Summer of Science Report 2025
Generative and Agentic AI (CS21)

Gunjan Kumar (24B4224)
Mentor : Harshil Solanki

July 2025

Figure 1: Genrative and Agentic AI

Contents

1 Intro to AI/ML Concepts

1.1 Introduction

Artificial intelligence (AI) and machine learning (ML) are the foundation of modern technological advancements.
Artificial Intelligence (AI): Artificial Intelligence is a broad field of computer science dedicated to creating intelligent machines that can perform tasks typically requiring human intelligence. These tasks include problem-solving, learning, decision-making, perception, and understanding language

Machine Learning (ML) : Machine Learning is a subset of AI that focuses on developing systems that can learn from data without being explicitly programmed. ML algorithms build a model based on sample data, known as "training data," to make predictions or decisions without being explicitly programmed to perform the task. ML used in everyday technologies, such as recommendation systems, spam filters, and facial recognition
ML offers a new way to solve problems, answer complex questions, and create new content. ML can predict the weather, estimate travel times, recommend songs, auto-complete sentences, summarize articles, and generate never-seen-before images.

1.2 Types of ML Systems

1. Supervised learning
2. Unsupervised learning
3. Reinforcement learning
4. Generative AI

- **Supervised Learning**: In this type, the model learns from labeled data, where each example has input features and a corresponding output label. The goal is to predict labels for new, unseen data.
    1. Regression : A regression model predicts a numeric value. For example, a weather model that predicts the amount of rain, in inches or millimeters, is a regression model.
        - Linear regression
        - Logistic Regression
    2. Classification : Classification models predict the likelihood that something belongs to a category. classification models output a value that states whether or not something belongs to a particular category. For example, classification models are used to predict if an email is spam or if a photo contains a cat.
- Unsupervised Learning: Here, the model learns from unlabeled data. It identifies patterns, clusters, or structures within the data without pre- defined labels.
    - Clustering , Association, Dimensionality resolution

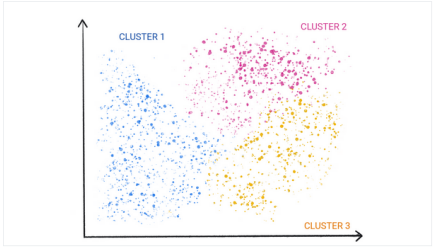


Figure 1. An ML model clustering similar data points.

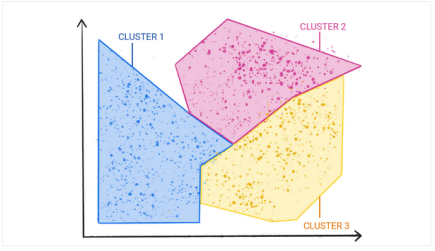


Figure 2. Groups of clusters with natural demarcations.

Figure 2: Clustering Example

- Reinforcement Learning: This learning paradigm involves an agent interacting with an environment. The agent receives rewards or penalties based on its actions, aiming to maximize cumulative rewards over time.
  - Q-Learning , Deep Q-Learning
  - Policy Gradient Method
- Generative AI is a class of models that creates content from user input. Generative AI can take a variety of inputs and create a variety of outputs, like text, images, audio, and video. It can also take and create combinations of these. For example, a model can take an image as input and create an image and text as output, or take an image and text as input and create a video as output.

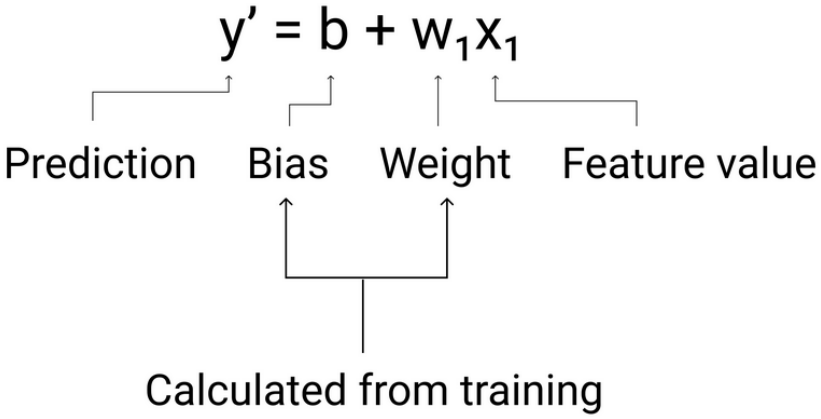
1.2.1 Some Important Terms

Features: Input variable (Measurable properties describing data examples e.g. height, weight, temperature etc.).
Labels: The values we want to predict (Output e.g., price, category).
Training Data: Used to train the model. It consists of labeled examples.
Testing Data: Used to evaluate the model's performance. It contains un- seen examples with known labels.
Overfitting: Model learns training data too well but performs poorly on new data.
Underfitting: Model is too simple and doesn't capture underlying patterns.

1.3 Linear regression

Regression is a method of fitting data points onto a curve such that the error associated is minimized. **Linear regression** is a statistical technique used to find the relationship between variables. In an ML context, linear regression finds the relationship between features and a label.

Linear regression equation :



  - Bias is the same concept as the y-intercept in the algebraic equation for a line.
  - Weight is the same concept as the slope in the algebraic equation for a line.

Models with multiple features

$$y' = b + \sum_{i=1}^{n} w_i x_i \tag{1}$$

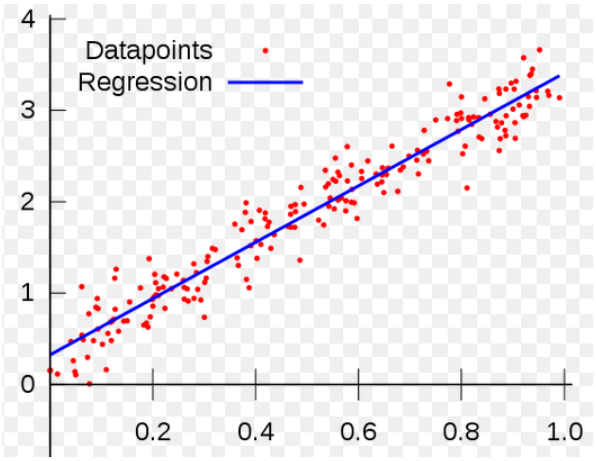


Figure 3: Linear Regression

1.3.1 Loss

Loss is a numerical metric that describes how wrong a model's predictions are.It measures the distance between the model's predictions and the actual labels

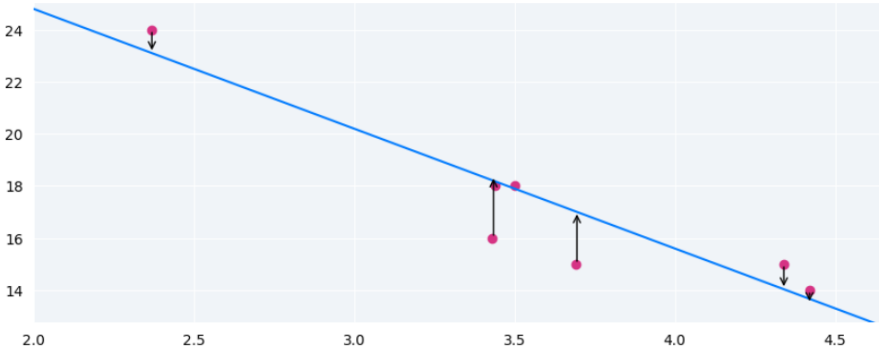


Figure 4: Linear regression Loss

Loss type	**Definition**	**Equation**
L_1 loss	The sum of the absolute values of the difference between the predicted values and the actual values.	$\sum \lvert\text{actual value} - \text{predicted value}\rvert$
Mean absolute error (MAE)	The average of L_1 losses across a set of N examples.	$\frac{1}{N}\sum \lvert\text{actual value} - \text{predicted value}\rvert$
L_2 loss	The sum of the squared difference between the predicted values and the actual values.	$\sum(\text{actual value} - \text{predicted value})^2$
Mean squared error (MSE)	The average of L_2 losses across a set of N examples.	$\frac{1}{N}\sum(\text{actual value} - \text{predicted value})^2$

Table 1: Loss Functions and Their Definitions

We need to find the best weight and bias to fit the data to minimize the error between the estimate by the model and the actual data. Gradient descent is a mathematical technique that iteratively finds the weights and bias that produce the model with the lowest loss.

1.3.2 Gradient descent

It is a numerical method which seeks to find the minima of a function by moving in the negative direction of steepest ascent at any given point. This approach, starting from any point, definitely leads to a local minimum.
The question is that if there are multiple local minima, then how do we handle it? For this reason, we prefer using Mean Squared error equations (cost functions) that have only one minima so that it becomes easier for the algorithm.
Convex functions have only one minima, so it is usually tried to use convex cost functions to train models.
We'll write the equation for making a prediction as : $f_{w,b}(x) = (w * x) + b$

- **MSE or Cost Function**

$$\frac{1}{M}\sum_{i=1}^{M}\left(f_{w,b}\left(x_{(i)}\right) - y_{(i)}\right)^2 \tag{2}$$

- **Weight derivative**

$$\frac{\partial}{\partial w}\frac{1}{M}\sum_{i=1}^{M}\left(f_{w,b}\left(x_{(i)}\right) - y_{(i)}\right)^2 \quad = \quad \frac{1}{M}\sum_{i=1}^{M}\left(f_{w,b}\left(x_{(i)}\right) - y_{(i)}\right) * 2x_{(i)} \tag{3}$$

- **Bias derivative**

$$\frac{\partial}{\partial b}\frac{1}{M}\sum_{i=1}^{M}\left(f_{w,b}\left(x_{(i)}\right) - y_{(i)}\right)^2 \quad = \quad \frac{1}{M}\sum_{i=1}^{M}\left(f_{w,b}\left(x_{(i)}\right) - y_{(i)}\right) * 2 \tag{4}$$

The model starts training by setting the weight and bias to zero : $w = 0, b = 0$

$$New\ weight = old\ weight - (small\ amount * weight\ slope)$$

$$New\ bias = old\ bias - (small\ amount * bias\ slope)$$

$$w_n = w_{n-1} - \alpha\frac{\partial MSE(w_{n-1}, b_{n-1})}{\partial w_{n-1}} \quad , \quad b_n = b_{n-1} - \alpha\frac{\partial MSE(w_{n-1}, b_{n-1})}{\partial b_{n-1}} \tag{5}$$

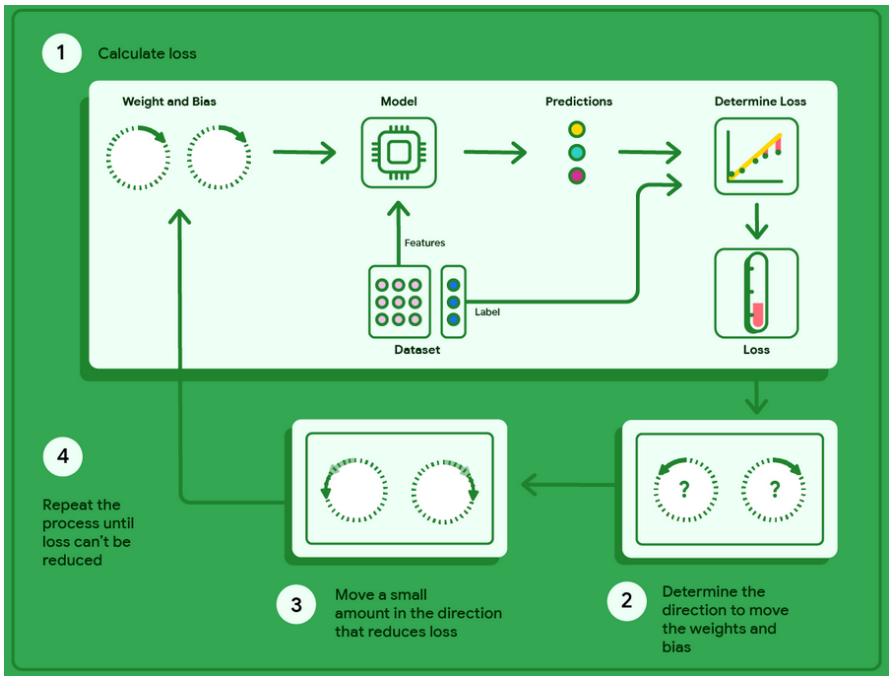


Figure 5: Gradient descent Process

- α is the learning rate and n denotes the number of iterations until then. The iterations continue till convergence. The choice of α is very important because a small value results in guarenteed convergence. α is a hyperparameter here. Upon convergence, we have found the optimal values for w and b and can thus apply the model to make predictions.

