DEEP LEARNING & NEURAL NETWORKS

What is Deep Learning?

Deep Learning is a subfield of **Machine Learning (ML)** that uses **artificial neural networks (ANNs)** with many layers (hence *deep*) to model and solve complex problems — such as image recognition, natural language processing, and game playing.

Deep Learning automatically learns features from data, unlike traditional ML which needs feature engineering.

What is a Neural Network?

A **Neural Network** is a set of algorithms modeled after the human brain that is designed to recognize patterns. It interprets sensory data through a kind of machine perception, labeling, or clustering of raw input.

Neural networks are the backbone of **Deep Learning**.

Input Layer → Hidden Layers → Output Layer

Output = Activation(Weight * Input + Bias)

Components of Neural Networks

Component	Description
Neuron (Node)	Processes input and applies activation
Weight	Controls the influence of input data
Bias	Allows model to fit better
Activation Function	Adds non-linearity (e.g., ReLU, Sigmoid)
Layers	Groups of neurons – Input, Hidden, Output

What is a Perceptron?

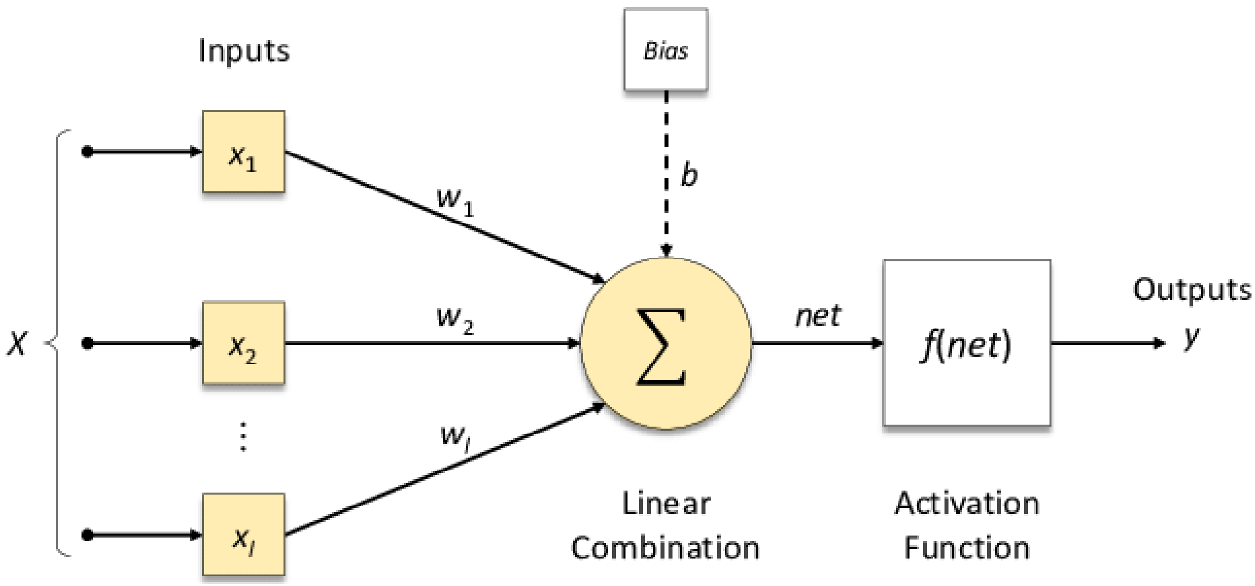
A **Perceptron** is the **simplest neural network** model. It's a type of **linear binary classifier** introduced by **Frank Rosenblatt** in 1958.

