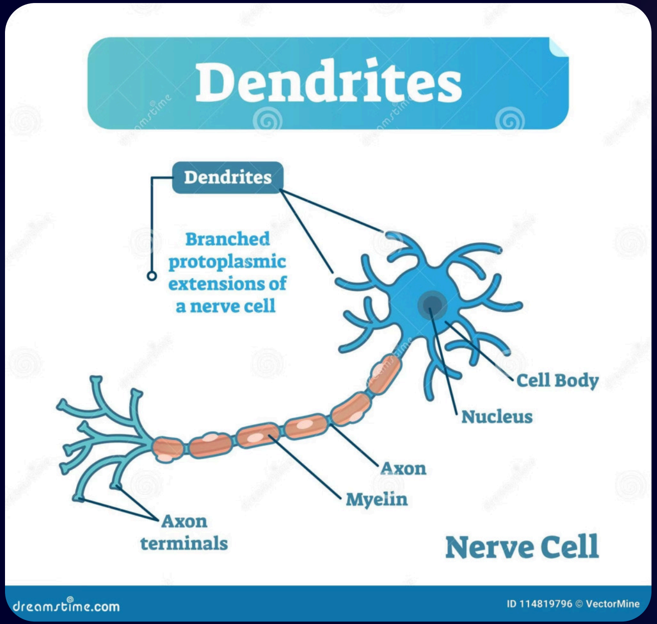
#### Materials Science

Predicting material properties from simulation data

#### Computational Biology

Image-based cell and tissue analysis

# From Neurons to Artificial Neurons



## Biological Inspiration

The human brain contains approximately 86 billion neurons, each performing a remarkably simple computation. A biological neuron integrates inputs from thousands of dendrites, applies a non-linear transformation, and produces an output signal through its axon.

This elegant mechanism inspired the development of artificial neural networks in the 1940s and continues to guide modern deep learning architectures.

## The Mathematical Abstraction

1 **Inputs → Features**

Data from previous neurons or raw features $x_i$

2 **Synaptic Weights → Parameters**

Learnable coefficients $w_i$ that amplify or diminish signals

3 **Activation → Non-linearity**

Function $f$ that introduces complexity

## The Artificial Neuron

$$y = f\left(\sum_i w_i x_i + b\right)$$

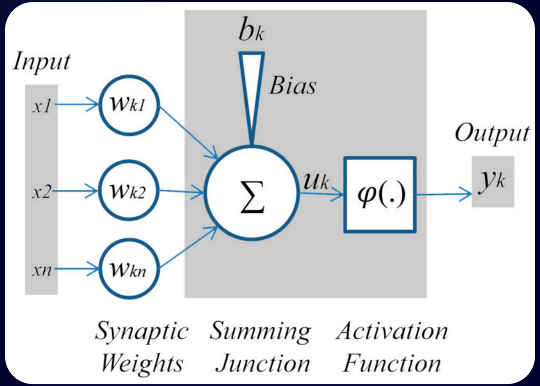
where $b$ is a bias term allowing the activation threshold to shift, and $f$ is a non-linear activation function.

# The Perceptron Model

## Historical Context

Introduced by Frank Rosenblatt in 1958, the perceptron represents the simplest form of a neural network. It performs binary classification by learning a linear decision boundary that separates two classes in feature space.

The perceptron's learning rule adjusts weights iteratively to minimize classification errors, laying the groundwork for modern gradient-based optimization.



## PyTorch Implementation

```
import torch

# Define XOR problem (not linearly separable)
x = torch.tensor([[0.,0.],[0.,1.],[1.,0.],[1.,1.]])
y = torch.tensor([[0.],[1.],[1.],[0.]])

# Single-layer perceptron
model = torch.nn.Sequential(
    torch.nn.Linear(2, 1),
    torch.nn.Sigmoid()
)

loss_fn = torch.nn.BCELoss()
opt = torch.optim.SGD(model.parameters(), lr=0.1)

for epoch in range(100):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    opt.zero_grad()
    loss.backward()
    opt.step()
```

**Critical Limitation:** The perceptron fails on the XOR problem because XOR is not linearly separable. This fundamental limitation motivated the development of multilayer networks capable of learning non-linear decision boundaries.

