Introduction to Neural Networks

Inspiration from Nature

Birds inspired humans to build airplanes. The tiny hooks on burrs sticking to a dog's fur led to the invention of **Velcro**. And just like that, nature has always been humanity's greatest engineer.

So, when it came to making machines that could **think, learn, and solve problems**, where did we look?

To the **human brain**.

That's how **neural networks** were born — machines inspired by **neurons** in our brains, built to recognize patterns, make decisions, and even learn from experience.

What is AI, ML, and DL?

Before we dive into neural networks, let's untangle these buzzwords.

Term	Stands for	Think of it as...
AI	Artificial Intelligence	The big umbrella: making machines "smart"
ML	Machine Learning	A subset of AI: machines that learn from data
DL	Deep Learning	A type of ML: uses neural networks

Let's simplify:

- **AI** is the dream: "Can we make machines intelligent?"
- **ML** is the method: "Let's give machines data and let them learn."
- **DL** is the tool: "Let's use neural networks that learn in layers — like the brain."

So, when we talk about **neural networks**, we're entering the world of **deep learning**, which is a part of **machine learning**, which itself is a part of **AI**.

So, What Are Neural Networks?

Imagine a bunch of simple decision-makers called **neurons**, connected together in layers.

Each neuron:

- Takes some input (like a number)
- Applies a little math (weights + bias)
- Passes the result through a rule (called an activation function)
- Sends the output to the next layer

By connecting many of these neurons, we get a **neural network**.

And what's amazing?

Even though each neuron is simple, when combined, the network becomes powerful — like how a bunch of ants can build a complex colony.

Why Are Neural Networks Useful?

Because they can **learn patterns**, even when we don't fully understand the patterns ourselves.

Examples:

- Recognize cats in photos
- Convert speech to text
- Translate languages
- Predict stock prices
- Generate art
- Power AI like ChatGPT

Structure of a Neural Network

Here's the basic anatomy of a neural network:

```
Input Layer       →       Hidden Layers        →       Output Layer
  (Data)              (Neurons doing math)             (Prediction)
```

Each **layer** is just a bunch of neurons working together. The more hidden layers, the **"deeper"** the network. Hence: **Deep Learning**.

Wait — Why Not Use Simple Code Instead?

Good question.

Sometimes, a simple formula or rule is enough (like `area = length × width`).

But what about:

- Recognizing handwritten digits?
- Understanding language?
- Diagnosing diseases from X-rays?

There are **no easy formulas** for these. Neural networks **learn the formula by themselves** from lots of examples.

How Do Neural Networks Learn?

Let's say the network tries to predict `y = x² + x`.

1. It starts with **random guesses** (bad predictions)
2. It checks how wrong it is (loss)
3. It adjusts the internal settings (weights) to be a little better
4. Repeat, repeat, repeat…

Over time, the network **figures out the relationship** between x and y.

This process is called **training** — and it's where the magic happens.

Are They Really Like the Brain?

Kind of — but **very simplified**.

- A biological brain neuron connects to 1000s of others
- It processes chemicals, spikes, timings
- It adapts and rewires itself

A neural network is a **mathematical model** — inspired by the brain, but way simpler. Still, the results are powerful.

Summary

Concept	Meaning
AI	Making machines act smart
ML	Letting machines learn from data
DL	Using multi-layered neural networks to learn complex stuff
Neural Network	A network of artificial neurons that learns from data

Perceptron – The Simplest Neural Network

What is a Perceptron?

The perceptron is the basic building block of a neural network. It's a simple computational model that takes several inputs, applies weights to them, adds a bias, and produces an output. It's essentially a decision-making unit.

Real-life Analogy

Imagine you're trying to decide whether to go outside based on:

- Is it sunny?
- Is it the weekend?
- Are you free today?

You assign importance (weights) to each factor:

- Sunny: 0.6
- Weekend: 0.3
- Free: 0.8

You combine these factors to make a decision: Go or Not Go.

This is what a perceptron does.

