Kalinga University

Department of Computer Science

Notes-I

Course-BCSAIMLCS/BCAAIML

Subject-Deep Learning

Subject Code- BCSAIMLCS502/BCAAIML502 **Sem-V**

What is a Neural Network?

Neural networks are machine learning models that mimic the complex functions of the human brain. These models consist of interconnected nodes or neurons that process data, learn patterns, and enable tasks such as pattern recognition and decision-making.

In this article, we will explore the fundamentals of neural networks, their architecture, how they work, and their applications in various fields. Understanding neural networks is essential for anyone interested in the advancements of artificial intelligence.

Understanding Neural Networks in Deep Learning

Neural networks are capable of learning and identifying patterns directly from data without pre-defined rules. These networks are built from several key components:

1. **Neurons**: The basic units that receive inputs, each neuron is governed by a threshold and an activation function.
2. **Connections**: Links between neurons that carry information, regulated by weights and biases.
3. **Weights and Biases**: These parameters determine the strength and influence of connections.
4. **Propagation Functions**: Mechanisms that help process and transfer data across layers of neurons.
5. **Learning Rule**: The method that adjusts weights and biases over time to improve accuracy.

Learning in neural networks follows a structured, three-stage process:

1. **Input Computation**: Data is fed into the network.

2. **Output Generation**: Based on the current parameters, the network generates an output.
3. **Iterative Refinement**: The network refines its output by adjusting weights and biases, gradually improving its performance on diverse tasks.

In an adaptive learning environment:

- The neural network is exposed to a simulated scenario or dataset.
- Parameters such as weights and biases are updated in response to new data or conditions.
- With each adjustment, the network's response evolves, allowing it to adapt effectively to different tasks or environments.

The image illustrates the analogy between a biological neuron and an artificial neuron, showing how inputs are received and processed to produce outputs in both systems.

Importance of Neural Networks

Neural networks are pivotal in identifying complex patterns, solving intricate challenges, and adapting to dynamic environments. Their ability to learn from vast amounts of data is transformative, impacting technologies like **natural language processing**, **self-driving vehicles**, and **automated decision-making**. Neural networks streamline processes, increase efficiency, and support decision-making across various industries. As a backbone of artificial intelligence, they continue to drive innovation, shaping the future of technology.

Evolution of Neural Networks

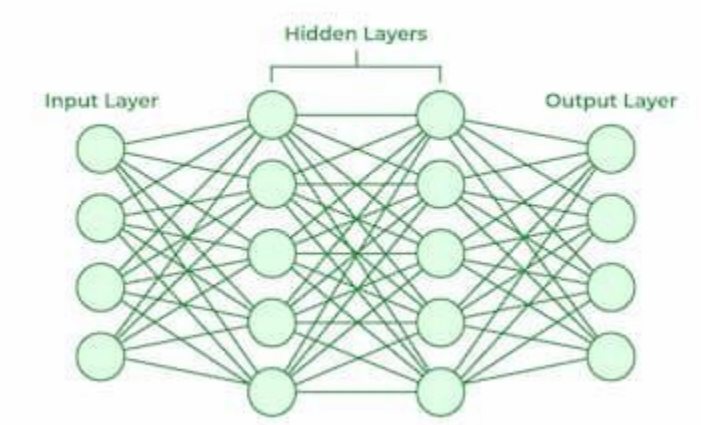
Neural networks have undergone significant evolution since their inception in the mid-20th century. Here's a concise timeline of the major developments in the field:

- **1940s-1950s**: The concept of neural networks began with McCulloch and Pitts' introduction of the first mathematical model for **artificial neurons**. However, the lack of computational power during that time posed significant challenges to further advancements.
- **1960s-1970s**: Frank Rosenblatt's worked on perceptrons. **Perceptrons** are simple single-layer networks that can solve linearly separable problems, but can not perform complex tasks.
- **1980s:** The development of **backpropagation** by Rumelhart, Hinton, and Williams revolutionized neural networks by enabling the training of multi-layer networks. This period also saw the rise of connectionism, emphasizing learning through interconnected nodes.
- **1990s:** Neural networks experienced a surge in popularity with applications across image recognition, finance, and more. However, this growth was tempered by a period known as the **"AI winter,"** during which high computational costs and unrealistic expectations dampened progress.
- **2000s:** A resurgence was triggered by the availability of larger datasets, advances in computational power, and innovative network architectures. Deep learning, utilizing multiple layers, proved highly effective across various domains.
- **2010s:** The landscape of machine learning has been dominated by deep learning with **CNNs** (Convolutional Neural Networks) excelling in image classification and **RNNs** (Recurrent Neural Networks) , **LSTMs**, and **GRUs** gaining traction in sequence-based tasks like language modeling and speech recognition.
- **2017: Transformer models**, introduced by Vaswani et al. in "Attention is All You Need," revolutionized NLP by using a self-attention mechanism for

parallel processing, improving efficiency. Models like **BERT**, **GPT**, and **T5** set new benchmarks in machine translation and text generation.

Layers in Neural Network Architecture

1. **Input Layer:** This is where the network receives its input data. Each input neuron in the layer corresponds to a feature in the input data.
2. **Hidden Layers:** These layers perform most of the computational heavy lifting. A neural network can have one or multiple hidden layers. Each layer consists of units (neurons) that transform the inputs into something that the output layer can use.
3. **Output Layer:** The final layer produces the output of the model. The format of these outputs varies depending on the specific task (e.g., classification, regression).



Working of Neural Networks

Forward Propagation

When data is input into the network, it passes through the network in the forward direction, from the input layer through the hidden layers to the output layer. This process is known as forward propagation. Here's what happens during this phase:

1. **Linear Transformation:** Each neuron in a layer receives inputs, which are multiplied by the weights associated with the connections. These products are summed together, and a bias is added to the sum. This can be represented mathematically as: z=w1x1+w2x2+…+wnxn+bz=$w_1x_1+w_2x_2+\ldots+w_n x_n+b$ where ww represents the weights, xx represents the inputs, and bb is the bias.
2. **Activation:** The result of the linear transformation (denoted as zz) is then passed through an activation function. The activation function is crucial because it introduces non-linearity into the system, enabling the network to

learn more complex patterns. Popular activation functions include ReLU, sigmoid, and tanh.

Backpropagation

After forward propagation, the network evaluates its performance using a loss function, which measures the difference between the actual output and the predicted output. The goal of training is to minimize this loss. This is where backpropagation comes into play:

1. **Loss Calculation:** The network calculates the loss, which provides a measure of error in the predictions. The loss function could vary; common choices are mean squared error for regression tasks or cross-entropy loss for classification.
2. **Gradient Calculation:** The network computes the gradients of the loss function with respect to each weight and bias in the network. This involves applying the chain rule of calculus to find out how much each part of the output error can be attributed to each weight and bias.
3. **Weight Update:** Once the gradients are calculated, the weights and biases are updated using an optimization algorithm like stochastic gradient descent (SGD). The weights are adjusted in the opposite direction of the gradient to minimize the loss. The size of the step taken in each update is determined by the learning rate.

Iteration

This process of forward propagation, loss calculation, backpropagation, and weight update is repeated for many iterations over the dataset. Over time, this iterative process reduces the loss, and the network's predictions become more accurate.

Through these steps, neural networks can adapt their parameters to better approximate the relationships in the data, thereby improving their performance on tasks such as classification, regression, or any other predictive modeling.

Example of Email Classification

Let's consider a record of an email dataset:

Email ID	Email Content	Sender	Subject Line	Label
1	"Get free gift cards now!"	spam@example.com	"Exclusive Offer"	1

To classify this email, we will create a feature vector based on the analysis of keywords such as "free," "win," and "offer."
The feature vector of the record can be presented as:

- "free": Present (1)
- "win": Absent (0)
- "offer": Present (1)

Email ID	Email Content	Sender	Subject Line	Feature Vector	Label
1	"Get free gift cards now!"	spam@example.com	"Exclusive Offer"	[1, 0, 1]	1

How Neurons Process Data in a Neural Network

In a **neural network**, input data is passed through multiple layers, including one or more **hidden layers**. Each **neuron** in these hidden layers performs several operations, transforming the input into a usable output.