# Building a Neural Network

## Network Architecture

A neural network is organized into distinct layers, each serving a specific role in the transformation of input data into meaningful predictions. This layered structure enables the network to build increasingly abstract representations.

01

### Input Layer

Receives raw feature vectors $x \in \mathbb{R}^n$

02

### Hidden Layers

Apply learned transformations to extract features

03

### Output Layer

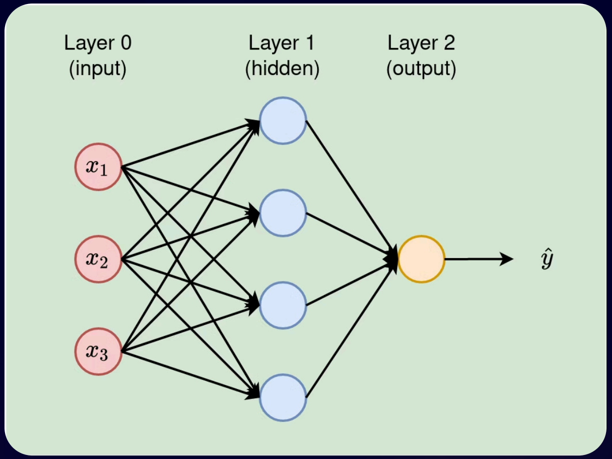
Produces final predictions $\hat{y}$

## The Forward Pass

Each layer computes an affine transformation followed by a non-linear activation:

$$a^{(l)} = f\left(W^{(l)}a^{(l-1)} + b^{(l)}\right)$$

where $W^{(l)}$ are weights, $b^{(l)}$ are biases, and $a^{(l)}$ is the activation at layer $l$.



## Activation Functions

Non-linear activation functions are essential for enabling networks to approximate complex functions. Without them, stacking multiple layers would be equivalent to a single linear transformation.

### Sigmoid

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

Outputs in (0,1), historically popular

### Tanh

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

Outputs in (-1,1), zero-centered

### ReLU

$\mathrm{ReLU}(x) = \max(0, x)$

Fast, efficient, now standard

# The Learning Process

## Training via Gradient Descent

Neural networks learn by iteratively adjusting their parameters to minimize a loss function $L(y, \hat{y})$ that quantifies the discrepancy between predictions and ground truth. This optimization process relies on computing gradients of the loss with respect to all parameters.

The fundamental update rule for each weight is:

$$w_i \leftarrow w_i - \eta \frac{\partial L}{\partial w_i}$$

where $\eta$ is the learning rate, a hyperparameter controlling the step size. Choosing an appropriate learning rate is crucial—too large and training diverges, too small and convergence is prohibitively slow.

## Backpropagation

Backpropagation is the algorithm that efficiently computes gradients by applying the chain rule recursively from the output layer back to the input. This elegant mathematical framework, formalized in the 1980s, made training deep networks computationally feasible.

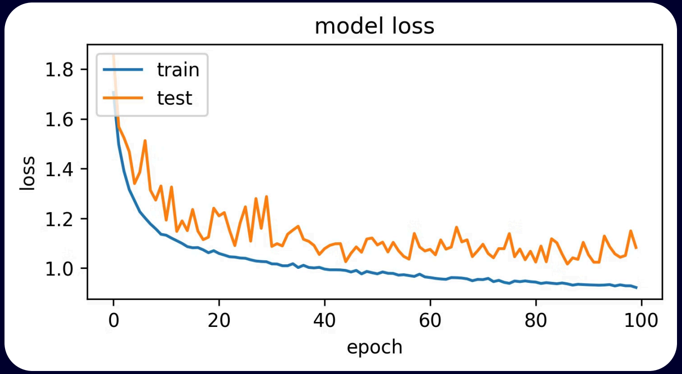
## PyTorch Implementation

```
loss_fn = torch.nn.MSELoss()
optimizer = torch.optim.SGD(
    model.parameters(),
    lr=0.01
)

for epoch in range(200):
    # Forward pass
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    # Backward pass
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if epoch % 50 == 0:
        print(f'Epoch {epoch}: Loss = {loss.item():.4f}')
```



The loss typically decreases exponentially in early epochs before plateauing as the model approaches a local minimum. Monitoring this curve helps diagnose training issues like overfitting or inadequate model capacity.

