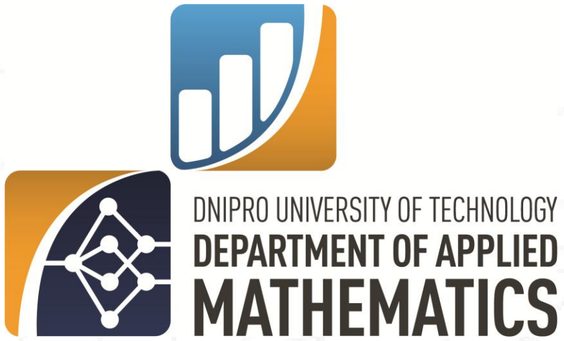


Introduction to Neural Networks

Course "Machine Learning: From Mathematical Foundations to Implementation in PYTHON"

Lecture by prof. Dmytro Babets

1. Limitations of Classical Machine Learning Models

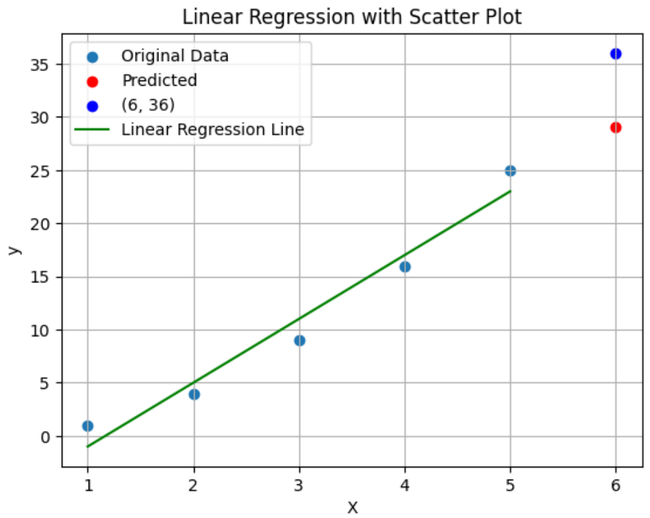
Linear Regression Model equation:

$$\hat{y} = w_0 + \sum_{i=1}^{n} w_i x_i$$

Optimization objective:

$$\min_{w} \; J(w) = \frac{1}{2m} \sum_{j=1}^{m} (\hat{y}^{(j)} - y^{(j)})^2$$

Limitation: only captures **linear relationships** between features and target.



Logistic Regression Model equation:

$$P(y = 1|x) = \sigma(w^T x + b) = \frac{1}{1 + e^{-(w^T x + b)}}$$

Decision boundary: **linear** in feature space.

Limitation: can't handle XOR-like (**non-linearly separable**) data.

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

No linear separator exists!

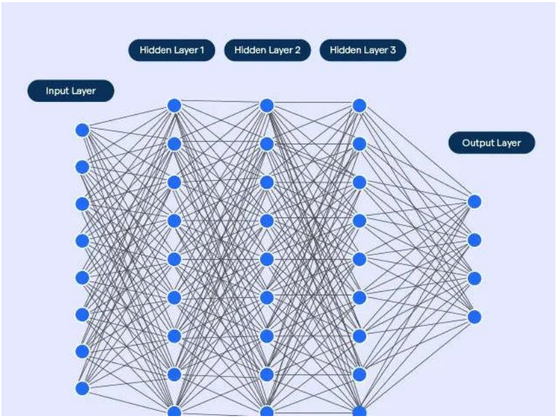
Support Vector Machines (SVMs).

Linear SVM objective:

$$\min_{w,b} \frac{1}{2}\|w\|^2 \quad \text{s.t.} \quad y_i(w^T x_i + b) \geq 1$$

Kernel trick can help, but requires manual choice of transformation

Limitation: can't automatically discover **hierarchical features** — unlike deep networks.

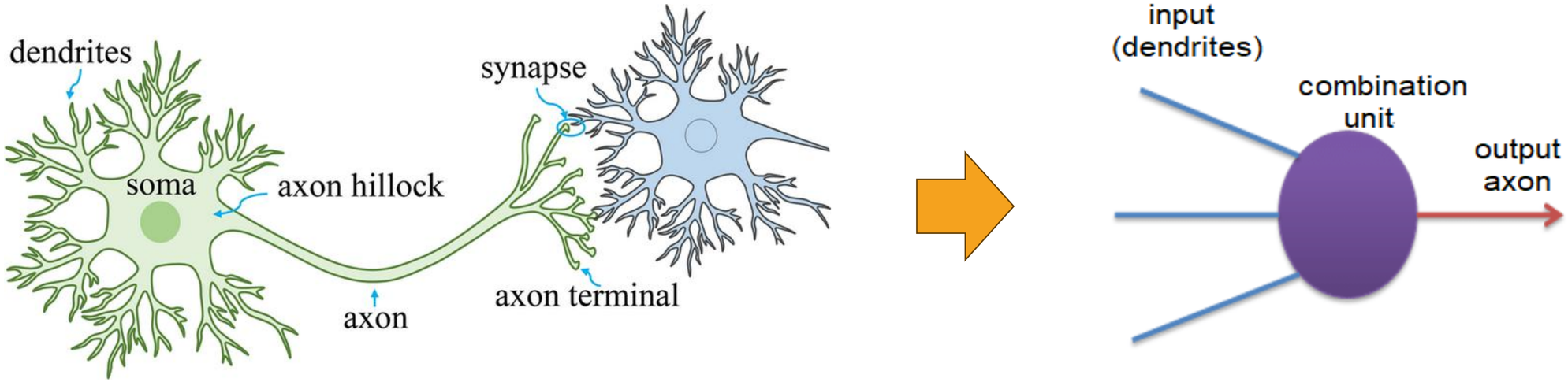


Biological Inspiration vs. Mathematical Abstraction

Key idea:

Instead of designing explicit features and transformations, **let the model learn internal representations** from data — inspired by how biological neurons process information.