1. Input Layer: The input layer contains 3 nodes that indicates the presence of each keyword.

2. Hidden Layer

- The input data is passed through one or more hidden layers.
- Each neuron in the hidden layer performs the following operations

  Weighted Sum: Each input is multiplied by a corresponding weight assigned to the connection. For example, if the weights from the input layer to the hidden layer neurons are as follows:
    - Weights for Neuron H1: [0.5, -0.2, 0.3]
    - Weights for Neuron H2: [0.4, 0.1, -0.5]

  Calculate Weighted Input:
    - For Neuron H1:
      - Calculation=(1×0.5)+(0×−0.2)+(1×0.3)=0.5+0+0.3=0.8Calculation=(1×0.5)+(0×−0.2)+(1×0.3)=0.5+0+0.3=0.8
    - For Neuron H2:

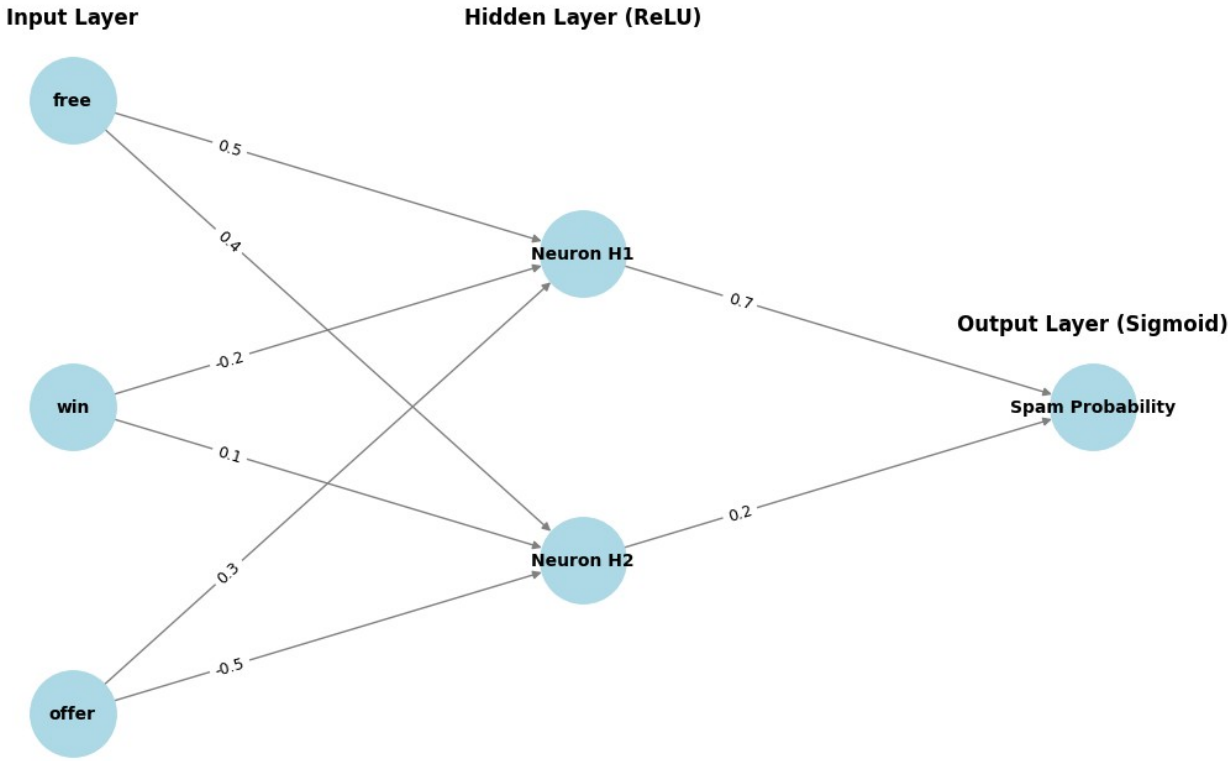
    - Calculation=(1×0.4)+(0×0.1)+(1×−0.5)=0.4+0−0.5=−0.1Calculation=(1×0.4)+(0×0.1)+(1×−0.5)=0.4+0−0.5=−0.1
  - **Activation Function**: The result is passed through an activation function (e.g., ReLU or sigmoid) to introduce non-linearity.
    - For H1, applying ReLU: ReLU(0.8)=0.8ReLU(0.8)=0.8
    - For H2, applying ReLU: ReLU(−0.1)=0ReLU(−0.1)=0

3. Output Layer

- The activated outputs from the hidden layer are passed to the output neuron.
- The output neuron receives the values from the hidden layer neurons and computes the final prediction using weights:
  - Suppose the output weights from hidden layer to output neuron are [0.7, 0.2].
  - Calculation:
    - Input=(0.8×0.7)+(0×0.2)=0.56+0=0.56Input=(0.8×0.7)+(0×0.2)=0.56+0=0.56
  - **Final Activation**: The output is passed through a sigmoid activation function to obtain a probability:
    - σ(0.56)≈0.636σ(0.56)≈0.636

4. Final Classification

- The output value of approximately **0.636** indicates the probability of the email being spam.
- Since this value is greater than 0.5, the neural network classifies the email as spam (1).



Neural Network for Email Classification Example

Learning of a Neural Network

1. Learning with Supervised Learning

In supervised learning, a neural network learns from labeled input-output pairs provided by a teacher. The network generates outputs based on inputs, and by comparing these outputs to the known desired outputs, an error signal is created. The network iteratively adjusts its parameters to minimize errors until it reaches an acceptable performance level.

2. Learning with Unsupervised Learning

Unsupervised learning involves data without labeled output variables. The primary goal is to understand the underlying structure of the input data (X). Unlike supervised learning, there is no instructor to guide the process. Instead, the focus is on modeling data patterns and relationships, with techniques like clustering and association commonly used.

3. Learning with Reinforcement Learning

Reinforcement learning enables a neural network to learn through interaction with its environment. The network receives feedback in the form of rewards or penalties, guiding it to find an optimal policy or strategy that maximizes

cumulative rewards over time. This approach is widely used in applications like gaming and decision-making.

Types of Neural Networks

There are *seven* types of neural networks that can be used.

- **Feedforward Networks:** A feedforward neural network is a simple artificial neural network architecture in which data moves from input to output in a single direction.
- **Singlelayer Perceptron:** A **single-layer perceptron** consists of only one layer of neurons . It takes inputs, applies weights, sums them up, and uses an activation function to produce an output**.**
- **Multilayer Perceptron (MLP):** MLP is a type of feedforward neural network with three or more layers, including an input layer, one or more hidden layers, and an output layer. It uses nonlinear activation functions.
- **Convolutional Neural Network (CNN):** A Convolutional Neural Network (CNN) is a specialized artificial neural network designed for image processing. It employs convolutional layers to automatically learn hierarchical features from input images, enabling effective image recognition and classification.
- **Recurrent Neural Network (RNN):** An artificial neural network type intended for sequential data processing is called a Recurrent Neural Network (RNN). It is appropriate for applications where contextual dependencies are critical, such as time series prediction and natural language processing, since it makes use of feedback loops, which enable information to survive within the network.
- **Long Short-Term Memory (LSTM):** LSTM is a type of RNN that is designed to overcome the vanishing gradient problem in training RNNs. It uses memory cells and gates to selectively read, write, and erase information.

Implementation of Neural Network using TensorFlow

Here, we implement simple feedforward neural network that trains on a sample dataset and makes predictions using following steps:

Step 1: Import Necessary Libraries

Import necessary libraries, primarily TensorFlow and Keras, along with other required packages such as NumPy and Pandas for data handling.

```
import numpy as np
import pandas as pd
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

Step 2: Create and Load Dataset

- Create or load a dataset. Convert the data into a format suitable for training (usually NumPy arrays).
- Define features (X) and labels (y).

```
data = {
   'feature1': [0.1, 0.2, 0.3, 0.4, 0.5],
   'feature2': [0.5, 0.4, 0.3, 0.2, 0.1],
   'label': [0, 0, 1, 1, 1]
}

df = pd.DataFrame(data)
X = df[['feature1', 'feature2']].values
y = df['label'].values
```

Step 3: Create a Neural Network

Instantiate a Sequential model and add layers. The input layer and hidden layers are typically created using Dense layers, specifying the number of neurons and activation functions.

```
model = Sequential()
model.add(Dense(8, input_dim=2, activation='relu'))  # Hidden layer
model.add(Dense(1, activation='sigmoid'))  # Output layer
```

Step 4: Compiling the Model

Compile the model by specifying the loss function, optimizer, and metrics to evaluate during training.

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

Step 5: Train the Model

Fit the model on the training data, specifying the number of epochs and batch size. This step trains the neural network to learn from the input data.

```
model.fit(X, y, epochs=100, batch_size=1, verbose=1)
```

Step 5: Make Predictions

Use the trained model to make predictions on new data. Process the output to interpret the predictions (e.g., convert probabilities to binary outcomes).

```
test_data = np.array([[0.2, 0.4]])
prediction = model.predict(test_data)
predicted_label = (prediction > 0.5).astype(int)
```

Complete Code for the Implementation

Let's have a complete code for the implementation.