# Deep Neural Networks

## What Makes a Network "Deep"?

Depth refers to the number of hidden layers in a network, with "deep" networks typically containing three or more. The power of depth lies in **hierarchical representation learning**—each successive layer builds upon the features extracted by previous layers, enabling the network to capture increasingly abstract concepts.

Consider image recognition: early layers detect simple edges and textures, middle layers recognize shapes and parts, while deep layers identify complex objects and scenes. This compositional hierarchy mirrors how the human visual cortex processes information.

Mathematically, depth provides exponential expressiveness: a deep network with relatively few neurons per layer can represent functions that would require exponentially many neurons in a shallow architecture.

### Layer 1: Low-Level Features

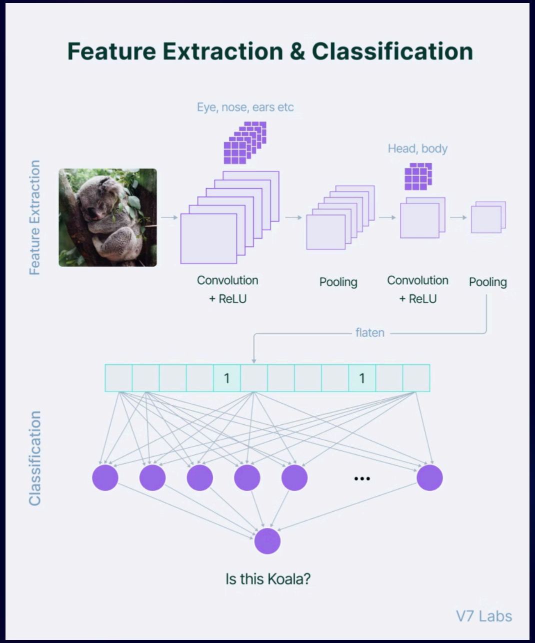
Edge detection, color gradients, basic textures

### Layer 2: Mid-Level Features

Corners, contours, simple shapes, patterns

### Layer 3: High-Level Features

Object parts, complex structures, semantic concepts



"The key idea behind deep learning is that representations should be learned automatically from data, and that deeper representations tend to be more abstract and invariant to nuisances in the input."

# Example: A Simple DNN in PyTorch

## Building a Fully Connected Deep Network

Let's construct a three-layer feedforward neural network for regression or classification tasks. This architecture demonstrates the modular design philosophy of modern deep learning frameworks.

```
import torch.nn as nn

class SimpleNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(10, 64),    # Input: 10 features
            nn.ReLU(),
            nn.Linear(64, 32),    # Hidden layer 1
            nn.ReLU(),
            nn.Linear(32, 1)      # Output: 1 prediction
        )

    def forward(self, x):
        return self.layers(x)

# Instantiate and inspect
model = SimpleNet()
print(model)
print(f'Total parameters: {sum(p.numel() for p in model.parameters())}')
```

### Architecture Details

1 **Input Layer**
10 → 64 neurons
704 parameters

2 **Hidden Layer**
64 → 32 neurons
2,080 parameters

3 **Output Layer**
32 → 1 neuron
33 parameters

**Total:** 2,817 learnable parameters

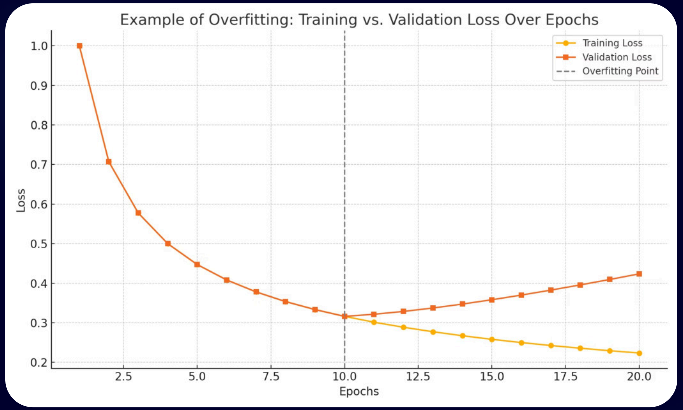
ReLU activations introduce non-linearity without the vanishing gradient problems that plague sigmoid and tanh functions, making them the default choice for modern deep networks.