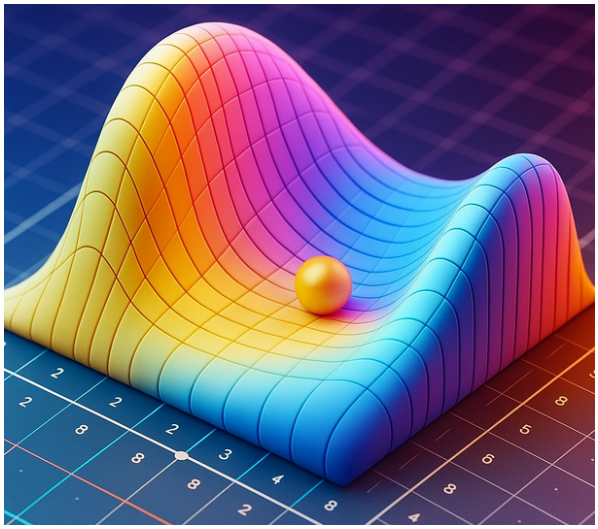


Figure 6: Convergence

1.4 Logistic Regression

Logistic regression, unlike its name suggests, is actually a classification algorithm. Given a set of data, it learns to classify them into groups based on their characteristics.
Many problems require a probability estimate as output. **Logistic regression** is an extremely efficient mechanism for calculating probabilities. Practically , we can use the returned probability in either of the following two ways:

- Applied "as is." For example, if a spam-prediction model takes an email as input and outputs a value of 0.932, this implies a probability of 93.2% that the email is spam.
- Converted to a binary category such as True or False, Spam or Not Spam

1.4.1 Sigmoid function

There is a family of functions called logistic function whose output is always a value between 0 and 1. The standard logistic function, also known as the sigmoid function (sigmoid means "s-shaped")

$$f(x) = \frac{1}{1 + e^{-x}} \tag{6}$$

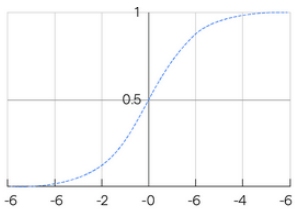


Figure 7: Sigmoid Function

However, no matter how large or how small the input value, the output will always be greater than 0 and less than 1.

Transforming linear output using the sigmoid function
The following equation represents the linear component of a logistic regression model:

$$z = b + w_1x_1 + w_2x_2 + \cdots + w_Nx_N \tag{7}$$

where:

- z is the output of the linear equation, also called the **log odds**.
- b is the bias.
- The w values are the model's learned weights.
- The x values are the feature values for a particular example.

To obtain the logistic regression prediction, the z value is then passed to the sigmoid function, yielding a value (a probability) between 0 and 1:

$$y = \frac{1}{1 + e^{-z}} \tag{8}$$

where:

- y is the output of the logistic regression model.
- z is the linear output (as calculated in the preceding equation).

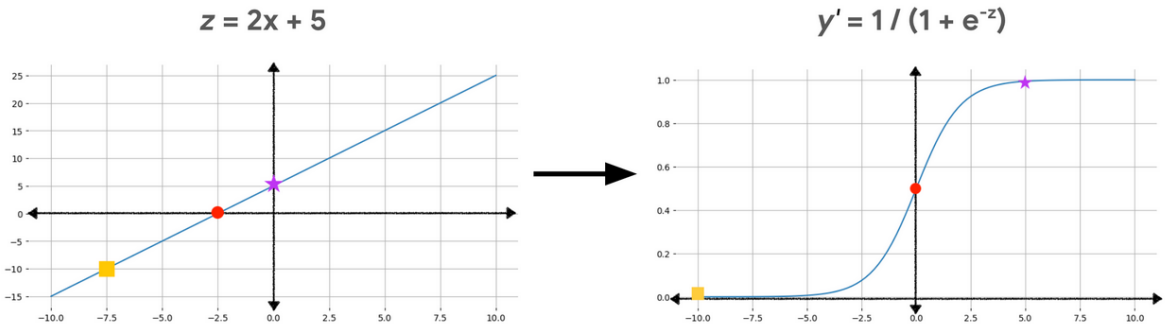


Figure 8: How linear output is transformed to logistic regression output

1.4.2 Log Loss and regularization

Squared loss works well for a linear model where the rate of change of the output values is constant. For example, given the linear model $y' = b + 3x_1$, each time you increment the input value by 1, the output value increases by 3.

However, the rate of change of a logistic regression model is not constant. As we saw , the sigmoid curve is s-shaped rather than linear. When the log-odds (z) value is closer to 0, small increases in z result in much larger changes to y than when z is a large positive or negative number.
If We use squared loss to calculate errors for the sigmoid function, as the output got closer and closer to 0 and 1, we need more memory to preserve the precision needed to track these values.
Instead, the loss function for logistic regression is Log Loss. The Log Loss equation returns the logarithm of the magnitude of the change, rather than just the distance from data to prediction.

$$\textbf{Log Loss} = \sum_{(x,y)\in D} -y\log(y') - (1-y)\log(1-y') \tag{9}$$

Where

- $(x, y) \in D$ is the dataset containing many labeled examples, which are (x, y) pairs.
- y is the label in a labeled example. Since this is logistic regression, every value of y must either be 0 or 1.
- y' is your model's prediction (somewhere between 0 and 1), given the set of features in x.

Regularization, a mechanism for penalizing model complexity during training, it is extremely important in logistic regression modeling. Without regularization, the asymptotic nature of logistic regression would keep driving loss towards 0 in cases where the model has a large number of features. Consequently, most logistic regression models use one of the following two strategies to decrease model complexity:

- L2 regularization
- Early stopping: Limiting the number of training steps to halt training while loss is still decreasing.

1.5 Classification

In the logistic regression, we learned how to use the sigmoid function to convert raw model output to a value between 0 and 1 to make probabilistic predictions, for example, predicting that a given email has a 75% chance of being spam. But what if Our goal is not to output probability but a category—for example, predicting whether a given email is "spam" or "not spam"?
To make this conversion, we choose a threshold probability, called a classification threshold. Examples with a probability above the threshold value are then assigned to the positive class (here, spam). Examples with a lower probability are assigned to the negative class, the alternative class (here, not spam).
what happens if the predicted score is equal to the classification threshold (for instance, a score of 0.5 where the classification threshold is also 0.5)? Handling for this case depends on the particular implementation chosen for the classification model.
Suppose the model scores one email as 0.99, predicting that email has a 99% chance of being spam, and another email as 0.51, predicting it has a 51% chance of being spam. If we set the classification threshold to 0.5, the model will classify both emails as spam. If we set the threshold to 0.95, only the email scoring 0.99 will be classified as spam.

2 Python for Data Analysis

2.1 What is data analysis

It's a process of checking, cleaning, transforming, and organizing data to find useful information, understand things better, and help in decision-making.

Figure 9: Data Analysis Tools

Applications :

- Unfamiliar detection (fraud, disease, etc.)
- Automation and decision-making (credit worthiness, etc.)
- Classifications (classifying emails as "important" or "junk")
- Forecasting (sales, revenue, etc.)
- Pattern detection (weather patterns, financial market patterns, etc.)
- Recognition (facial, voice, text, etc.)
- Recomendation (Based on learned prefrences, suggest movies, books etc.)

2.2 Python

It is a beginner-friendly and easy to use programming language with a vast resource for libraries for all kinds of applications. It is the language that is used by most machine learning developers and data scientists.
Why Python for Data Analysis?

- very simple and intuitive to learn
- "correct" language
- Powerful libraries (not just for Data Analysis)
- Free and open source
- Amazing community, docs and conferences

We use **Jupyter Notebooks**, an open source interactive computing platform that is used throughout machine learning and data interpretation. It has cells divided into code blocks and text blocks written in markdown. Users can run specific parts of the code which allows for flexibility and easy debugging, while also making it more readable and simpler to understand.

The libraries we use...

- Pandas : The cornerstone of our Data Analysis job with Python
- Matplotlib : The foundational library for visualizations. Other libraries we'll use will be built on top of matplotlib.

- Numpy : The numeric library that serves as the foundation of all calculations in Python.
- Scikit-learn : The most popular machine learning library for Python (not deep learning)
- Seaborn : A statistical visualization tool built on top of matplotlib.
- Statsmodels : A library with many advanced statistical functions.
- Scipy : Advanced scientific computing, including functions for optimization, linear algebra, image processing and much more.

2.2.1 Intro to NumPy

NumPy major contributions are:

- Efficient numeric computation with C primitives
- Efficient collections with vectorized operations
- An integrated and natural Linear Algebra API
- A C API for connecting NumPy with libraries written in C, C++, or FORTRAN.

In Python, everything is an object, which means that even simple ints are also objects, with all the required machinery to make object work. Some common objects in NumPy are :

1. **numpy.ndarray** : n-dimensional array which stores data for access and manipulation. This is the fundamental data structure in NumPy
2. **numpy.shape** : Determines the dimensions (shape) of an array
3. **numpy.dtype** : Check the data type of array elements
4. **numpy.reshape** : Reshapes arrays to change their dimensions. Total number of elements remain same, but the way they are packaged changes
5. **numpy.arange** : create arrays with regularly spaced values.
6. **numpy.mean** : Returns the average value of the series of data given
7. **numpy.max and numpy.min** : Return the highest and lowest values in an array, respectively
8. **numpy.sum** : Returns the sum of the series of data given
9. **numpy.dot** : Perform the dot product operation of vectors or even matrix multiplication
10. **numpy.linspace** : Generates evenly spaced values within a given range
11. **numpy.random.rand** : Creates an array of random values between 0 and unity.

2.2.2 Intro to Pandas

Pandas is an open-source Python library providing high-performance, easy-to-use data structures and data analysis tools. It is particularly well-suited for working with tabular data.

Series: a one-dimensional labeled array holding data of any type such as integers, strings, Python objects etc.
DataFrame: a two-dimensional data structure that holds data as a two dimensional array or a table with rows and columns.

1. **head()**: used to view the first few data entries. Default value is 5, but can be modified
2. **shape**: returns the dimensions of the array
3. **fillna()**: Replace missing values in a DataFrame with specified values.

