INTERNSHIP TASK ARTIFICIAL INTELLIGENCE

OBJECTIVE

The objective of this internship was to gain practical experience in the field of artificial intelligence, specifically focusing on machine learning algorithms, data analysis, pattern recognition, and computer vision. By working on real-world projects, the aim was to apply theoretical knowledge and develop technical skills. Additionally, the internship sought to enhance problem-solving abilities, foster collaboration with experienced professionals, and understand the ethical implications of AI technologies.

ABSTRACT

Artificial Intelligence (AI) is a transformative technology that enables machines to perform tasks traditionally requiring human intelligence. These tasks include perception, reasoning, learning, and decision-making. AI encompasses a variety of subfields such as machine learning, natural language processing, and robotics, each contributing to the development of intelligent systems. The applications of AI are vast, ranging from healthcare and finance to autonomous vehicles and smart cities. As AI continues to evolve, it raises important ethical and philosophical questions about the nature of intelligence and the future of human-machine interaction.

EXECUTIVE SUMMARY

This report summarizes the key experiences and achievements during my AI internship at TeReSol. Over the course of 6 weeks], I had the opportunity to work on various tasks involving machine learning, deep learning, data analysis, and computer vision. My primary responsibilities included developing predictive models, analysing large datasets, and implementing AI algorithms to solve real-world problems.

One of the significant projects I contributed to was ECG signal generation using autoencoders, where I developed a deep learning model that improved accuracy by 98%. Additionally, I understood OpenCV for digital image processing and computer vision.

Throughout the internship, I gained valuable insights into the practical applications of AI technologies and honed my technical skills in Python, TensorFlow, and other relevant tools. This experience has not only deepened my understanding of AI but also reinforced my passion for pursuing a career in this dynamic field.

In conclusion, this internship provided a comprehensive learning experience, allowing me to apply theoretical knowledge in practical settings, contribute to impactful projects, and develop both technical and professional skills essential for a successful career in artificial intelligence.

COURSE LEARNING OBJECTIVES

1. **Understand Key Concepts**: Students will be able to explain fundamental concepts of artificial intelligence, including machine learning, neural networks, and natural language processing.
2. **Apply AI Techniques**: Students will be able to apply machine learning algorithms to real-world datasets and interpret the results.
3. **Develop AI Models**: Students will be able to design, implement, and evaluate AI models using programming languages such as Python and frameworks like TensorFlow.
4. **Analyse Ethical Implications**: Students will be able to discuss the ethical considerations and societal impacts of AI technologies.
5. **Collaborate on AI Projects**: Students will be able to work effectively in teams to develop AI solutions for complex problems.

COURSE OUTCOMES

1. **Knowledge Acquisition**: Students will be able to explain fundamental concepts of artificial intelligence, including machine learning, neural networks, and natural language processing.
2. **Practical Application**: Students will be able to apply machine learning algorithms to real-world datasets and interpret the results.
3. **Model Development**: Students will be able to design, implement, and evaluate AI models using programming languages such as Python and frameworks like TensorFlow.
4. **Ethical Understanding**: Students will be able to discuss the ethical considerations and societal impacts of AI technologies.
5. **Collaboration Skills**: Students will be able to work effectively in teams to develop AI solutions for complex problems.

OBJECTIVES:

1. **Data Preparation**: Collecting, cleaning, and preprocessing data for training AI models. This involves ensuring data quality and removing any noise.
2. **Model Development:** Assisting in the design, implementation, and training of machine learning models using various techniques such as supervised, unsupervised, and semi-supervised learning.
3. **Algorithm Implementation:** Developing and adapting machine learning algorithms to meet specific business needs.

4. **Experimentation:** Conducting experiments to evaluate the effectiveness of AI models and algorithms and making necessary adjustments to improve performance.
5. **Feature Engineering:** Implementing feature engineering techniques to enhance model accuracy and performance.
6. **Collaboration:** Working closely with data scientists, engineers, and other team members to complete projects and solve complex problems.
7. **Documentation:** Documenting the processes, methodologies, and results of your work to ensure reproducibility and reporting the progress to supervisor.

SOFTWARE REQUIREMENTS

Python 3.12 IDE
Google Collab
Command Prompt

COURSE CONTENT

1. Towards Data science
2. Machine Learning with Andrew Nag
3. Aurelien-Geron-Hands-On-Machine-Learning-with-Scikit-Learn-Keras-and-Tensorflow_-Concepts-Tools-and-Techniques-to-Build-Intelligent-Systems-OReilly-Media-2019 (Book)

TASKS
WEEK 1
RESEARCH REVIEW FOR EEG BASED BRAIN COMPUTER INTERFACES

The research paper provides a comprehensive review of the integration of artificial intelligence (AI) techniques, particularly machine learning (ML) and deep learning (DL), in the development and application of electroencephalography (EEG)-based brain-computer interfaces (BCIs).

Key points from the paper include:
1. Machine Learning Overview: The paper discusses how ML aims to automatically identify brain patterns associated with specific tasks, moving away from traditional statistical methods. It categorizes ML applications in EEG-based BCIs into classification tasks and individual adaptive tasks.
2. Deep Learning Techniques: Various deep learning models, including convolutional neural networks (CNNs) and recurrent neural networks (RNNs), are explored for their effectiveness in processing EEG data. For instance, a study mentioned achieved 97%

accuracy in classifying EEG driving fatigue levels using a spatial-temporal convolutional neural network.
3. Transfer Learning: The paper highlights the importance of transfer learning in overcoming inter-subject variability in EEG data. It discusses methods that allow for the transfer of knowledge from one subject to another, thereby reducing the need for extensive training data.
4. Applications: The review covers diverse applications of EEG-based BCIs, including visual object classification, natural language processing, and brain-controlled robotics. It emphasizes the potential for BCIs to enhance human-computer interaction and improve cognitive computing insights from EEG signals.
5. Conclusions: The authors conclude that the combination of ML and DL techniques can significantly advance the understanding and application of EEG data, leading to improved BCI systems with implications for multimedia, semantic, and clinical applications. They also note the need for further research to optimize training processes and enhance the reliability of BCI systems.
Overall, the paper underscores the transformative potential of AI in enhancing the capabilities and applications of EEG-based BCIs, paving the way for innovative solutions in various fields.

MOTIVES OF BRAIN COMPUTER INTERFACES

The motives behind the development and application of brain-computer interfaces (BCIs) are diverse and multifaceted. Here are some key motives:
1. Enhancing Communication: BCIs aim to provide a communication channel for individuals with severe motor disabilities, enabling them to interact with their environment and communicate effectively without relying on traditional methods.
2. Restoring Motor Function: One of the primary goals of BCIs is to restore motor functions in individuals with paralysis or neurological disorders. By translating brain signals into commands for prosthetic devices or exoskeletons, BCIs can help regain mobility and independence.
3. Improving Human-Computer Interaction: BCIs seek to revolutionize the way humans interact with computers and technology. By allowing direct brain control of devices, BCIs can create more intuitive and efficient interfaces, enhancing user experience in various applications, including gaming and virtual reality.
4. Neurofeedback and Cognitive Enhancement: BCIs can be used for neurofeedback applications, where users receive real-time feedback on their brain activity. This can help in training cognitive functions, improving focus, and managing stress or anxiety.
5. Research and Understanding of Brain Function: BCIs provide valuable insights into brain activity and cognitive processes. Researchers use BCIs to study brain functions, understand neurological disorders, and develop new therapeutic approaches.

6. Applications in Robotics and Automation: BCIs can be integrated into robotic systems, allowing for brain-controlled robots that can assist in various tasks, from rehabilitation to industrial applications, enhancing efficiency and safety.
7. Entertainment and Gaming: The gaming industry is exploring BCIs to create immersive experiences where players can control games using their thoughts, leading to new forms of entertainment and engagement.
8. Advancements in AI and Machine Learning: The integration of AI and machine learning with BCIs aims to improve the accuracy and efficiency of interpreting brain signals, leading to more sophisticated applications and better user experiences.
Overall, the motives of BCIs encompass a wide range of goals, from improving quality of life for individuals with disabilities to advancing technology and understanding of the human brain.

CLASSES OF AI:

Artificial Intelligence (AI) can be classified into several categories based on its capabilities and functionalities. Here are the main classes of AI:

Based on Capabilities

1. **Artificial Narrow Intelligence (ANI)**:
    - Also known as Weak AI, ANI is designed to perform a specific task. It operates under a narrow set of constraints and limitations.
2. **Artificial General Intelligence (AGI)**:
    - Also known as Strong AI, AGI refers to a machine with the ability to understand, learn, and apply knowledge across a wide range of tasks, like human intelligence.
3. **Artificial Superintelligence (ASI)**:
    - ASI surpasses human intelligence in all aspects, including creativity, problem-solving, and emotional intelligence.

Based on Functionalities

1. **Reactive Machines**:
    - These AI systems can only react to specific inputs and do not have the ability to form memories or use past experiences to inform current decisions.
2. **Limited Memory**:
    - These AI systems can use past experiences to inform future decisions. Most modern AI applications, such as self-driving cars, fall into this category as they use data to make predictions and decisions.
3. **Theory of Mind**:
    - This type of AI is still in development and aims to understand human emotions, beliefs, and intentions. It would be able to interact more naturally with humans by recognizing and responding to emotional cues.

4. **Self-Aware AI**:
    - The most advanced form of AI, self-aware AI, would have its own consciousness, self-awareness, and understanding of its existence. This type of AI remains theoretical and is a subject of philosophical and ethical discussions.

TYPES OF AI

VISUAL AI (Computer Vision)
Recognize Patterns, visual literals.
LIBRARIES
OpenCv
Pytorch
Tensorflow
APPLICATIONS
Object detection & Recognition
Image Classification
Scene Understanding
Anomaly Detection

LITERAL AI (National Language Processing)
Ability to handle images, texts and pdfs
LIBRARIES
OpenAI
Lang Chain
Hugging Face
MOTION AI
Process and detect motion
LIBRARIES
OpenNN
PyBrain
Theano

ARTIFICIAL INTELLIGENCE

Artificial Intelligence (AI) is a branch of computer science focused on creating systems capable of performing tasks that typically require human intelligence. These tasks include learning, reasoning, problem-solving, perception, and language understanding.

MACHINE LEARNING

Machine learning (ML) is a subfield of artificial intelligence (AI) that focuses on developing algorithms and statistical models that enable computers to learn from and make predictions or decisions based on data. Here are some key points about machine learning:

Definition

Machine learning involves training algorithms on data so they can learn patterns and make decisions without being explicitly programmed for specific tasks.

Types of Machine Learning

1. **Supervised Learning**:
    - The algorithm is trained on labelled data, meaning the input comes with the correct output. The goal is to learn a mapping from inputs to outputs. Examples include classification and regression tasks.
2. **Unsupervised Learning**:
    - The algorithm is trained on unlabelled data and must find patterns and relationships within the data. Examples include clustering and dimensionality reduction.
3. **Semi-Supervised Learning**:
    - Combines both labelled and unlabelled data during training. This approach is useful when acquiring a fully labelled dataset is expensive or time-consuming.
4. **Reinforcement Learning**:
    - The algorithm learns by interacting with an environment and receiving rewards or penalties based on its actions. This method is often used in robotics and game playing.

Linear Regression: Predicts a continuous output based on the linear relationship between input variables.

Logistic Regression: Used for binary classification problems.

Support Vector Machines (SVM): Finds the hyperplane that best separates different classes.

Decision Trees: Splits data into branches to make predictions.

Random Forest: An ensemble method using multiple decision trees to improve accuracy.

K-Means Clustering: Groups data into clusters based on similarity.

Hierarchical Clustering: Builds a hierarchy of clusters.

Principal Component Analysis (PCA): Reduces the dimensionality of data while preserving variance.

DEEP LEARNING

Deep learning is a specialized subset of machine learning that focuses on using neural networks with many layers (hence "deep") to model and understand complex patterns in data. Here are some key points about deep learning:

Definition

Deep learning involves training artificial neural networks with multiple layers to learn representations of data. These networks can automatically discover the features needed for tasks such as classification, regression, and more.

Key Components

1. **Neural Networks**: The backbone of deep learning, consisting of interconnected nodes (neurons) organized in layers. Each layer transforms the input data into a more abstract representation.
2. **Layers**:
    - **Input Layer**: The initial layer that receives the raw data.
    - **Hidden Layers**: Intermediate layers that process the data through various transformations.
    - **Output Layer**: The final layer that produces the prediction or classification.