- Neurons are basic units of the brain and nervous system.
- Receive input signals via dendrites, process them, and produce an output through the axon.
- A neuron “fires” if total stimulation exceeds a certain threshold.
- Synaptic connections have different strengths → represent weights in the artificial model.



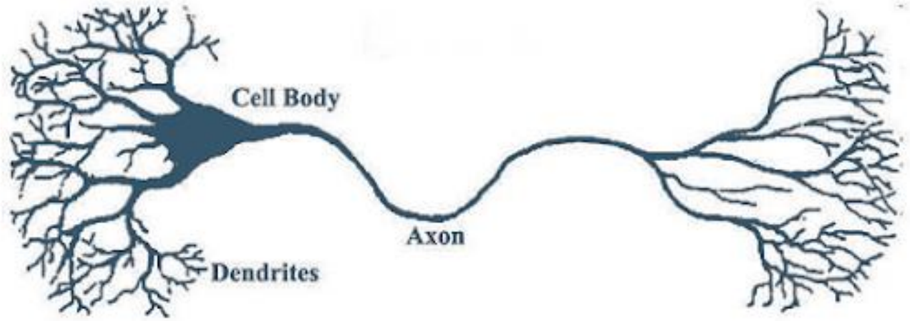
Mathematical abstraction of a neuron

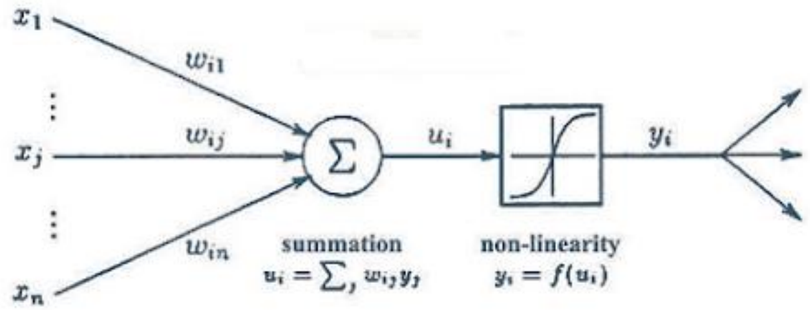
- Artificial neuron = simplified computational model.
- Each input x_i has an associated weight w_i.
- The neuron computes: $z = \sum_{i=1}^{n} w_i x_i + b$
- Output determined by **activation function**: $y=f(z)$

Computational abstraction: $y = f\left(\sum_{i=1}^{n} w_i x_i + b\right),$

where f is an activation function (Step, sigmoid, ReLU, etc).

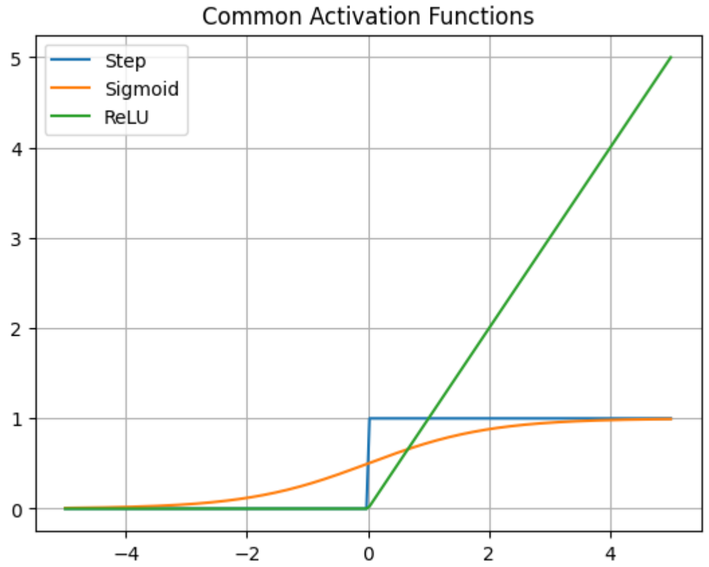
In the artificial neuron, we replace biological signals with numerical values. Each connection has a weight that measures how strong that signal is. The neuron calculates a weighted sum of its inputs and adds a bias term. Then we pass the result through an activation function to decide the neuron's output. This equation $y=f(\sum w_i x_i+b)$ - is the mathematical heart of every neural network.





Activation functions – making neurons nonlinear

Function	Formula	Description
Step	*f(z)=1 if z≥0; 0 otherwise*	Binary decision, perceptron model
Sigmoid	$f(z) = \frac{1}{1+e^{-z}}$	Smooth transition, probabilistic output
ReLU	*f(z)=max(0,z)*	Efficient, dominant in deep learning



Activation functions introduce nonlinearity.

Without them, the entire network would be equivalent to one big linear transformation - useless for complex tasks.

The step function mimics early biological firing, the sigmoid gives smooth probabilistic outputs, and ReLU became standard for deep architectures because of computational efficiency and gradient stability.

From biology to computation

Biological neuron → conceptual inspiration.

Artificial neuron → mathematical and computational abstraction.

Core idea remains:

- Weighted inputs
- Summation
- Nonlinear activation

Forms the building block of all neural networks.

We don’t simulate biological details - we abstract them.

Yet the key principle remains: signals are combined, weighted, and passed through a nonlinear rule.

Stacking many such neurons gives rise to multilayer architectures capable of approximating almost any function.

