Module 1

Deep Learning Basics

Learning Objectives

By the end of this module, you should be able to:

- Explain what Machine Learning (ML) and Deep Learning (DL) are.
- Describe the structure of a simple neural network.
- Train a small neural network in Python using TensorFlow/Keras.
- Understand how model size and accuracy trade-offs matter for microcontrollers.

1.1 What is Machine Learning?

Machine Learning (ML) is a method of teaching computers to recognize patterns and make decisions without explicit programming for every scenario. Instead of following rigid rules, ML systems learn from data and experience.

Traditional Programming vs. Machine Learning

In traditional programming:

- Humans write explicit rules: "If this condition is true, then do that action."
- The program follows these rules step by step.

In machine learning:

- We provide the computer with *examples* (data) rather than rules.
- The computer learns the rules itself by adjusting internal parameters.

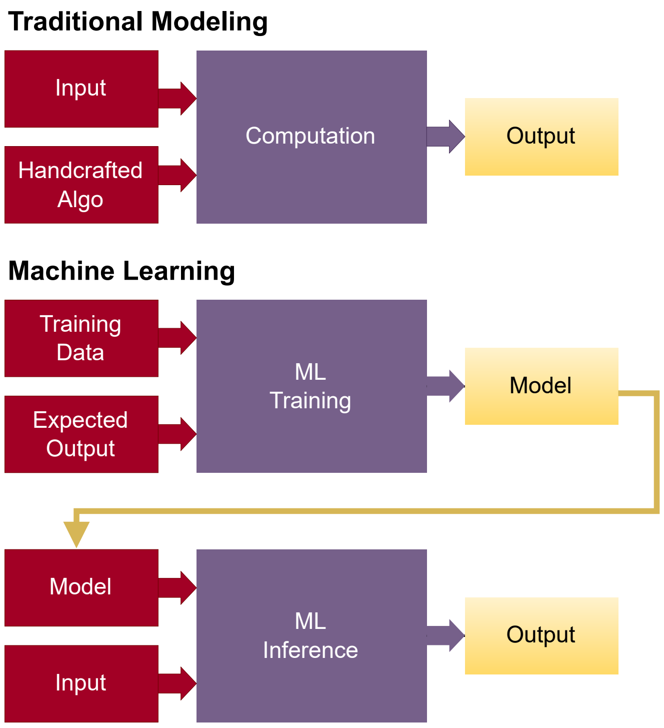

Figure 1.1: Traditional vs ML

Analogy: Imagine teaching someone to recognize birds.

- Traditional: you write rule like ”if it has feathers and a beak, then it is a bird.”
- Machine learning: you show them thousands of bird/non-bird images, and they learn the common patterns features themselves.

From Machine Learning to Deep Learning

- **Machine Learning:** Uses algorithms like decision trees or regression, but often requires human input to select important data features.
- **Deep Learning:** A subset of ML using **multi-layer neural networks** that automatically extract features from raw data (e.g., images, audio).

TinyML is the fusion of machine learning and embedded systems, enabling on-device intelligence for resource-constrained devices. This is achieved through:

- Model compression (reducing size from GBs to kBs),
- Optimization (running on devices with milliwatt power and real-time response).

1.2 Why TinyML?

Cloud AI is powerful, but many applications require **edge computing**—processing data locally on small, battery-powered devices.

Key Benefits

- **Low Power:** Microcontrollers consume milliwatts, allowing devices to run for months or even years on a small battery.
- **Privacy:** Data (like voice recordings) can be processed locally, without sending it to the cloud.
- **Low Latency:** Decisions can be made instantly on-device, without waiting for internet round-trips.
- **Cost Efficiency:** No need for expensive servers or continuous internet access.

Table 1.1: Cloud AI vs TinyML

	Cloud AI	**TinyML (Edge)**
Compute	GPUs/servers	Microcontrollers (kB of RAM)
Latency	Depends on internet	Real-time, immediate
Privacy	Data sent to cloud	Data stays on device
Power Use	High (datacenters)	Very low (years on battery)
Cost	Server + internet fees	One-time hardware cost

Where TinyML Excels:

- **Keyword Spotting:** "Hey Arduino" on a RM10 microcontroller
- **Gesture Recognition:** Accelerometer-based movement detection, e.g. flick vs. shake.
- **Environmental Monitoring:** detecting machinery faults, bird sounds
- **Healthcare Devices:** Low-power irregular heartbeats detection

TinyML vs. Cloud AI: A Partnership

TinyML doesn't replace cloud AI but complements it:

- **Cloud**: trains large models, aggregates data, updates firmware.
- **Edge (TinyML)**: runs optimized lightweight models locally.