```
import numpy as np
import pandas as pd
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

```
data = {
    'feature1': [0.1, 0.2, 0.3, 0.4, 0.5],
    'feature2': [0.5, 0.4, 0.3, 0.2, 0.1],
    'label': [0, 0, 1, 1, 1]
}

df = pd.DataFrame(data)

X = df[['feature1', 'feature2']].values
y = df['label'].values

model = Sequential()

model.add(Dense(8, input_dim=2, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

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

model.fit(X, y, epochs=100, batch_size=1, verbose=1)

test_data = np.array([[0.2, 0.4]])

prediction = model.predict(test_data)
predicted_label = (prediction > 0.5).astype(int)
print(f"Predicted label: {predicted_label[0][0]}")
```

Output:

Predicted label: 1

Advantages of Neural Networks

Neural networks are widely used in many different applications because of their many benefits:

- **Adaptability:** Neural networks are useful for activities where the link between inputs and outputs is complex or not well defined because they can adapt to new situations and learn from data.
- **Pattern Recognition:** Their proficiency in pattern recognition renders them efficacious in tasks like as audio and image identification, natural language processing, and other intricate data patterns.
- **Parallel Processing:** Because neural networks are capable of parallel processing by nature, they can process numerous jobs at once, which speeds up and improves the efficiency of computations.
- **Non-Linearity:** Neural networks are able to model and comprehend complicated relationships in data by virtue of the non-linear activation

functions found in neurons, which overcome the drawbacks of linear models.

Disadvantages of Neural Networks

Neural networks, while powerful, are not without drawbacks and difficulties:

- **Computational Intensity:** Large neural network training can be a laborious and computationally demanding process that demands a lot of computing power.
- **Black box Nature:** As "black box" models, neural networks pose a problem in important applications since it is difficult to understand how they make decisions.
- **Overfitting:** Overfitting is a phenomenon in which neural networks commit training material to memory rather than identifying patterns in the data. Although regularization approaches help to alleviate this, the problem still exists.
- **Need for Large datasets:** For efficient training, neural networks frequently need sizable, labeled datasets; otherwise, their performance may suffer from incomplete or skewed data.

Applications of Neural Networks

Neural networks have numerous applications across various fields:

1. **Image and Video Recognition**: CNNs are extensively used in applications such as facial recognition, autonomous driving, and medical image analysis.
2. **Natural Language Processing (NLP)**: RNNs and transformers power language translation, chatbots, and sentiment analysis.
3. **Finance**: Predicting stock prices, fraud detection, and risk management.
4. **Healthcare**: Neural networks assist in diagnosing diseases, analyzing medical images, and personalizing treatment plans.
5. **Gaming and Autonomous Systems**: Neural networks enable real-time decision-making, enhancing user experience in video games and enabling autonomous systems like self-driving cars.

Limits of Traditional Computing

Traditional computing has long been the backbone of technology, built upon the principles of **algorithmic logic**, **procedural programming**, and **deterministic execution**. In this model, a **programmer explicitly defines all instructions** the computer must follow to solve a problem. While this approach works remarkably well for structured, well-defined problems (e.g., arithmetic, database queries, payroll processing), it becomes inadequate when faced with **ambiguous, noisy, or complex real-world tasks** like facial recognition, speech-to-text, and autonomous driving.

As technology evolved, the demand for systems that can **learn**, **adapt**, and **operate under uncertainty** increased—highlighting the **limits of traditional computing** and laying the foundation for **machine learning and deep learning**.

◆ . Architecture of Traditional Computing

A traditional program works based on the **Von Neumann architecture**, following a basic "input → process → output" cycle. The processor executes a list of instructions stored in memory, one at a time, using control logic.

Basic Workflow:

```
text
CopyEdit
Input → Program Logic (Rules) → Output
```

Example:
A program designed to calculate tax would:

- Accept income as input
- Apply a pre-defined formula (fixed rules)
- Output the tax amount

There's **no learning** involved. The system only follows instructions.

◆ . Core Characteristics of Traditional Systems

Aspect	Description
Determinism	Always gives the same output for the same input.
Static Logic	Requires hardcoded logic for all operations.
Rule-based	Must define every condition, edge case, and exception manually.
No Learning	Cannot adapt to new data unless reprogrammed.
Low Flexibility	Poor handling of unexpected or ambiguous inputs.
Structured Input	Best suited for clearly formatted data like numbers or fixed strings.

◆ . Key Limitations of Traditional Computing

A. Rule Explosion in Complex Systems

As the complexity of a problem increases, the number of required rules **grows exponentially**. This becomes unmanageable.

📌 Example: Creating a handwriting recognition program would require separate rules for every possible style and stroke of each letter.

B. Lack of Adaptability

Traditional systems cannot adapt or update themselves based on experience or new data. If the environment changes, the code must be **manually rewritten or patched**.

📌 Example: A fraud detection program based on fixed patterns may miss new or evolving fraud schemes.

C. Handling Noisy or Unstructured Data

Traditional systems work best on **clean, numerical data**. They struggle with **unstructured inputs** such as:

- Natural language (text with slang, grammar errors)
- Images and videos (variation in color, angle, lighting)
- Audio (background noise, pitch changes)

📌 Example: Voice command systems fail in noisy environments without machine learning models trained on variable datasets.

D. Manual Feature Engineering

In traditional computing, the programmer must decide **what data attributes (features)** are important and how to process them. This limits generalization and automation.

📌 Example: In facial recognition, a rule-based system would require manual detection of nose length, eye spacing, etc.

E. Inability to Generalize

Traditional systems follow exact instructions. They **don't generalize** beyond what they are explicitly programmed for.

📌 Example: A chess engine built with hardcoded rules won't learn new strategies unless manually added by the developer.

F. Not Scalable for Big Data

With increasing data volumes, rule-based systems become:

- Computationally expensive
- Difficult to maintain
- Poorly optimized for real-time decision-making

📌 Example: Processing millions of customer interactions using fixed logic is inefficient and error-prone.

◆ . Why Deep Learning is the Solution

Deep learning overcomes traditional limitations by learning **directly from data**, rather than relying on explicitly programmed logic.

Limitation in Traditional Systems	Deep Learning Capability
Rule explosion	Learns patterns from data, no need for manual rules
No adaptability	Models update automatically with new data
Poor handling of unstructured data	Excels at processing images, speech, and text
Manual feature engineering	Automatically extracts features using neural layers

Limitation in Traditional Systems	Deep Learning Capability
No generalization	Generalizes well to unseen data if trained properly
Not scalable	Scales effectively with large data and computing power

◆ . Real-World Comparisons

Task	Traditional Approach	Deep Learning Approach
Spam detection	Fixed keywords and sender rules	Learns patterns in text, sender behavior
Image classification	Rule-based pixel pattern matching	Learns features like edges, shapes, objects
Language translation	Predefined grammar/phrases	Learns from large parallel text corpora
Stock market prediction	Basic indicators and thresholds	Learns from historical trends and time series
Face recognition	Rule for eye/nose distance, ratios	Learns from image datasets with variation

◆ . Example: Handwriting Recognition

Traditional System Logic:

- Define shape and line rules for each letter (e.g., loops for “e”)
- Hard to distinguish between “e” and “c”, or messy writing

Deep Learning System:

- Learns patterns from thousands of samples

- Tolerates variations in shape, slant, or spacing
- Performs well even on messy handwriting

◆ . Summary

Traditional computing is:

- **Static** (fixed logic)
- **Manual** (requires detailed programming)
- **Rigid** (cannot learn or improve)

In contrast, modern AI systems like deep learning are:

- **Dynamic** (learn from experience)
- **Scalable** (work with massive datasets)
- **Versatile** (handle vision, speech, text)

While traditional computing still plays a role in structured and predictable environments (e.g., compilers, database management), deep learning is crucial for **intelligent systems**, **automation**, and **decision-making in complex domains**.

Machine learning is a branch of Artificial Intelligence that focuses on developing models and algorithms that let computers learn from data without being explicitly programmed for every task. In simple words, ML teaches the systems to think and understand like humans by learning from the data.

It can be broadly categorized into four types:

- **Supervised Learning:** Trains models on labeled data to predict or classify new, unseen data.
- **Unsupervised Learning**: Finds patterns or groups in unlabeled data, like clustering or dimensionality reduction.
- **Reinforcement Learning**: Learns through trial and error to maximize rewards, ideal for decision-making tasks.