Types of Neural Networks

1. **Convolutional Neural Networks (CNNs)**: Primarily used for image and video recognition tasks. They are designed to automatically and adaptively learn spatial hierarchies of features.
2. **Recurrent Neural Networks (RNNs)**: Suitable for sequential data like time series or natural language. They have connections that form directed cycles, allowing them to maintain a memory of previous inputs.
3. **Transformers**: Used in natural language processing tasks. They rely on self-attention mechanisms to process input data in parallel, making them highly efficient for tasks like translation and text generation.

GENERATIVE ARTIFICIAL NEURAL NETWORKS

GANs consist of two neural networks, the **Generator** and the **Discriminator**, which are trained simultaneously through adversarial training. The Generator creates fake data samples, while the Discriminator evaluates them against real data samples[1][2].

How GANs Work

1. **Generator**: This network generates new data samples from random noise. Its goal is to produce data that is indistinguishable from real data.
2. **Discriminator**: This network evaluates the data samples and tries to distinguish between real and fake data. It provides feedback to the Generator on how to improve its samples.

The two networks are trained in a zero-sum game: the Generator tries to fool the Discriminator, while the Discriminator tries to correctly identify real vs. fake data. This adversarial process continues until the Generator produces highly realistic data.

LONG SHORT-TERM MEMORY:

Long Short-Term Memory (LSTM) networks are a type of recurrent neural network (RNN) designed to handle and process sequential data, making them particularly effective for tasks involving time series, natural language processing, and other sequence prediction problems. Here's an overview of LSTMs:

Key Components of LSTM

1. **Cell State**: The cell state acts as a conveyor belt, carrying information across the sequence. It allows the network to maintain long-term dependencies.
2. **Gates**: LSTMs use gates to control the flow of information. These gates are:
   Forget Gate: Decides what information to discard from the cell state.
   Input Gate: Determines which new information to add to the cell state.
   Output Gate: Controls what information to output based on the cell state.

How LSTMs Work

LSTMs address the vanishing gradient problem found in traditional RNNs by allowing gradients to flow unchanged. This is achieved by the cell state and gates, which regulate the flow of information and maintain long-term dependencies.

WORKLOW OF MACHINE LEARNING

The workflow of a machine learning (ML) project involves several key stages, each crucial for building, training, and deploying effective models. Here's a detailed overview:

1. Problem Definition

- **Understand Business Goals**: Clearly define the problem you aim to solve and understand the business objectives. This helps in selecting the right approach and metrics for success.

2. Data Collection

- **Gather Data**: Collect relevant data from various sources such as databases, APIs, or web scraping. The quality and quantity of data significantly impact the model's performance.

3. Data Preprocessing

- **Data Cleaning**: Handle missing values, remove duplicates, and correct errors in the data.
- **Data Transformation**: Normalize or standardize data and convert categorical variables into numerical formats.
- **Feature Engineering**: Create new features or modify existing ones to improve model performance.

4. Dataset Splitting

- **Training Set**: Used to train the model.

- **Validation Set**: Used to tune hyperparameters and validate the model during training.
- **Test Set**: Used to evaluate the final model's performance.

5. Model Selection

- **Choose Algorithms**: Select appropriate machine learning algorithms based on the problem type (e.g., regression, classification, clustering).
- **Baseline Model**: Start with a simple model to establish a performance baseline.

6. Model Training

- **Train the Model**: Use the training dataset to train the model by adjusting weights and biases to minimize error.
- **Hyperparameter Tuning**: Optimize hyperparameters to improve model performance using techniques like grid search or random search.

7. Model Evaluation

- **Performance Metrics**: Evaluate the model using metrics such as accuracy, precision, recall, F1-score, or RMSE (Root Mean Squared Error).
- **Cross-Validation**: Use cross-validation techniques to ensure the model generalizes well to unseen data.

8. Model Refinement

- **Iterate and Improve**: Based on evaluation results, refine the model by adjusting features, tuning hyperparameters, or selecting different algorithms.

9. Model Deployment

- **Deploy the Model**: Integrate the model into a production environment where it can make real-time predictions.
- **Monitor Performance**: Continuously monitor the model's performance to detect and address issues like model drift.

10. Maintenance and Updates

- **Regular Updates**: Periodically retrain the model with new data to maintain its accuracy and relevance.
- **Feedback Loop**: Incorporate feedback from users and stakeholders to improve the model continuously.

Best Practices

- **Documentation**: Keep detailed documentation of all steps, decisions, and changes made during the project.
- **Version Control**: Use version control systems to track changes in code and data.
- **Collaboration**: Work closely with cross-functional teams to ensure the model meets business requirements and integrates smoothly into existing systems.

DATA PREPARATION & PREPROCESSING

Data preprocessing is the process of cleaning, transforming, and organizing raw data to make it suitable for analysis and modelling. It typically involves several steps:

1. **Data Cleaning**:

- **Handling Missing Values**: Techniques include removing rows with missing values, imputing missing values with statistical methods (mean, median, mode), or using machine learning algorithms to predict missing values[1].
- **Removing Duplicates**: Identifying and removing duplicate records to ensure data integrity.
- **Handling Outliers**: Detecting and treating outliers that can skew the results of the analysis.

2. **Data Transformation**:
   - **Normalization**: Scaling data to a standard range, typically [0, 1], to ensure that no single feature dominates the model.
   - **Standardization**: Transforming data to have a mean of 0 and a standard deviation of 1.
   - **Encoding Categorical Data**: Converting categorical variables into numerical formats using techniques like one-hot encoding or label encoding.
3. **Data Integration**:
   - Combining data from different sources into a cohesive dataset. This may involve resolving schema conflicts and ensuring consistency across datasets.
4. **Data Reduction**:
   - **Dimensionality Reduction**: Reducing the number of features while retaining important information using techniques like Principal Component Analysis (PCA).
   - **Feature Selection**: Selecting the most relevant features for the model to improve performance and reduce complexity.

Data Preparation

Data preparation is a broader term that includes data preprocessing but also encompasses other steps to get data ready for analysis:

1. **Data Collection**:
   - Gathering data from various sources such as databases, APIs, web scraping, or manual entry.
2. **Data Exploration**:
   - Conducting exploratory data analysis (EDA) to understand the data distribution, identify patterns, and detect anomalies. This involves visualizing data using plots and charts.
3. **Data Splitting**:
   - Dividing the dataset into training, validation, and test sets to evaluate the model's performance and ensure it generalizes well to new data.

Importance of Data Preprocessing and Preparation

- **Improves Model Accuracy**: Clean and well-prepared data leads to more accurate and reliable models.
- **Reduces Noise**: Eliminates errors and inconsistencies in the dataset, making it easier for algorithms to find patterns.
- **Enhances Efficiency**: Streamlines the data analysis process, saving time and computational resources.

Epochs

- **Definition**: An epoch is one complete pass through the entire training dataset. During an epoch, the model sees each training example once.
- **Example**: If you have a dataset of 1,000 images and you set the number of epochs to 5, the model will go through all 1,000 images five times during training.

Batches

- **Definition**: A batch is a subset of the training dataset. Instead of passing the entire dataset to the model at once, the data is divided into smaller batches.
- **Example**: If your dataset has 1,000 images and you choose a batch size of 100, the dataset will be divided into 10 batches, each containing 100 images.

Iterations

- **Definition**: An iteration is a single update of the model's parameters. The number of iterations is determined by the number of batches needed to complete one epoch.
- **Example**: If you have 1,000 images and a batch size of 100, it will take 10 iterations to complete one epoch (since 1,000 / 100 = 10).

Passes

- **Definition**: The term "passes" is often used interchangeably with epochs, referring to the number of times the entire dataset is passed through the model.
- **Example**: If you train your model for 5 epochs, it means the dataset has made 5 passes through the model.

Relationship Between These Terms

- **Epochs**: Number of times the entire dataset is passed through the model.
- **Batches**: Subsets of the dataset used to train the model in smaller chunks.
- **Iterations**: Number of batches needed to complete one epoch.
- **Passes**: Another term for epochs, indicating how many times the dataset is passed through the model.

Why These Concepts Matter

- **Memory Efficiency**: Using batches allows training on large datasets that wouldn't fit into memory if processed all at once.
- **Faster Training**: Smaller batches can be processed faster, leading to quicker updates of the model's parameters.
- **Stability and Convergence**: Properly tuning the number of epochs, batch size, and iterations helps in achieving a balance between underfitting and overfitting.

UNDERFITTING, OVERFITTING, BIAS, VARIANCE

Underfitting

- **Definition**: Underfitting occurs when a model is too simple to capture the underlying patterns in the data. It performs poorly on both the training and test datasets.
- **Causes**:
    - Using a model that is too simple.
    - Insufficient training data.
    - Poor feature selection or inadequate feature engineering.

Overfitting

- **Definition**: Overfitting occurs when a model is too complex and captures noise in the training data, leading to poor generalization to new data.
- **Causes**:
    - Using a model that is too complex.
    - Too many features relative to the number of training examples.

Bias

- **Definition:** Bias refers to the error introduced by approximating a real-world problem, which may be complex, by a simplified model.
- **High Bias:** Leads to underfitting. The model is too simplistic and fails to capture the complexity of the data.
- **Low Bias:** Indicates that the model is flexible enough to capture the underlying patterns in the data**.**

Bias-Variance Trade-off

- **Trade-off:** There is a trade-off between bias and variance. Increasing model complexity reduces bias but increases variance, and vice versa.
- **Goal:** Find a balance where both bias and variance are minimized to achieve good generalization on new data**.**

Training Data

- **Purpose:** Used to train the model. The model learns the patterns and relationships in the data by adjusting its parameters.
- **Process:** The model iteratively processes the training data, minimizing the error between its predictions and the actual outcomes.
- **Size:** Usually, the largest portion of the dataset to ensure the model has enough information to learn effectively.

Validation Data

- **Purpose:** Used to tune the model's hyperparameters and evaluate its performance during training. It helps in selecting the best model configuration.
- **Process:** After each training iteration (or epoch), the model is evaluated on the validation data. This helps in monitoring the model's performance and preventing overfitting.
- **Size:** Typically, a smaller portion of the dataset, often around 10-20%.

Testing Data

- **Purpose:** Used to evaluate the final model's performance. It provides an unbiased assessment of how well the model generalizes to new, unseen data.
- **Process:** Once the model is trained and validated, it is tested on the testing data to measure its accuracy, precision, recall, F1-score, etc.
- **Size:** Like the validation set, often around 10-20% of the dataset.

Workflow

1. **Data Splitting:** The original dataset is split into training, validation, and testing sets. Common splits are 70% training, 15% validation, and 15% testing, but this can vary based on the dataset size and specific requirements.
2. **Model Training:** The model is trained using the training data.
3. **Hyperparameter Tuning:** The model's hyperparameters are tuned using the validation data. This step helps in finding the optimal settings for the model.
4. **Model Evaluation:** The final model is evaluated on the testing data to assess its performance and generalization ability.

Importance

- **Training Data:** Ensures the model learns the underlying patterns in the data.
- **Validation Data:** Helps in tuning the model and preventing overfitting by providing feedback during training.
- **Testing Data**: Provides an unbiased evaluation of the model's performance on new, unseen data.
-

CONFUSION MATRIX

A confusion matrix is typically a square matrix that compares the actual class labels with the predicted class labels.

Key Terms

1. **True Positive (TP)**: The model correctly predicts the positive class.
2. **True Negative (TN)**: The model correctly predicts the negative class.
3. **False Positive (FP)**: The model incorrectly predicts the positive class (Type I error).
4. **False Negative (FN)**: The model incorrectly predicts the negative class (Type II error).

GETTING STARTED WITH PANDAS, NUMPY, POLARS, MATPLOTLIB & SEABORN

1. **Pandas**:
    - **Purpose**: Pandas is a powerful data manipulation library for Python.
    - **Strengths**:
        - Provides data structures (like DataFrames) to handle tabular data efficiently.

        - Offers flexible data cleaning, transformation, and aggregation.
        - Integrates seamlessly with other libraries.
    - **Use Cases**: Data preprocessing, exploration, and analysis.
    - **Reference**: You can learn more about Pandas here.
2. **NumPy**:
    - **Purpose**: NumPy is the fundamental package for scientific computing in Python.
    - **Strengths**:
        - Efficiently handles large arrays and matrices.
        - Provides mathematical functions and linear algebra operations.
        - Essential for numerical computations.
    - **Use Cases**: Numerical simulations, machine learning, and scientific computing.
    - **Reference**: Explore NumPy here.
3. **Matplotlib**:
    - **Purpose**: Matplotlib is a versatile plotting library.
    - **Strengths**:
        - Customizable and low-level – you have full control over plot details.
        - Supports various plot types (line plots, scatter plots, histograms, etc.).
        - Widely used for creating publication-quality visualizations.
    - **Use Cases**: Data visualization, exploratory analysis, and presentations.
    - **Reference**: Check out Matplotlib's documentation here.
4. **Polars**:
    - **Purpose**: Polars is a high-performance DataFrame library.
    - **Strengths**:
        - Faster than Pandas for certain operations (e.g., reading CSV files).
        - Supports parallel processing and optimized memory usage.
        - Integrates well with other Python libraries.
    - **Use Cases**: Large-scale data processing, analytics, and ETL pipelines.
    - **Reference**: Learn more about Polars here.
5. **Seaborn**:
    - **Purpose**: Seaborn is a statistical data visualization library.
    - **Strengths**:
        - Built on top of Matplotlib, providing higher-level functions.
        - Simplifies creating attractive statistical plots (e.g., bar plots, heatmaps).
        - Great for exploring relationships in data.
    - **Use Cases**: Statistical analysis, pattern discovery, and visual storytelling.