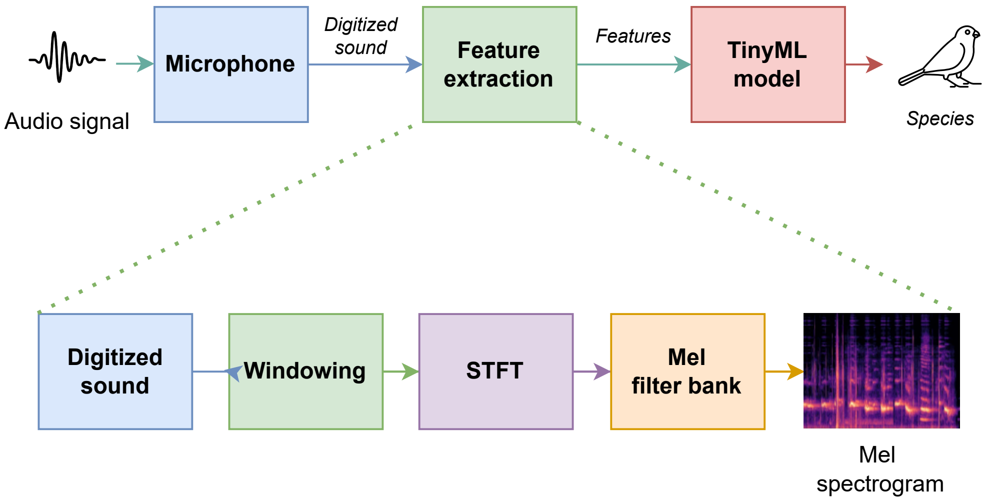

Figure 1.2: Bird sound recognition pipeline

Case Study: Google Speech Commands

One of the most popular examples of TinyML is **keyword spotting**. Google released the *Speech Commands* dataset: thousands of short audio clips containing words like “yes”, “no”, “up”, “down”, and “stop”. On low-power devices, a small neural network runs continuously on the microphone input to detect a **wake word** (e.g. “Hey Google”).

Why is this important?

- **Privacy**: The device only sends audio to the cloud *after* it hears the wake word.
- **Efficiency**: Reduces power use, bandwidth, and latency.
- **Scalability**: Works on cheap microcontrollers, not just smartphones.

In this course, we will build similar projects where your STM32 Nucleo boards listen for patterns in data, just like Google’s speech models.

1.3 Neural Networks: The Intuition

Neural networks are inspired by the way biological neurons work, but they are much simpler and easier to analyze. At their core, they combine inputs, apply weights, add a bias, and then pass the result through a nonlinear function called an *activation*.

From Linear Algebra to Intelligence

You can think of a neural network as a sequence of operations:

1. Take several inputs (like sensor readings or pixels).
2. Multiply each input by a **weight** that determines its importance.
3. Add the results together and include a **bias** term.
4. Pass the result through an **activation function** to decide the output.

Mathematically, for inputs $x_1, x_2, \ldots, x_n$ with weights $w_1, w_2, \ldots, w_n$, and bias b, the neuron output is:

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

where $f(\cdot)$ is the activation function.

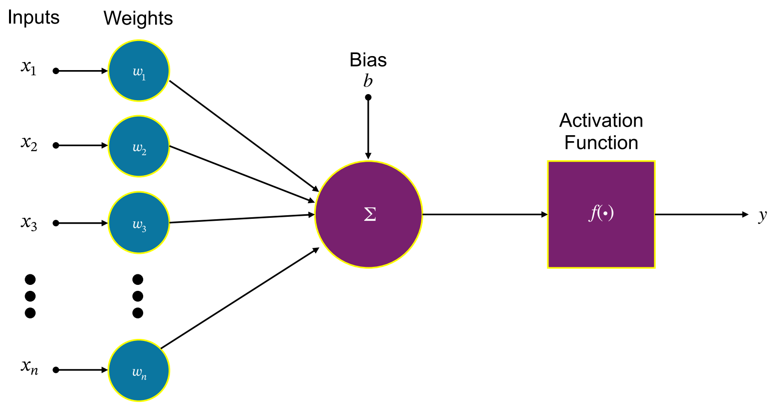

Figure 1.3: Artifical neural network structure

This diagram shows how inputs are weighted, summed, and passed through an activation function to produce an output.