Perceptron Formula

A perceptron takes inputs (x1, x2, ..., xn), multiplies each by its corresponding weight (w1, w2, ..., wn), adds a bias (b), and passes the result through an activation function.

y = f(w1*x1* + *w2*x2 + ... + wn*xn + b)

Where:

- xi: input features
- wi: weights
- b: bias
- f: activation function (e.g., step function)

Step-by-step Example: Binary Classification

Let's say we want a perceptron to learn this simple table:

Input (x1, x2)	Output (y)
(0, 0)	0
(0, 1)	0
(1, 0)	0
(1, 1)	1

This is the behavior of a logical AND gate.

We will use:

- Inputs: x1, x2
- Weights: w1, w2
- Bias: b
- Activation Function: Step function

Step function:

```
def step(x):
    return 1 if x >= 0 else 0
```

Code: Simple Perceptron from Scratch

```
def step(x):
    return 1 if x >= 0 else 0

def perceptron(x1, x2, w1, w2, b):
    z = x1 * w1 + x2 * w2 + b
    return step(z)

# Try different weights and bias to match the AND logic
print(perceptron(0, 0, 1, 1, -1.5))  # Expected: 0
print(perceptron(0, 1, 1, 1, -1.5))  # Expected: 0
print(perceptron(1, 0, 1, 1, -1.5))  # Expected: 0
print(perceptron(1, 1, 1, 1, -1.5))  # Expected: 1
```

This matches the AND logic perfectly.

Summary

- A perceptron is the simplest form of a neural network.
- It performs a weighted sum of inputs, adds a bias, and passes the result through an activation function to make a decision.
- It can model simple binary functions like AND, OR, etc.

Common Terms in Deep Learning

Before diving into neural networks, let's clarify some common terms you'll encounter:

Perceptron

A **perceptron** is the simplest type of neural network — just one neuron. It takes inputs, multiplies them by weights, adds a bias, and applies an activation function to make a decision (e.g., classify 0 or 1).

Neural Network

A **neural network** is a collection of interconnected layers of perceptrons (neurons). Each layer transforms its inputs using weights, biases, and activation functions. Deep neural networks have multiple hidden layers and can model complex patterns.

Hyperparameters

These are settings we configure **before training** a model. They are not learned from the data. Examples include:

- Learning rate: How much to adjust weights during training
- Number of epochs: How many times the model sees the entire training dataset
- Batch size: How many samples to process before updating weights

- Number of layers or neurons: How many neurons are in each layer of the network

---

Learning Rate (η)

This controls **how much we adjust the weights** after each training step. A learning rate that's too high may overshoot the solution; too low may make training very slow.

Training

This is the process where the model **learns patterns from data** by updating weights based on errors between predicted and actual outputs.

Backpropagation

Backpropagation is the algorithm used to **update weights** in a neural network. It calculates the gradient of the loss function with respect to each weight by applying the chain rule, allowing the model to learn from its mistakes.

Inference

Inference is when the trained model is used to **make predictions** on new, unseen data.

Activation Function

This function adds **non-linearity** to the output of neurons, helping networks model complex patterns.

Common activation functions:

- `ReLU` : Rectified Linear Unit
- `Sigmoid` : squashes output between 0 and 1
- `Tanh` : squashes output between -1 and 1

> Perceptrons typically use a **step function** as the activation, but modern neural nets often use `ReLU` .

Epoch

One **epoch** means one full pass over the entire training dataset. Multiple epochs are used so the model can keep refining its understanding.