WEEK 2

UNDERSTANDING THE LEARNING CURVES

A learning curve plots the model's performance on the y-axis against the number of training iterations or the size of the training dataset on the x-axis. It helps visualize how well the model is learning and generalizing to new data.

Components of a Learning Curve

1. **Training Learning Curve**: Shows the model's performance on the training dataset over time.
2. **Validation Learning Curve**: Shows the model's performance on the validation dataset over time.

Interpreting Learning Curves

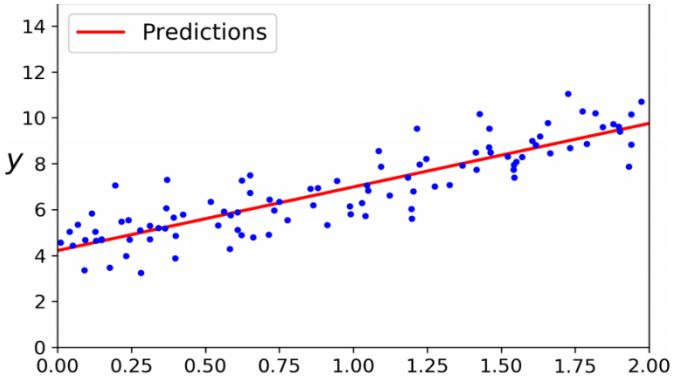
1. **Underfitting**:
    - **Indication**: Both training and validation curves converge to a high error rate.
    - **Cause**: The model is too simple to capture the underlying patterns in the data.
    - **Solution**: Increase model complexity, add more features, or reduce regularization.
2. **Overfitting**:
    - **Indication**: The training curve shows low error, but the validation curve shows high error.
    - **Cause**: The model is too complex and captures noise in the training data.
    - **Solution**: Simplify the model, use regularization techniques, or add more training data.
3. **Good Fit**:
    - **Indication**: Both training and validation curves converge to a low error rate.
    - **Cause**: The model is well-balanced and generalizes well to new data.

Diagnosing Model Behaviour

- **High Bias**: If both curves have high error rates, the model has high bias and is underfitting.
- **High Variance**: If the training curve has low error but the validation curve has high error, the model has high variance and is overfitting.

LINEAR REGRESSION

Linear regression aims to find the best-fitting straight line (linear relationship) between the dependent variable (often denoted as (y)) and the independent variable(s) (often denoted as (x)). The simplest form is **simple linear regression**, which involves one independent variable. When there are multiple independent variables, it is called **multiple linear regression**.



$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

COST

$$\mathrm{MSE}(\mathbf{X}, h_{\boldsymbol{\theta}}) = \frac{1}{m} \sum_{i=1}^{m} \left(\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)}\right)^2$$

GRADIENT DESCENT

Gradient Descent is a very generic optimization algorithm capable of finding optimal solutions to a wide range of problems. The general idea of Gradient Descent is to tweak parameters iteratively to minimize a cost function.

$$\frac{\partial}{\partial \theta_j} \mathrm{MSE}(\boldsymbol{\theta}) = \frac{2}{m} \sum_{i=1}^{m} \left(\boldsymbol{\theta}^T \mathbf{x}^{(i)} - y^{(i)}\right) x_j^{(i)}$$

ALGORITHM

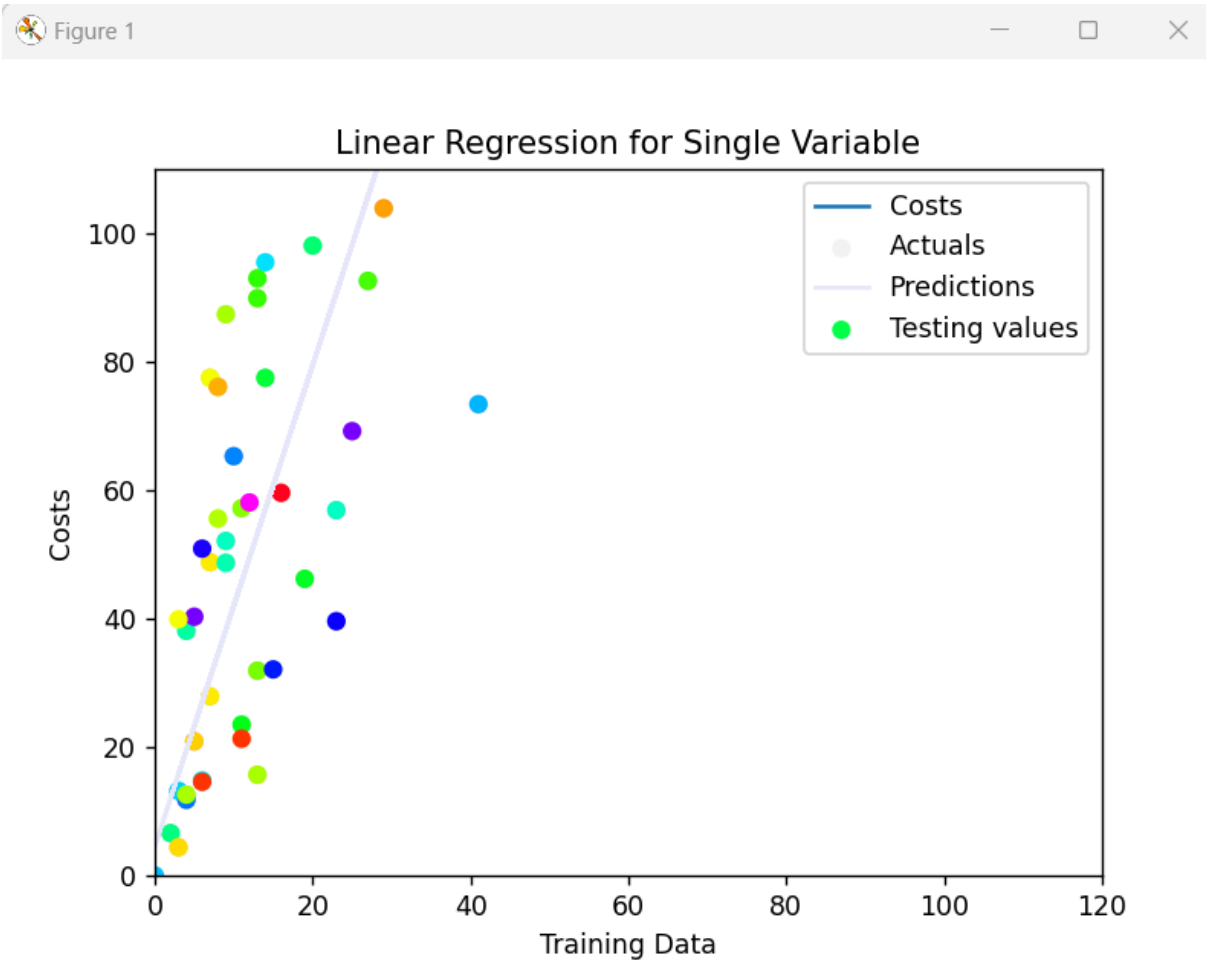
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
data  = pd.read_csv(r"C:\Users\soban\Downloads\swedish_insurance.csv")
X = data["X"].values
Y = data["Y"].values
X = np.c_[np.ones(len(X)), X]
X = X.T
```

```
def hypothesis (thetha, X):
  predictions = np.matmul(thetha.T,X)
  return predictions

def compute_cost (predictions, actual):
  m = len(actual)
  errors = predictions - actual
  squared = errors*errors
  mse = np.sum(squared)/(2*m)
  return mse

def gradient_descent(X, Y, learning_rate, iterations):
  actual = Y
  costs = []
  m = len(actual)
  thetha = np.zeros(X.shape[0])
  for i in range(iterations):
    predictions = hypothesis(thetha, X)
    errors = predictions - actual
    costs.append(compute_cost(predictions, actual))
    gradients = (1/m)*np.matmul(X,errors)
    thetha = thetha - (learning_rate*gradients)
  return thetha, costs

learning_rate = 0.0005
iterations = 1000
thetha,costs = gradient_descent(X, Y, learning_rate, iterations)

plt.plot(np.arange(1000), costs, label = 'Costs')

predictions = np.matmul(thetha.T, X)
colors = np.random.randint(10, 100, size = (1, len(Y)))
plt.scatter(X[1,:], Y, c = colors, cmap = "Pastel1", label = 'Actuals')

plt.plot(X[1,:], predictions, color = 'lavender', label = 'Predictions')
plt.axis([0, 120, 0, 110])

plt.xlabel("Training Data")
plt.ylabel("Costs")
plt.title("Linear Regression for Single Variable")
```

```
data1  = pd.read_csv("C:\\Users\\soban\\Downloads\\test.csv")
X1 = data["X"].values
Y_test = data["Y"].values
X_test = np.c_[np.ones(len(X1)), X1]
X_test = X_test.T

colors = np.random.randint(10, 100, size = (1, len(Y)))
plt.scatter(X_test[1,:], Y_test, c = colors, cmap = "gist_rainbow", label="Testing values")
prediction_t = hypothesis(thetha, X_test)
plt.legend()
plt.show()

i = 0

for pred in X1:
   print("The predicted values for ", X1[i], " are : ", prediction_t[i])
   i = i + 1
   print()

N = Y_test.size
MAE = (1/N)*np.sum(np.abs(prediction_t - Y_test))
print(MAE)

MSE = (1/N)*np.sum(np.square(prediction_t - Y_test))
print(MSE)

RMSE = np.sqrt(MSE)
print(RMSE)

R2 = 1 - ((np.sum(np.square(prediction_t - Y_test)))/(np.sum(np.square(prediction_t -
np.mean(Y_test)))))
print(R2)
```

DATASET
Swedish Insurance

OUTPUT



ERROR & R2 SCORE