Note: *Self-supervised learning is not one of the original three, but it has become a major category in deep learning and fields like NLP and computer vision.*

Semi-Supervised Learning: *The model generates its own labels from the data, so we don't need human-annotated labels.*

Module 1: Machine Learning Pipeline

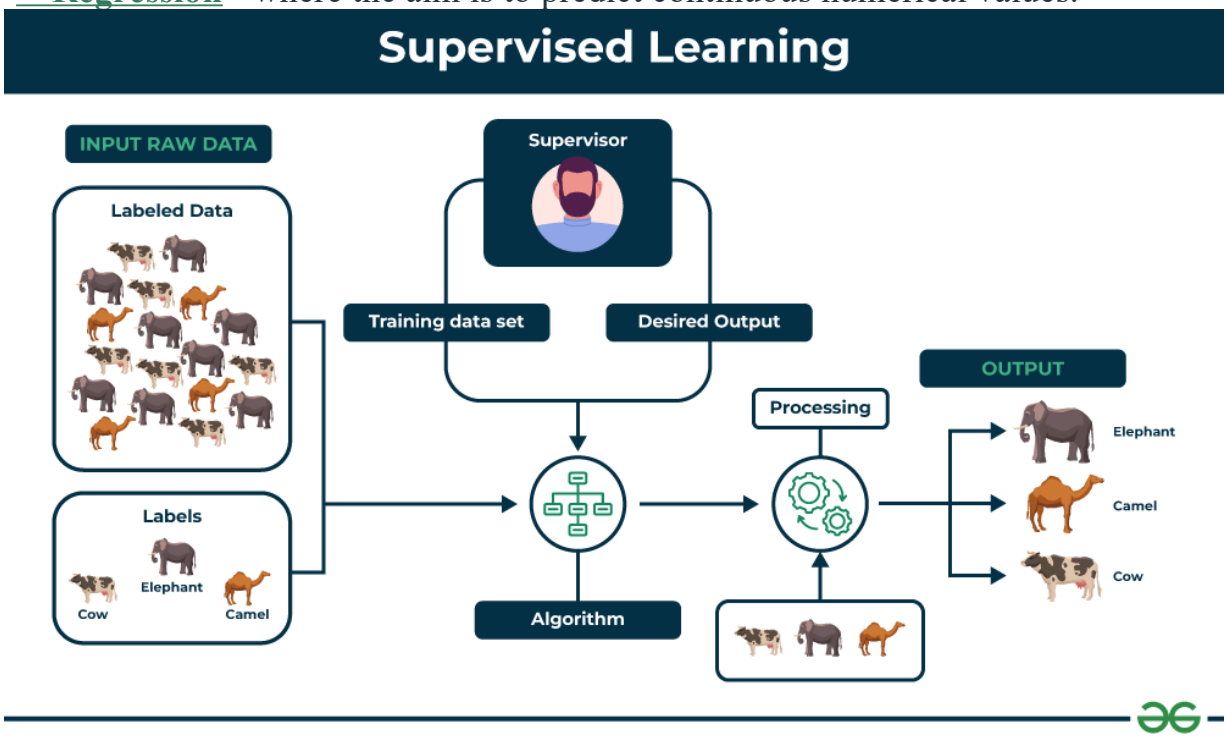
In order to make predictions there are some steps through which data passes in order to produce a machine learning model that can make predictions.

1. **ML workflow**
2. **Data Cleaning**
3. **Feature Scaling**
4. **Data Preprocessing in Python**

Module 2: Supervised Learning

Supervised learning algorithms are generally categorized into **two main types:**

- **Classification** - where the goal is to predict discrete labels or categories
- **Regression** - where the aim is to predict continuous numerical values.



Supervised Learning

There are many algorithms used in supervised learning each suited to different types of problems. Some of the most commonly used supervised learning algorithms are:

1. Linear Regression

This is one of the simplest ways to predict numbers using a straight line. It helps find the relationship between input and output.

- Introduction to Linear Regression
- Gradient Descent in Linear Regression
- Multiple Linear Regression
- Ridge Regression
- Lasso regression
- Elastic net Regression

2. Logistic Regression

Used when the output is a "yes or no" type answer. It helps in predicting categories like pass/fail or spam/not spam.

- Understanding Logistic Regression
- Cost function in Logistic Regression

3. Decision Trees

A model that makes decisions by asking a series of simple questions, like a flowchart. Easy to understand and use.

- Decision Tree in Machine Learning
- Types of Decision tree algorithms
- Decision Tree - Regression (Implementation)
- Decision tree - Classification (Implementation)

4. Support Vector Machines (SVM)

A bit more advanced—it tries to draw the best line (or boundary) to separate different categories of data.

- Understanding SVMs
- SVM Hyperparameter Tuning - GridSearchCV
- Non-Linear SVM

5. k-Nearest Neighbors (k-NN)

This model looks at the closest data points (neighbors) to make predictions. Super simple and based on similarity.

- Introduction to KNN
- Decision Boundaries in K-Nearest Neighbors (KNN)

6. Naïve Bayes

A quick and smart way to classify things based on probability. It works well for text and spam detection.

- Introduction to Naive Bayes
- Gaussian Naive Bayes
- Multinomial Naive Bayes
- Bernoulli Naive Bayes
- Complement Naive Bayes

7. Random Forest (Bagging Algorithm)

A powerful model that builds lots of decision trees and combines them for better accuracy and stability.

- Introduction to Random forest
- Random Forest Classifier
- Random Forest Regression
- Hyperparameter Tuning in Random Forest

Introduction to Ensemble Learning

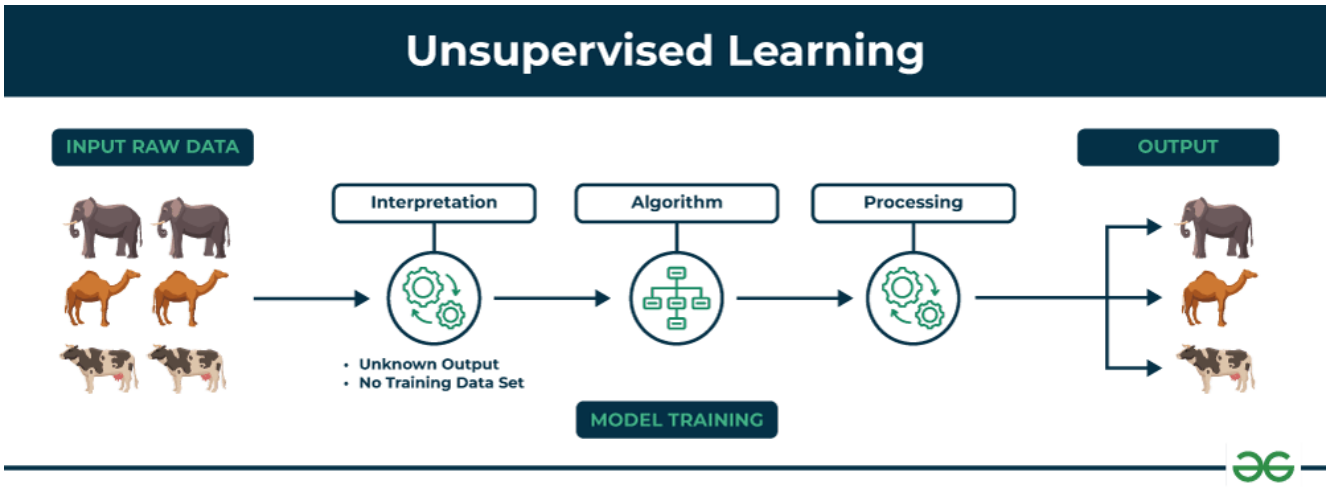
Ensemble learning combines multiple simple models to create a stronger, smarter model. There are mainly two types of ensemble learning:

- Bagging that combines multiple models trained independently.
- Boosting that builds models sequentially each correcting the errors of the previous one.

Module 3: Unsupervised learning

Unsupervised learning are again divided into **three main categories** based on their purpose:

- Clustering
- Association Rule Mining
- Dimensionality Reduction.



Unsupervised learning

1. Clustering

Clustering algorithms group data points into clusters based on their similarities or differences. Types of clustering algorithms are:

Centroid-based Methods:

- K-Means clustering
- Elbow Method for optimal value of k in KMeans
- K-Means++ clustering
- K-Mode clustering
- Fuzzy C-Means (FCM) Clustering

Distribution-based Methods:

- Gaussian mixture models
- Expectation-Maximization Algorithm
- Dirichlet process mixture models (DPMMs)

Connectivity based methods:

- Hierarchical clustering
- Agglomerative Clustering
- Divisive clustering
- Affinity propagation

Density Based methods:

- DBSCAN (Density-Based Spatial Clustering of Applications with Noise)
- OPTICS (Ordering Points To Identify the Clustering Structure)

2. Dimensionality Reduction

Dimensionality reduction is used to simplify datasets by reducing the number of features while retaining the most important information.