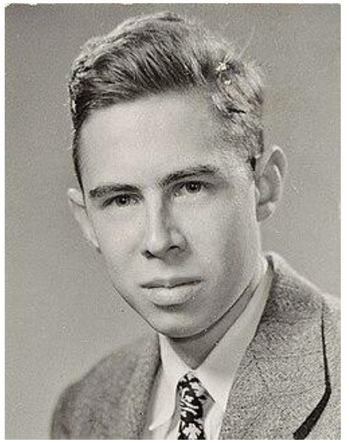
Every neural network — from a simple perceptron to a transformer — is built on the same equation:

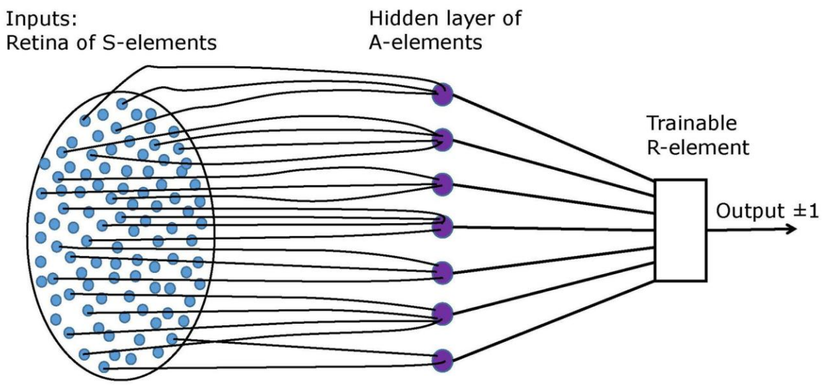
$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

The rest is architecture, scale, and training

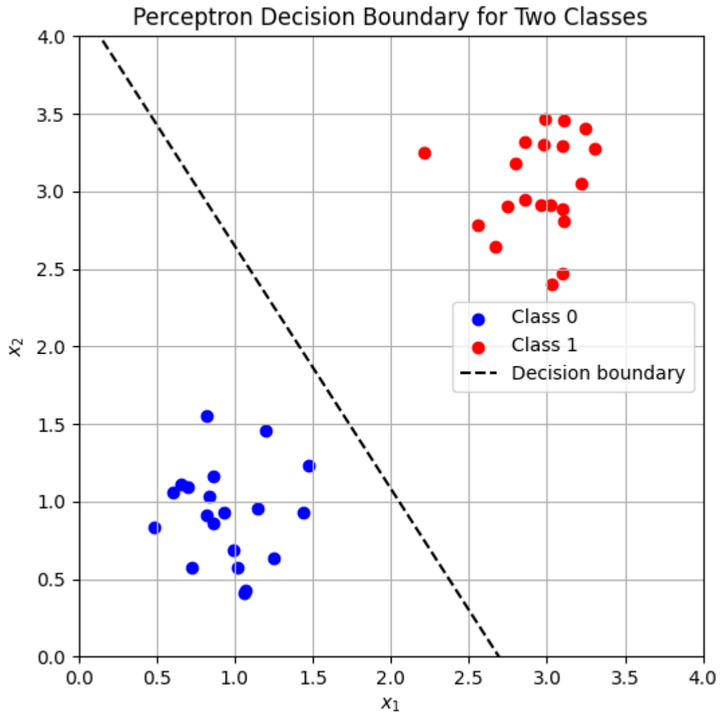
The Perceptron Model

- Introduced by Frank Rosenblatt (1958).
- The first trainable artificial neuron.
- Goal: find a linear decision boundary separating two classes.
- Inspired by biological neurons, but trained via data.





https://en.wikipedia.org/wiki/Frank_Rosenblatt



- Two Gaussian clusters represent two classes (blue and red).
- The perceptron adjusts weights until it finds a line that separates the classes.
- The dashed black line is the decision boundary: $w_1x_1+w_2x_2+b=0$

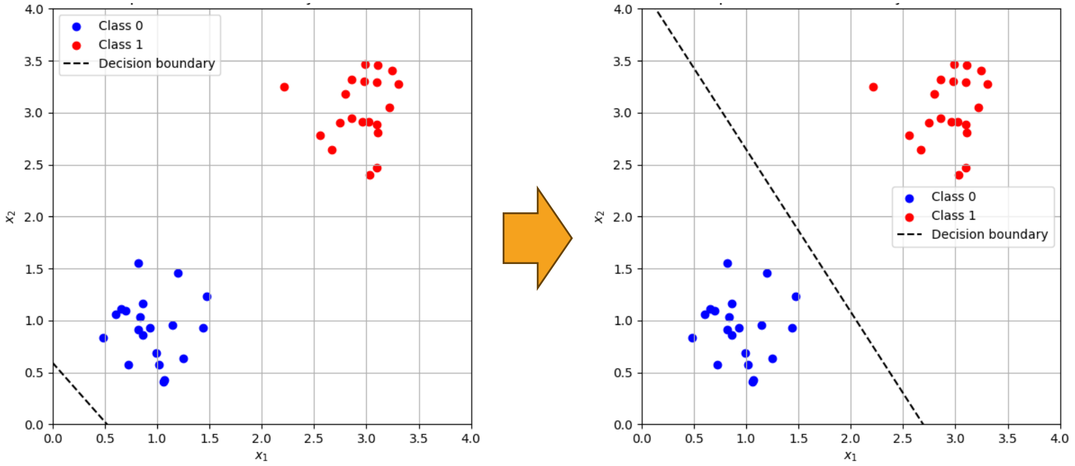
Perceptron mathematical model

For an input vector $x = [x_1, x_2, \ldots, x_n]$:

$$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b = w^Tx + b$$

Output:

$$\hat{y} = f(z) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases}$$



Training rule (Rosenblatt learning):

Given training examples $(x^{(i)}, y^{(i)})$, update after each sample:

$$w \leftarrow w + \eta(y - \hat{y})x, \qquad b \leftarrow b + \eta(y - \hat{y})$$

where:

- η – learning rate (0.0–1.0);
- y – true label;
- $\hat{y}$ - predicted output

Learning intuition:

If $y = 1,\ \hat{y} = 0$: increase weights for active features.

