Key differences: deep learning vs. neural networks

The terms *deep learning* and *neural networks* are used interchangeably because all deep learning systems are made of neural networks. However, technical details vary. There are several different types of neural network technology, and all may not be used in deep learning systems.

For this comparison, the term *neural network* refers to a feedforward neural network. Feedforward neural networks process data in one direction, from the input node to the output node. Such networks are also called *simple neural networks*.

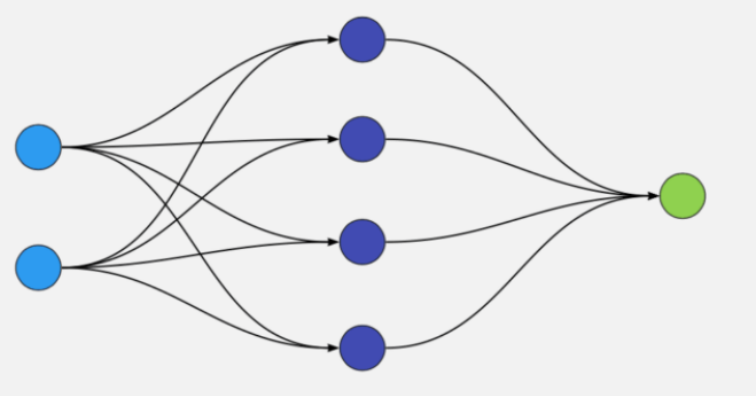


Figure** Error! No text of specified style in document.**.1 simple feedforward neural network.

Next are some key differences between feedforward neural networks and deep learning systems.

Architecture

In a simple neural network, every node in one layer is connected to every node in the next layer. There is only a single hidden layer. In contrast, deep learning systems have several hidden layers that make them deep.

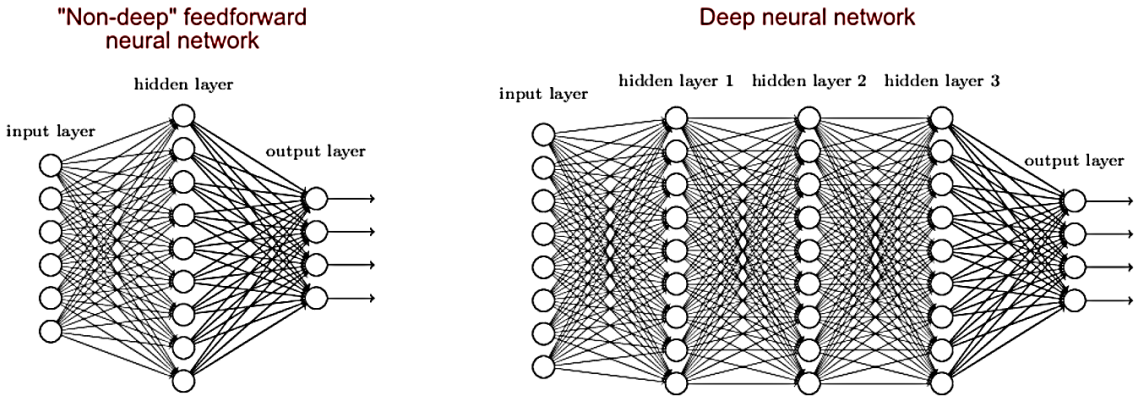
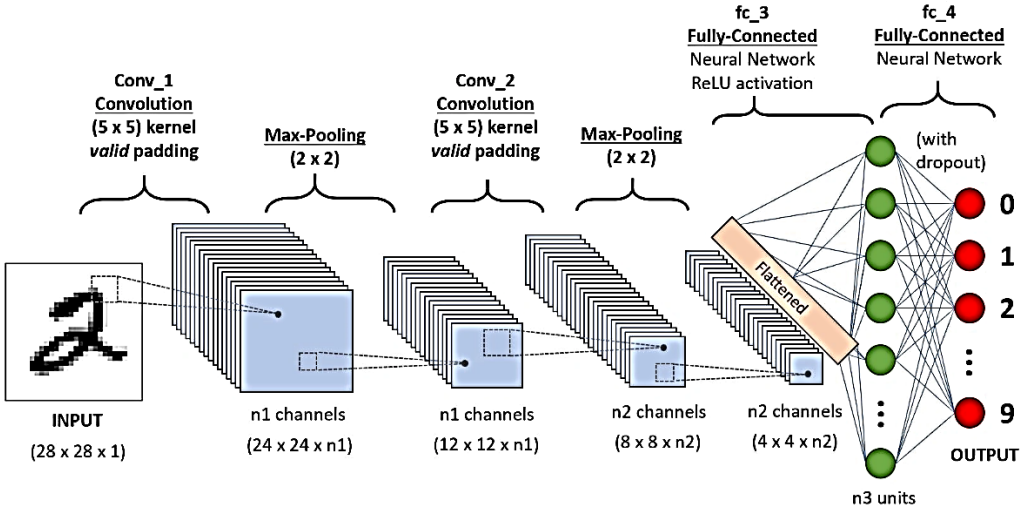


Figure** Error! No text of specified style in document.**.2 Simple NN vs. Deep NN Architecture

There are two main types of deep learning systems with differing architectures—convolutional neural networks (CNNs) and recurrent neural networks (RNNs).