## Overfitting and Regularization

### The Bias–Variance Trade-Off

Deep neural networks with thousands or millions of parameters possess enormous capacity to memorize training data, including noise and outliers. This phenomenon, called **overfitting**, results in excellent training performance but poor generalization to unseen data.

The bias-variance trade-off captures this tension: simple models (high bias) underfit the data, while complex models (high variance) overfit. Our goal is finding the sweet spot where the model captures true patterns without memorizing noise.



The telltale sign of overfitting: training loss continues decreasing while validation loss increases, indicating the model is learning dataset-specific quirks rather than general patterns.

### Regularization Techniques

#### Dropout

Randomly deactivates neurons during training, forcing the network to learn robust, redundant representations

#### Weight Decay (L2)

Penalizes large weights by adding $\lambda\|w\|^2$ to the loss function

#### Early Stopping

Halts training when validation performance stops improving

### Implementation

```
# Dropout layer (30% probability)
nn.Dropout(p=0.3)

# Weight decay in optimizer
optimizer = torch.optim.Adam(
 model.parameters(),
 lr=0.001,
 weight_decay=1e-4 # L2 regularization
)
```

# Neural Networks in Physics: Applications and Future Directions

## Current Applications

### Particle Physics

Event classification and anomaly detection in collider data. Deep learning models process millions of collision events at CERN, identifying rare particle signatures that could reveal new physics beyond the Standard Model.

### Fluid Dynamics

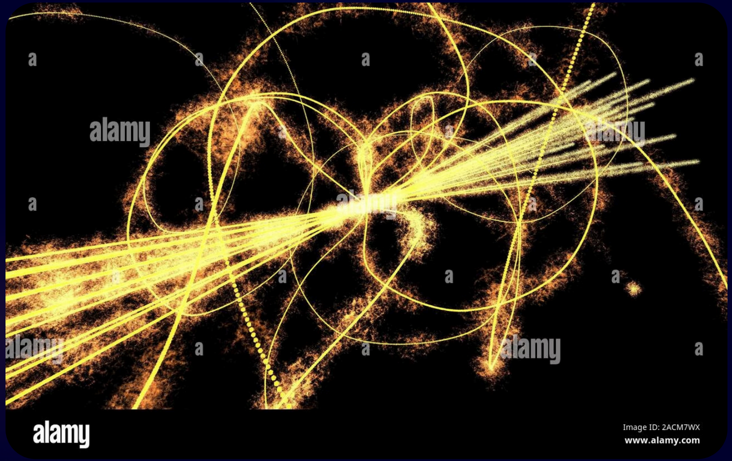
Turbulence prediction and flow pattern recognition. Neural networks can learn to approximate computationally expensive fluid simulations, accelerating design cycles in aerospace and climate modeling.

### Materials Science

Predicting material properties from atomic structure and simulation data, enabling rapid screening of novel materials for batteries, superconductors, and catalysts.

### Computational Biology

Cell segmentation, tracking, and classification in microscopy images. Automated analysis of terabyte-scale imaging datasets reveals dynamic cellular processes.



## Physics-Informed Neural Networks

An emerging paradigm combines data-driven learning with physical laws. By incorporating differential equations directly into the loss function, these networks learn solutions that satisfy both observational data and fundamental physics:

$$L = L_{\text{data}} + \lambda ||N(u_\theta) - 0||^2$$

where $N$ is a differential operator encoding physical laws (e.g., Navier-Stokes, Maxwell's equations).

## Looking Forward

**Next class:** Convolutional Neural Networks (CNNs) for spatial data—how architecture innovations enable state-of-the-art image recognition and beyond.