```
37.03610649341637
0.8194868940157835
```

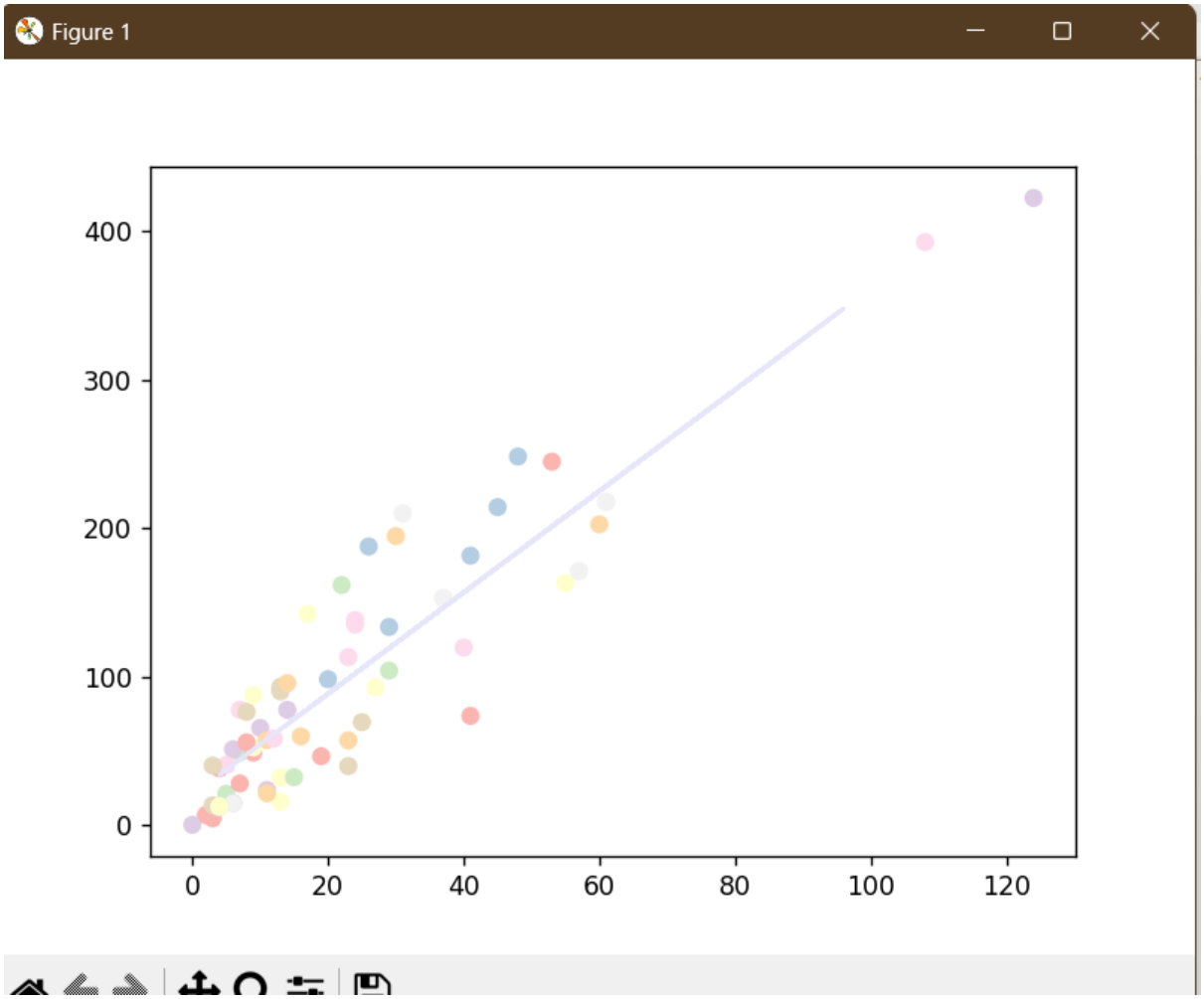
USING SK-LEARN

CODE

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model
from sklearn.metrics import mean_squared_error, r2_score
lr = linear_model.LinearRegression()

data = pd.read_csv(r"C:\Users\soban\Downloads\swedish_insurance.csv")
X = data["X"].values
X = X.reshape(X.size, 1)
Y = data["Y"].values
Y = Y.reshape(Y.size, 1)
```

```
lr.fit(X, Y)

data1 = pd.read_csv("C:\\Users\\soban\\Downloads\\test.csv")
X_test = data1["X"].values
X_test = X_test.reshape(X_test.size, 1)
Y_test = data1["Y"].values
Y_test = Y_test.reshape(Y_test.size, 1)

prediction = lr.predict(X_test)
print("Mean Squared Error: %.2f" % mean_squared_error(Y_test, prediction))
print("Coefficients of determination: %.2f" % r2_score(Y_test, prediction))

colors = np.random.randint(10, 100, size = (1, len(Y_test)))
plt.scatter(X_test, Y_test, c = colors, cmap = "Pastel1", label = "Y_Test")
plt.plot(X_test, prediction, color = "lavender", label = "Predictions")
plt.show()

i = 0

for pred in X_test:
  print("The predicted values for ", X_test[i], " are : ", prediction[i])
  i = i + 1
  print()

print(lr.score(X,Y))
print(lr.score(X_test, Y_test))
```

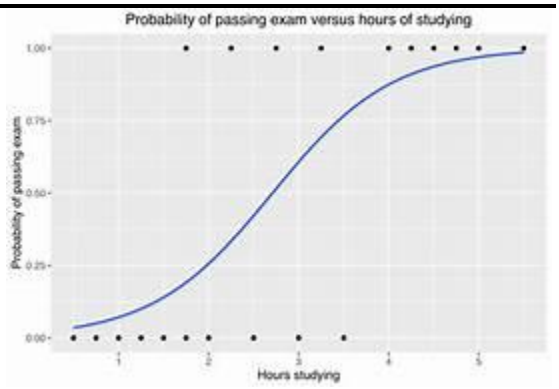
OUTPUT



R2 SCORE

0.8333466719794502

LOGISTIC REGRESSION



Logistic Regression is a supervised machine learning algorithm used primarily for binary classification tasks. It predicts the probability that a given instance belongs to a particular class. Here's a detailed overview:

Key Concepts

1. **Binary Classification**:
   - Logistic regression is used when the dependent variable is binary (e.g., 0 or 1, Yes or No).
   - It models the probability that an instance belongs to a specific class.
2. **Logistic Function (Sigmoid Function)**:

- The logistic function maps any real-valued number into a value between 0 and 1.
- The formula for the logistic function is:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

It is based on newton Raphson method, for reducing the loss.

CODE:

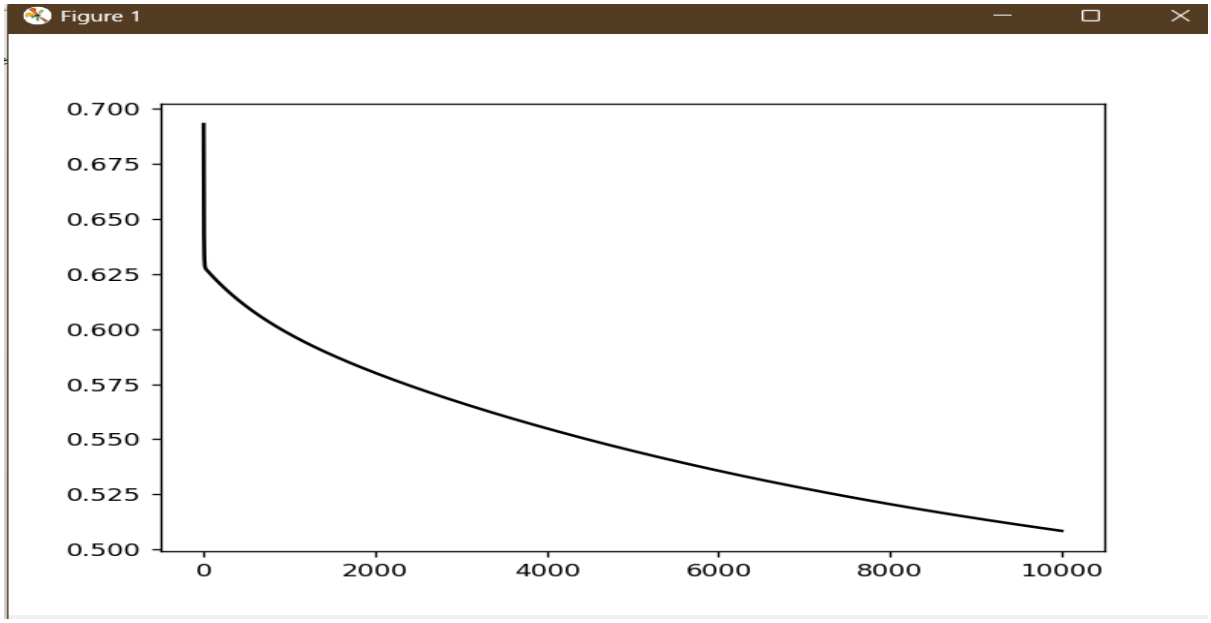
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

X_train = pd.read_csv("C:\\Users\\soban\\Downloads\\train_X.csv")
Y_train = pd.read_csv("C:\\Users\\soban\\Downloads\\train_Y.csv")
X_test = pd.read_csv("C:\\Users\\soban\\Downloads\\test_X.csv")
Y_test = pd.read_csv("C:\\Users\\soban\\Downloads\\test_Y.csv")

X_train = X_train.drop("Id", axis = 1)
Y_train = Y_train.drop("Id", axis = 1)
X_test = X_test.drop("Id", axis = 1)
Y_test = Y_test.drop("Id", axis = 1)

X_train = X_train.values
Y_train = Y_train.values
X_test = X_test.values
Y_test = Y_test.values

X_train = X_train.T
Y_train = Y_train.reshape(1, X_train.shape[1])

X_test = X_test.T
Y_test = Y_test.reshape(1, X_test.shape[1])

def model(X, Y, iterations, learning_rate):
    m = X_train.shape[1]
    n = X_train.shape[0]

    W = np.zeros((n,1))
```

```
    B = 0
    costs = []

    for i in range(iterations):
        Z = np.dot(W.T, X) + B
        A = sigmoid(Z)
        cost = -(1/m)*np.sum(Y*np.log(A) + (1 - Y)*np.log(1 - A))
        dW = (1/m)*np.dot(A - Y, X.T)
        dB = (1/m)*np.sum(A - Y)
        W = W - learning_rate * (dW.T)
        B = B - learning_rate * dB
        costs.append(cost)

        if(i%(iterations/10) == 0):
            print("Cost after", i, "iteration is: ", cost)

    return W, B, costs
def accuracy(X, Y, W, B):
    Z = np.dot(W.T, X) + B
    A= sigmoid(Z)
    A = np.where(A>0.5, 1, 0)
    Accuracy = (1 - np.sum(np.absolute(A - Y)/Y.shape[1]))*100
    print("Accuracy of the model is: ",  Accuracy, "%")
def sigmoid(X):
    return (1/(1+np.exp(-X)))

iterations = 10000
learning_rate = 0.0012
W, B, costs = model(X_train, Y_train, iterations, learning_rate)
accuracy(X_test, Y_test, W, B)
plt.figure()
plt.plot(np.arange(iterations), costs, color = 'black')
plt.show()
```

OUTPUT

```
= RESTART: D:\Pandas\LogisticRegression.py
Cost after 0 iteration is:  0.6931471805599454
Cost after 1000 iteration is:  0.5978162155052676
Cost after 2000 iteration is:  0.5801527540114927
Cost after 3000 iteration is:  0.5665452186192215
Cost after 4000 iteration is:  0.5549403313739917
Cost after 5000 iteration is:  0.5447502360016205
Cost after 6000 iteration is:  0.535721254363657 7
Cost after 7000 iteration is:  0.5276906292177218
Cost after 8000 iteration is:  0.5205305465613481
Cost after 9000 iteration is:  0.5141335016514218
Accuracy of the model is:  78.4688995215311 %
```

OUTPUT RESPONSE



USING SKLEARN

CODE:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model
from sklearn.metrics import mean_squared_error, r2_score, accuracy_score

lr = linear_model.LogisticRegression()

X_train = pd.read_csv("C:\\Users\\soban\\Downloads\\train_X.csv")
X_train = X_train.drop("Id", axis = 1)
X_train = np.array(X_train)

Y_train = pd.read_csv("C:\\Users\\soban\\Downloads\\train_Y.csv")
Y_train = Y_train.drop("Id", axis = 1)
Y_train = np.array(Y_train)

lr.fit(X_train, Y_train)

X_test = pd.read_csv("C:\\Users\\soban\\Downloads\\test_X.csv")
X_test = X_test.drop("Id", axis = 1)
age = np.mean(X_test["Age"])
```

```
fare = np.mean(X_test["Fare"])
X_test = X_test.fillna({"Age": age, "Fare": fare})
X_test = np.array(X_test)

prediction = lr.predict(X_test)

Y_test = pd.read_csv("C:\\Users\\soban\\Downloads\\test_Y.csv")
Y_test = Y_test.drop("Id", axis = 1)
Y_test = np.array(Y_test)

print("Mean Squared Error: %.2f" % mean_squared_error(Y_test, prediction))
print("Coefficients of determination: %.2f" % r2_score(Y_test, prediction))
print(lr.score(X_train,Y_train))
print(lr.score(X_test, Y_test))
print(accuracy_score(Y_test, prediction))

plt.figure()
plt.plot(X_test[:,0], prediction, color = "lavender", label = "Predictions")
```

OUTPUT

```
Mean Squared Error: 0.05
Coefficients of determination: 0.80
0.7946127946127947
0.9545454545454546
0.9545454545454546
```