- **Principal Component Analysis (PCA)**
- **t-distributed Stochastic Neighbor Embedding (t-SNE)**
- **Non-negative Matrix Factorization (NMF)**
- **Independent Component Analysis (ICA)**
- **Isomap**
- **Locally Linear Embedding (LLE)**

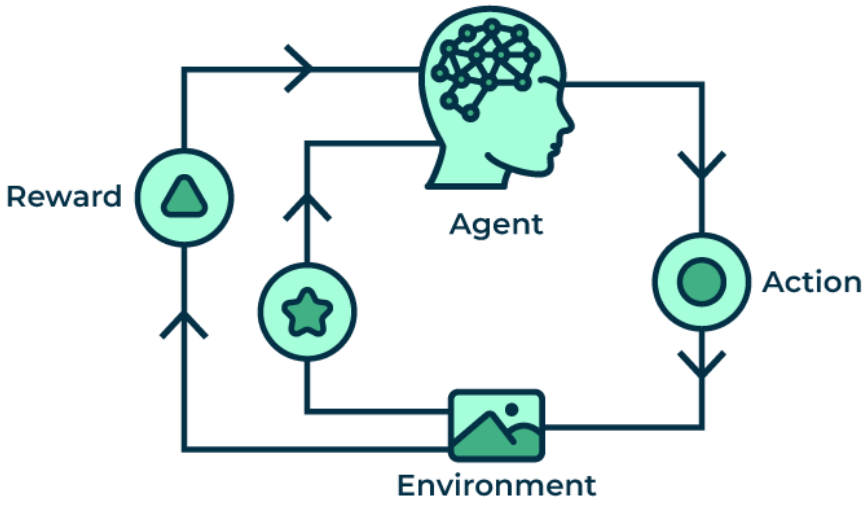
3. Association Rule

Find patterns between items in large datasets typically in market basket analysis.

- **Apriori algorithm**
- **Implementing apriori algorithm**
- **FP-Growth (Frequent Pattern-Growth)**
- **ECLAT (Equivalence Class Clustering and bottom-up Lattice Traversal)**

Module 4: Reinforcement Learning

Reinforcement learning interacts with environment and learn from them based on rewards.



Reinforcement Learning

1. Model-Based Methods

These methods use a model of the environment to predict outcomes and help the agent plan actions by simulating potential results.

- **Markov decision processes (MDPs)**
- **Bellman equation**

- Value iteration algorithm
- Monte Carlo Tree Search

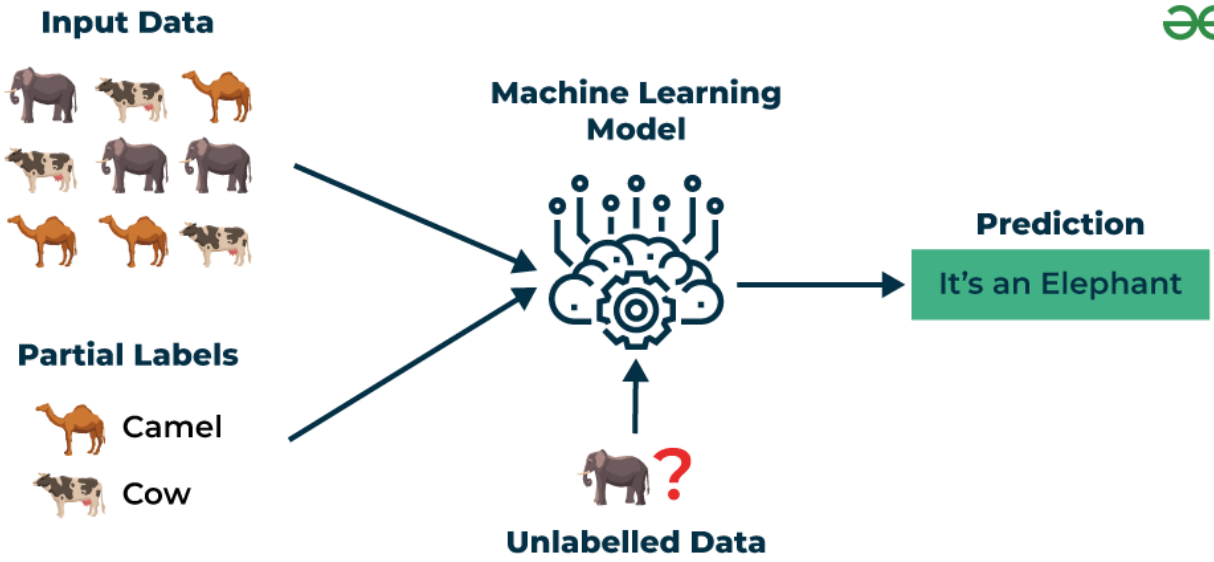
2. Model-Free Methods

The agent learns directly from experience by interacting with the environment and adjusting its actions based on feedback.

- Q-Learning
- SARSA
- Monte Carlo Methods
- Reinforce Algorithm
- Actor-Critic Algorithm
- Asynchronous Advantage Actor-Critic (A3C)

Module 5: Semi Supervised Learning

It uses a mix of labeled and unlabeled data making it helpful when labeling data is costly or it is very limited.



Semi Supervised Learning

- Semi Supervised Classification
- Self-Training in Semi-Supervised Learning
- Few-shot learning in Machine Learning

Module 6: Deployment of ML Models

The trained ML model must be integrated into an application or service to make its predictions accessible.

- Machine learning deployement
- Deploy ML Model using Streamlit Library

- **Deploy ML web app on Heroku**
- **Create UIs for prototyping Machine Learning model with Gradio**

APIs allow other applications or systems to access the ML model's functionality and integrate them into larger workflows.

- **Deploy Machine Learning Model using Flask**
- **Deploying ML Models as API using FastAPI**

MLOps ensure they are deployed, monitored and maintained efficiently in real-world production systems.

- **MLOps**
- **Continuous Integration and Continuous Deployment (CI/CD) in MLOps**
- **End-to-End MLOps**

Neuron

◆ 1. Introduction

In artificial intelligence and deep learning, the **neuron** is the **fundamental processing unit** of an artificial neural network (ANN). Inspired by the functioning of biological neurons in the human brain, artificial neurons are designed to **simulate the way humans learn and process information**.

A biological neuron receives input from neighboring neurons, processes it, and transmits an electrical signal to other neurons. Similarly, in artificial systems, a **neuron receives inputs (numerical values), processes them using weights and biases**, applies an **activation function**, and **produces an output** that is passed on to the next layer.

Artificial neurons **learn patterns from data** by **adjusting weights** and **biases** during training. The effectiveness of deep learning largely depends on the **architecture**, **quantity**, and **interconnection** of these neurons.

◆ 2. Biological vs Artificial Neuron

Attribute	Biological Neuron	Artificial Neuron
Input	Dendrites receive signals from other neurons	Input values (features) from previous layers
Processing Center	Soma (cell body) integrates signals	Computes a weighted sum of inputs + bias
Output Mechanism	Axon transmits electrical impulses	Produces a scalar output after activation
Learning Process	Synaptic strength changes via plasticity	Weights and bias updated via backpropagation
Activation	Triggered when potential exceeds threshold	Activation function decides firing

◆ 3. Mathematical Model of a Neuron

Let:

- x1,x2,...,xnx_1, x_2, ..., x_nx1,x2,...,xn be the inputs to the neuron
- w1,w2,...,wnw_1, w_2, ..., w_nw1,w2,...,wn be the weights
- bbb be the bias term
- fff be the activation function

◆ Weighted Sum:

z=∑i=1n(xi⋅wi)+bz = \sum_{i=1}^{n}(x_i \cdot w_i) + bz=i=1∑n(xi⋅wi)+b

◆ Activation:

a=f(z)a = f(z)a=f(z)

- zzz: Linear combination of inputs and weights + bias
- aaa: Activated output of the neuron

This output aaa becomes the input to the next neuron in the network.

◆ 4. Components of a Neuron

Component	Role
Inputs (xxx)	Numerical data features passed from input or previous layers
Weights (www)	Determines the importance of each input to the neuron's decision
Bias (bbb)	Adjusts the output, helps shift the activation curve
Summation (zzz)	Computes weighted input plus bias
Activation Function	Decides whether the neuron should be activated (fired)
Output (aaa)	The result passed to next layer

◆ 5. Activation Functions – The Neuron's Decision Maker

Activation functions decide **whether and how much** a neuron "fires." They introduce **non-linearity**, allowing networks to solve complex, real-world problems.

Common Activation Functions:

a. Sigmoid Function

f(z)=11+e−zf(z) = \frac{1}{1 + e^{-z}}f(z)=1+e−z1

- Range: (0, 1)
- Pros: Smooth and differentiable
- Cons: Saturates for large inputs; slow gradient
- Use: Binary classification, output layers

b. Tanh (Hyperbolic Tangent)

f(z)=ez−e−zez+e−zf(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}f(z)=ez+e−zez−e−z

- Range: (-1, 1)
- Pros: Zero-centered
- Cons: Still suffers from saturation
- Use: Hidden layers

c. ReLU (Rectified Linear Unit)

f(z)=max(0,z)f(z) = \max(0, z)f(z)=max(0,z)

- Range: [0, ∞)
- Pros: Fast, sparse activation, computationally efficient
- Cons: "Dying ReLU" problem where neurons output 0 always
- Use: Most hidden layers

d. Softmax

f(zi)=ezi∑jezjf(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}f(zi)=∑jezjezi

- Converts outputs into probability distribution
- Used in **multi-class classification** output layers

◆ 6. Visual Representation of a Single Neuron

```
lua
CopyEdit
          x1 ----\
          x2 ----|  \
          x3 ----|   >---- [ ∑(xi*wi) + b ] ----> f(z) ----> Output
          .      |  /
          xn ----/
```

◆ 7. How a Neuron Learns – Learning Process

A neuron learns via **backpropagation and gradient descent**.