If $y = 0,\ \hat{y} = 1$: decrease weights.

Repeats for all samples → line moves until all points are correctly classified

! if data is linearly separable

Perceptron algorithm

Pseudocode:

```
initialize w, b = 0
for epoch in range(max_epochs):
    for each (x, y) in training_data:
        z = w · x + b
        y_hat = 1 if z >= 0 else 0
        w = w + η * (y - y_hat) * x
        b = b + η * (y - y_hat)
```

This simple loop implements the perceptron learning rule.

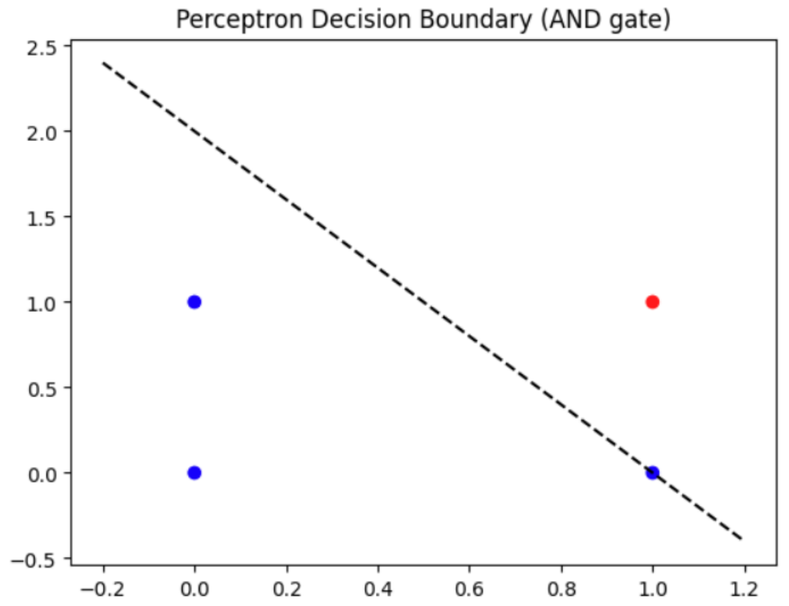
Note that it updates weights after **each sample** — this is an early form of stochastic gradient descent.

Python example:

```python
import numpy as np
import matplotlib.pyplot as plt

# Linearly separable dataset
X = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([0, 0, 0, 1])  # AND logic

# Parameters
w = np.zeros(2)
b = 0
eta = 0.1

for epoch in range(10):
    for xi, target in zip(X, y):
        z = np.dot(w, xi) + b
        y_hat = 1 if z >= 0 else 0
        w += eta * (target - y_hat) * xi
        b += eta * (target - y_hat)
print("Weights:", w, "Bias:", b)
```

```
Weights: [0.2 0.1] Bias: -0.20000000000000004
```

```python
x1 = np.linspace(-0.2, 1.2, 50)
x2 = -(w[0]*x1 + b)/w[1]
plt.scatter(X[:,0], X[:,1], c=y, cmap='bwr')
plt.plot(x1, x2, 'k--')
plt.title("Perceptron Decision Boundary (AND gate)")
plt.show()
```



This shows that the AND function is linearly separable — it can be perfectly divided by a straight line, so the perceptron can learn it

Limitations of the perceptron

- Works only if data are linearly separable.
- Fails on the XOR problem: No single line can separate (0,1) and (1,0) from (0,0) and (1,1).
- Output is discrete → non-differentiable → can’t use gradient methods directly.
- Leads to development of multi-layer perceptrons (MLP) and backpropagation.

Key takeaway

The perceptron is simple yet foundational.

It introduced:

- The concept of learnable weights.
- Iterative error correction.
- The idea of neurons as basic computational units.

But it also revealed the need for nonlinearity and depth — leading to **modern neural networks**

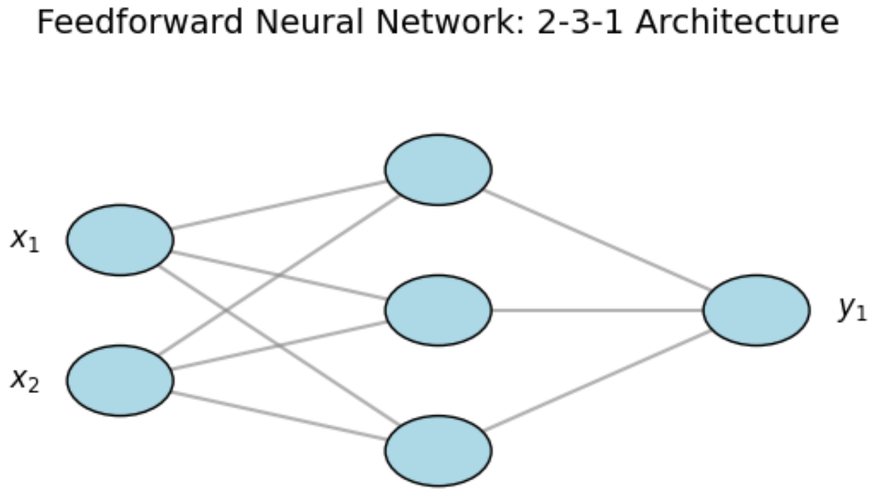
Multilayer Neural Networks. Feedforward Architecture

Input layer → Hidden layer(s) → Output layer

Each neuron computes: $a_j = f\left(\sum_i w_{ji} x_i + b_j\right)$

Each layer takes the output of the previous one, applies a linear transformation (weights and biases), and then a nonlinear activation function $f(z)$.