It maps input features to an output using: output = activation($w \cdot x + b$)

Perceptron Architecture

☐ Components:

- **Inputs ($x_1, x_2, ..., x_n$)**
- **Weights ($w_1, w_2, ..., w_n$)**
- **Bias (b)**
- **Activation Function** (usually step function for single-layer perceptron)
- **Output (y)**



Training Process (Step-by-Step)

1. **Initialize weights and bias** (randomly or zeros)
2. **Forward pass**: compute `output = activation(w·x + b)`
3. **Error Calculation**: `error = y_true - y_pred`
4. **Weight Update Rule**: w_new = w_old + learning_rate * error * x

b_new = b_old + learning_rate * error

Python Implementation: Perceptron for AND Gate:

```
import numpy as np


# Step activation function

def step_function(x):

    return 1 if x >= 0 else 0


# Perceptron class

class Perceptron:

    def __init__(self, input_size, learning_rate=0.1):

        self.weights = np.zeros(input_size)

        self.bias = 0

        self.lr = learning_rate


    def predict(self, x):

        summation = np.dot(x, self.weights) + self.bias

        return step_function(summation)


    def train(self, X, y, epochs=10):

        for epoch in range(epochs):

            print(f"\nEpoch {epoch+1}")

            for i in range(len(X)):

                y_pred = self.predict(X[i])

                error = y[i] - y_pred
```

```
        self.weights += self.lr * error * X[i]

        self.bias += self.lr * error

        print(f"Input: {X[i]}, Predicted: {y_pred}, Error: {error}, Weights: {self.weights}, Bias: {self.bias}")


# AND gate data

X = np.array([[0,0], [0,1], [1,0], [1,1]])

y = np.array([0, 0, 0, 1])


# Train perceptron

p = Perceptron(input_size=2)

p.train(X, y)


# Test predictions

print("\nFinal Predictions:")

for i in range(len(X)):

  print(f"{X[i]} --> {p.predict(X[i])}")
```

Output:

Epoch 1

Input: [0 0], Predicted: 0, Error: 0, Weights: [0. 0.], Bias: 0.0

Input: [0 1], Predicted: 0, Error: 0, Weights: [0. 0.], Bias: 0.0

Input: [1 0], Predicted: 0, Error: 0, Weights: [0. 0.], Bias: 0.0

Input: [1 1], Predicted: 0, Error: 1, Weights: [0.1 0.1], Bias: 0.1

...

Final Predictions:

[0 0] --> 0

[0 1] --> 0

[1 0] --> 0

[1 1] --> 1

What is a Multi-Layer Perceptron (MLP)?

A **Multi-Layer Perceptron (MLP)** is a **feedforward neural network** with one or more **hidden layers** between input and output layers. It can solve **non-linearly separable problems** like XOR.

MLP Architecture:

Input Layer → Hidden Layer(s) → Output Layer

Each neuron in one layer is **fully connected** to the next layer.

Training Process (Using Backpropagation)

1. **Initialize weights and biases**
2. **Forward pass**:
   - Compute activations layer by layer
3. **Loss calculation** (e.g., Mean Squared Error)
4. **Backward pass**:
   - Use chain rule to calculate gradients (backpropagation)
5. **Update weights** using optimizer (e.g., gradient descent)

Python Code: MLP for XOR (No Libraries):

```
import numpy as np



# Activation functions

def sigmoid(x):

    return 1 / (1 + np.exp(-x))
```

```python
def sigmoid_derivative(x):
    return x * (1 - x)


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


# Initialize parameters
input_size = 2
hidden_size = 2
output_size = 1
learning_rate = 0.5
epochs = 10000


# Weights and biases
np.random.seed(1)
W1 = np.random.uniform(size=(input_size, hidden_size))     # weights from input to hidden
b1 = np.zeros((1, hidden_size))                     # bias for hidden layer
W2 = np.random.uniform(size=(hidden_size, output_size))    # weights from hidden to output
b2 = np.zeros((1, output_size))                     # bias for output layer


# Training loop
for epoch in range(epochs):
```

```
# Forward pass
z1 = np.dot(X, W1) + b1
a1 = sigmoid(z1)

z2 = np.dot(a1, W2) + b2
a2 = sigmoid(z2)

# Compute error
error = y - a2
loss = np.mean(np.square(error))

# Backpropagation
d_output = error * sigmoid_derivative(a2)
d_hidden = d_output.dot(W2.T) * sigmoid_derivative(a1)

# Update weights and biases
W2 += a1.T.dot(d_output) * learning_rate
b2 += np.sum(d_output, axis=0, keepdims=True) * learning_rate
W1 += X.T.dot(d_hidden) * learning_rate
b1 += np.sum(d_hidden, axis=0, keepdims=True) * learning_rate

# Print loss occasionally
if epoch % 1000 == 0:
    print(f"Epoch {epoch} - Loss: {loss:.4f}")
```

```
# Final predictions

print("\nPredictions after training:")

for i in range(len(X)):

    z1 = np.dot(X[i], W1) + b1

    a1 = sigmoid(z1)

    z2 = np.dot(a1, W2) + b2

    a2 = sigmoid(z2)

    print(f"Input: {X[i]} → Output: {a2.round(3)}")
```