Steps:

1. **Forward Pass**: Compute output using weights and activation
2. **Loss Calculation**: Compare predicted output with actual label using loss function (e.g., MSE, Cross-Entropy)
3. **Backward Pass**: Calculate gradient of loss with respect to weights using **chain rule**
4. **Weight Update**: Adjust weights to reduce loss using:

   wi:=wi−η⋅∂L∂wiw_i := w_i - \eta \cdot \frac{\partial L}{\partial w_i}wi:=wi−η⋅∂wi∂L

   where:

   - η\etaη = learning rate
   - LLL = loss function

◆ 8. Role of Bias in a Neuron

The **bias** allows the activation function to be shifted left or right. This makes the neuron more flexible and helps in fitting data that **does not pass through the origin**.

- Without bias, all activation functions are restricted to a specific shape.
- Bias ensures better performance and **faster convergence** during training.

◆ 9. Types of Neurons (Based on Activation)

Type	Activation Function Used	Use Case
Linear Neuron	f(z)=zf(z) = zf(z)=z	Rare; used in regression
Binary Neuron	Step or Threshold	Early models (Perceptron)
Sigmoid Neuron	Sigmoid	Binary outputs; old binary classifiers
ReLU Neuron	ReLU	Most deep networks today
Softmax Neuron	Softmax	Multi-class classification output

◆ 10. Applications of Neurons in Neural Networks

- **Convolutional Layers** (CNNs): Specialized neurons detect patterns in images.
- **Recurrent Neurons** (RNNs): Maintain state over sequences for time-series data.
- **Transformer Neurons**: Use attention mechanisms to weigh input importance in NLP tasks.
- **Autoencoders**: Compress and reconstruct data using bottleneck neuron layers.

◆ 11. Challenges with Neurons

- **Vanishing Gradient**: Especially in sigmoid/tanh activations, gradients become too small, slowing learning.
- **Dead Neurons**: In ReLU, if input is negative too often, neuron stops firing.

- **Overfitting**: Too many neurons can memorize training data rather than generalize.
- **Hyperparameter Sensitivity**: Neuron behavior is influenced by learning rate, initialization, etc.

◆ 12. Real-World Analogy

Think of a neuron as a **decision unit**:

- Inputs = ingredients in a recipe
- Weights = how important each ingredient is
- Bias = chef's twist (personal adjustment)
- Activation = final decision whether the dish is ready to serve
- Output = the dish passed to next kitchen station (layer)

◆ 13. Summary

- A **neuron** is the smallest computational unit in a neural network.
- It performs a **weighted sum + bias**, then applies an **activation function**.
- Neurons are organized in layers and enable **learning from data**.
- Proper design of neurons and their activation is critical to the success of deep learning models.

Feedforward Neural Network

Feedforward Neural Network (FNN) is a type of artificial neural network in which information flows in a single direction—from the input layer through hidden layers to the output layer—without loops or feedback. It is mainly used for pattern recognition tasks like image and speech classification.

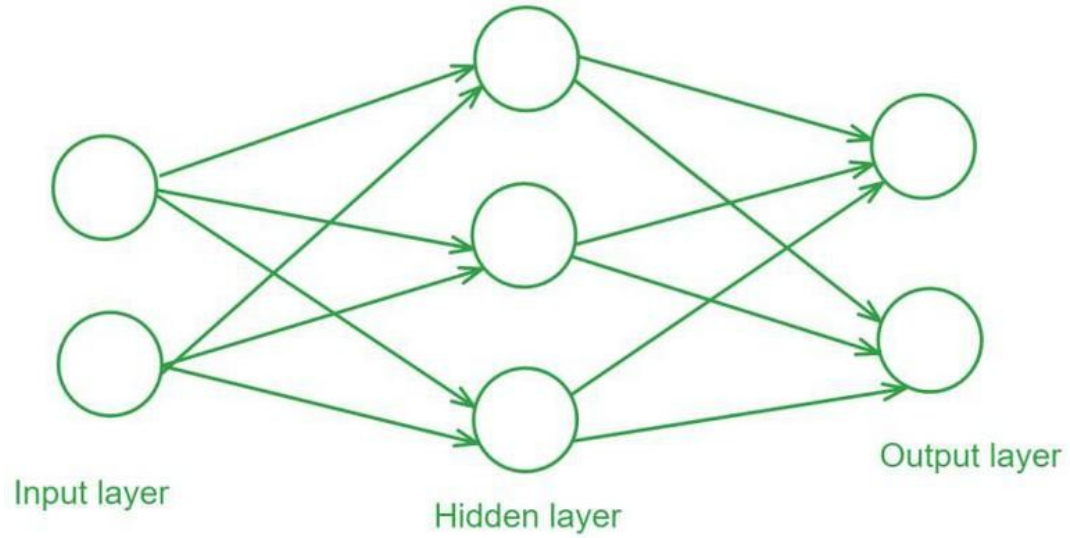
For example in a credit scoring system banks use an FNN which analyze users' financial profiles—such as income, credit history and spending habits —to determine their creditworthiness.
Each piece of information flows through the network's layers where various calculations are made to produce a final score.

Structure of a Feedforward Neural Network

Feedforward Neural Networks have a structured layered design where data flows sequentially through each layer.

1. **Input Layer**: The input layer consists of neurons that receive the input data. Each neuron in the input layer represents a feature of the input data.
2. **Hidden Layers**: One or more hidden layers are placed between the input and output layers. These layers are responsible for learning the complex patterns in the data. Each neuron in a hidden layer applies a weighted sum of inputs followed by a non-linear activation function.
3. **Output Layer**: The output layer provides the final output of the network. The number of neurons in this layer corresponds to the number of classes in a classification problem or the number of outputs in a regression problem.

Each connection between neurons in these layers has an associated weight that is adjusted during the training process to minimize the error in predictions.



Feed Forward Neural Network

Activation Functions

Activation functions introduce non-linearity into the network enabling it to learn and model complex data patterns.

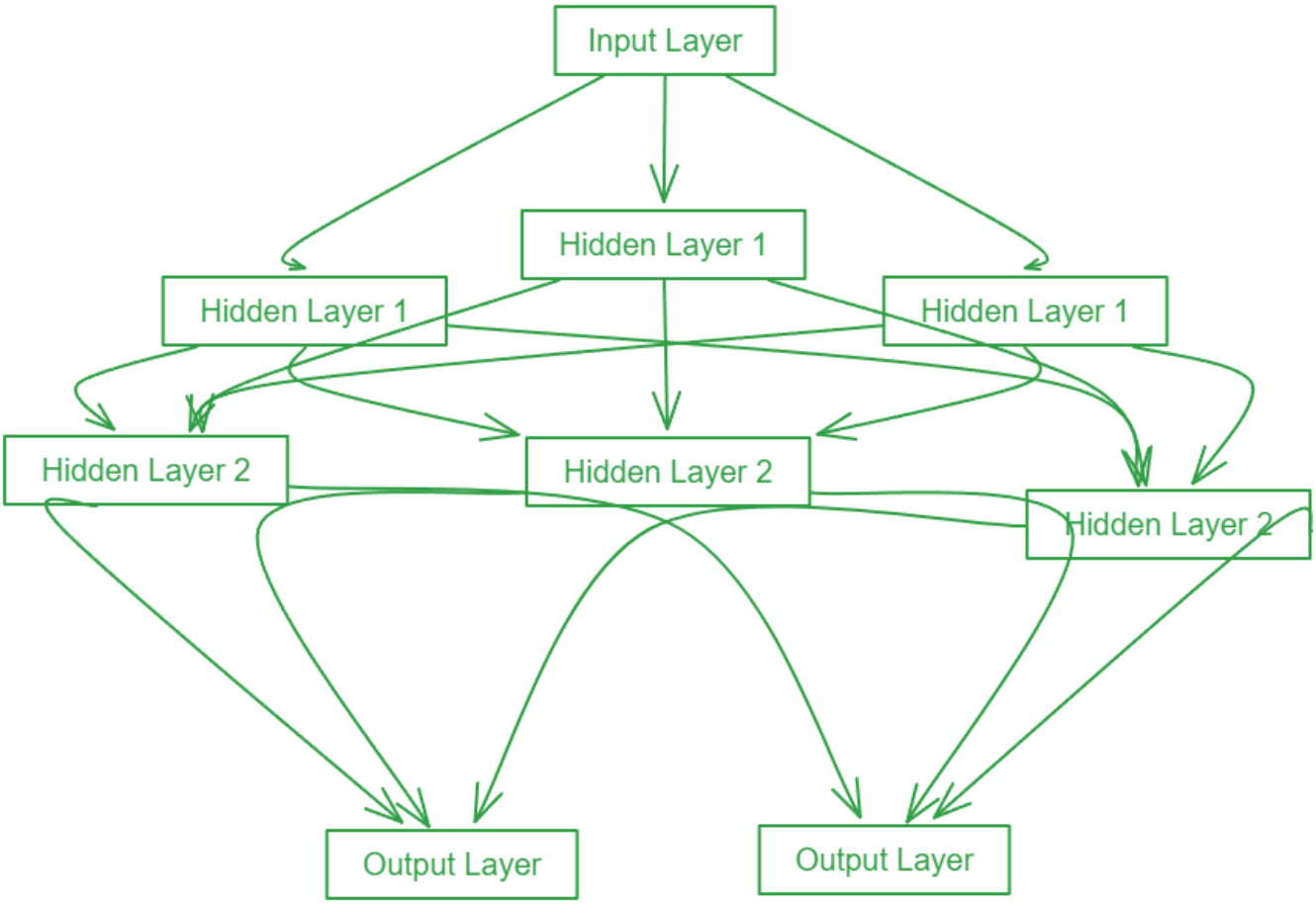
Common activation functions include:

- **Sigmoid**: $\sigma(x)=\sigma(x)=11+e-x\sigma(x)=1+e-x1$
- **Tanh**: $\tanh(x)=ex-e-xex+e-x\tanh(x)=ex+e-xex-e-x$
- **ReLU**: $\text{ReLU}(x)=\max(0,x)\text{ReLU}(x)=\max(0,x)$

Training a Feedforward Neural Network

Training a Feedforward Neural Network involves adjusting the weights of the neurons to minimize the error between the predicted output and the actual output. This process is typically performed using backpropagation and gradient descent.

1. **Forward Propagation**: During forward propagation the input data passes through the network and the output is calculated.
2. **Loss Calculation**: The loss (or error) is calculated using a loss function such as Mean Squared Error (MSE) for regression tasks or Cross-Entropy Loss for classification tasks.
3. **Backpropagation**: In backpropagation the error is propagated back through the network to update the weights. The gradient of the loss function with respect to each weight is calculated and the weights are adjusted using gradient descent.



Forward Propagation

Gradient Descent

Gradient Descent is an optimization algorithm used to minimize the loss function by iteratively updating the weights in the direction of the negative gradient. Common variants of gradient descent include:

- **Batch Gradient Descent**: Updates weights after computing the gradient over the entire dataset.
- **Stochastic Gradient Descent (SGD)**: Updates weights for each training example individually.
- **Mini-batch Gradient Descent**: It Updates weights after computing the gradient over a small batch of training examples.

Evaluation of Feedforward neural network

Evaluating the performance of the trained model involves several metrics:

- **Accuracy**: The proportion of correctly classified instances out of the total instances.

- **Precision**: The ratio of true positive predictions to the total predicted positives.
- **Recall**: The ratio of true positive predictions to the actual positives.
- **F1 Score**: The harmonic mean of precision and recall, providing a balance between the two.
- **Confusion Matrix:** A table used to describe the performance of a classification model, showing the true positives, true negatives, false positives, and false negatives.

Code Implementation of Feedforward neural network

This code demonstrates the process of building, training and evaluating a neural network model using TensorFlow and Keras to classify handwritten digits from the MNIST dataset.

The model architecture is defined using the **Sequential API** consisting of:

- a Flatten layer to convert the 2D image input into a 1D array
- a Dense layer with 128 neurons and ReLU activation
- a final Dense layer with 10 neurons and softmax activation to output probabilities for each digit class.

Model is compiled with the Adam optimizer, SparseCategoricalCrossentropy loss function and SparseCategoricalAccuracy metric and then trained for 5 epochs on the training data.

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.losses import SparseCategoricalCrossentropy
from tensorflow.keras.metrics import SparseCategoricalAccuracy

# Load and prepare the MNIST dataset
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

# Build the model
model = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])

# Compile the model
```

```
model.compile(optimizer=Adam(),
              loss=SparseCategoricalCrossentropy(),
              metrics=[SparseCategoricalAccuracy()])