“Feedforward” means information moves strictly in one direction — no feedback loops.



Each circle represents a neuron.

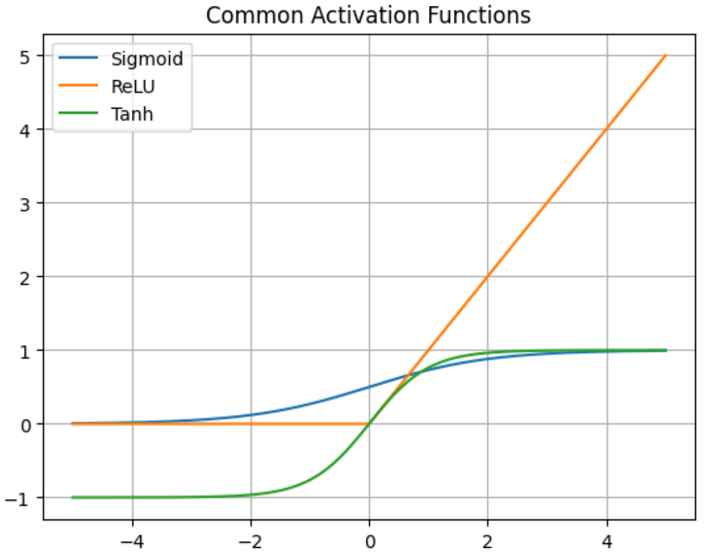
Gray lines show the flow of information (weights) from one layer to the next.

The layout [2, 3, 1] corresponds to:

- Input layer: 2 neurons (x_1, x_2)
- Hidden layer: 3 neurons
- Output layer: 1 neuron (y_1)

Activation Functions

Function	Formula	Description
Tanh	$f(z)=tanh(z)$	Zero-centered
Sigmoid	$f(z) = \frac{1}{1+e^{-z}}$	Smooth transition, probabilistic output
ReLU	$f(z)=max(0,z)$	Efficient, dominant in deep learning



Nonlinearity is key!

Without a nonlinear activation, multiple layers would collapse into a **single linear transformation**.

Differentiability and Gradient Descent

To train a network, we minimize a **loss function** — typically Mean Squared Error or Cross-Entropy.

1. Mean Squared Error (MSE)
Used mainly for **regression problems** (continuous outputs).

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Where:
y_i — true target value
$\hat{y}_i$ — predicted output of the network
N — number of training examples

The MSE measures the average squared difference between predictions and true values.
Minimizing it makes the network's outputs get closer to actual targets.
It's smooth and differentiable — convenient for gradient-based optimization.

2. Cross-Entropy Loss
Used for classification problems.
(a) Binary classification:

$$\mathcal{L}_{\text{CE}} = -\frac{1}{N}\sum_{i=1}^{N}\Big[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\Big]$$

(b) Multi-class classification:

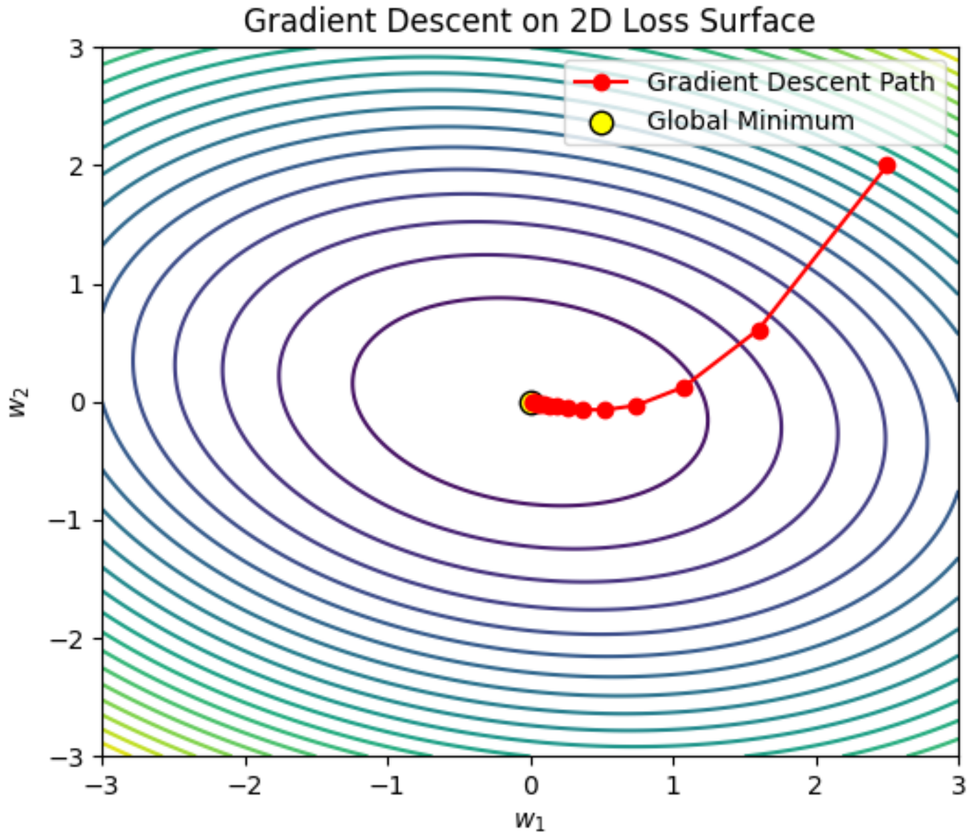
$$\mathcal{L}_{\text{CE}} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik}\log(\hat{y}_{ik})$$