Sample Output:

```
Epoch 0 - Loss: 0.2565

...

Epoch 9000 - Loss: 0.0013

Epoch 10000 - Loss: 0.0011

Predictions after training:

Input: [0 0] → Output: [0.014]

Input: [0 1] → Output: [0.986]

Input: [1 0] → Output: [0.987]

Input: [1 1] → Output: [0.012]
```

What is an Activation Function?

An **activation function** introduces **non-linearity** into a neural network so it can learn complex patterns and relationships in data.

Without activation functions, a neural network would be **just a linear regression model**.

Common Activation Functions:

Function	Pros	Cons
Sigmoid	Smooth, probability output	Vanishing gradient, slow
Tanh	Zero-centered	Still suffers from gradient issues
ReLU	Fast, works well in practice	Dying ReLU problem
Leaky ReLU	Fixes ReLU dying issue	Small slope on negative side
ELU	Smoother than ReLU	Computationally expensive
Swish	Smooth, trainable	Slightly slower
Softmax	Outputs probabilities (multiclass)	Not used in hidden layers

When to Use What?

Layer Type	Common Activation
Hidden Layer	ReLU / Leaky ReLU / Swish
Output (binary)	Sigmoid
Output (multi-class)	Softmax
Regression Output	Linear (no activation)

Python Code to Visualize All:

```
import numpy as np

import matplotlib.pyplot as plt



x = np.linspace(-10, 10, 100)
```

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def tanh(x):
    return np.tanh(x)

def relu(x):
    return np.maximum(0, x)

def leaky_relu(x):
    return np.where(x > 0, x, x * 0.01)

def elu(x, alpha=1.0):
    return np.where(x >= 0, x, alpha * (np.exp(x) - 1))

def swish(x):
    return x * sigmoid(x)

def softmax(x):
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum()

# Plotting
plt.figure(figsize=(12, 8))
```

```python
plt.subplot(2, 3, 1)
plt.plot(x, sigmoid(x))
plt.title("Sigmoid")
plt.subplot(2, 3, 2)
plt.plot(x, tanh(x))
plt.title("Tanh")
plt.subplot(2, 3, 3)
plt.plot(x, relu(x))
plt.title("ReLU")

plt.subplot(2, 3, 4)
plt.plot(x, leaky_relu(x))
plt.title("Leaky ReLU")

plt.subplot(2, 3, 5)
plt.plot(x, elu(x))
plt.title("ELU")

plt.subplot(2, 3, 6)
plt.plot(x, swish(x))
plt.title("Swish")

plt.tight_layout()
plt.show()
```

Forward & Backward Propagation in Neural Networks

These are the **two core processes** in training a neural network:

- **Forward Propagation**: Calculates output predictions.
- **Backward Propagation**: Adjusts weights to reduce error.

Python Code: Forward & Backward Propagation (XOR):

```
import numpy as np


# Sigmoid and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def sigmoid_derivative(x):
    return x * (1 - x)


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


# Seed and initialize weights and biases
np.random.seed(0)
W1 = np.random.rand(2, 2)
b1 = np.zeros((1, 2))
W2 = np.random.rand(2, 1)
b2 = np.zeros((1, 1))
```