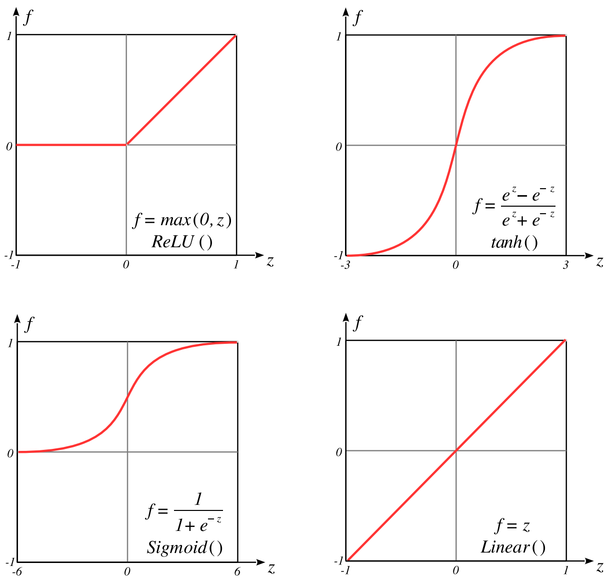

Figure 1.4: Four widely used activation functions

Why Activation Functions Matter

Without nonlinear activations, a network is just a stack of linear operations — equivalent to a single linear function. Activations introduce nonlinearity, allowing neural networks to approximate complex patterns. Common examples include:

- **Sigmoid:** Smooth step between 0 and 1.
- **ReLU (Rectified Linear Unit):** Passes positive values, zeros out negative ones.
- **tanh:** Like sigmoid but centered at zero.

From One Neuron to a Network

By connecting many neurons into layers, and stacking layers into a network, we create a system that can learn very complex mappings from inputs to outputs. Training adjusts the weights w_i and biases b so that the network performs the desired task, such as recognizing speech commands or detecting gestures.

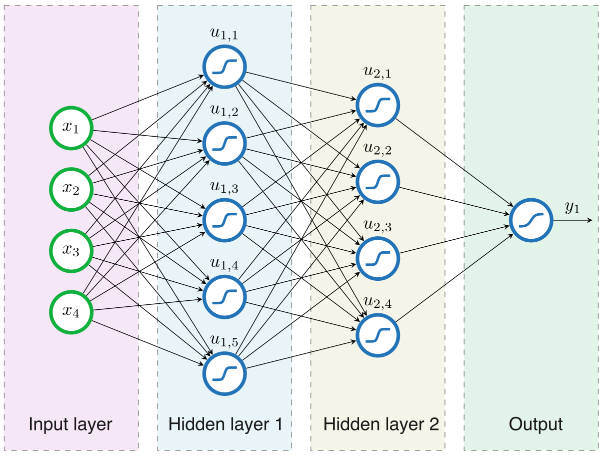

Figure 1.5: Combining neurons to build a deep learning network

From Neural Networks to Deep Learning

A simple neural network may have just one hidden layer, which is often enough for straightforward tasks. Deep learning refers to networks with two or more hidden layers stacked together. Each hidden layer transforms the representation it receives: the first may detect simple patterns (like edges in an image), the next may combine them into shapes, and deeper layers may recognize entire objects.

In other words, as information flows from the **input layer** through multiple **hidden layers** to the **output layer**, the network gradually builds up more abstract and powerful features. This layered hierarchy is what gives deep learning its ability to solve complex tasks, such as image recognition, speech processing, and natural language understanding.

Deep Learning vs. Convolutional Neural Networks (CNNs)

Deep learning is the broad idea of using neural networks with many hidden layers to learn complex patterns from data. A convolutional neural network (CNN) is a specific type of deep learning model designed for data with spatial structure, such as images or spectrograms.

Instead of fully connecting every neuron between layers, CNNs use small filters (convolutions) that slide over the input. This makes them efficient, less prone to overfitting, and especially good at detecting local patterns like edges, textures, or repeating sound features. In short: **deep learning is the family, CNN is a member specialized for structured data**.