4. **describe()**: Generate summary statistics (count, mean, std, min, max) for numeric and object columns.

5. **groupby()**: Groups data based on one or more columns. Enables aggregation and analysis within groups.

6. **apply()**: Applies a function to each element or row/column of a DataFrame.

7. **merge()**: Merge/join DataFrames based on common columns. Essential for combining data from different sources.

8. **drop()**: Removes specified rows or columns from a DataFrame. Useful for data cleaning

Listing 1: Panadas Example code

```
import pandas as pd
 # Create a DataFrame
 data = {
 'Name': ['Gunjan', 'Pritam', 'Samarth', 'Rohan'],
 'Age': [19, 18, 19, 18],
 'City': ['Darbhanga', 'Mysore', 'Surat', 'Gopalganj']
 }
 df = pd.DataFrame(data)
 print("Original DataFrame:\n", df)
 # Select a column
 print("\nAge column:\n", df['Age'])
 # Filter data
 print("\nPeople older than 18:\n", df[df['Age'] > 18])
 # Group by city and calculate average age
 print("\nAverage age by city:\n", df.groupby('City')['Age'].mean())
```

2.2.3 Intro to Matplotlib

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. It finds extensive use in presenting data in a succinct and attractive manner that enables people to understand the situation better.

Listing 2: Matplolib Example code

```
import matplotlib.pyplot as plt
import numpy as np

# Create some data for sine wave
x = np.linspace(0, 10, 100)
y = np.sin(x)
# Create a line plot
plt.figure(figsize=(8, 6))
plt.plot(x, y, label='sin(x)', color='blue', linestyle='--')
plt.title('Sine Wave')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.grid(True)
plt.legend()
plt.savefig('sine_wave.png')
plt.close()
# Create some data for scatter plot
np.random.seed(42)
n_points = 50
x_scatter = np.random.rand(n_points) * 10
y_scatter = np.random.rand(n_points) * 10
colors = np.random.rand(n_points)
areas = (20 * np.random.rand(n_points))**2
```

```
# Create a scatter plot
plt.figure(figsize=(8, 6))
plt.scatter(x_scatter, y_scatter, s=areas, c=colors, alpha=0.7, cmap='viridis')
plt.title('Scatter Plot')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.colorbar(label='Color Intensity')
plt.grid(True)
plt.savefig('scatter_plot.png')
plt.close()
```

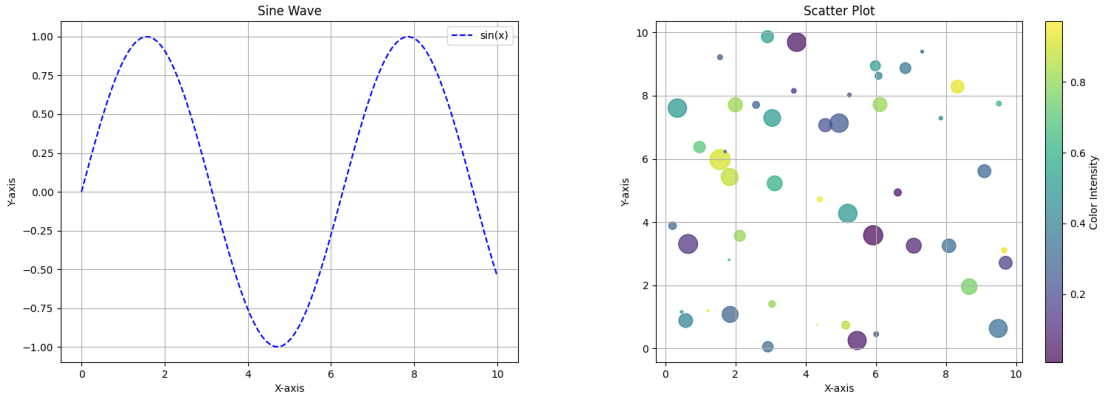
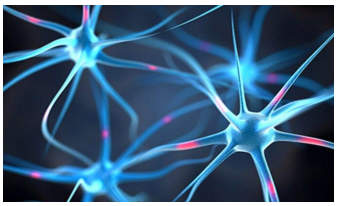


Figure 10: Example Code Output

3 Neural Network

A neural network is a computational model inspired by the structure and functioning of the human brain. Designed to find nonlinear patterns in data. During training of a neural network, the model automatically learns the optimal feature crosses to perform on the input data to minimize loss.



Neural Networks are networks of interconnected neurons, in human brains.

Artificial Neural Networks are highly connected to other neurons, and performs computations by combining signals from other neurons.

It consists of basically:

- Neurons: which pass input values through functions and output the result.
- Weights: which carry values (real-number) between neurons.

Neurons consists of three layers :

1. Input layers
2. Hidden layers
3. Output Layers

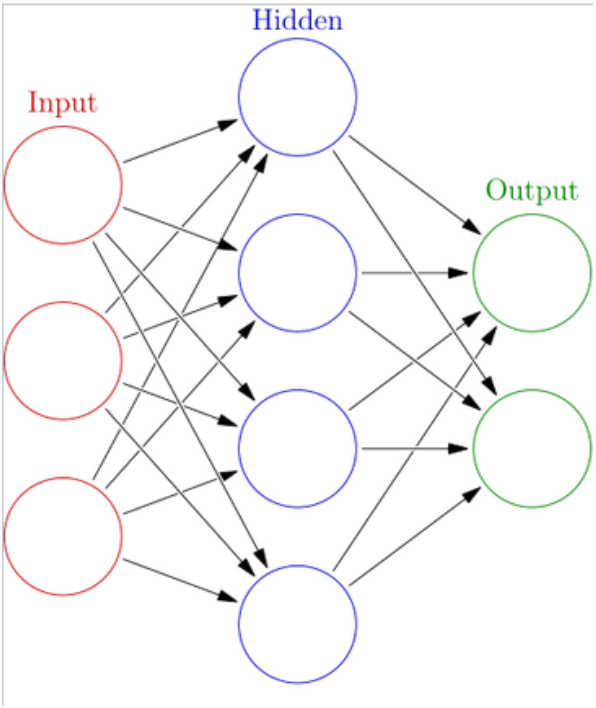


Figure 11: Neurons Layers

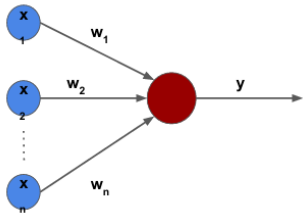
Artificial Neural Networks are composed of **artificial neurons** which are conceptually derived from biological neurons. Each artificial neuron has inputs and produces a single output which can be sent to multiple other neurons.
To find the output of the neuron we take the weighted sum of all the inputs, weighted by the weights of the connections from the inputs to the neuron. We add a bias term to this sum. This weighted sum is sometimes called the activation. This weighted sum is then passed through a (usually nonlinear) activation function to produce the output. The initial inputs are external data, such as images and documents. The ultimate outputs accomplish the task, such as recognizing an object in an image.

3.1 The Perceptron Model

A perceptron is the simplest form of an artificial neuron and is a fundamental building block of neural networks. It,s a computing element where input are associated with the weights and the cell having a threshold value.

$$y' = \begin{cases} 1, & \text{if } \sum w_i x_i > \text{threshold} \\ 0, & \text{otherwise} \end{cases} \tag{10}$$

- Rewrite $\sum w_i x_i$ as w.x
- Replace threshold = -b
- b : Bias, a prior inclination towards some decision.



$$y' = \begin{cases} 1, & \text{if } w.x + b > 0 \\ 0, & \text{otherwise} \end{cases} \tag{11}$$

$$y' = f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{12}$$

3.1.1 Perceptron Training

Step 1: Absorb bias b as weight.
Step 2: Start with a random weight value W_j.
Step 3: Predict for each input x_i: If the prediction is correct for every x, then return w
Step 4: In case of a mistake for given input x , update as follows:

- mistake in positive (y = 1), update $W_(j+1) = W_j + X$
- mistake in negative (y = 0), update $W_(j+1) = W_j - X$

3.1.2 Example:A simple decision via Perceptron

Q : Should I Go to the Movies?
Feature (Input) :

1. Extra lecture this weekend ? (x_1)
2. Friends want to go ? (x_2)
3. Pending Assignment due this weekend ? (x_3)

x_1	x_2	x_3	**Should you go?** (y)
1	0	1	0 (No)
0	1	0	1 (Yes)
1	1	1	0 (No)
0	1	1	0 (No)
0	1	0	1 (Yes)

Table 2: Training Data

Initial Setup

- **Weights:** $\mathbf{w} = [0, 0, 0]$
- **Bias:** $b = 0$

Iteration : Go through each example and update if there's a mistake
Iteration 1: $\mathbf{x} = [1, 0, 1],\ y = 0$
Prediction:

$$z = \mathbf{w} \cdot \mathbf{x} + b = (0)(1) + (0)(0) + (0)(1) + 0 = 0 \Rightarrow y' = 1$$

Mistake! $(y = 0,\ y' = 1)$
Update:

$$\mathbf{w} = \mathbf{w} - \mathbf{x} = [0, 0, 0] - [1, 0, 1] = [-1, 0, -1]$$

$$b = b - 1 = -1$$

Iteration 2: $\mathbf{x} = [0, 1, 0],\ y = 1$
Prediction:

$$z = (-1)(0) + (0)(1) + (-1)(0) - 1 = -1 \Rightarrow y' = 0$$

Mistake! $(y = 1,\ y' = 0)$
Update:

$$\mathbf{w} = \mathbf{w} + \mathbf{x} = [-1, 0, -1] + [0, 1, 0] = [-1, 1, -1]$$

$$b = b + 1 = 0$$

Iteration 3: $\mathbf{x} = [1, 1, 1],\ y = 0$
Prediction:

$$z = (-1)(1) + (1)(1) + (-1)(1) + 0 = -1 \Rightarrow y' = 0$$

Correct, no update
Iteration 4: $\mathbf{x} = [0, 1, 1],\ y = 0$

Prediction:

$$z = (-1)(0) + (1)(1) + (-1)(1) + 0 = 0 \Rightarrow y' = 1$$

Mistake! $(y = 0,\ y' = 1)$
Update:

$$\mathbf{w} = \mathbf{w} - \mathbf{x} = [-1, 1, -1] - [0, 1, 1] = [-1, 0, -2]$$

$$b = b - 1 = -1$$

Iteration 5: $\mathbf{x} = [0, 1, 0],\ y = 1$
Prediction:

$$z = (-1)(0) + (0)(1) + (-2)(0) - 1 = -1 \Rightarrow y' = 0$$

Mistake! $(y = 1,\ y' = 0)$
Update:

$$\mathbf{w} = \mathbf{w} + \mathbf{x} = [-1, 0, -2] + [0, 1, 0] = [-1, 1, -2]$$

$$b = b + 1 = 0$$

Final Parameters

- **Weights:** $[-1,\ 1,\ -2]$
- **Bias:** 0

Learned Rule

$$z = -x_1 + x_2 - 2x_3 + 0, \quad y' = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$