Training a Perceptron with scikit-learn

1. Import Libraries

```
from sklearn.linear_model import Perceptron
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
```

---

2. Create and Split Data

```
X, y = make_classification(n_samples=1000, n_features=10, n_classes=2,
random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
```

---

3. Initialize the Perceptron

```
clf = Perceptron(
    max_iter=1000,    # Maximum number of epochs
    eta0=0.1,         # Learning rate
    random_state=42,  # For reproducibility
    tol=1e-3,         # Stop early if improvement is smaller than this
    shuffle=True      # Shuffle data each epoch
)
```

---

4. Train the Model

```
clf.fit(X_train, y_train)
```

Under the hood, this performs the following steps:

- Loops through the data up to `max_iter` times (epochs)
- Computes predictions
- If a prediction is wrong, updates weights

---

5. Evaluate the Model

```
accuracy = clf.score(X_test, y_test)
print(f"Accuracy: {accuracy:.2f}")
```

Important Hyperparameters Recap:

Hyperparameter	Description
`max_iter`	Number of epochs (passes over training data)
`eta0`	Learning rate
`tol`	Tolerance for stopping early
`shuffle`	Whether to shuffle data between epochs
`random_state`	Seed for reproducibility

TensorFlow vs Keras vs PyTorch

Deep learning has transformed industries—from self-driving cars to language models like ChatGPT. But behind the scenes, there are powerful libraries that make all of this possible: **TensorFlow**, **Keras**, and **PyTorch**. Today, we'll explore:

- Why and how each library was created
- How they differ in philosophy and design
- What a "backend" means in Keras
- Which one might be right for you

A Brief History

1. TensorFlow

- **Released**: 2015 by **Google Brain**
- **Language**: Python (but has C++ core)
- **Goal**: Provide an **efficient**, **production-ready**, and **scalable** library for deep learning.
- TensorFlow is a **computational graph framework**, meaning it represents computations as nodes in a graph.
- Open sourced in 2015, TensorFlow quickly became the go-to library for many companies and researchers.

Fun Fact: TensorFlow was a spiritual successor to Google's earlier tool called **DistBelief**.

---

2. Keras

- **Released**: 2015 by **François Chollet**

- **Goal**: Make deep learning **simple**, **intuitive**, and **user-friendly**.
- Keras was originally just a high-level wrapper over **Theano** and **TensorFlow**, making it easier to build models with fewer lines of code.
- Keras introduced the idea of writing deep learning models like stacking Lego blocks.

Keras was not a full deep learning engine—it needed a "backend" to actually do the math

---

3. PyTorch

- **Released**: 2016 by **Facebook AI Research (FAIR)**
- **Language**: Python-first, with a strong integration to NumPy
- **Goal**: Make deep learning **flexible**, **dynamic**, and **easier for research**.
- PyTorch uses **dynamic computation graphs**, meaning the graph is built **on the fly**, allowing for more intuitive debugging and flexibility.

PyTorch gained massive popularity in academia and research because of its Pythonic nature and simplicity.

In a nut shell...

Feature	TensorFlow	Keras	PyTorch
Developed By	Google	François Chollet (Google)	Facebook (Meta)
Level	Low-level & high-level	High-level only	Low-level (with some high-level APIs)
Computation Graph	Static (TensorFlow 1.x), Hybrid (2.x)	Depends on backend	Dynamic
Ease of Use		Very high	High

Feature	TensorFlow	Keras	PyTorch
	Medium (Better in 2.x)		
Debugging	Harder (in 1.x), Easier in 2.x	Easy	Very easy (Pythonic)
Production-ready	Yes	Yes (via TensorFlow backend)	Gaining ground
Research usage	Moderate	Moderate	Very high

What Is a "Backend" in Keras?

Keras itself **doesn't perform computations** like matrix multiplications or gradient descent. It's more like a **user interface** or a **frontend**. It delegates the heavy lifting to a **backend engine**.

Supported Backends Over Time:

- **Theano** (now discontinued)
- **TensorFlow** (default backend now)
- **CNTK** (Microsoft, also discontinued)
- **PlaidML** (experimental support)

You can think of Keras as the *steering wheel*, while TensorFlow or Theano was the *engine* under the hood.

In **TensorFlow 2.0 and later**, Keras is fully integrated as `tf.keras`, eliminating the need to manage separate backends.