Where:
$y_{ik} = 1$, if sample i belongs to class k, otherwise 0
$\hat{y}_{ik}$ - predicted probability that sample i belongs to class k
K - number of classes

Cross-entropy penalizes confident but wrong predictions. It measures how well the predicted probability distribution $\hat{y}$ matches the true distribution y. It's derived from information theory — minimizing it corresponds to maximizing likelihood

We adjust each weight in the direction that reduces the loss, using its gradient: $w_{ij}^{(new)} = w_{ij}^{(old)} - \eta\frac{\partial \mathcal{L}}{\partial w_{ij}}$

Differentiability and Gradient Descent



This plot shows a simple quadratic loss surface.

Each contour line represents equal loss values — like elevation on a map.

The red path shows how the algorithm gradually updates the parameters (w_1, w_2) in the direction of the negative gradient, step by step, moving toward the global minimum.

The yellow dot is the minimum of the loss function.

We adjust each weight in the direction that reduces the loss, using its gradient.

That’s the principle of gradient descent.

Backpropagation

Backpropagation is *not* part of the network architecture.

It's an **algorithm** used to train the feedforward network — to *adjust weights and biases* so that the network minimizes a loss function.

Backpropagation efficiently applies the **chain rule of calculus** to compute the gradient of the loss $\mathcal{L}$ with respect to every weight in the network:

$$\frac{\partial L}{\partial w_{ji}} = \frac{\partial L}{\partial a_j} \cdot \frac{\partial a_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ji}} \quad \Rightarrow \quad \frac{\partial L}{\partial w_{ji}} = \delta_j \, x_i \quad \text{where} \quad \delta_j = \frac{\partial L}{\partial a_j} \cdot f'(z_j)$$ - is the **error term** for neuron j.

$z_j = \sum_i w_{ji} \, x_i + b_j$ is the *weighted sum* before activation.

Step-by-step Interpretation:

$\frac{\partial L}{\partial a_j}$ — how sensitive the loss is to the output of neuron j (this comes from the next layer - it's *back-propagated*);

$\frac{\partial a_j}{\partial z_j} = f'(z_j)$ — how the activation function changes with its input.

$\frac{\partial z_j}{\partial w_{ji}} = x_i$ — because z_j depends linearly on its input weights.

Propagating the Error Backward (Summary)

Each neuron produces output $a_j = f\left(\sum_i w_{ji}\, x_i + b_j\right)$.

During learning, backpropagation finds how each w_{ji} affects the total loss.

Using the chain rule, we compute

$$\frac{\partial L}{\partial w_{ji}} = \delta_j x_i$$

and update weights to reduce the loss.

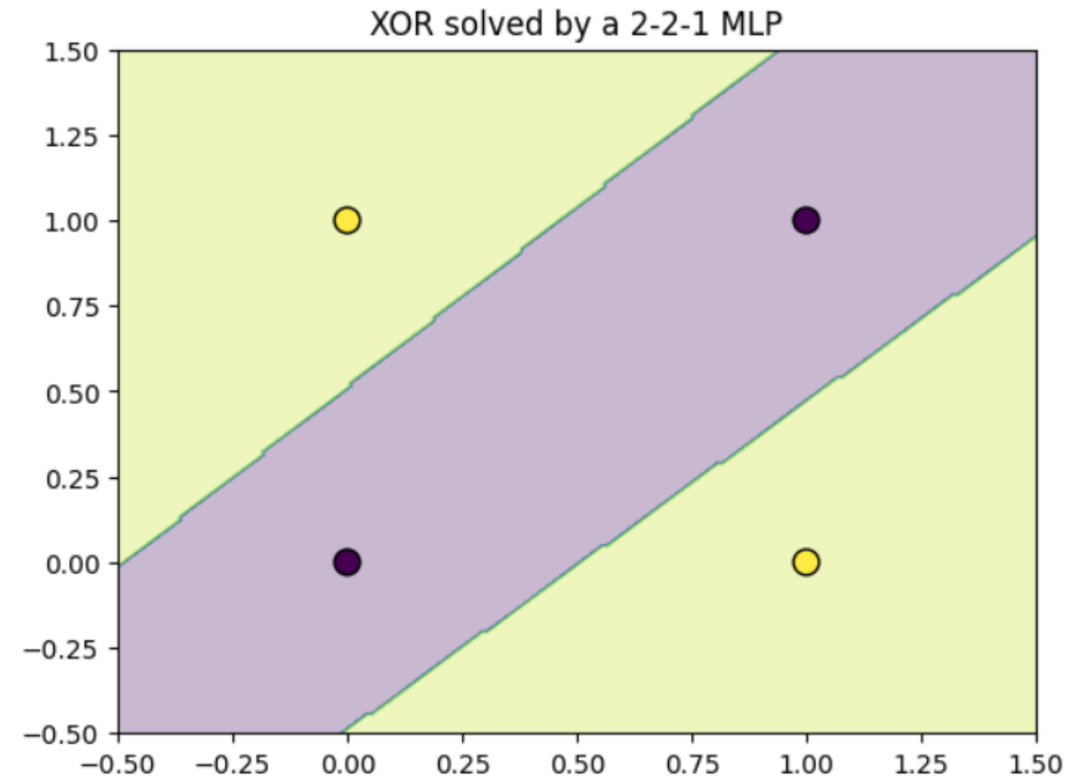
Errors (δ_j) move backward through the network — hence the name backpropagation.