Listing 3: Prceptron Training Python code

```python
import numpy as np

# Training data
# x1: Extra lecture , x2: Friend wants to go , x3: Pending assignments
X = np.array([
    [1, 0, 1],
    [0, 1, 0],
    [1, 1, 1],
    [0, 1, 1],
    [0, 1, 0]
])
# Outputs: 1 = Yes, 0 = No
y = np.array([0, 1, 0, 0, 1])
# Initialize weights and bias
w = np.zeros(X.shape[1])  # [0, 0, 0]
b = 0
learning_rate = 1
# Function (step)
def step_function(z):
    return 1 if z > 0 else 0
# Training the perceptron
for epoch in range(10):  # Run for 10 epochs or until convergence
    errors = 0
    for i in range(len(X)):
        x_i = X[i]
        y_pred = step_function(np.dot(w, x_i) + b)
        error = y[i] - y_pred
        if error != 0:
            w += learning_rate * error * x_i
            b += learning_rate * error
            errors += 1
    if errors == 0:
```

```
        break

# Final weights and bias
print("Trained weights:", w)
print("Trained bias:", b)
# Prediction function
def predict(x_input):
    z = np.dot(w, x_input) + b
    return step_function(z)
# Test new inputs
test_inputs = np.array([
    [0, 1, 0],
    [1, 0, 1],
    [0, 0, 0]
])
print("\nPredictions:")
for test in test_inputs:
    result = predict(test)
    print(f"Input: {test} = {'Yes' if result == 1 else 'No'}")
```

Listing 4: Output

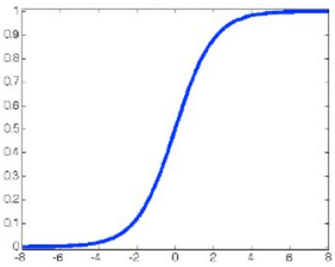
```
Trained weights: [-1.  1. -2.]
Trained bias: 1

Predictions:
Input: [0 1 0] = Yes
Input: [1 0 1] = No
Input: [0 0 0] = Yes
```

3.2 Activation Function

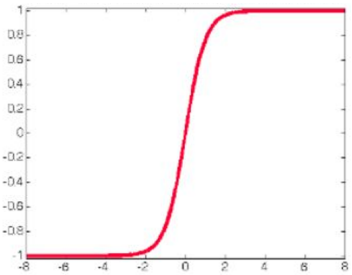
Activation function decide whether a neuron should be activated or not. It helps the network to use the useful information and suppress the irrelevant information.Activation functions introduce non-linearity into the neural network, allowing it to learn complex patterns and relationships in the data. Without activation functions, a neural network would simply be a linear regression model, regardless of the number of layers.

3.2.1 Common Activation Functions

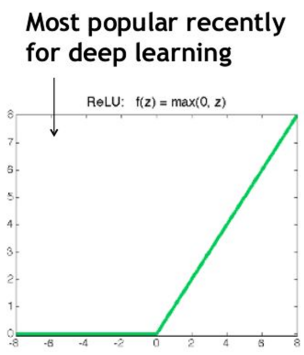
1. **Sigmoid function** : $\sigma = \frac{1}{1+e^{-z}}$
   - Range 0 to 1
   - Continuously differentiable
   - not symmetric around origin



2. **Tanh (Hyperbolic Tangent)** : $F(x) = tanh(x) = \frac{2}{1+e^{-2x}} - 1.$
   - Squashes values between - 1 and 1
   - Scaled version of sigmoid
   - Symmetric around origin
   - vanishing gradient



3. ReLU (Rectified linear Unit) : $F(x) = max(0, x)$



- It is a non-linear activation function (piecewise linear function), allowing neural networks to learn complex patterns and making backpropagation more efficient.
- Trivial to implement
- Sparse representation
- Avoid the problem of vanishing gradients
- ReLU is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. At a time only a few neurons are activated making the network sparse making it efficient and easy for computation.

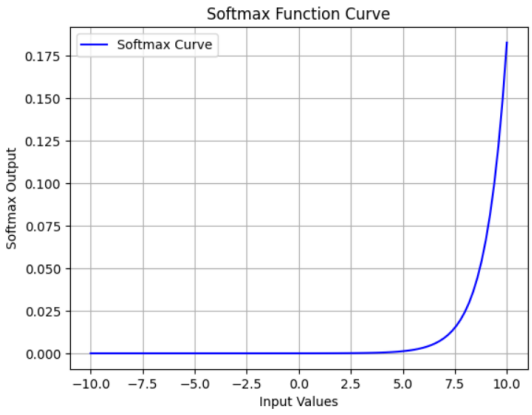
3.2.2 Why is Non-Linearity Important in Neural Networks?

Neural networks consist of neurons that operate using weights, biases and activation functions. In the learning process these weights and biases are updated based on the error produced at the output—a process known as backpropagation. Activation functions enable backpropagation by providing gradients that are essential for updating the weights and biases.
Without non-linearity even deep networks would be limited to solving only simple, linearly separable problems. Activation functions help neural networks to model highly complex data distributions and solve advanced deep learning tasks. Adding non-linear activation functions introduce flexibility and enable the network to learn more complex and abstract patterns from data.

3.2.3 Exponential Activation Functions

1. Softmax function : is designed to handle multi-class classification problems. It transforms raw output scores from a neural network into probabilities. It works by squashing the output values of each class into the range of 0 to 1 while ensuring that the sum of all probabilities equals 1.

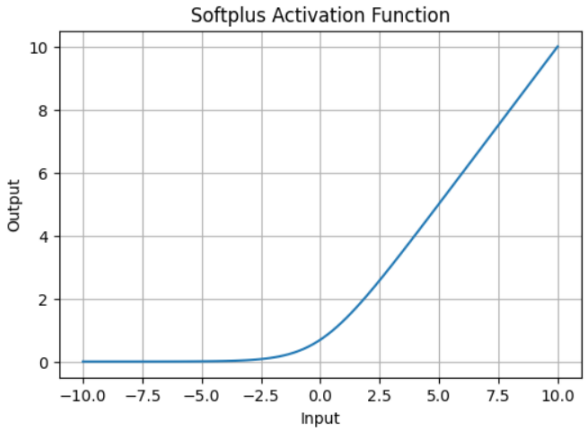
- Softmax is a non-linear activation function
- It ensures that each class is assigned a probability, helping to identify which class the input belongs to.



2. SoftPlus Function : $F(x) = log(1 + e^x)$.
This equation ensures that the output is always positive and differentiable at all points which is an advantage over the traditional ReLU function.

- The Softplus function is non-linear and outputs values in the range $(0,\infty)(0,\infty)$, similar to ReLU

- Softplus is a smooth, continuous function, meaning it avoids the sharp discontinuities of ReLU which can sometimes lead to problems during optimization.



3.2.4 Impact of Activation Functions on Model Performance

The choice of activation function has a direct impact on the performance of a neural network in several ways:

- **Convergence Speed**: Functions like ReLU allow faster training by avoiding the vanishing gradient problem while Sigmoid and Tanh can slow down convergence in deep networks.
- **Gradient Flow**: Activation functions like ReLU ensure better gradient flow, helping deeper layers learn effectively. In contrast Sigmoid can lead to small gradients, hindering learning in deep layers.
- **Model Complexity**: Activation functions like Softmax allow the model to handle complex multi-class problems, whereas simpler functions like ReLU or Leaky ReLU are used for basic layers.

Activation functions are the backbone of neural networks enabling them to capture non-linear relationships in data. From classic functions like Sigmoid and Tanh to modern variants like ReLU and Swish, each has its place in different types of neural networks. The key is to understand their behavior and choose the right one based on the needs of our model.

3.3 Backpropagation

Back Propagation is also known as "Backward Propagation of Errors" is a fundamental algorithm for training artificial neural networks. It is an iterative process that adjusts the weights of the network to minimize the difference between the predicted output and the actual target output. This is achieved by calculating the gradient of the loss function with respect to each weight and then updating the weights in the direction that reduces the loss.
Back Propagation plays a critical role in how neural networks improve over time. Here's why:

1. **Efficient Weight Update**: It computes the gradient of the loss function with respect to each weight using the chain rule making it possible to update weights efficiently.
2. **Scalability**: The Back Propagation algorithm scales well to networks with multiple layers and complex architectures making deep learning feasible.
3. **Automated Learning**: With Back Propagation the learning process becomes automated and the model can adjust itself to optimize its performance.

3.3.1 Steps of Backpropagation

- **Forward Pass**: Input data is fed through the network, and the output is calculated.
- **Calculate Loss**: The difference between the predicted output and the actual target output is calculated using a loss function (e.g., Mean Squared Error, Cross Entropy).
- **Backward Pass** (Gradient Calculation): The error is propagated backward through the network, and the gradients of the loss with respect to each weight are calculated using the chain rule of calculus.

- Chain Rule

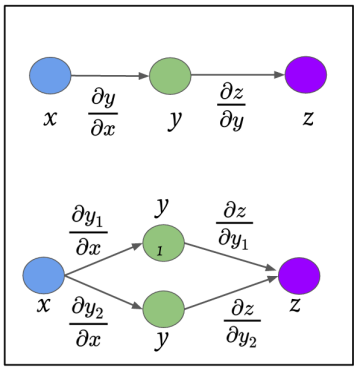
Single Path

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x}$$

Multiple Path

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1} \cdot \frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2} \cdot \frac{\partial y_2}{\partial x}$$

$$\frac{\partial z}{\partial x} = \sum_{t=1}^{T} \frac{\partial z}{\partial y_t} \cdot \frac{\partial y_t}{\partial x}$$



- **Weight Update**: The weights are updated using an optimization algorithm (e.g., Gradient Descent) in the direction opposite to the gradient, effectively reducing the loss

4 Deep Learning

4.1 Introduction

Deep Learning is transforming the way machines understand, learn and interact with complex data. Deep learning mimics neural networks of the human brain.

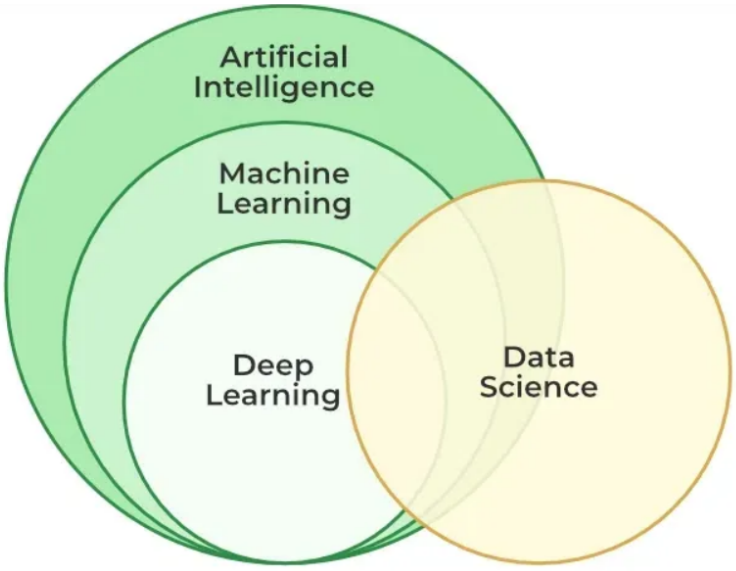


Figure 12: Deep Learning as a part of ML

Machine Learning	Deep Learning
Apply statistical algorithms to learn the hidden patterns and relationships in the dataset.	Uses artificial neural network architecture to learn the hidden patterns and relationships in the dataset.
Can work on the smaller amount of dataset.	Requires the larger volume of dataset compared to machine learning.
Better for the low-label task.	Better for complex tasks like image processing, natural language processing, etc.
Takes less time to train the model.	Takes more time to train the model.
A model is created by relevant features which are manually extracted from images to detect an object in the image.	Relevant features are automatically extracted from images. It is an end-to-end learning process.
Less complex and easy to interpret the result.	More complex, it works like the black box interpretations of the result are not easy.
It can work on the CPU or requires less computing power as compared to deep learning.	It requires a high-performance computer with GPU.