TensorFlow vs PyTorch: The Real Battle

Area	TensorFlow	PyTorch
Ease of Deployment	TensorFlow Serving, TFX, TensorFlow Lite	TorchServe, ONNX, some catching up
Mobile Support	Excellent (TF Lite, TF.js)	Improving but limited
Dynamic Graphs	TF 1.x: No, TF 2.x: Yes (via Autograph)	Native support
Community	Large, especially in production	Massive in academia
Performance	Highly optimized	Also strong, especially for GPUs

---

Which One Should You Learn First?

- **Beginner?** → Start with **Keras** via **TensorFlow 2.x (`tf.keras`)**. It's simple and production-ready.
- **Researcher or experimenting a lot?** → Use **PyTorch**.
- **Looking for deployment & scalability?** → **TensorFlow** is very robust.

Installing TensorFlow 2.0

Today we will install TensorFlow 2.0, which is a powerful library for machine learning and deep learning tasks. TensorFlow 2.0 simplifies the process of building and training models, making it more user-friendly compared to its predecessor.

Understanding and Visualizing the MNIST Dataset with TensorFlow

The **MNIST** dataset is like the "Hello World" of deep learning. It contains **70,000 grayscale images** of handwritten digits (0–9), each of size **28x28 pixels**. Before we train a neural network, it's important to *understand what we're working with*.

Today we'll:

- Load the MNIST dataset using TensorFlow
- Visualize a few digit samples
- Understand the data format

Step 1: Load the Dataset

TensorFlow provides a built-in method to load MNIST, so no extra setup is needed.

```
import tensorflow as tf
import matplotlib.pyplot as plt

# Load dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Check the shape
print("Training data shape:", x_train.shape)
print("Training labels shape:", y_train.shape)
```

Output:

```
Training data shape: (60000, 28, 28)
Training labels shape: (60000,)
```

- We have 60,000 training images and 10,000 test images.
- Each image is 28x28 pixels.
- Each label is a number from 0 to 9.

Step 2: Visualize Sample Digits

Let's look at a few images to get a feel for the dataset:

```
# Plot first 10 images with their labels
plt.figure(figsize=(10, 2))
for i in range(10):
    plt.subplot(1, 10, i + 1)
    plt.imshow(x_train[i], cmap="gray")
    plt.axis("off")
    plt.title(str(y_train[i]))
plt.tight_layout()
plt.show()
```

This will display the first 10 handwritten digits with their corresponding labels above them.

What You Should Notice

- The digits vary in **writing style**, which makes this dataset great for teaching computers to generalize.
- All images are normalized 28x28 grayscale—**no color channels**.
- The label is **not embedded** in the image; it's provided separately.

Your First Neural Network

TensorFlow is an open-source deep learning library developed by Google. It's widely used in industry and academia for building and training machine learning models. **TensorFlow 2.0** brought significant improvements in ease of use, especially with **eager execution** and tight integration with Keras.

In this tutorial, we'll create a **simple neural network** that learns to classify handwritten digits using the **MNIST dataset**. This dataset contains 28x28 grayscale images of digits from 0 to 9.

Key Features of TensorFlow 2.0

- **Eager execution** by default (no more complex session graphs!)
- **Keras as the official high-level API** (`tf.keras`)
- Better **debugging and simplicity**
- Great for both beginners and professionals

What is a Neural Network?

A **neural network** is a collection of layers that learn to map input data to outputs. Think of layers as filters that extract meaningful patterns. Each layer applies transformations using **weights** and **activation functions**.

Installation

```
pip install tensorflow
```

Importing Required Libraries

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
```

Loading and Preparing the Data

```
# Load the data
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalize the input data
x_train = x_train / 255.0
x_test = x_test / 255.0

# One-hot encode the labels
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

Building a Simple Neural Network

```
model = Sequential([
    Flatten(input_shape=(28, 28)),     # 28x28 images to 784 input features
    Dense(128, activation='relu'),     # Hidden layer with 128 neurons
    Dense(10, activation='softmax')    # Output layer for 10 classes
])
```

Compiling the Model

```
model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
```

Training the Model

```
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

Evaluating the Model

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc:.4f}")
```