CODE:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

df = pd.read_csv("C:\\Users\\soban\\Downloads\\loanfile.csv")
df["LoanAmount"] = df["LoanAmount"].fillna(df["LoanAmount"].mean())
df["Credit_History"] = df["Credit_History"].fillna(df["Credit_History"].median())
df.dropna(inplace = True)

plt.figure()
plt.subplot(2,3,1)
sns.countplot(x = df['Gender'], hue = df['Loan_Status'], palette = 'Pastel1')
```

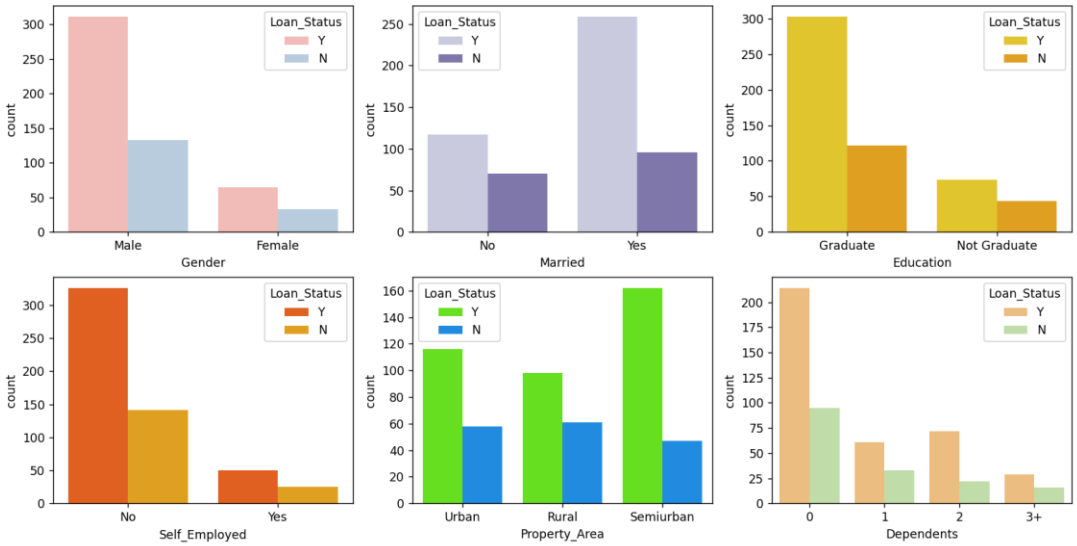
```
plt.subplot(2,3,2)
sns.countplot(x = df['Married'], hue = df['Loan_Status'], palette = 'Purples')

plt.subplot(2,3,3)
sns.countplot(x = df['Education'], hue = df['Loan_Status'], palette = 'Wistia')

plt.subplot(2,3,4)
sns.countplot(x = df['Self_Employed'], hue = df['Loan_Status'], palette = 'autumn')

plt.subplot(2,3,5)
sns.countplot(x = df['Property_Area'], hue = df['Loan_Status'], palette = 'gist_rainbow')

plt.subplot(2,3,6)
sns.countplot(x = df['Dependents'], hue = df['Loan_Status'], palette = 'Spectral')

plt.show()

df['Loan_Status'].map({'Y':1,'N':0})

df.Gender = df.Gender.map({'Male': 1, 'Female': 0})
df.Married = df.Married.map({'Yes': 1, 'No': 0})
df.Dependents = df.Dependents.map({'0': 0, '1': 1, '2': 2, '3+': 3})
df.Self_Employed= df.Self_Employed.map({'Yes': 1, 'No': 0})
df.Education = df.Education.map({'Graduate': 1, 'Non Graduate': 0})
df.Property_Area = df.Property_Area.map({'Urban': 1, 'Rural': 0})

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
df.dropna(inplace = True)
X = df.iloc[1:542, 1:11].values
Y = df.iloc[1:542, 12].values

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.3, random_state = 0)
model = LogisticRegression()
model.fit(X_train, Y_train)

lr_Prediction = model.predict(X_test)
print("LR ACCURACY = ", metrics.accuracy_score(lr_Prediction, Y_test))
print()
print("Y_Predicted Values", lr_Prediction)
```

```
print()
print("Y_test", Y_test)
```

DATASET:

Loan File dataset

OUTPUT:



ACCURACY:

```
LR ACCURACY =  0.8571428571428571

Y_Predicted Values ['Y' 'Y' 'N' 'Y' 'Y' 'N' 'Y' 'Y' 'Y' 'N' 'Y' 'Y' 'Y' 'Y' 'Y'
'N' 'Y' 'Y'
 'Y' 'Y' 'Y' 'N' 'Y' 'Y' 'Y' 'Y' 'Y' 'Y' 'Y' 'Y' 'N' 'Y' 'Y' 'Y' 'Y' 'N'
 'N' 'Y' 'N' 'Y' 'Y' 'Y' 'N' 'N' 'N' 'Y' 'Y' 'Y' 'Y' 'N' 'Y' 'Y' 'Y' 'Y'
 'N' 'Y' 'N' 'Y' 'Y' 'Y' 'Y' 'Y' 'Y' 'Y' 'Y' 'Y' 'Y' 'Y' 'Y' 'Y' 'Y' 'Y'
 'Y' 'Y' 'Y' 'N' 'Y']

Y_test ['Y' 'Y' 'N' 'N' 'Y' 'N' 'Y' 'Y' 'Y' 'N' 'Y' 'Y' 'Y' 'Y' 'N' 'N' 'Y' 'Y'
 'Y' 'Y' 'Y' 'Y' 'Y' 'Y' 'Y' 'Y' 'Y' 'Y' 'Y' 'Y' 'Y' 'Y' 'Y' 'Y' 'Y' 'N'
 'N' 'Y' 'N' 'Y' 'Y' 'N' 'N' 'N' 'Y' 'Y' 'Y' 'N' 'Y' 'N' 'Y' 'Y' 'Y' 'Y'
 'N' 'Y' 'N' 'Y' 'Y' 'Y' 'Y' 'N' 'N' 'Y' 'Y' 'Y' 'N' 'Y' 'N' 'Y' 'Y' 'Y'
 'Y' 'Y' 'Y' 'N' 'Y']
```

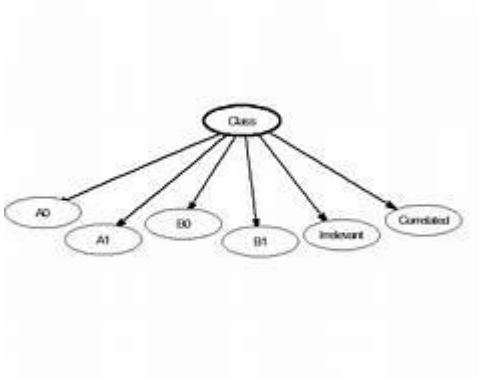
GENERATIVE LEARNING ALGORITHM (CDA)

Generative learning algorithms are a class of machine learning models that aim to understand and model the underlying distribution of data. Unlike discriminative models, which focus on distinguishing between different classes, generative models can generate new data instances that resemble the training data. Here's a detailed overview:

Key Concepts

1. **Generative Models**:
    - **Definition**: Generative models learn the joint probability distribution (P(X, Y)) of the input features (X) and the output labels (Y). They can generate new data points by sampling from this distribution[1].
    - **Examples**: Generative Adversarial Networks (GANs), Variational Autoencoders (VAEs), and Hidden Markov Models (HMMs)[2].
2. **Discriminative Models**:
    - **Definition**: Discriminative models learn the conditional probability (P(Y | X)), focusing on predicting the label (Y) given the input features (X). They are used for classification tasks[1].
    - **Examples**: Logistic Regression, Support Vector Machines (SVMs), and Neural Networks

NAIVE BAYES ALGORITHM:



Naive Bayes is a probabilistic machine learning algorithm based on Bayes' Theorem. It is widely used for classification tasks due to its simplicity and effectiveness. Here's a detailed overview:

Key Concepts

1. **Bayes' Theorem**:
    - Bayes' Theorem describes the probability of an event based on prior knowledge of conditions related to the event. The formula is:

$$P(A|B) = P(B)P(B|A) \cdot P(A)$$

    - Here, (P(A|B)) is the posterior probability, (P(B|A)) is the likelihood, (P(A)) is the prior probability, and (P(B)) is the marginal likelihood.
2. **Naive Assumption**:
    - The "naive" part of Naive Bayes comes from the assumption that all features are independent of each other given the class label.

CODE:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme()

plt.figure()
data = pd.read_csv(r"C:\Users\soban\Downloads\Breast_cancer_data.csv")
plt.hist(data["diagnosis"])
plt.show()
```

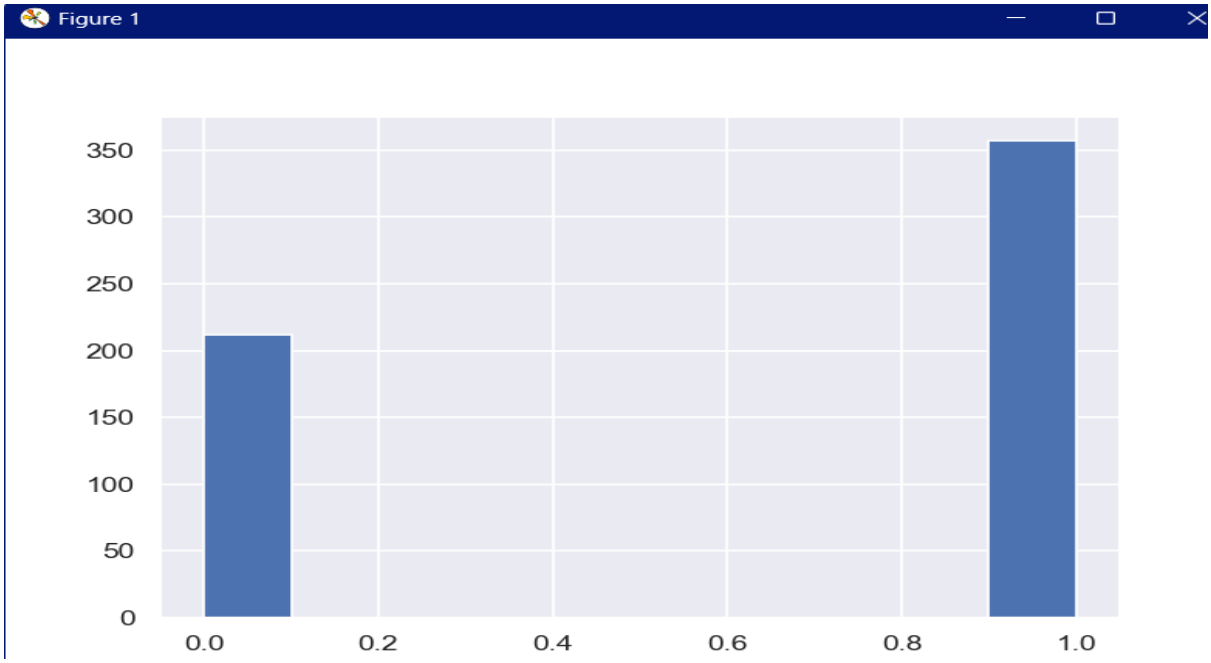
```
plt.figure()
corr = data.corr(method="pearson")
c = sns.diverging_palette(220, 20, as_cmap = True)
sns.heatmap(corr, cmap = c, square = True, linewidths = 0.2)
plt.show()

data = data[["mean_radius", "mean_texture", "mean_smoothness", "diagnosis"]]
plt.figure()
plt.subplot(1,3,1)
sns.histplot(data["mean_radius"], kde = True, color = 'r')
plt.subplot(1,3,2)
sns.histplot(data["mean_smoothness"], kde = True, color = 'b')
plt.subplot(1,3,3)
sns.histplot(data["mean_texture"], kde = True, color = 'm')
plt.show()

def calculate_prior(df, Y):
  classes = sorted(list(df[Y].unique()))
  prior = []
  for i in classes:
    prior.append(len(df[df[Y]==i])/len(df))
  return prior

def calculate_likelihood_gaussian(df, feat_name, feat_val, Y, label):
  feat = list(df.columns)
  df = df[df[Y]==label]
  mean, std = df[feat_name].mean(), df[feat_name].std()
  p_x_given_y = (1 / (np.sqrt(2 * np.pi) * std)) *  np.exp(-((feat_val-mean)**2 / (2 * std**2
)))
  return p_x_given_y

def naive_bayes_gaussian(df, X, Y):
  features = list(df.columns)[:-1]

  prior = calculate_prior(df, Y)

  Y_pred = []
  for x in X:
    labels = sorted(list(df[Y].unique()))
    likelihood = [1]*len(labels)
```

```
        for j in range(len(labels)):
            for i in range(len(features)):
                likelihood[j] *= calculate_likelihood_gaussian(df, features[i], x[i], Y, labels[j])
        post_prob = [1]*len(labels)
        for j in range(len(labels)):
            post_prob[j] = likelihood[j] * prior[j]

        Y_pred.append(np.argmax(post_prob))

    return np.array(Y_pred)

from sklearn.model_selection import train_test_split
train, test = train_test_split(data, test_size=.2, random_state=41)

X_test = test.iloc[:,:-1].values
Y_test = test.iloc[:,-1].values
Y_pred = naive_bayes_gaussian(train, X=X_test, Y="diagnosis")

from sklearn.metrics import confusion_matrix, f1_score
print(confusion_matrix(Y_test, Y_pred))
print(f1_score(Y_test, Y_pred))
```