CNN architecture



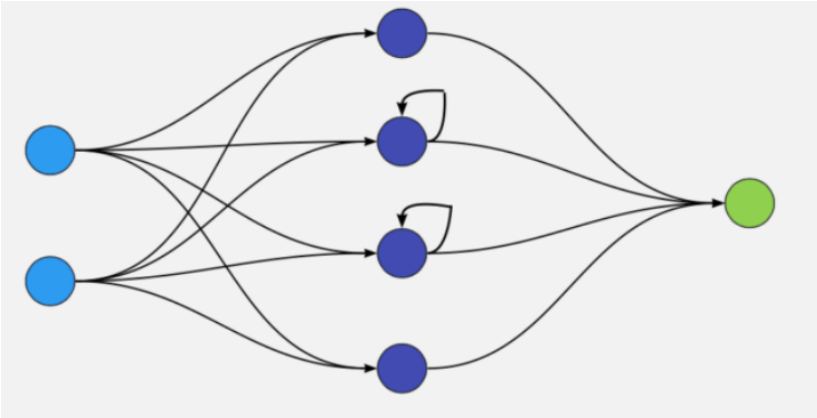
Example on CNN architecture

CNNs have three-layer groups:

- Convolutional layers extract information from data you input, using preconfigured filters.
- Pooling layers reduce the dimensionality of data, breaking down data into different parts or regions.
- Fully connected layers create additional neural pathways between layers. This allows the network to learn complex relationships between features and make high-level predictions.

You can use CNN architecture when you process images and videos, as it can handle varying inputs in dimension and size.

RNN architecture



Example on RNN architecture

The architecture of an RNN can be visualized as a series of recurrent units.

Each unit is connected to the previous unit, forming a directed cycle. At each time step, the recurrent unit takes the current input and combines it with the previous hidden state. The unit produces an output and updates the hidden state for the next time step. This process is repeated for each input in the sequence, which allows the network to capture dependencies and patterns over time.

RNNs excel at natural language functions like language modeling, speech recognition, and sentiment analysis.

	Deep learning systems	Simple neural networks
Architecture	Consists of several hidden layers arranged for convolution or recurrence.	Neural networks consist of an input, hidden, and output layer. They mimic the human brain in structure.
Complexity	Depending on its function, a deep learning network is highly complicated and has structures like long short-term memory (LSTM) and autoencoders.	Neutral networks are less complicated, as they have only a few layers.
Performance	A deep learning algorithm can solve complex issues across large data volumes.	Neural networks perform well when solving simple problems.
Training	It costs a lot of money and resources to train a deep learning algorithm.	The simplicity of a neural network means it costs less to train.

Epoch, Batch Size, and Iterations

Before diving into the different learning rules, we must first explain some important notations:
In Neural Networks (NNs), the terms epoch, batch size, and iterations are crucial in understanding how training works. Here's a clear explanation of each term, along with an example.

- ***Epoch***

An epoch refers to one complete pass of the entire training dataset through the neural network.

Example:

Suppose we have a dataset with 1000 training samples.
If we set the number of epochs = 5, it means the model will go through the entire dataset 5 times during training.
More epochs usually help the model learn better, but too many can lead to overfitting.

- ***Batch Size***

The batch size refers to the number of training samples processed before updating the model's weights.

Example:

If batch size = 200, then:
The dataset of 1000 samples is divided into 5 batches (since 1000/200 = 5).
The model updates weights after every 200 samples.
Choosing a batch size affects memory usage and training speed:
Small batch size (e.g., 16, 32): More updates, requires less memory but slower training.
Large batch size (e.g., 256, 512): Faster training but may generalize poorly.

- ***Iterations***

An iteration refers to one update of the model's weights, which happens after processing one batch.

Example:

Given 1000 samples and batch size = 200, we have:
5 batches per epoch → 5 iterations per epoch.
If we train for 5 epochs, the total iterations = 5 × 5 = 25.

In conclusion:

- **Epoch** = One full pass through the entire dataset.
- **Batch size** = Number of samples processed before updating weights.
- **Iteration** = One batch processed, leading to one weight update.

Perceptron Learning Rule

The Perceptron Learning Rule is a supervised learning algorithm used to train a single-layer perceptron for binary classification problems. It works by iteratively adjusting the weights based on the difference between the predicted output and the actual target.

The weights and bias are updated using the formula:

$$w_{i,j} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$$

$$b = b + \eta(y_j - \hat{y}_j)$$

Where:

$w_{i,j}$ is the weight between the *ith* input and *jth* output neuron,

x_i is the ith input value,

y_j is the actual value (target), and $\hat{y}_j$ is the predicted value,

η is the learning rate

Code:

```python
import numpy as np
# Generate synthetic data (2D points, 2 classes)
np.random.seed(42)
X = np.random.rand(100, 2)  # 100 samples, 2 features
y = (X[:, 0] + X[:, 1] > 1).astype(int)  # Simple decision boundary

# Initialize weights and bias
weights = np.random.randn(2)
bias = np.random.randn()

# Hyperparameters
learning_rate = 0.1
epochs = 75
batch_size = 20  # Mini-batch training
iterations_per_epoch = len(X) // batch_size  # Total iterations per epoch

# Training Loop
for epoch in range(epochs):
    print(f"\nEpoch {epoch+1}/{epochs}")

    # Shuffle data for mini-batch training
    indices = np.random.permutation(len(X))
    X_shuffled, y_shuffled = X[indices], y[indices]

    for i in range(iterations_per_epoch):
        start = i * batch_size
        end = start + batch_size
        X_batch, y_batch = X_shuffled[start:end], y_shuffled[start:end]

        # Forward pass
        predictions = np.dot(X_batch, weights) + bias
        predictions = np.where(predictions > 0, 1, 0)  # Step activation

        # Compute error
        errors = y_batch - predictions

        # Update weights and bias using perceptron learning rule
        weights += learning_rate * np.dot(X_batch.T, errors)
        bias += learning_rate * np.sum(errors)

        print(f" Iteration {i+1}/{iterations_per_epoch}, Batch Error: {np.sum(errors)}")

print("\nFinal Weights:", weights)
print("Final Bias:", bias)
```