Practical Example – XOR Classification

We can now solve the XOR problem using a small **Multilayer Perceptron (MLP)**.

```python
import numpy as np
from sklearn.neural_network import MLPClassifier
import matplotlib.pyplot as plt

# XOR dataset
X = np.array([[0,0], [0,1], [1,0], [1,1]])
y = np.array([0, 1, 1, 0])

# Train MLP with one hidden layer of 2 neurons
model = MLPClassifier(hidden_layer_sizes=(2,), activation='tanh',
                      solver='sgd', learning_rate_init=0.1, max_iter=100000)
model.fit(X, y)

# Predictions
print("Predictions:", model.predict(X))

# Plot decision regions
xx, yy = np.meshgrid(np.linspace(-0.5, 1.5, 200),
                     np.linspace(-0.5, 1.5, 200))
Z = model.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.3)
plt.scatter(X[:,0], X[:,1], c=y, edgecolor='k', s=100)
plt.title("XOR solved by a 2-2-1 MLP")
plt.show()
```

```
Predictions: [0 1 1 0]
```



The MLP correctly learns the XOR function. The plot shows a nonlinear decision boundary dividing the (0,0)/(1,1) vs (0,1)/(1,0) regions.

Key Takeaways

- Hidden layers allow **nonlinear feature transformations**.
- Activation functions provide **nonlinearity** and gradient flow.
- Backpropagation + gradient descent = core of deep learning.
- Even a small MLP can solve problems unsolvable by a single perceptron.

Scikit-learn Neural Network Classes Overview

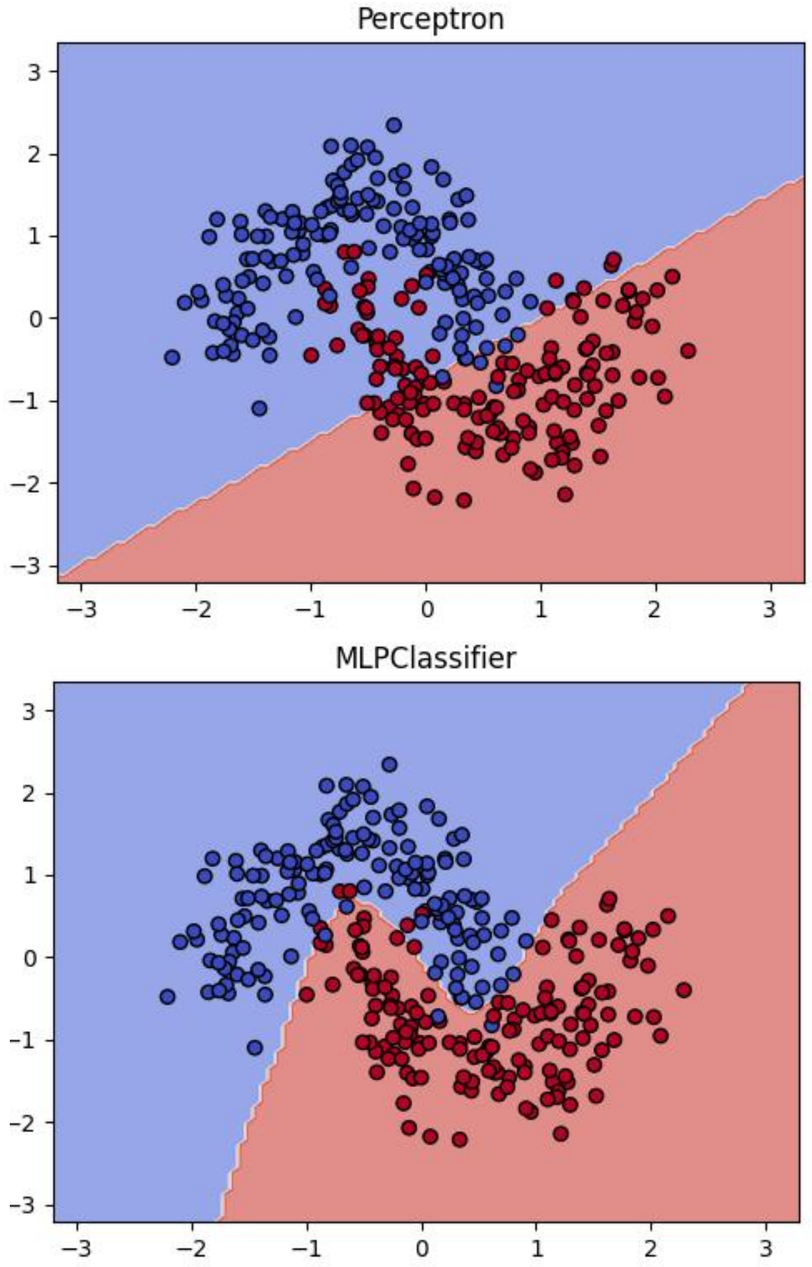
Aspect	Perceptron (sklearn.linear_model.Perceptron)	MLPClassifier / MLPRegressor (sklearn.neural_network)
Purpose	Implements the classic single-layer perceptron (linear classifier). Suitable for linearly separable data.	Implements a Multi-Layer Perceptron trained by backpropagation. Supports multiple hidden layers, nonlinear activation, and various solvers.
Type	Linear binary/multiclass classifier	Nonlinear classifier (MLPClassifier) or regressor (MLPRegressor)
Training algorithm	Stochastic Gradient Descent (SGD) using the perceptron learning rule	Backpropagation with different optimizers: 'adam', 'sgd', or 'lbfgs'
Loss function	Hinge-like loss (similar to linear SVM)	For classifier: log-loss (cross-entropy); for regressor: squared loss
Activation	Sign function (implicitly)	'identity', 'logistic', 'tanh', 'relu'
Hidden layers	None — only direct connection from inputs to outputs	Configurable: e.g. hidden_layer_sizes=(100,) (1 hidden layer, 100 neurons)
Regularization	alpha — L2 penalty on weights	alpha — L2 regularization term (default 0.0001)
Learning rate	eta0 (initial learning rate); learning_rate can be 'constant', 'optimal', or 'invscaling'	learning_rate_init (default 0.001) and schedule: 'constant', 'invscaling', 'adaptive'
Solver	'sgd' (default)	'adam' (default), 'sgd', or 'lbfgs'
Epochs / Iterations	max_iter — number of passes over training data (default 1000)	max_iter — maximum iterations (default 200)
Batch size	Not applicable (uses online or mini-batch SGD implicitly)	batch_size — number of samples per gradient update (default 'auto')
Early stopping	No	early_stopping=True stops training if validation score does not improve
Shuffle	shuffle=True shuffles samples each epoch	shuffle=True also available
Momentum	Optional: momentum and nesterovs_momentum (if solver='sgd')	Also supported when using solver='sgd'
Learning rate control	learning_rate, eta0	learning_rate, learning_rate_init
Output layer activation	Implicit sign function	Determined by task:• 'softmax' (for classification)• Linear (for regression)
Methods	fit(X, y) — trainpredict(X) — labelsscore(X, y) — accuracypartial_fit(X, y) — online update	fit(X, y) — trainpredict(X) — predictpredict_proba(X) — class probabilitiesscore(X, y) — accuracy/R^2partial_fit(X, y) — incremental learning
Attributes after fitting	coef_ — weight matrixintercept_ — bias vectorclasses_ — class labels	coefs_ — list of weight matrices per layerintercepts_ — list of bias vectorsn_iter_, loss_, n_layers_, out_activation_
Initialization	Random weights	Xavier-like initialization internally
Scaling input data	Strongly recommended (use StandardScaler)	**Essential** for convergence and stable learning