1.4 Running Code in Google Colab

Before we train our first neural network, we need a simple environment where everything "just works." Instead of installing Python, TensorFlow, and drivers on your laptop, we will use **Google Colab**, a free, cloud-based Jupyter notebook service.

What is Google Colab?

- A free platform from Google for running Python notebooks in the cloud.
- Pre-installed with TensorFlow, NumPy, and many other scientific libraries.
- Provides free access to GPUs (Graphics Processing Units) for faster training.
- Only requires a Google account and a web browser.

Getting Started

1. Go to: `https://colab.research.google.com`
2. Sign in with your Google account.
3. Create a new notebook: `File → New Notebook`.
4. You now have a coding environment with a Python interpreter ready to go.

Checking TensorFlow

In a Colab cell, type:

```
import tensorflow as tf
print(tf.__version__)
```

This should display a version like `2.x`, which is what we need.

Why Use Colab?

- No installation headaches.
- Consistent setup for all students.
- Access to GPU without extra hardware.
- Easy to share notebooks with others (just like Google Docs).

Tip

Colab notebooks automatically save in your Google Drive. You can also download them as `.ipynb` files or export them to `.py` scripts if needed.

With Colab ready, you can now run the MNIST example in the next section directly in your browser.

Enabling GPU Acceleration (Optional but Recommended)

1. Click `Runtime → Change runtime type`.
2. Under `Hardware accelerator`, select `GPU`.
3. Click `Save`.

You will now have access to a free GPU instance. Training will run much faster.

1.5 Your First Neural Network in Code

To make things concrete, let us train a very simple neural network on a classic dataset: **MNIST**, which contains 70,000 grayscale images of handwritten digits (0–9), each of size 28×28 pixels.

Even though MNIST is not something we would run on a microcontroller directly, it is the "Hello World" of deep learning. Once you understand this, you can later replace digits with sensor data or audio features.

Step 1: Import Libraries and Load Data

We use `TensorFlow/Keras`, which gives us convenient access to the MNIST dataset. Open a new notebook in Google Colab and paste this cell:

```
import tensorflow as tf
from tensorflow import keras
import numpy as np

(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data
    ()
```

At this point:

- `x_train` is an array of 60,000 images (training data).
- `y_train` are the corresponding labels (digits 0–9).
- `x_test`, `y_test` form the test set (10,000 images).

Step 2: Preprocess the Data

Neural networks work better when inputs are normalized.

```
x_train = x_train.reshape(-1, 28*28).astype("float32") / 255.0
x_test = x_test.reshape(-1, 28*28).astype("float32") / 255.0
```

Here:

- Each 28×28 image is flattened into a 784-element vector.
- Pixel values are scaled to the range $[0, 1]$.

Step 3: Define the Model

We start with a simple network: a single hidden layer.

```
model = keras.Sequential([
    keras.layers.Dense(128, activation="relu", input_shape=(784,)),
    keras.layers.Dense(10, activation="softmax")
])
```

Explanation:

- `Dense(128)`: A fully connected layer with 128 neurons and ReLU activation.
- `Dense(10)`: Output layer with 10 neurons (for digits 0–9), using softmax to produce probabilities.

Step 4: Compile the Model

We specify the optimizer, loss function, and metrics.

```
model.compile(optimizer="adam",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
```

- **Optimizer (Adam):** Adjusts weights efficiently.
- **Loss (Cross-Entropy):** Measures how well predictions match labels.
- **Accuracy:** Human-friendly metric for evaluation.

Step 5: Train the Model

Now we let the model learn from the data.

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

Each *epoch* means the model sees all 60,000 images once. The *batch size* means it processes 32 images at a time before updating weights.

Step 6: Evaluate on Test Data

Finally, we test the trained model on unseen data.

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

You should see around **97–98% accuracy** after just a few epochs, which is quite impressive for such a small network.

1.6 Exercises

1. Change the number of neurons in the hidden layer.
2. Change the activation function from ReLU to sigmoid.
3. Train for 1, 5, and 10 epochs. Compare results.

1.7 Reflection

- Why do smaller models train faster but may have lower accuracy?
- Why does accuracy sometimes stop improving after many epochs?
- How do these trade-offs matter for microcontrollers?

1.8 Resources

- TensorFlow Beginner Tutorial
- Warden, Pete & Situnayake, Daniel. *TinyML* (Chapters 1–2).