Table 3: Comparison of Machine Learning and Deep Learning

4.1.1 Deep Learning Applications

1. **Computer vision**
   In computer vision, deep learning models enable machines to identify and understand visual data. Some of the main applications of deep learning in computer vision include:

   - Deep learning models are used to identify and locate objects within images and videos, making it possible for machines to perform tasks such as self-driving cars, surveillance and robotics.
   - classify images into categories such as animals, plants and buildings. This is used in applications such as medical imaging, quality control and image retrieval.
   - Image segmentation into different regions, making it possible to identify specific features within images.

2. **Natural language processing (NLP)**
   In NLP, deep learning model enable machines to understand and generate human language. Some of the main applications of deep learning in NLP include:

   - Automatic Text Generation: learn the corpus of text and new text like summaries, essays can be automatically generated using these trained models.
   - Language translation: Translate text from one language to another, making it possible to communicate with people from different linguistic backgrounds.
   - Sentiment analysis: Analyze the sentiment of a piece of text, making it possible to determine whether the text is positive, negative or neutral.
   - Speech recognition: Recognize and transcribe spoken words, making it possible to perform tasks such as speech-to-text conversion, voice search and voice-controlled devices.

3. **Reinforcement learning**
   In reinforcement learning, deep learning works as training agents to take action in an environment to maximize a reward. Some of the main applications of deep learning in reinforcement learning include:

   - Game playing: Deep reinforcement learning models have been able to beat human experts at games such as Go, Chess and Atari.

- Robotics: Used to train robots to perform complex tasks such as grasping objects, navigation and manipulation.
- Control systems: Used to control complex systems such as power grids, traffic management and supply chain optimization.

4.2 Deep learning Models

4.2.1 Convolutional Neural Network (CNN)

Convolutional Neural Networks (CNNs) are deep learning models designed to process data with a grid-like topology such as images. They are the foundation for most modern computer vision applications to detect features within visual data.
Components :

1. **Convolutional Layers**: These layers apply convolutional operations to input images using filters or kernels to detect features such as edges, textures and more complex patterns. Convolutional operations help preserve the spatial relationships between pixels.

2. **Pooling Layers**: They downsample the spatial dimensions of the input, reducing the computational complexity and the number of parameters in the network. Max pooling is a common pooling operation where we select a maximum value from a group of neighboring pixels.

3. **Activation Functions**: They introduce non-linearity to the model by allowing it to learn more complex relationships in the data.

4. **Fully Connected Layers**: These layers are responsible for making predictions based on the high-level features learned by the previous layers. They connect every neuron in one layer to every neuron in the next layer.

Training of a Convolutional Neural Network : CNNs are trained using a supervised learning approach. This means that the CNN is given a set of labeled training images. The CNN learns to map the input images to their correct labels.

- Data Preparation: The training images are preprocessed to ensure that they are all in the same format and size.
- Loss Function: A loss function is used to measure how well the CNN is performing on the training data.
- Optimizer: An optimizer is used to update the weights of the CNN in order to minimize the loss function.
- Backpropagation: Calculate the gradients of the loss function with respect to the weights of the CNN. The gradients are then used to update the weights of the CNN using the optimizer.

Evaluation CNN Models : Efficiency of CNN can be evaluated using a variety of criteria.

- Accuracy: Accuracy is the percentage of test images that the CNN correctly classifies.
- Precision: Precision is the percentage of test images that the CNN predicts as a particular class and that are actually of that class.
- Recall: Recall is the percentage of test images that are of a particular class and that the CNN predicts as that class.
- F1 Score: The F1 Score is a harmonic mean of precision and recall. It is a good metric for evaluating the performance of a CNN on classes that are imbalanced.

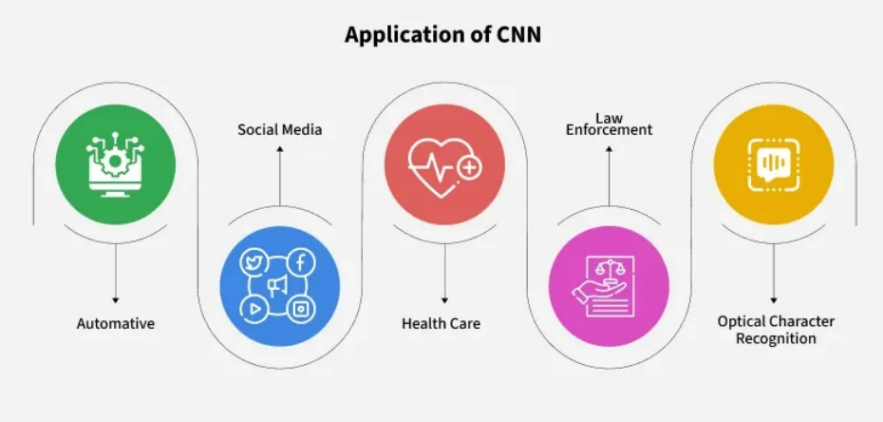
e wsDA W 

Figure 13: CNN

4.2.2 Recurrent Neural Networks

Imagine reading a sentence and you try to predict the next word, you don't rely only on the current word but also remember the words that came before. RNNs work similarly by "remembering" past information and passing the output from one step as input to the next i.e it considers all the earlier words to choose the most likely next word. This memory of previous steps helps the network understand context and make better predictions.

1. Recurrent Neurons : The fundamental processing unit in RNN is a Recurrent Unit. They hold a hidden state that maintains information about previous inputs in a sequence. Recurrent units can "remember" information from prior steps by feeding back their hidden state, allowing them to capture dependencies across time.

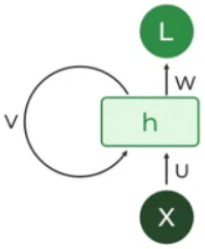


Figure 14: Recurrent Neuron

2. RNN Unfolding : RNN unfolding or unrolling is the process of expanding the recurrent structure over time steps. This unrolling enables backpropagation through time (BPTT) a learning process where errors are propagated across time steps to adjust the network's weights enhancing the RNN's ability to learn dependencies within sequential data

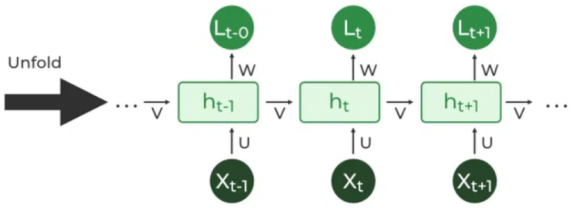


Figure 15: RNN Unfolding

Architecture : RNNs use shared weights across time steps, allowing them to remember information over sequence. The hidden state H_i is calculated for every input X_i to retain sequential dependencies.

1. **Hidden State Calculation:**

$$h = \sigma(U \cdot X + W \cdot h_{t-1} + B) \tag{13}$$

   - h represents the current hidden state.
   - U and W are weight matrices.
   - B is the bias.

2. **Output Calculation:**

$$Y = O(V \cdot h + C) \tag{14}$$

   The output Y is calculated by applying O, an activation function, to the weighted hidden state where V and C represent weights and bias.

3. **Overall Function:**

$$Y = f(X, h, W, U, V, B, C) \tag{15}$$

   This function defines the entire RNN operation where the state matrix S holds each element s_i representing the network's state at each time step i.

How RNN Works
At each time step, Recurrent Neural Networks (RNNs) process inputs using a fixed activation function. These units maintain an internal **hidden state** that serves as memory, retaining information from previous time steps. This memory enables the network to store past knowledge and adapt based on new inputs.
Updating the Hidden State in RNNs : The current hidden state h_t depends on the previous hidden state h_{t-1} and the current input x_t.

1. **State Update:**

$$h_t = f(h_{t-1}, x_t) \tag{16}$$

   where:

   - h_t is the current hidden state,
   - h_{t-1} is the previous hidden state,
   - x_t is the input at the current time step.

2. **Activation Function Application:**

$$h_t = \tanh(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t) \tag{17}$$

   Here:

   - W_{hh} is the weight matrix for the recurrent connection,
   - W_{xh} is the weight matrix for the input connection.

3. **Output Calculation:**

$$y_t = W_{hy} \cdot h_t \tag{18}$$

   where:

   - y_t is the output at time step t,
   - W_{hy} is the weight matrix at the output layer.

These parameters are updated using **backpropagation**. However, since RNNs operate on sequential data, an adapted form known as **Backpropagation Through Time (BPTT)** is used for training the network over multiple time steps.
The loss function $L(\theta)$ depends on the final hidden state h_3, and each hidden state relies on the preceding ones, forming a sequential dependency chain:

$$h_3 \text{ depends on } h_2, \quad h_2 \text{ depends on } h_1, \quad h_1 \text{ depends on } h_0$$

Initial Gradient Expression : The partial derivative of the loss function with respect to the weight matrix W is:

$$\frac{\partial L(\theta)}{\partial W} = \frac{\partial L(\theta)}{\partial h_3} \cdot \frac{\partial h_3}{\partial W} \tag{19}$$

Final Gradient Expression : Since h_3 indirectly depends on all previous hidden states (h_2, h_1, h_0), we apply the chain rule across time steps:

$$\frac{\partial L(\theta)}{\partial W} = \frac{\partial L(\theta)}{\partial h_3} \cdot \sum_{k=1}^{3} \frac{\partial h_3}{\partial h_k} \cdot \frac{\partial h_k}{\partial W} \tag{20}$$

.

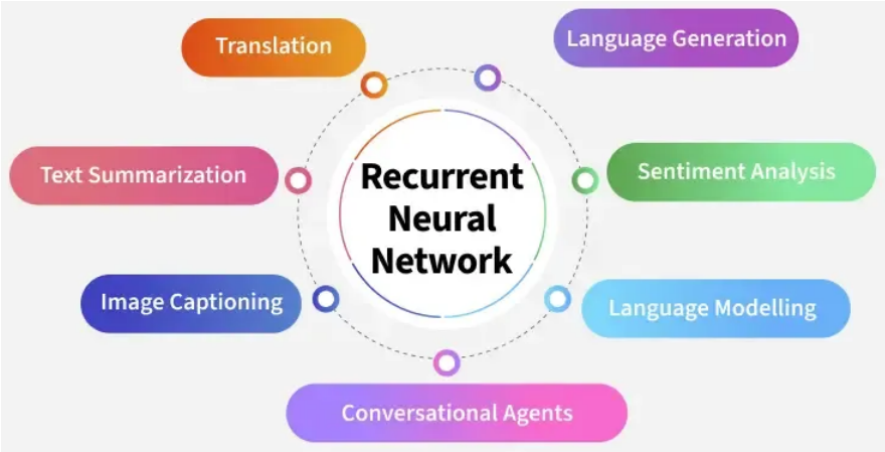


Figure 16: Applications of RNN

4.2.3 Transformer

Transformer Architecture uses self-attention to transform one whole sentence into a single sentence. This is useful where older models work step by step and it helps overcome the challenges seen in models like RNNs and LSTMs. Traditional models like RNNs (Recurrent Neural Networks) suffer from the vanishing gradient problem which leads to long-term memory loss.