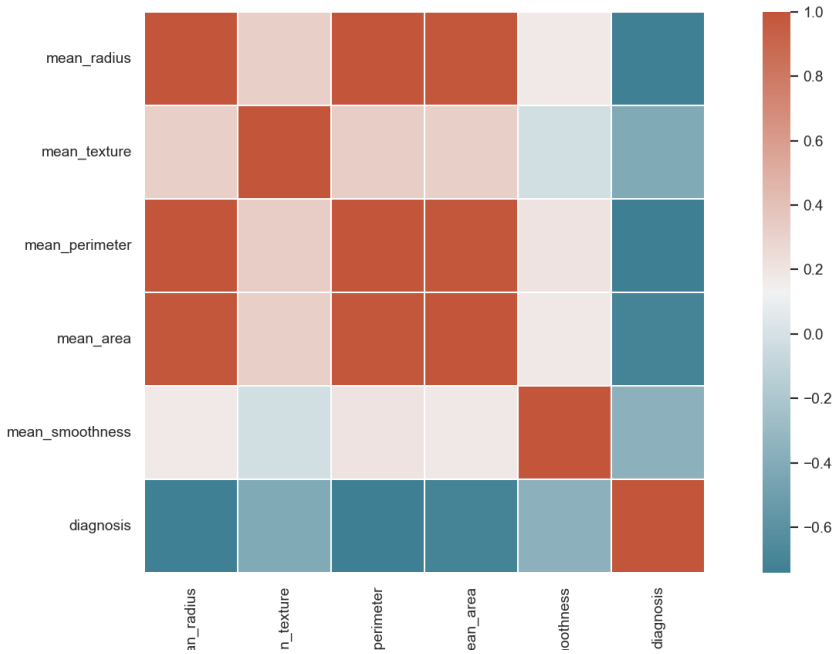
DATASET

Breast Cancer prediction dataset

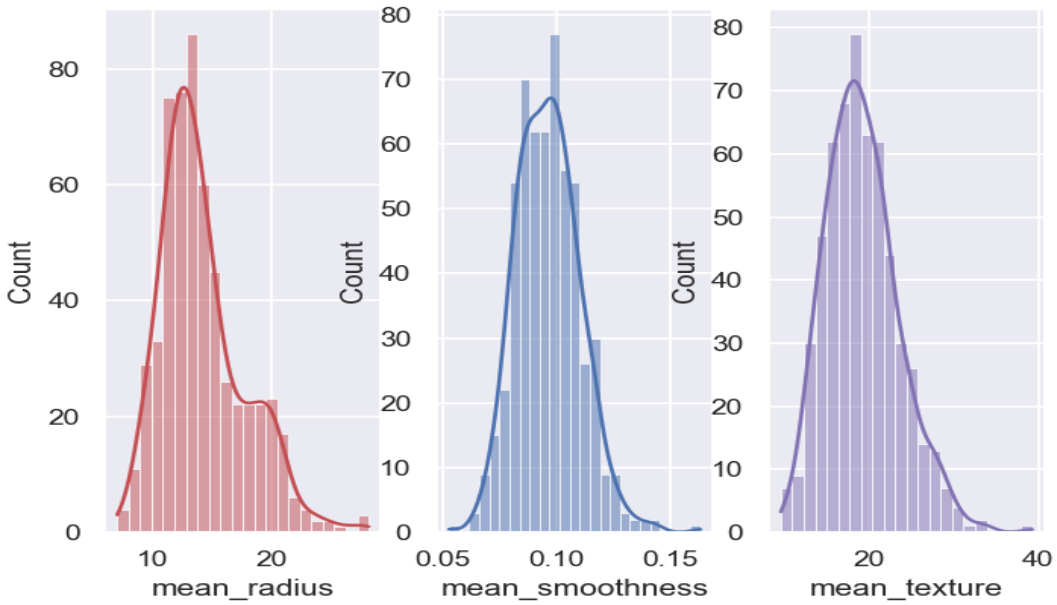
OUTCOME



HEATMAP: Showing correlation among different parameters need to be eliminated.



GAUSSIAN DISTRIBUTION: Applying gaussian distribution to the naive bayes.



CONFUSION MATRIX AND ACCURACY SCORE

```
[[36  4]
 [ 0 74]]
0.9736842105263158
```

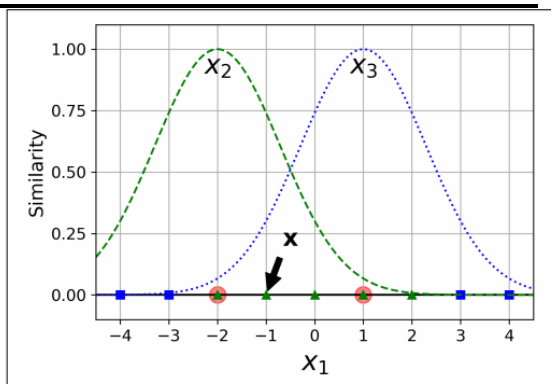
CODE:

```
from sklearn import metrics
```

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, f1_score
from sklearn.naive_bayes import GaussianNB
import pandas as pd
import numpy as np

data = pd.read_csv("C:\\Users\\soban\\Downloads\\archive
(5)\\Breast_cancer_data.csv")
X = data.iloc[:, :-1]
X = np.array(X)

Y = data.iloc[:, -1]
Y = np.array(Y)

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.2, random_state = 41)
Nb = GaussianNB()

Nb.fit(X_train, Y_train)
Y_pred = Nb.predict(X_test)

print(confusion_matrix(Y_test, Y_pred))
print(f1_score(Y_test, Y_pred))
```

OUTPUT

```
[[36  4]
 [ 2 72]]
0.96
```



SUPPORT VECTOR MACHINES

Support Vector Machines (SVM) are powerful supervised learning models used for classification and regression tasks. They are particularly effective in high-dimensional spaces and are known for their robustness in handling both linear and non-linear data. Here's a detailed overview:

$$\phi_\gamma(\mathbf{x}, \ell) = \exp\left(-\gamma \| \mathbf{x} - \ell \|^2\right)$$

Key Concepts

1. **Hyperplane**:

- A hyperplane is a decision boundary that separates different classes in the feature space. In a 2D space, it's a line; in 3D, it's a plane; and in higher dimensions, it's a hyperplane.

2. **Support Vectors**:
   - Support vectors are the data points that are closest to the hyperplane. These points are critical as they define the position and orientation of the hyperplane.
3. **Margin**:
   - The margin is the distance between the hyperplane and the nearest data points from each class. SVM aims to maximize this margin to ensure the best separation between classes.

Functional Margin

The functional margin of a training example ((x^{(i)}, y^{(i)})) with respect to a hyperplane defined by ((w, b)) is given by:

$$\hat{\gamma}^{(i)} = y^{(i)}(w^T x^{(i)} + b)$$

- **Interpretation**: The functional margin measures how confident the model is in its classification. A larger functional margin indicates a more confident classification.

Geometric Margin

The geometric margin normalizes the functional margin by the norm of the weight vector (w):

- $$\gamma^{(i)} = \frac{\hat{\gamma}^{(i)}}{\|w\|} = \frac{y^{(i)}(w^T x^{(i)} + b)}{\|w\|}$$

  Interpretation: The geometric margin represents the actual distance from the data point to the hyperplane. Maximizing the geometric margin ensures that the hyperplane is optimally positioned to separate the classes.

SVM Optimization

The goal of SVM is to find the hyperplane that maximizes the geometric margin. This can be formulated as an optimization problem:

$$\min_{w,b} \frac{1}{2}\|w\|^2$$

subject to:

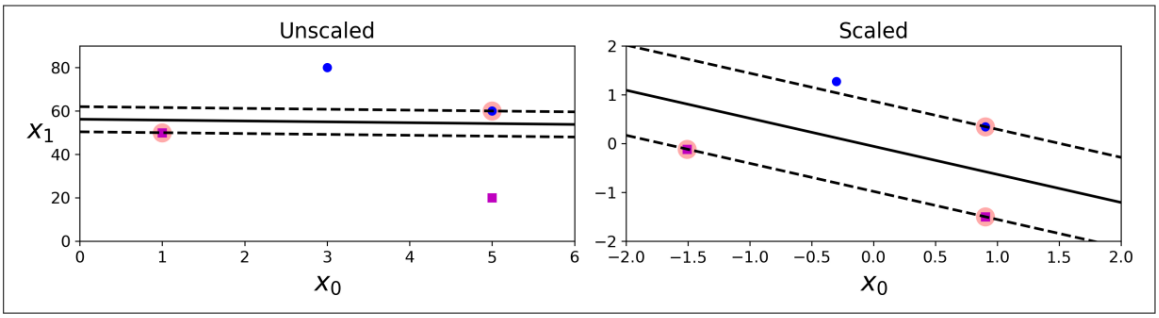
$$y^{(i)}(w^T x^{(i)} + b) \geq 1 \quad \forall i$$

This optimization problem ensures that the hyperplane not only separates the classes but does so with the maximum possible margin.

Kernel Trick

For non-linearly separable data, SVM uses the kernel trick to transform the data into a higher-dimensional space where a linear hyperplane can separate the classes.
Common kernels include:

- **Linear Kernel**: (K (x, x') = x^T x')
- **Polynomial Kernel**: (K (x, x') = (x^T x' + c) ^d)

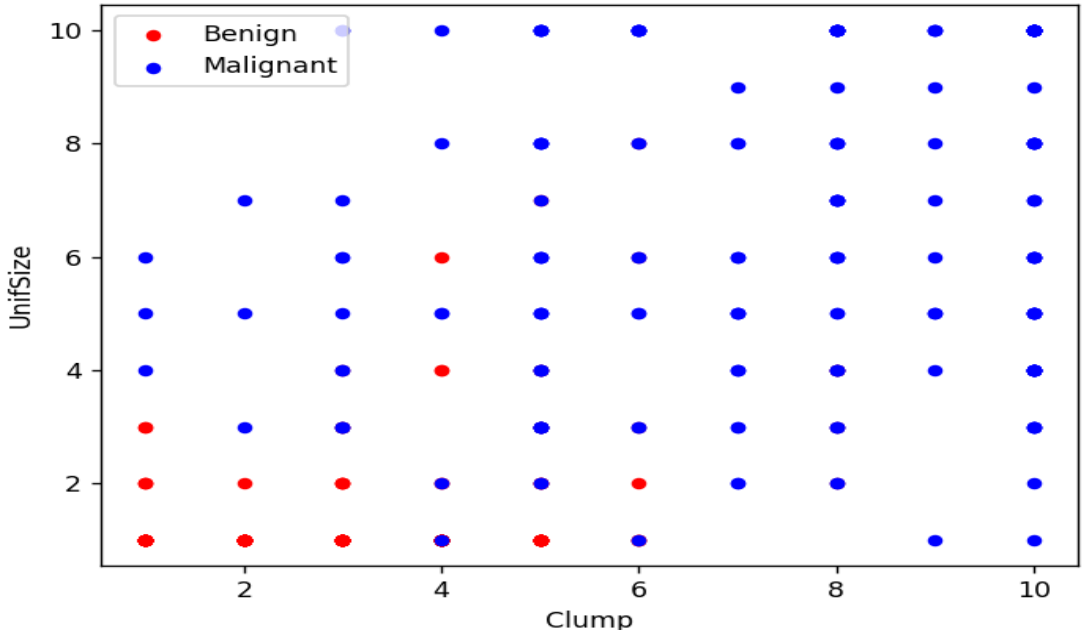


CODE

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

data = pd.read_csv("C:\\Users\\soban\\Downloads\\cell_samples.csv")
mag_data = data[data['Class']== 4][0:200]
beni_data = data[data['Class']== 2][0:200]

axes = beni_data.plot(kind = 'scatter',  x = 'Clump', y = 'UnifSize', color = 'r', label = 'Benign')
mag_data.plot(kind = 'scatter',  x = 'Clump', y = 'UnifSize', color = 'b', label = 'Malignant', ax = axes)
plt.show()

data = data[pd.to_numeric(data['BareNuc'], errors = 'coerce').notnull()]
data['BareNuc']  = data['BareNuc'].astype('int')

feature = data[['Clump', 'UnifSize', 'UnifShape', 'MargAdh', 'SingEpiSize', 'BareNuc', 'BlandChrom', 'NormNucl', 'Mit']]
X = np.asarray(feature)

Y = np.array(data['Class'])

from sklearn.model_selection import train_test_split
```

```
from sklearn import metrics
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn import svm
SV = svm.SVC()
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.3, random_state =  0)
SV.fit(X_train, Y_train)
prediction = SV.predict(X_test)
print(SV.support_vectors_)
print(confusion_matrix(Y_test, prediction))
print(accuracy_score(Y_test, prediction))
```

OUTPUT

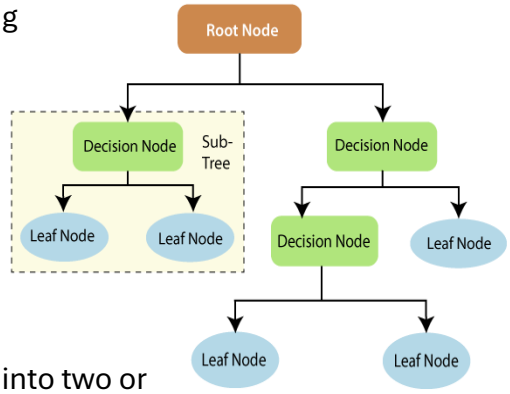


CONFUSION MATRIX & ACCURACY SCORE

```
[[124    6]
 [   4   71]]
0.951219512195219
```

DECISION FOREST

A **decision tree** is a non-parametric supervised learning algorithm used for both classification and regression tasks. It models decisions and their possible consequences in a tree-like structure. Here's a detailed overview:



Structure of a Decision Tree

1. **Root Node**:
   - The topmost node in a decision tree. It represents the entire dataset and is split into two or more homogeneous sets.
2. **Internal Nodes**:
   - Nodes that represent decisions or tests on attributes. Each internal node splits the data based on a specific feature.
3. **Branches**:
   - The connections between nodes. Each branch represents the outcome of a decision or test.
4. **Leaf Nodes (Terminal Nodes)**:
   - Nodes that represent the outcome or prediction. They do not split further.