```
# Hyperparameters
lr = 0.5
epochs = 10000

# Training
for epoch in range(epochs):
    # ---- FORWARD ----
    z1 = np.dot(X, W1) + b1
    a1 = sigmoid(z1)

    z2 = np.dot(a1, W2) + b2
    y_pred = sigmoid(z2)

    # ---- LOSS ----
    loss = np.mean((y - y_pred) ** 2)

    # ---- BACKWARD ----
    d_y_pred = (y_pred - y) * sigmoid_derivative(y_pred)
    d_W2 = np.dot(a1.T, d_y_pred)
    d_b2 = np.sum(d_y_pred, axis=0, keepdims=True)

    d_hidden = np.dot(d_y_pred, W2.T) * sigmoid_derivative(a1)
    d_W1 = np.dot(X.T, d_hidden)
    d_b1 = np.sum(d_hidden, axis=0, keepdims=True)
```

```
    # ---- UPDATE ----
    W2 -= lr * d_W2
    b2 -= lr * d_b2
    W1 -= lr * d_W1
    b1 -= lr * d_b1

    if epoch % 1000 == 0:
        print(f"Epoch {epoch} - Loss: {loss:.4f}")

# Final predictions
print("\nPredictions after training:")
for i in range(len(X)):
    output = sigmoid(np.dot(sigmoid(np.dot(X[i], W1) + b1), W2) + b2)
    print(f"{X[i]} → {output.round(3)}")
```

Sample Output:

```
Epoch 0 - Loss: 0.2572
...
Epoch 9000 - Loss: 0.0012
Predictions after training:
[0 0] → [0.01]
[0 1] → [0.99]
[1 0] → [0.99]
[1 1] → [0.01]
```

What is a Loss Function?

A **loss function** is a mathematical function that measures the **difference between the predicted output** and the **actual target value**.

In simple terms, **loss = error**
Smaller loss → better model performance

During training, the model:

1. Makes predictions (forward pass)
2. Calculates **loss**
3. Uses **backpropagation** to adjust weights
4. Repeats (over many epochs) to **minimize the loss**

Categories of Loss Functions

Problem Type	Common Loss Functions
Regression	MSE, MAE, Huber Loss
Binary Classification	Binary Cross-Entropy
Multi-class Classification	Categorical Cross-Entropy
Multi-label Classification	Binary Cross-Entropy (with sigmoid)

1. Mean Squared Error (MSE)

Use: Regression problems

- Penalizes large errors
- Differentiable & smooth

```
from sklearn.metrics import mean_squared_error

loss = mean_squared_error(y_true, y_pred)
```

2. Mean Absolute Error (MAE)

Use: Regression (less sensitive to outliers)

```
from sklearn.metrics import mean_absolute_error
```

loss = mean_absolute_error(y_true, y_pred)

3. Binary Cross-Entropy (Log Loss)

Use: Binary classification (sigmoid activation)

from sklearn.metrics import log_loss

loss = log_loss(y_true, y_pred)

4. Categorical Cross-Entropy

Use: Multi-class classification (softmax activation)

from tensorflow.keras.losses import CategoricalCrossentropy

loss_fn = CategoricalCrossentropy()

loss = loss_fn(y_true, y_pred).numpy()

5. Sparse Categorical Cross-Entropy

Use: Multi-class classification with integer labels

from tensorflow.keras.losses import SparseCategoricalCrossentropy

loss_fn = SparseCategoricalCrossentropy()

6. Huber Loss

Use: Regression, robust to outliers

from tensorflow.keras.losses import Huber

loss_fn = Huber(delta=1.0)

7. KL Divergence (Relative Entropy)

Use: Measure difference between two probability distributions

from tensorflow.keras.losses import KLDivergence

What is an Optimizer?

An **optimizer** is an algorithm that updates the **model's weights and biases** to **minimize the loss function** during training.

It's the brain behind learning — it guides how the model learns from the errors.

Optimizer Workflow (in training loop)

1. Do a **forward pass** to calculate predictions.
2. Compute the **loss**.
3. Do a **backward pass** to compute gradients.
4. **Optimizer uses gradients** to update model parameters (weights & biases).

Popular Optimizers

Optimizer	Type	Pros	Cons
SGD	Basic	Simple, efficient	Slow, sensitive to learning rate
Momentum	Modified SGD	Faster convergence	Needs tuning of momentum
AdaGrad	Adaptive	Good for sparse data	Learning rate decays too fast
RMSprop	Adaptive	Great for RNNs	Needs tuning
Adam	Adaptive	Fast, widely used	Uses more memory
Nadam	Adaptive	Adam + Nesterov momentum	Slightly slower sometimes
Adadelta	Adaptive	No need to set learning rate	Not as popular anymore

Simple Python Example: Optimizer

import numpy as np

Example: Minimize f(x) = x^2