For example: In the sentence: **XYZ went to France in 2019 when there were no cases of COVID and there he met the president of that country** the word "that country" refers to "France".
However RNN would struggle to link **that country** to **France** since it processes each word in sequence leading to losing context over long sentences. This limitation prevents RNNs from understanding the full meaning of the sentence.

Core Concepts of Transformers:

1. **Self-Attention Mechanism** : The self-attention mechanism allows transformers to determine which words in a sentence are most relevant to each other using a **scaled dot-product attention** approach. Each word in a sequence is mapped to three vectors:

- Query (Q)
- Key (K)
- Value (V)

The attention score determine how much attention each word should pay to others, is calculated as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{21}$$

where d_k is the dimension of the key vectors.

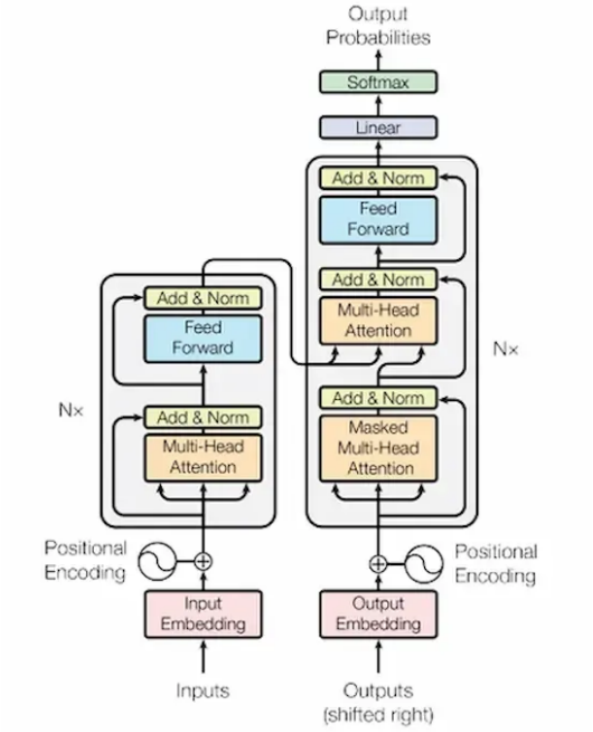


Figure 17: The Transformer Model

2. **Positional Encoding**

Unlike RNNs, transformers lack an inherent sense of word order as they process input in parallel. To address this, **positional encodings** are added to token embeddings to provide information about the position of each token in a sequence.
Example of Positional Encoding : Suppose we have a Transformer model which translates English sentences into French. [**The cat sat on the mat**.]
Tokenization ["**The**", "**cat**" , "**sat**", "**on**", "**the**" ,"**mat**"]

After that each token is mapped to a high-dimensional vector representation through an embedding layer.
Embeddings =$[E1, E2, E3, E4, E5, E6]$, where each E_i is a 4-dimensional vector.

Positional Encoding Formula For each position p in the sequence and for each dimension $2i$ and $2i + 1$ in the encoding vector:

$$\text{Even-indexed dimensions} : \text{PE}(p, 2i) = \sin\left(\frac{p}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \tag{22}$$

$$\text{Odd-indexed dimensions} : \text{PE}(p, 2i+1) = \cos\left(\frac{p}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \tag{23}$$

Here,

- p is the position index in the input sequence.
- i is the dimension index in the embedding vector.
- d_{model} is the total dimension of the model's embedding space.

This encoding allows the model to learn relative and absolute positional relationships between tokens.

Listing 5: Python code Implementation

```python
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

def positional_encoding(position, d_model):
    angle_rads = np.arange(position)[:, np.newaxis] / np.power(10000,
        (2 * (np.arange(d_model)[np.newaxis, :] // 2)) / np.float32(
            d_model))

    angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])  #for even indices
    angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])  #for odd indices

    pos_encoding = angle_rads[np.newaxis, ...]  # (1, position, d_model)
    return tf.cast(pos_encoding, dtype=tf.float32)

sentence = ["My", "name", "is", "Gunjan"]
position = len(sentence)      # 4 tokens
d_model = 8                   # simple small embedding dimension

# Get Positional Encoding
pos_encoding = positional_encoding(position, d_model)

# Simulate dummy word embeddings (random or zeros)
word_embeddings = tf.random.uniform((1, position, d_model))

# Add Positional Encoding to Embeddings
final_embeddings = word_embeddings + pos_encoding

print("Positional Encoding Shape:", pos_encoding.shape)   # (1, 4, 8)
print("Final Embedding Shape:", final_embeddings.shape)   # (1, 4, 8)
```

3. **Multi-Head Attention**

   Instead of one attention mechanism, transformers use multiple attention heads running in parallel. Each head captures different relationships or patterns in the data, enriching the model's understanding.

4. **Position-wise Feed-Forward Networks**

   These consist of two linear transformations with a ReLU activation in between, applied independently to each position in the sequence.

   Mathematically:

   $$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

   where W_1, W_2 are weight matrices and b_1, b_2 are bias terms.
   This transformation helps refine the encoded representation at each position.

5. **Encoder-Decoder Architecture**

   The encoder-decoder structure is key to transformer models. The encoder processes the input sequence into a vector, while the decoder converts this vector back into a sequence. Each encoder and decoder layer includes self-attention and feed-forward layers.

Intuition with Example : For instance in the sentence "**The cat didn't chase the mouse, because it was not hungry**" the word 'it' refers to 'cat'. The self-attention mechanism helps the model correctly associate 'it' with 'cat' ensuring an accurate understanding of sentence structure.

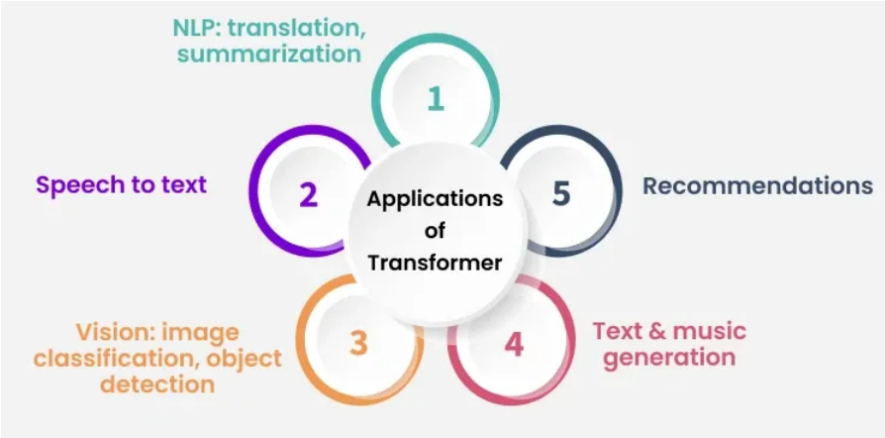


Figure 18: Transformer Application

5 Generative AI

5.1 Introduction

Generative artificial intelligence (Generative AI or gen AI), is a type of AI that can create new content like conversations, stories, images, videos, and music. It can learn about different topics such as languages, programming, art, science, and more, and use this knowledge to solve new problems.
For example: It can learn about popular design styles and create a unique logo for a brand or an organisation.

5.1.1 Evolution

- **Generative Adversarial Networks (GANs)**, introduced in 2014, use two AI systems that work together: one generates new content, and the other checks if it looks real. This made generative AI much better at creating realistic images, videos, and sounds. Example: GANs can create life like images of people who don't exist. (filters used in apps like Snapchat)
- **Large Language Models (LLMs)** : Models like GPT-3 and GPT-4 can understand and generate human-like text. They are trained on massive amounts of data from books, websites, and other sources. AI can now hold conversations, write essays, generate code, and much more. Example: ChatGPT can write a poem.
- **Multimodal Generative AI** : New AI models can handle multiple types of data at once—text, images, audio, and video. This allows AI to create content that combines different formats. Example: AI creating videos and songs from written scripts the help of different models integrating together.

5.1.2 Types of Generative AI Models

Generative AI is versatile, with different models designed for specific tasks.

- **Text-to-Text**: Widely used for tasks like drafting emails, summarizing lengthy documents, translating languages, or even writing creative content.For example tools like ChatGPT.
- **Text-to-Image**: Tools like DALL-E, Midjourney can create a custom digital image based on prompts.

- **Image-to-Image**: These models enhance or transform images based on input image . For example Adobe Firefly.
- **Image-to-Text**: AI tools like captionBot analyze and describe the content of images in text form.
- **Speech-to-Text**: This application converts spoken words into written text. It powers virtual assistants like Siri.
- **Text-to-Audio**: Generative AI can create music, sound effects, or audio narrations from textual prompts. For example ElevenLabs.
- **Text-to-Video**: These models allow users to generate video content by describing their ideas in text. For example Sora , PikaLabs and Runway.
- **Multimodal AI**: Tools like GPT-4o, Gemini 1.5, and Claude 3.5 integrate multiple input and output formats, like text, images, and audio, into a unified interface. For instance, an educational platform could let students ask questions via text and receive answers as interactive visuals or audio explanations, enhancing learning experiences.

In today's world, Generative AI has become a trusted best friend for humans, working alongside us to achieve incredible things. Imagine a painter creating a masterpiece, while they focus on the vision, Generative AI acts as their assistant, mixing colors, suggesting designs, or even sketching ideas. The painter remains in control, but the AI makes the process faster and more exciting.

Generative AI is not here to replace humans but to empower them. It takes on repetitive tasks, offers endless possibilities, and helps people achieve results they might not have imagined alone. At the same time, humans bring their intuition, creativity, and ethical judgment, ensuring the AI's contributions are meaningful and responsible.

5.2 Generative Adversarial Network (GAN)

It is introduced by Ian Goodfellow and his team in 2014 and they have transformed how computers generate images, videos, music and more. Unlike traditional models that only recognize or classify data, they take a creative way by generating entirely new content that closely resembles real-world data.

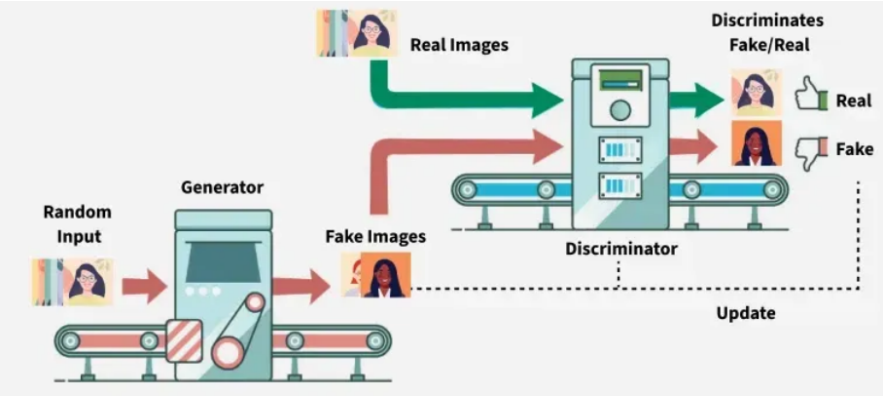


Figure 19: Generative Adversarial Network

5.2.1 Architecture of GANs

1. **Generator Model** : The generator is a deep neural network that takes random noise as input to generate realistic data samples like images or text. It learns the underlying data patterns by adjusting its internal parameters during training through backpropagation.
   Generator Loss Function: The generator tries to minimize this loss

$$J_G = \frac{-1}{m} \sum_{i=1}^{m} logD(G(z_i)) \tag{24}$$

Where,

- J_G measure how well the generator is fooling the discriminator.
- $G(z_i)$ is the generated sample from random noise z_i
- $D(G(z_i))$ is the discriminator's estimated probability that the generated sample is real.