Parameter descriptions (summary)

Parameter	Description	Common values / notes
hidden_layer_sizes	Tuple specifying number of neurons in each hidden layer.	(100,) = 1 hidden layer, 100 neurons; (50, 30, 10) = 3 layers
activation	Activation function for hidden layers.	'identity', 'logistic', 'tanh', 'relu'
solver	Optimization algorithm for weight updates.	'adam' (adaptive), 'sgd', 'lbfgs' (quasi-Newton)
alpha	L2 regularization parameter.	Smaller → less regularization.
learning_rate_init / eta0	Initial learning rate.	e.g., 0.01 or 0.001
max_iter	Maximum number of training iterations (epochs).	Increase if model doesn't converge.
tol	Tolerance for stopping criteria (min improvement).	Default 1e-4
batch_size	Number of samples per gradient update (for MLP only).	'auto' or integer
shuffle	Shuffle samples each epoch.	Default True
momentum	Momentum factor for gradient updates (if sgd).	Typical 0.9
early_stopping	Stop training when validation score stops improving.	Default False
validation_fraction	Fraction of training data used for validation when early_stopping=True.	Default 0.1
random_state	Seed for reproducibility.	Integer or None
verbose	Prints progress messages.	Default False

Example (Comparing Perceptron and MLP on Nonlinear Data)

```python
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
from sklearn.linear_model import Perceptron
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.inspection import DecisionBoundaryDisplay

# Data
X, y = make_moons(n_samples=300, noise=0.2, random_state=42)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Models
p = Perceptron(max_iter=1000, random_state=0)
p.fit(X_scaled, y)

mlp = MLPClassifier(hidden_layer_sizes=(50,), activation='relu', solver='adam', max_iter=5000, random_state=0)
mlp.fit(X_scaled, y)

# Visualization
fig, axes = plt.subplots(1, 2, figsize=(10, 4))
for model, title, ax in zip([p, mlp], ['Perceptron', 'MLPClassifier'], axes):
    DecisionBoundaryDisplay.from_estimator(model, X_scaled, response_method='predict', ax=ax, cmap='coolwarm', alpha=0.6)
    ax.scatter(X_scaled[:, 0], X_scaled[:, 1], c=y, edgecolor='k', cmap='coolwarm')
    ax.set_title(title)
plt.tight_layout()
plt.show()
```



In this example, we generate a nonlinearly separable dataset using make_moons() and compare the performance of a single-layer Perceptron with a Multilayer Perceptron (MLP).

By introducing hidden layers and nonlinear activations, MLPs can model complex, nonlinearly separable relationships that simple Perceptrons cannot.

НТУ "ДНІПРОВСЬКА ПОЛІТЕХНІКА"
КАФЕДРА ПРИКЛАДНОЇ МАТЕМАТИКИ

Let’s proceed to the practical exercises

link:
https://colab.research.google.com/drive/1UfWHQZKqWHOiA2No9-AdP3MZDYTqDiii?usp=sharing

REFERENCES

1. Aggarwal, C. C. (2018). Neural Networks and Deep Learning. Springer International Publishing. https://doi.org/10.1007/978-3-319-94463-0
2. Cohen, M. X. (2022). *Practical linear algebra for data science: From Core Concepts to Applications Using Python*. “O’Reilly Media, Inc.”
3. Cohen, M. X. (2021). *Linear algebra: theory, intuition, code*.