# Train the model
model.fit(x_train, y_train, epochs=5)

# Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f'\nTest accuracy: {test_acc}')
```

Output:

Test accuracy: *0.9767000079154968*

By understanding their architecture, activation functions, and training process, one can appreciate the capabilities and limitations of these networks. Continuous advancements in optimization techniques and activation functions have made feedforward networks more efficient and effective, contributing to the broader field of artificial intelligence.

Types of Neurons in Deep Learning

◆ 1. Introduction

A **neuron** in a neural network is a **mathematical function** that mimics the behavior of its biological counterpart in the human brain. However, unlike biological neurons, artificial neurons process **numeric vectors**, and their functionality is defined entirely by **mathematical operations**.

The **type of a neuron** in deep learning is primarily determined by the **activation function** it uses. Activation functions play a **crucial role** in determining how the output of a neuron is shaped and how information flows through the network.

Why Different Types of Neurons?

Different neuron types exist because:

- **Different tasks** (e.g., classification, regression, segmentation) require different mathematical behavior.

- Some functions perform better in **certain layers** (e.g., ReLU in hidden layers, Softmax in output).
- Neurons with different activations have **different differentiability, saturation, and computational properties**.

◆ 2. Anatomy of a Neuron

A neuron computes:

z=∑i=1nxiwi+bz = \sum_{i=1}^n x_i w_i + bz=i=1∑nxiwi+b a=f(z)a = f(z)a=f(z)

Where:

- xix_ixi: input feature
- wiw_iwi: corresponding weight
- bbb: bias term
- fff: activation function
- aaa: activated output

The **choice of fff** defines the **type of neuron**.

◆ 3. Classification of Neurons by Activation Function

Table of Neuron Types

Neuron Type	Activation Function	Output Range	Differentiable?	Use Case
Linear Neuron	f(z)=zf(z) = zf(z)=z	(−∞,+∞)(-\infty, +\infty)(−∞,+∞)	✓	Regression tasks
Binary Step Neuron	Step/Threshold	0 or 1	✗	Perceptrons, rule-based classifiers
Sigmoid Neuron	11+e−z\frac{1}{1+e^{-z}}1+e−z1	(0, 1)	✓	Binary classification
Tanh	tanh(z)\tanh(z)tanh(z)	(−1, 1)	✓	Normalized

Neuron Type	Activation Function	Output Range	Differentiable?	Use Case
Neuron				feature extraction
ReLU Neuron	max(0,z)\max(0, z)max(0,z)	[0, ∞)	✓ (partially)	Default for hidden layers
Leaky ReLU Neuron	zzz or 0.01z0.01z0.01z	(−∞, ∞)	✓	Solves ReLU's dying neuron problem
Softmax Neuron	ezi∑ezj\frac{e^{z_i}}{\sum e^{z_j}}∑ezjezi	(0, 1), sum=1	✓	Output layer for multi-class classification

◆ 4. In-Depth Theory of Each Neuron Type

◆ 4.1 Linear Neuron

- **Activation**: f(z)=zf(z) = zf(z)=z
- Outputs the **raw score** without any transformation.
- **Pros**: Used in **regression problems** where output is continuous.
- **Cons**: Cannot model non-linear relationships → not useful in hidden layers.

Linear neurons are still used in **output layers of regression models** but are almost never used inside hidden layers.

◆ 4.2 Binary Step Neuron (Threshold Neuron)

- **Activation**:

  f(z)={1,if z≥θ0,if z<θf(z) = \begin{cases} 1, & \text{if } z \geq \theta \\ 0, & \text{if } z < \theta \end{cases}f(z)={1,0,if z≥θif z<θ

- Mimics the all-or-none firing of a biological neuron.

- **Limitation**: **Not differentiable**, hence **cannot be trained with gradient descent**.
- Historically used in **Perceptron** models but now obsolete for deep learning.

◆ 4.3 Sigmoid Neuron

- **Activation**:

  f(z)=11+e−zf(z) = \frac{1}{1 + e^{-z}}f(z)=1+e−z1

- Squashes the input to a range between **0 and 1**.
- **Use Case**: Binary classification problems; **output layer for 2-class problems**.
- **Issue**:
  - Gradients become very small as output saturates → **vanishing gradient problem**.
  - Outputs are **not zero-centered** → slow convergence.

Despite limitations, sigmoid is still used when output needs to represent **probability**.

◆ 4.4 Tanh Neuron

- **Activation**:

  f(z)=ez−e−zez+e−z=tanh(z)f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \tanh(z)f(z)=ez+e−zez−e−z=tanh(z)

- Range: (−1, 1) → **zero-centered**
- **Advantages**:
  - Better convergence than sigmoid
- **Limitations**:
  - Still suffers from **vanishing gradient** problem

Often used in older architectures like vanilla RNNs.

◆ 4.5 ReLU Neuron (Rectified Linear Unit)

- **Activation**:

f(z)=max(0,z)f(z) = \max(0, z)f(z)=max(0,z)

- Allows only **positive signals** to pass through.
- **Advantages**:
  - **Computationally efficient** (no exponentials)
  - Helps mitigate vanishing gradients
  - Creates **sparsity** (many outputs are 0)
- **Drawback**: **Dying ReLU** – neuron gets stuck when inputs are always negative

ReLU is the **most widely used** activation in modern deep learning models.

◆ 4.6 Leaky ReLU Neuron

- **Activation**:

  f(z)={zif z>0αzif z≤0,α≈0.01f(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha z & \text{if } z \leq 0 \end{cases}, \alpha \approx 0.01f(z)={zαzif z>0if z≤0,α≈0.01

- Fixes the **dying ReLU** problem by allowing a small gradient when z<0z < 0z<0
- Works well in **deep convolutional neural networks** (CNNs)

◆ 4.7 Softmax Neuron

- **Activation**:

  f(zi)=ezi∑j=1nezjf(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}f(zi)=∑j=1nezjezi

- Converts logits into **probabilities** such that:

  ∑i=1nf(zi)=1\sum_{i=1}^{n} f(z_i) = 1i=1∑nf(zi)=1

- Used in **output layer for multi-class classification**
- Each output neuron corresponds to one class; value represents **confidence**.

◆ 5. Summary Table

Neuron Type	Differentiable	Suitable For	Key Strength
Linear	✓	Regression tasks	Outputs continuous values
Step	✗	Historical perceptrons	Mimics simple threshold decision
Sigmoid	✓	Binary classification	Outputs probabilities
Tanh	✓	Hidden layers	Zero-centered; smoother than sigmoid
ReLU	✓ (partially)	Deep networks (CNNs, DNNs)	Fast convergence, efficient training
Leaky ReLU	✓	Deep networks with sparse input	Solves ReLU's zero-gradient issue
Softmax	✓	Output layer (multi-class)	Outputs class-wise probabilities

◆ 6. Practical Considerations for Choosing Neuron Type

- Use **ReLU** or **Leaky ReLU** in hidden layers for **fast convergence**
- Use **Sigmoid** or **Softmax** in output layer depending on classification type:
  - **Sigmoid**: binary classification
  - **Softmax**: multi-class classification
- Use **Tanh** if data is **centered around 0**
- Avoid **Step functions** in modern architectures due to non-differentiability
- Use **Linear neurons** only in output layer for **regression problems**

◆ 7. Visual Intuition

Plotting activation functions shows:

- **ReLU** rises sharply after 0
- **Sigmoid** and **Tanh** flatten for large inputs

- **Softmax** turns scores into relative probabilities

Understanding these behaviors is key to selecting the **right neuron type for your architecture**.

Softmax Output Layer

(Unit I – Deep Learning | BCAAIML502 / BCSAMLCS502)

◆ 1. Introduction

In deep learning, particularly for **multi-class classification**, we need a way to transform the output of a neural network into a **probability distribution** over all possible classes. This is exactly what the **Softmax activation function** does. It is generally used in the **output layer** of a neural network to represent **mutually exclusive class probabilities**.

The Softmax function transforms raw prediction scores (also known as **logits**) into **probabilities that sum up to 1**, making it easier to interpret the model's predictions.

◆ 2. Why Use the Softmax Function?

Neural networks produce **real-valued scores** at the final layer. These scores:

- Are not bounded (can be negative or positive)
- Do not represent probabilities
- Cannot be directly used to make meaningful decisions

The **Softmax function solves this by**:

- Squashing outputs into the range (0, 1)
- Ensuring the total sum of all output values is 1
- Allowing us to interpret each output as a **probability of belonging to a specific class**

◆ 3. Softmax Function – Definition

Given a vector of logits z=[z1,z2,...,zK]z = [z_1, z_2, ..., z_K]z=[z1,z2,...,zK], the Softmax function is defined as:

σ(zi)=ezi∑j=1Kezj\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}σ(zi)=∑j=1Kezjezi

Where:

- ziz_izi is the raw score (logit) for class iii
- KKK is the total number of classes
- σ(zi)\sigma(z_i)σ(zi) is the output probability for class iii

---

◆ 4. Intuition Behind the Formula

- Each logit is exponentiated: ezie^{z_i}ezi ensures **non-negative output**
- The sum in the denominator normalizes all values
- This creates a **smooth distribution** over the possible classes

The effect of exponentiation is to **magnify differences** between larger and smaller values, making the largest value dominate — which helps the model confidently select the most likely class.

◆ 5. Example Calculation

Let's say a network produces the logits:

z=[2.0,1.0,0.1]z = [2.0, 1.0, 0.1]z=[2.0,1.0,0.1]

Then:

e2.0=7.39,e1.0=2.72,e0.1=1.11e^{2.0} = 7.39,\quad e^{1.0} = 2.72,\quad e^{0.1} = 1.11e2.0=7.39,e1.0=2.72,e0.1=1.11

Sum = 7.39 + 2.72 + 1.11 = 11.22

σ(z1)=7.3911.22≈0.658\sigma(z_1) = \frac{7.39}{11.22} \approx 0.658 σ(z1)=11.227.39 ≈0.658 σ(z2)=2.7211.22≈0.242\sigma(z_2) = \frac{2.72}{11.22} \approx 0.242 σ(z2)=11.222.72≈0.242 σ(z3)=1.1111.22≈0.099\sigma(z_3) = \frac{1.11}{11.22} \approx 0.099σ(z3)=11.221.11≈0.099

✅ Interpretation:

- Class 1 has the highest probability (~66%), so it is chosen as the predicted class.

◆ 6. Key Properties

Property	Description
Range	Each output is in (0, 1)
Normalization	Sum of all outputs is exactly 1
Differentiability	The function is smooth and differentiable – essential for backpropagation
Relative Scoring	Shows confidence: larger logits = higher probabilities
Interconnected Outputs	A change in one logit's value affects all output probabilities

◆ 7. Practical Use in Deep Learning

◆ Use Case: Multi-class Classification

- Softmax is used in problems where an input is assigned to **one of several** mutually exclusive classes.

◆ Typical Applications:

- Handwritten digit recognition (MNIST: 10 classes)
- Language translation (each word as a class)
- Image classification (e.g., 1000 classes in ImageNet)

◆ Example:

If your neural network is classifying images into {Cat, Dog, Rabbit}, the final output layer would consist of **3 neurons**, and the Softmax function will convert the logits into class probabilities.

◆ 8. Output Layer + Loss Function: Why Softmax Works Best with Cross-Entropy

The Softmax output is typically **paired with the Categorical Cross-Entropy loss**, which measures how far off the predicted probabilities are from the actual class label.

Cross-Entropy Loss:

L=−∑i=1Kyilog(y^i)L = -\sum_{i=1}^{K} y_i \log(\hat{y}_i)L=−i=1∑Kyilog(y^i)

Where:

- yiy_iyi is the true class label (one-hot encoded)
- y^i\hat{y}_iy^i is the predicted Softmax probability

This combination makes the optimization process smoother and more accurate.

◆ 9. Softmax vs Other Activation Functions

Feature	Softmax	Sigmoid	ReLU
Output Type	Probability distribution	Independent probabilities	Unbounded positive numbers
Output Sum	= 1	Not guaranteed	No relation
Best For	Multi-class classification	Binary or multi-label	Hidden layers
Range	(0, 1), sum = 1	(0, 1)	[0, ∞)
Use in Output Layer	✓	✓ (binary only)	✗ (not for classification)

◆ 10. Numerical Stability: The max(z)\max(z)max(z) Trick

Exponentiation in Softmax can lead to **overflow** (very large values), which can crash the training process.

Solution:

Subtract the maximum logit value before applying Softmax.

σ(zi)=ezi−max(z)∑jezj−max(z)\sigma(z_i) = \frac{e^{z_i - \max(z)}}{\sum_{j} e^{z_j - \max(z)}}σ(zi)=∑jezj−max(z)ezi−max(z)

This **does not affect the final output**, but ensures **numerical safety** and stability.

◆ 11. Summary

- The **Softmax function** converts raw output scores (logits) from a neural network into **probabilities**.
- It is essential in **multi-class classification tasks**, particularly when only **one class should be predicted**.
- It ensures that:
  - All output values are positive
  - The total sum of outputs = 1
- Works best with **categorical cross-entropy** for training
- Must be implemented with **numerical care** to prevent overflow