The generator aims to maximize $D(G(z_i))$ meaning it wants the discriminator to classify its fake data as real (probability close to 1).

2. **Discriminator Model** : The discriminator acts as a binary classifier helps in distinguishing between real and generated data. It learns to improve its classification ability through training and refines its parameters to detect fake samples more accurately.
Discriminator Loss Function: The discriminator tries to minimize this loss

$$J_D = \frac{-1}{m}\sum_{i=1}^{m} logD(x_i) - \frac{1}{m}\sum_{i=1}^{m} log(1 - D(G(z_i)) \tag{25}$$

Where,

- J_D measures how well the discriminator classifies real and fake samples.
- x_i is a real data sample.
- $G(z_i)$ is a fake sample from the generator.
- $D(x_i)$ criminator's probability that $x_i$is real.
- $D(G(z_i))$ is the discriminator's probability that the fake sample is real.

The discriminator wants to correctly classify real data as real (maximize $logD(x_i)$ and fake data as fake (maximize $log(1 - D(G(zi)))$.

GANs are trained using a MinMax Loss between the generator and discriminator

$$min_G max_D(G, D) = [E_{x\sim pdata}[logD(x)] + E_{z\sim p_z(z)}[log(1D(g(z)))] \tag{26}$$

Where,

- G is generator network and is D is the discriminator network
- $p_{data}(x)$ = true data distribution
- $p_z(z)$ = distribution of random noise (usually normal or uniform)
- D(x) = discriminator's estimate of real data
- D(G(z)) = discriminator's estimate of generated data

The generator tries to minimize this loss (to fool the discriminator) and the discriminator tries to maximize it (to detect fakes accurately).

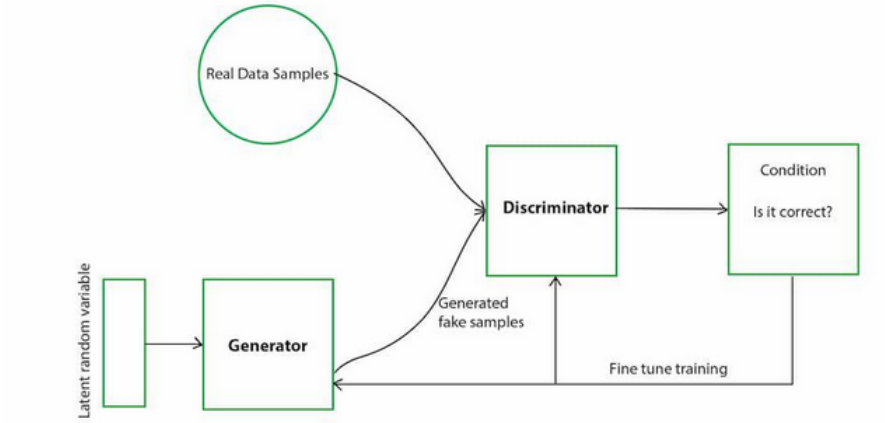


Figure 20: GANs Architecture

5.2.2 Working of GAN

GANs train by having two networks the Generator (G) and the Discriminator (D) compete and improve together. Here is the step-by-step process

1. **Generator's First Move**

   The generator starts with a random noise vector like random numbers. It uses this noise as a starting point to create a fake data sample such as a generated image. The internal layers of the generator transform this noise into something that looks like real data.

2. **Discriminator's Turn**

   The discriminator receives two types of data

   - Real samples from the actual training dataset.
   - Fake samples created by the generator.

   Discriminator analyze each input and find whether it's real data or something G cooked up. It outputs a probability score between 0 and 1.

3. **Adversarial Learning**

   If the discriminator correctly classifies real and fake data it gets better at its job.
   If the generator fools the discriminator by creating realistic fake data, it receives a positive update and the discriminator is penalized for making a wrong decision.

4. **Generator's Improvement**

   Through many iterations, the generator improves and creates more convincing fake samples.

5. **Discriminator's Adaptation**

   The discriminator also learns continuously by updating itself to better spot fake data. This constant back-and-forth makes both networks stronger over time.

6. **Training Progression**

   As training continues, the generator becomes highly proficient in producing realistic data. Eventually, the discriminator struggling to distinguish real from fake shows that the GAN has reached a well-trained state.
   At this point, the generator can produce high-quality synthetic data that can be used for different applications.

Figure 21: Application of GANs

5.3 Generative Pre-trained Transformer (GPT)

The Generative Pre-trained Transformer (GPT) is a model, developed by Open AI to understand and generate human-like text. GPT has revolutionized how machines interact with human language making more meaningful communication possible between humans and computers.

GPT is based on the **transformer** architecture and the core idea behind it is the use of **self-attention** mechanisms that processes words in relation to all other words in a sentence whereas the traditional methods that process words in sequential order. This allows the model to weigh the importance of each word no matter its position in the sentence, leading to a more nuanced understanding of language.

It is extremely useful for applications like creating written content, generating creative writing or even simulating dialogue.

5.3.1 Architecture of GPT

Important elements of this architecture consist of:

- **Self-Attention System** : This enables the model to evaluate the significance of each word within the context of the complete input sequence. It makes it possible for the model to comprehend word linkages and dependencies which is essential for producing content that is logical and suitable for its context.
- **Layer normalization and residual connections** : By reducing problems such as disappearing and exploding gradients, these characteristics aid in training stabilization and enhance network convergence.
- **Feedforward Neural Networks** : These networks process the output of the attention mechanism and add another layer of abstraction and learning capability. They are positioned between self-attention layers.

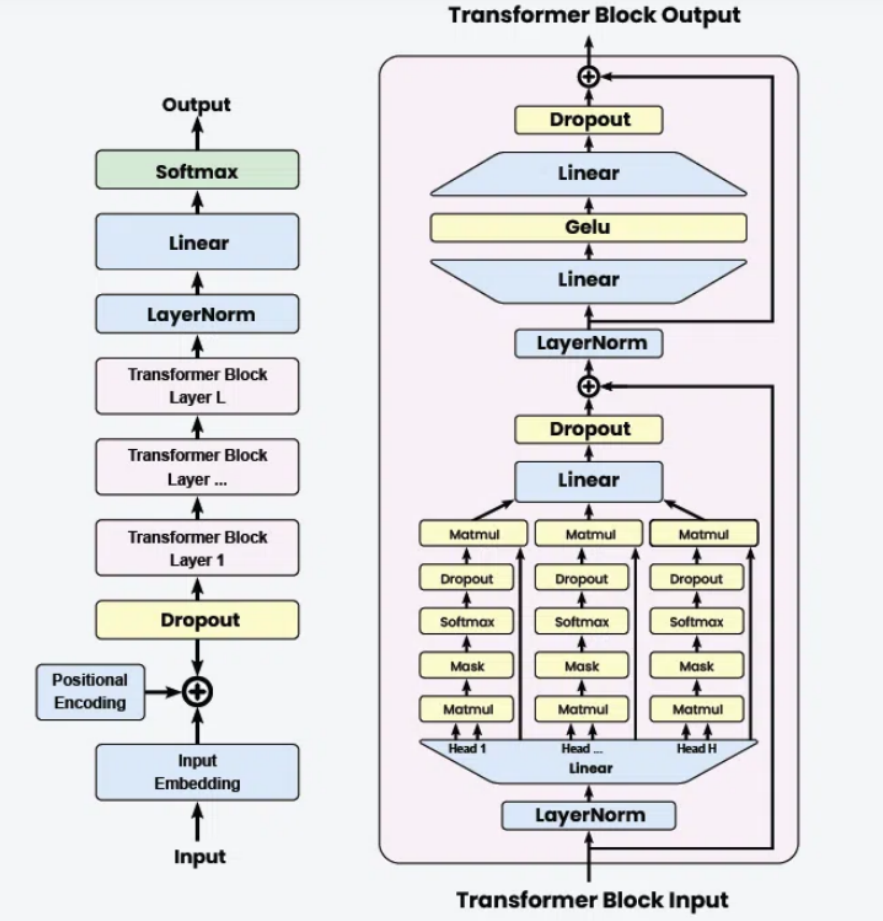


Figure 22: GPT Architecture

1. **Input Embedding**
    - **Input:** The raw text input is tokenized into individual tokens (words or subwords).
    - **Embedding:** Each token is converted into a dense vector representation using an embedding layer.
2. **Positional Encoding**
   Since transformers do not inherently understand the order of tokens, positional encodings are added to the input embeddings to retain the sequence information.
3. **Dropout Layer**
   A dropout layer is applied to the embeddings to prevent overfitting during training.
4. **Transformer Blocks**
    - **LayerNorm :** Each transformer block starts with a layer normalization.
    - **Multi-Head Self-Attention :** Core component where the input passes through multiple attention heads.
    - **Add & Norm :** Output of attention is added back to the input (residual connection) and normalized again.
    - **Feed-Forward Network :** Position-wise feed-forward layer with two linear transformations and a GeLU activation in between.
    - **Dropout :** Applied to the output of the feed-forward network.
5. **Layer Stack**
   Transformer blocks are stacked to form a deeper model, capturing more complex patterns and dependencies.
6. **Final Layers**
    - **LayerNorm:** Final layer normalization is applied.
    - **Linear:** Output is mapped to the vocabulary size.
    - **Softmax:** Produces the final probabilities for each token in the vocabulary.

5.3.2 Training Process of GPT

Large-scale text corpora are used for unsupervised learning to train GPT models. The training consists of two main stages:

1. **Pre-training:** Also known as language modeling, this stage trains the model to predict the next word in a sentence using a diverse internet dataset. This enables the model to generate human-like text across various domains.
2. **Fine-tuning:** While GPT performs well in zero-shot and few-shot scenarios, fine-tuning improves its performance on specific tasks or domains by training on custom datasets.

6 Agentic AI

6.1 Introduction

Agentic AI is specialized AI that focuses on becoming really good at one specific task, rather than knowing a little about everything. Think of it like this instead of being a general doctor, it iss like a heart surgeon who is an expert in just heart operations.

- **Specialized mastery**: It's trained and fine-tuned to handle a particular type of problem with great accuracy.
- **Tool usage**: It can connect with and use specific tools like software, APIs, databases, etc to achieve its goal.
- **Goal-oriented actions**: Instead of just giving information, it actively takes steps toward completing the task.
- **Efficient problem-solving**: Because it's specialized, it can work faster and more effectively in its field than a general AI.

6.2 Applications of Agentic AI

The potential applications of Agentic AI are vast and varied. Here are some few examples:

1. **Autonomous Vehicles:** Agentic AI can be used in self-driving cars, where the AI acts as the driver making real-time decisions based on traffic conditions, road signs and other environmental factors.
2. **Healthcare**: In healthcare, it could assist in patient diagnosis and treatment planning by autonomously analyzing medical data and recommending personalized treatment options.
3. **Finance**: It could be used in the financial sector for algorithmic trading, where the AI independently makes trading decisions based on market trends and data analysis.
4. **Robotics**: In robotics, Agentic AI could power robots that perform complex tasks autonomously such as search and rescue missions in disaster-stricken areas.
5. **Smart Home Systems**: It could enhance smart home systems by autonomously managing energy consumption, security and other home automation features.