How Decision Trees Work

1. **Splitting**:
   - The process of dividing a node into two or more sub-nodes based on certain conditions. The goal is to create subsets of the data that are as homogeneous as possible.
2. **Attribute Selection Measures**:
   - **Gini Index**: Measures the impurity of a node. A lower Gini index indicates a purer node.
   - **Information Gain**: Measures the reduction in entropy or impurity. It is used to decide which feature to split on at each step in the tree.
   - **Chi-Square**: Measures the statistical significance of the differences between observed and expected frequencies.
3. **Pruning**:
   - The process of removing branches that have little importance to reduce the complexity of the model and prevent overfitting. Pruning can be done by setting a threshold for the minimum number of samples required to split a node or by using cross-validation.

Advantages and Disadvantages

Advantages:

- **Easy to Understand**: The tree structure is intuitive and easy to interpret.
- **Non-Parametric**: Does not assume any underlying distribution of the data.
- **Handles Both Numerical and Categorical Data**: Can be used for various types of data.

Disadvantages:

- **Overfitting**: Decision trees can easily overfit the training data, especially if they are deep.
- **Instability**: Small changes in the data can result in a completely different tree.
- **Bias**: Can be biased towards features with more levels.

CODE

```
training_data = [
  ['Green', 3, 'Apple'],
  ['Yellow', 3, 'Apple'],
  ['Red', 1, 'Grape'],
  ['Red', 1, 'Grape'],
  ['Yellow', 3, 'Lemon'],
]

header = ["color", "diameter", "label"]


def unique_vals(rows, col):
  return set([row[col] for row in rows])

def class_counts(rows):
  counts = {}
  for row in rows:
    label = row[-1]
    if label not in counts:
      counts[label] = 0
    counts[label] += 1
  return counts


def is_numeric(value):
  """Test if a value is numeric."""
  return isinstance(value, int) or isinstance(value, float)

class Question:

  def __init__(self, column, value):
    self.column = column
    self.value = value
```

```
    def match(self, example):
        val = example[self.column]
        if is_numeric(val):
            return val >= self.value
        else:
            return val == self.value

    def __repr__(self):
        condition = "=="
        if is_numeric(self.value):
            condition = ">="
        return "Is %s %s %s?" % (
            header[self.column], condition, str(self.value))

def partition(rows, question):
    true_rows, false_rows = [], []
    for row in rows:
        if question.match(row):
            true_rows.append(row)
        else:
            false_rows.append(row)
    return true_rows, false_rows

def gini(rows):
    counts = class_counts(rows)
    impurity = 1
    for lbl in counts:
        prob_of_lbl = counts[lbl] / float(len(rows))
        impurity -= prob_of_lbl**2
    return impurity

def info_gain(left, right, current_uncertainty):
    p = float(len(left)) / (len(left) + len(right))
    return current_uncertainty - p * gini(left) - (1 - p) * gini(right)

def find_best_split(rows):
    best_gain = 0
    best_question = None
    current_uncertainty = gini(rows)
    n_features = len(rows[0]) - 1
```

```
    for col in range(n_features):

        values = set([row[col] for row in rows])

        for val in values:

            question = Question(col, val)

            true_rows, false_rows = partition(rows, question)

            if len(true_rows) == 0 or len(false_rows) == 0:
                continue

            gain = info_gain(true_rows, false_rows, current_uncertainty)

            if gain >= best_gain:
                best_gain, best_question = gain, question

    return best_gain, best_question

class Leaf:
    def __init__(self, rows):
        self.predictions = class_counts(rows)


class Decision_Node:

    def __init__(self,
                 question,
                 true_branch,
                 false_branch):
        self.question = question
        self.true_branch = true_branch
        self.false_branch = false_branch


def build_tree(rows):
    gain, question = find_best_split(rows)

    if gain == 0:
```

```
        return Leaf(rows)

    true_rows, false_rows = partition(rows, question)

    true_branch = build_tree(true_rows)

    false_branch = build_tree(false_rows)

    return Decision_Node(question, true_branch, false_branch)


def print_tree(node, spacing=""):

    if isinstance(node, Leaf):
        print (spacing + "Predict", node.predictions)
        return

    print (spacing + str(node.question))

    print (spacing + '--> True:')
    print_tree(node.true_branch, spacing + "  ")
    print (spacing + '--> False:')
    print_tree(node.false_branch, spacing + "  ")


def classify(row, node):
    """See the 'rules of recursion' above."""

    if isinstance(node, Leaf):
        return node.predictions

    if node.question.match(row):
        return classify(row, node.true_branch)
    else:
        return classify(row, node.false_branch)

def print_leaf(counts):
    """A nicer way to print the predictions at a leaf."""
    total = sum(counts.values()) * 1.0
    probs = {}
    for lbl in counts.keys():
```

```
        probs[lbl] = str(int(counts[lbl] / total * 100)) + "%"
    return probs

if __name__ == '__main__':

    my_tree = build_tree(training_data)

    print_tree(my_tree)

    testing_data = [
        ['Green', 3, 'Apple'],
        ['Yellow', 4, 'Apple'],
        ['Red', 2, 'Grape'],
        ['Red', 1, 'Grape'],
        ['Yellow', 3, 'Lemon'],
    ]

    for row in testing_data:
        print ("Actual: %s. Predicted: %s" %
               (row[-1], print_leaf(classify(row, my_tree))))
```

OUTPUT

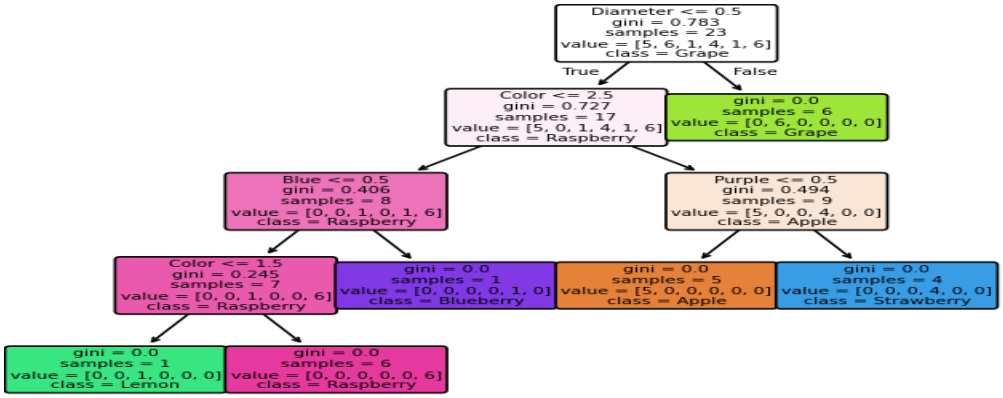
```
Is diameter >= 3?
--> True:
  Is color == Yellow?
  --> True:
    Predict {'Apple': 1, 'Lemon': 1}
  --> False:
    Predict {'Apple': 1}
--> False:
  Predict {'Grape': 2}
Actual: Apple. Predicted: {'Apple': '100%'}
Actual: Apple. Predicted: {'Apple': '50%', 'Lemon': '50%'}
Actual: Grape. Predicted: {'Grape': '100%'}
Actual: Grape. Predicted: {'Grape': '100%'}
Actual: Lemon. Predicted: {'Apple': '50%', 'Lemon': '50%'}
```

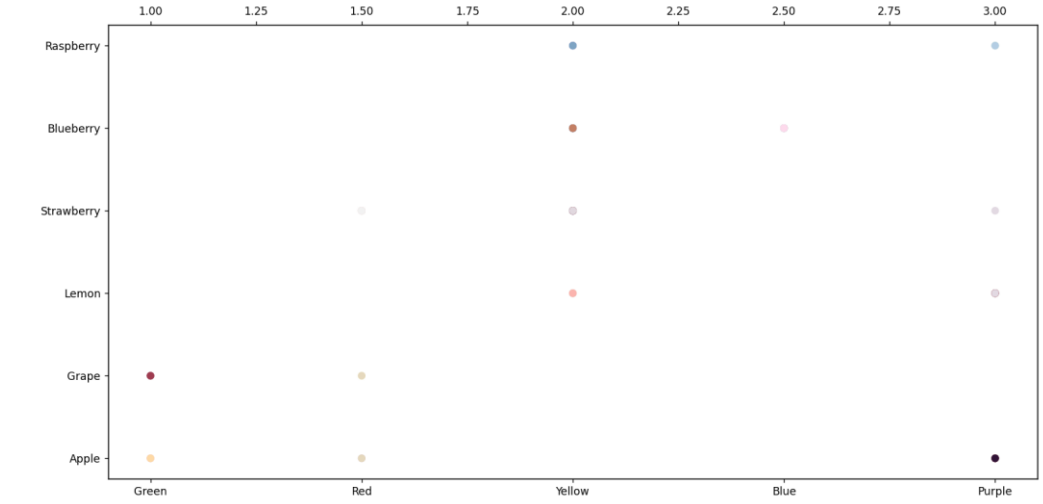
CODE

```
from sklearn import tree
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
```

```
dt = tree.DecisionTreeClassifier()
data = pd.read_csv("C:\\Users\\soban\\OneDrive - National University of Sciences &
Technology\\Desktop\\Pandas\\fruit.csv")

columns = ['Color', 'Diameter', 'Fruit']

fig, ax = plt.subplots()
ay = ax.twiny()
colors = np.random.randint(10,100, data['Color'].size)
ax.scatter(data['Color'], data['Fruit'], c = colors, cmap = 'Pastel1')
ay.scatter(data['Diameter'], data['Fruit'], c = colors, cmap = 'twilight')
plt.show()

from sklearn.preprocessing import OneHotEncoder
one_hot = OneHotEncoder()
enc = one_hot.fit_transform(data[["Color"]])
data[one_hot.categories_[0]] = enc.toarray()

X = data
X = X.drop(columns=['Color', 'Fruit'])
X = X.values
Y = data["Fruit"].values
Y = Y.reshape(Y.size, 1)

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, stratify = Y, test_size = 0.3,
random_state = 0)

dt.fit(X_train, Y_train)
Y_pred = dt.predict(X_test)

print(accuracy_score(Y_test, Y_pred))
print(classification_report(Y_test, Y_pred))
print(dt.predict_proba(X_test))

plt.figure()
tree.plot_tree(dt, feature_names = data.columns, class_names = data['Fruit'].unique(),
filled = True, rounded = True)
plt.show()
```

OUTPUT






```
0.9090909090909091
                precision    recall  f1-score   support

         Apple       0.75      1.00      0.86         3
    Blueberry       1.00      1.00      1.00         2
         Grape       1.00      1.00      1.00         1
         Lemon       1.00      1.00      1.00         2
     Raspberry       1.00      1.00      1.00         1
    Strawberry       1.00      0.50      0.67         2

      accuracy                           0.91        11
     macro avg       0.96      0.92      0.92        11
  weighted avg       0.93      0.91      0.90        11

[[1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0.]
 [1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0.]
 [1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1.]
 [1. 0. 0. 0. 0. 0.]]
```

RANDOM FOREST

Random Forest is a popular machine learning algorithm that combines the outputs of multiple decision trees to produce a single prediction or result. Here are the key points:

1. **Ensemble of Decision Trees**:
    - Random Forest builds an ensemble (or a “forest”) of decision trees.
    - Each tree is trained on a random subset of the data.