```
x = 10

lr = 0.1

epochs = 20


for epoch in range(epochs):

    grad = 2 * x

    x = x - lr * grad

    print(f"Epoch {epoch+1}: x = {x:.4f}, f(x) = {x**2:.4f}")
```

Output:

```
Epoch 1: x = 8.0000, f(x) = 64.0000

Epoch 2: x = 6.4000, f(x) = 40.9600

...

Epoch 20: x = 0.1220, f(x) = 0.0149
```

Working Example: Feedforward Neural Network with Keras

```
import numpy as np

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense


# Dummy dataset (X: input features, Y: binary labels)

X = np.array([[0,0], [0,1], [1,0], [1,1]])

Y = np.array([[0], [1], [1], [0]])  # XOR pattern
```

```
# Build model
model = Sequential()
model.add(Dense(4, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# Compile and train
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X, Y, epochs=100, verbose=0)

# Predict
print(model.predict(X))
```

Output:

```
[[0.01]
 [0.98]
 [0.97]
 [0.05]]
```

Case Study:

Image Classification for Plant Disease Detection

Python Code:

```
import tensorflow as tf

from tensorflow.keras.preprocessing.image import
ImageDataGenerator

import matplotlib.pyplot as plt

import numpy as np

import os


# Load dataset

train_path = 'plant_disease_dataset/train'

test_path = 'plant_disease_dataset/test'


# Preprocess: Resize and normalize images

img_size = (128, 128)

batch_size = 32


train_datagen = ImageDataGenerator(rescale=1./255)
```

```
test_datagen = ImageDataGenerator(rescale=1./255)

train_data = train_datagen.flow_from_directory(
    train_path,
    target_size=img_size,
    batch_size=batch_size,
    class_mode='binary'
)

test_data = test_datagen.flow_from_directory(
    test_path,
    target_size=img_size,
    batch_size=batch_size,
    class_mode='binary'
)

# Build CNN Model
model = tf.keras.Sequential([
```

```
    tf.keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(128,128,3)),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),

    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')  # Binary classification
])

# Compile
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Train
history = model.fit(train_data, validation_data=test_data, epochs=10)
```

```python
# Evaluate
loss, acc = model.evaluate(test_data)
print(f"\nTest Accuracy: {acc:.2f}")
```

Output:

```
Found 800 images belonging to 2 classes.
Found 200 images belonging to 2 classes.
Epoch 1/10
...
Test Accuracy: 0.92
```

Predict on Custom Image:

```python
import cv2

def predict_image(img_path):
    img = cv2.imread(img_path)
    img = cv2.resize(img, img_size)
    img = img / 255.0
    img = img.reshape(1, 128, 128, 3)
```

```
    pred = model.predict(img)[0][0]
    class_name = "Diseased" if pred > 0.5 else "Healthy"
    print(f"Prediction: {class_name} ({pred:.2f})")

# Try with a custom image
predict_image("sample_leaf.jpg")
```

Output:

Prediction: Diseased (0.94)