6.3 Agentic AI Architecture

Agentic AI systems operate through various steps such as:

Figure 23: Working of Agents

Unlike traditional AI which depends on human input, agentic AI can observe, reason, learn and make decisions independently without needing specific instructions. This diagram illustrates the complete cycle of how an Agentic AI agent operates within its environment.

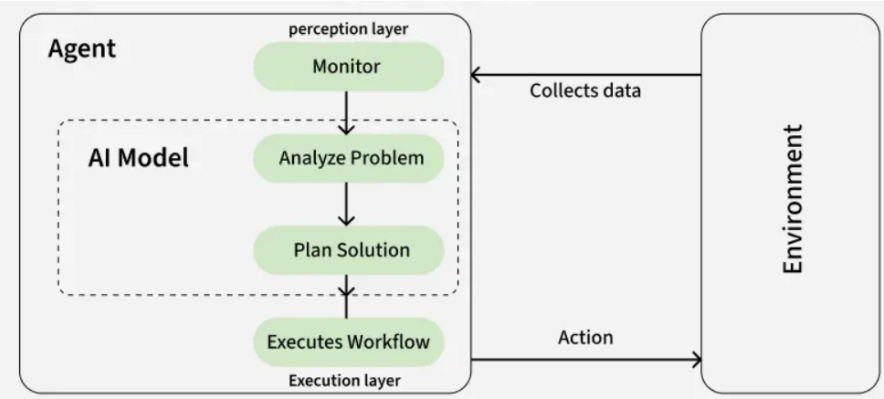


Figure 24: Agentic Search

6.3.1 Types of Agentic Architectures

There are several types of agentic architectures each with its own strengths and weaknesses, suitable for different tasks and environments.

1. **Single-agent architecture**: A single AI system that functions independently, making decisions and taking actions without the involvement of other agents.
2. **Multi-agent architecture**: This architecture involves multiple AI systems interacting with each other, collaborating and coordinating their actions to achieve common goals.

Its sub types are

- **Vertical architecture**: This approach involves agentic AI systems organized in a hierarchical structure with higher-level agents overseeing and guiding the actions of lower-level agents.
- **Horizontal architecture**: This involves agentic AI systems operating on the same level without any hierarchical structure, communicating and coordinating their actions as needed.
- **Hybrid architecture**: This involves a combination of different agentic architecture types and using the strengths of each to achieve optimal performance in complex environments.

6.3.2 Principles of Agentic AI Architecture

The principles behind the Agentic AI architecture are mentioned below:

- **Autonomy**: Agentic AI works independently within set limits and hence reducing the need for human involvement. It adapts to changing conditions while following ethical and safety guidelines.
- **Goal-Directed Behavior**: The system focuses on clear objectives, using them to guide its perception, reasoning and planning. Goals can be set by users or inferred from the context.
- **Adaptability**: Agentic systems improve over time by learning from feedback. Methods like online learning or meta-learning allow them to continuously evolve.
- Modularity: A modular design allows components to be developed, tested and updated independently. This enhances scalability and facilitates integration with existing systems.
- Transparency: To build trust, agentic AI provides understandable outputs that explain its reasoning and actions.

6.4 Single Agentic AI Architecture

Diagram presents the comprehensive architecture of an Agentic AI system, showcasing how it processes user requests through three interconnected layers.

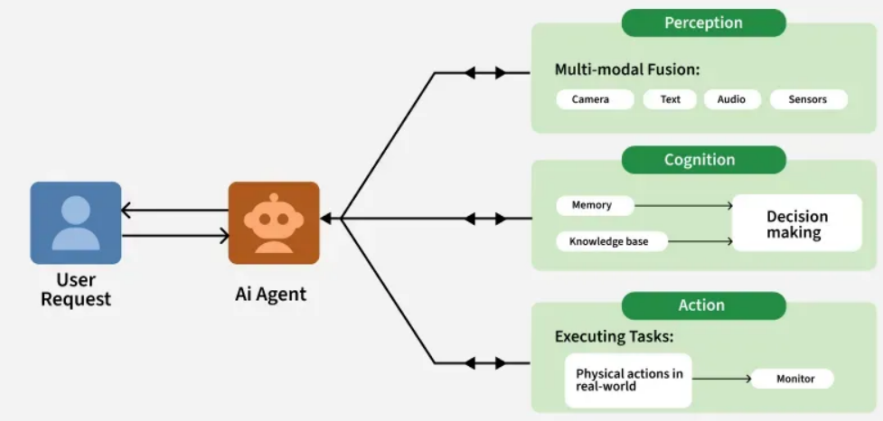


Figure 25: Overview of Core Components

1. **Perception**
   The way by which the agent collects information from its surroundings and using inputs like images, sound, text or sensor data is perception. Systems use sensors, data streams and external databases to understand their environment and recognize changes or events that need a response.

   - **Sensors**: These may include cameras, microphones, motion detectors or specialized sensors etc.
   - **Data Integration**: The AI system integrates data from multiple sources allowing for a comprehensive understanding of the situation. This can involve data from IoT devices, external APIs and historical datasets.

2. **Cognitive Layer**
   After understanding its environment agent must analyze the data and decide the best action. This process involves assessing the current situation, considering potential outcomes and selecting the best action based on predefined goals or objectives.
   Agentic AI uses techniques such as:

   - **Rule-Based Systems**: Simple systems that follow predefined rules to make decisions.
   - **Machine Learning Models**: More advanced systems that use statistical techniques to learn patterns from data and make predictions.
   - **Reinforcement Learning**: Agentic AI systems often use reinforcement learning where they learn through trial and error by receiving feedback i.e rewards or penalties based on their actions.

3. **Action and Execution**
   The action component executes the decisions made by the agent. Once the agent processes the data and chooses an action, it takes action in the environment. This could involve sending commands to physical systems like a robotic arm or self-driving car and then handling data or communicating it with other systems.

   - **Robotics**: In physical environments it can control robotic systems to perform tasks such as assembly, navigation and interaction with humans.
   - **Software Automation**: In virtual environments it can control software systems to automate processes such as decision-making in business operations, customer service chatbots or IT systems management.

4. **Learning and Adaptation**
   The systems need to adapt and get better over time by learning from past experiences. This enables them to handle new situations that may not have been specifically programmed. Learning mechanisms in agentic AI can be:

   - Supervised Learning
   - Unsupervised Learning
   - Reinforcement Learning

6.5 Multi-Agent Architectures

In multi-agent architectures, multiple specialized agents work together, each handling its own domain.

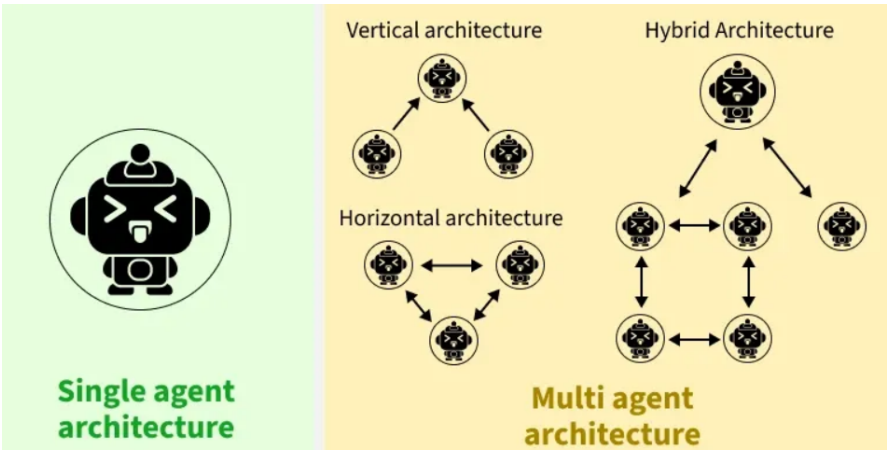


Figure 26: Single Agent vs Multiple Agent

Agents divide a large problem into smaller tasks, collaborate to solve them and adapt their roles as tasks evolve for flexibility and fast response. Each agent may use unique methods, one might employ **natural language processing (NLP)**, another computer vision, others use **retrieval augmented generation (RAG)** to pull external data.

1. **Vertical AI Architecture**

   In a vertical architecture, a leader agent oversees subtasks and decisions, with agents reporting back for centralized control.

   - **Key features**
     - Hierarchy: Roles are clearly defined.
     - Centralized communication: Agents report to the leader.
   - **Strengths**
     - Task efficiency: Ideal for sequential workflows.
     - Clear accountability: Leader aligns objective.
   - **Weaknesses**
     - Bottlenecks: Leader reliance can slow progress.
     - Single point of failure: Vulnerable to leader failure leading to whole system failure.
   - **Use cases**
     - Workflow automation: Multistep approvals.
     - Document generation: Sections overseen by a leader.

2. **Horizontal AI Architecture**

   Horizontal AI Architecture uses peer collaboration model in which agents work as equals in a decentralized system, collaborating freely to solve tasks.

- **Key features**
  - Distributed collaboration: All agents share resources and ideas.
  - Decentralized decisions: Group-driven decision-making for collaborative autonomy.
- **Strengths**
  - Dynamic problem solving: Fosters innovation.
  - Parallel processing: Agents work on tasks simultaneously.
- **Weaknesses**
  - Coordination challenges: Mismanagement can cause inefficiencies.
  - Slower decisions: Too much deliberation.
- **Best Use cases**
  - Brainstorming: Generating diverse ideas.
  - Complex problem solving: Tackling interdisciplinary challenges.

3. **Hybrid AI Architecture**

   Combines structured leadership with collaborative flexibility. Here leadership shifts based on task requirements.

   - **Key features**
     - Dynamic leadership: Leadership adapts to the phase of the task.
     - Collaborative leadership: Leaders engage their peers openly.
   - **Strengths**
     - Versatility: Combines strengths of both models.
     - Parallel processing: Agents work on tasks simultaneously.
   - **Weaknesses**
     - Complexity: Balancing leadership roles and collaboration requires robust mechanisms.
     - Resource management: More demanding.
   - **Best Use cases**
     - Versatile tasks: Strategic planning or team projects.
     - Dynamic processes: Balancing structured and creative demands.

6.6 Agentic Framework

Agentic frameworks refer models that define how agents (whether artificial or natural) can perform tasks, make decisions and interact with their environment in an autonomous, intelligent manner. These frameworks provide the structure and guidelines for how agents operate, reason and adapt in various settings.

- **Reactive Architectures**: It link immediate environmental input to direct responses. These agents operate on instinct, acting solely on current stimuli without using memory or planning. They are unable to learn from the past or anticipate future events.
- **Deliberative Architectures**: It enable agents to act by reasoning, strategizing and building internal representations of the world. Unlike reactive systems, these agents assess their surroundings, consider possible outcomes and make decisions after thoughtful analysis.
- **Cognitive Architectures**: It simulate human-like intelligence. They combine perception, reasoning, learning and decision-making abilities, allowing agents to handle complex tasks by thinking and adapting much like people do.

References

[1] Google ML Crash Course

[2] GeeksforGeeks GAN Tutorial

[3] GfG Deep Learning

[4] IIT Patna Neural Network

[5] Coursera Generative and Agentic AI

[6] Youtube FreeCodeCamp For Python