    - These individual trees work together to make a more accurate and stable prediction.
2. **Classification and Regression**:
    - Random Forest can handle both classification and regression tasks.
    - In classification, it predicts class labels (e.g., spam or not spam).
    - In regression, it predicts continuous values (e.g., house prices).
3. **Randomness and Diversity**:
    - While growing the trees, Random Forest adds extra randomness.
    - Instead of searching for the most important feature during node splitting, it considers a random subset of features.
    - This diversity leads to better overall performance.
4. **Hyperparameters**:
    - Random Forest has hyperparameters similar to decision trees and bagging classifiers.
    - You can control aspects like tree depth, number of trees, and feature sampling.
5. **Feature Importance**:
    - Random Forest provides feature importance scores, helping you understand which features contribute most to predictions.
6. **Applications**:
    - Random Forest is widely used in various domains, including finance, healthcare, and natural language processing.

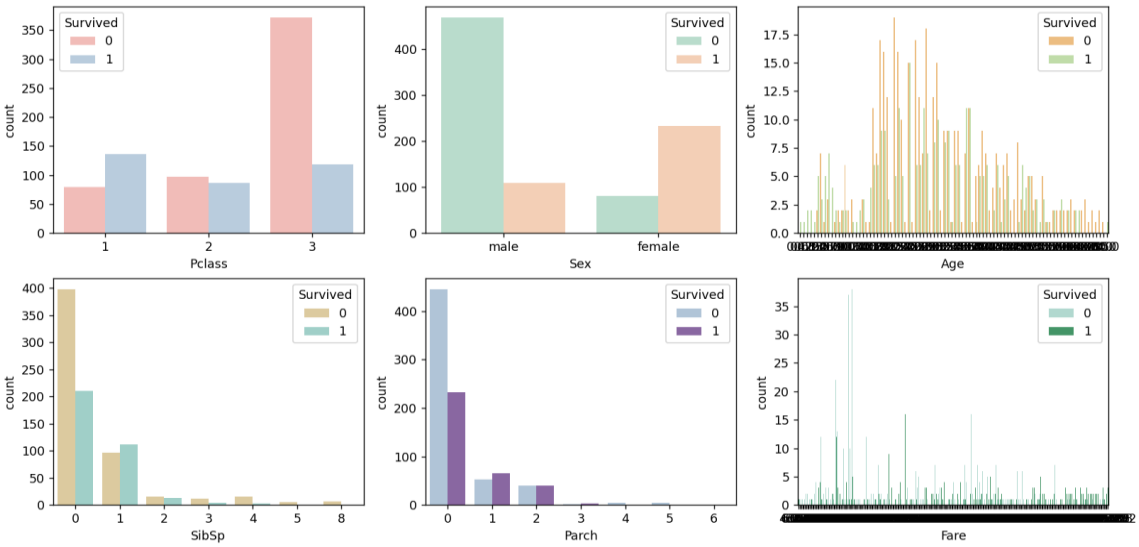
CODE

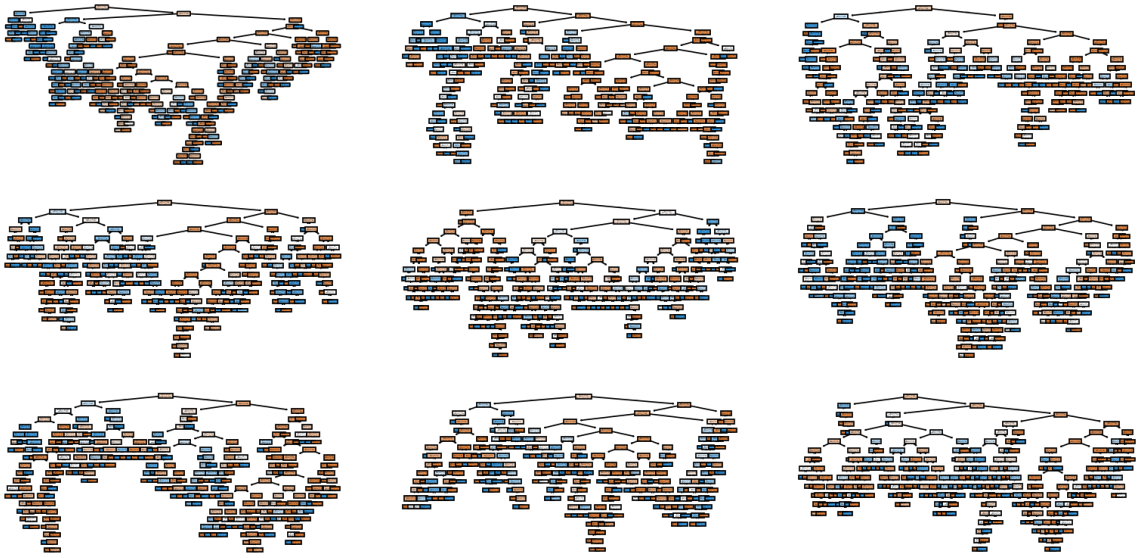
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')

titanic_data = pd.read_csv("C:\\Users\\soban\\Downloads\\titanic.csv")

titanic_data = titanic_data.dropna(subset=['Survived'])

X = titanic_data[['Pclass', 'Sex', 'Age', 'SibSp', 'Parch', 'Fare']]
y = titanic_data['Survived']

import seaborn as sns
plt.figure()
```

```
plt.subplot(2,3,1)
sns.countplot(x = titanic_data['Pclass'], hue = titanic_data['Survived'], palette =
'Pastel1')
plt.subplot(2,3,2)
sns.countplot(x = titanic_data['Sex'], hue = titanic_data['Survived'], palette = 'Pastel2')
plt.subplot(2,3,3)
sns.countplot(x = titanic_data['Age'], hue = titanic_data['Survived'], palette = 'Spectral')
plt.subplot(2,3,4)
sns.countplot(x = titanic_data['SibSp'], hue = titanic_data['Survived'], palette = 'BrBG')
plt.subplot(2,3,5)
sns.countplot(x = titanic_data['Parch'], hue = titanic_data['Survived'], palette = 'BuPu')
plt.subplot(2,3,6)
sns.countplot(x = titanic_data['Fare'], hue = titanic_data['Survived'], palette = 'BuGn')
plt.show()

columns = ['Pclass', 'Sex', 'Age', 'SibSp', 'Parch', 'Fare']

X.loc[:, 'Sex'] = X['Sex'].map({'female': 0, 'male': 1})

X.loc[:, 'Age'].fillna(X['Age'].median(), inplace=True)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

rf_classifier = RandomForestClassifier()

rf_classifier.fit(X_train, y_train)

y_pred = rf_classifier.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
classification_rep = classification_report(y_test, y_pred)

print(accuracy)
print(classification_rep)

from sklearn import tree

plt.figure()
plt.subplot(3, 3, 1)
tree.plot_tree(rf_classifier.estimators_[0], feature_names = titanic_data.columns, filled
= True, rounded = True);
```

```
plt.subplot(3, 3, 2)
tree.plot_tree(rf_classifier.estimators_[12], feature_names = titanic_data.columns,
filled = True);

plt.subplot(3, 3, 3)
tree.plot_tree(rf_classifier.estimators_[24], feature_names = titanic_data.columns,
filled = True);

plt.subplot(3, 3, 4)
tree.plot_tree(rf_classifier.estimators_[36], feature_names = titanic_data.columns,
filled = True);

plt.subplot(3, 3, 5)
tree.plot_tree(rf_classifier.estimators_[48], feature_names = titanic_data.columns,
filled = True);

plt.subplot(3, 3, 6)
tree.plot_tree(rf_classifier.estimators_[60], feature_names = titanic_data.columns,
filled = True);

plt.subplot(3, 3, 7)
tree.plot_tree(rf_classifier.estimators_[72], feature_names = titanic_data.columns,
filled = True);

plt.subplot(3, 3, 8)
tree.plot_tree(rf_classifier.estimators_[84], feature_names = titanic_data.columns,
filled = True);

plt.subplot(3, 3, 9)
tree.plot_tree(rf_classifier.estimators_[99], feature_names = titanic_data.columns,
filled = True);
plt.show()
```

OUTPUT





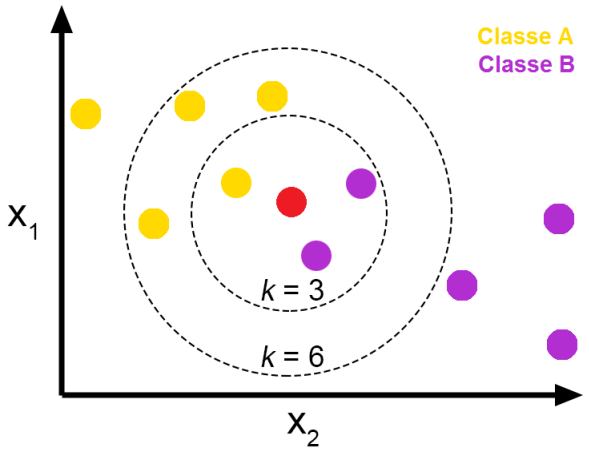
0.8044692737430168

	precision	recall	f1-score	support
0	0.82	0.85	0.84	105
1	0.77	0.74	0.76	74
accuracy			0.80	179
macro avg	0.80	0.80	0.80	179
weighted avg	0.80	0.80	0.80	179

K-NEAREST NEIGHBOUR

1. **Algorithm:**

- KNN is a **lazy learning** algorithm, meaning it doesn't build an explicit model during training.
- Instead, it memorizes the entire training dataset and uses it for predictions.



2. **Distance Metrics**:
   - KNN relies on a distance metric (usually Euclidean distance) to measure similarity between data points.
   - Other distance metrics include Manhattan distance, Minkowski distance, and cosine similarity.
3. **Choosing k**:
   - The choice of k (number of neighbours) is crucial.
   - A small k can lead to overfitting (reacting to noise), while a large k can lead to underfitting (smoothing out decision boundaries).
4. **Decision Rule**:
   - For classification, KNN uses majority voting among the k nearest neighbours.
   - For regression, it averages the target values of those neighbours.
5. **Scaling Features**:
   - Normalize or standardize features to ensure equal importance across dimensions.
   - Otherwise, features with larger scales dominate the distance calculation.
6. **Pros and Cons**:
   - **Pros**: Simple, intuitive, and works well for certain datasets.
   - **Cons**: Sensitive to outliers, computationally expensive during prediction.

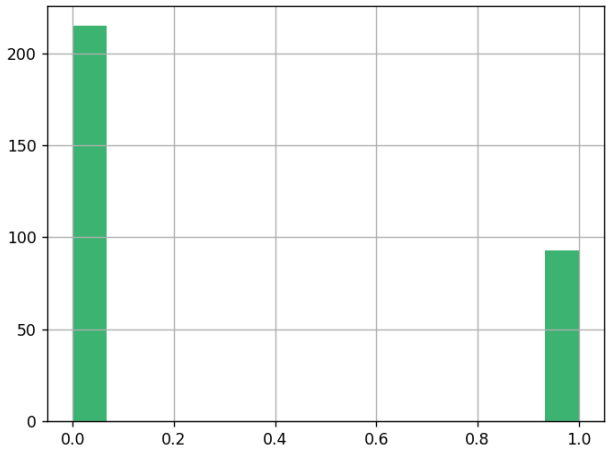
CODE

```
import pandas as pd
import numpy as np
from sklearn.metrics import f1_score, accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier

import warnings
warnings.filterwarnings('ignore')
```

```
data = pd.read_csv("C:\\Users\\soban\\Downloads\\Training.csv")
col = ['Glucose','BloodPressure', 'SkinThickness', 'Insulin', 'BMI', 'Age']
for i in col:
  data[col] = data[col].replace(0, np.nan)
  mean = data[col].mean(skipna = True)
  data[col] = data[col].replace(np.nan, mean)

X_train = data.iloc[:, :-1]
Y_train = data.iloc[:, -1].values
Y_train = Y_train.reshape(len(Y_train), 1)

data = pd.read_csv("C:\\Users\\soban\\Downloads\\Testing.csv")
col = ['Glucose','BloodPressure', 'SkinThickness', 'Insulin', 'BMI', 'Age']
for i in col:
  data[col] = data[col].replace(0, np.nan)
  mean = data[col].mean(skipna = True)
  data[col] = data[col].replace(np.nan, mean)

X_test = data.iloc[:, :-1]
Y_test = data.iloc[:, -1].values
Y_test = Y_test.reshape(len(Y_test), 1)

KN = KNeighborsClassifier(n_neighbors = 17 , p = 2, metric = 'euclidean')
sc = StandardScaler()
X_train = sc.fit_transform(X_train)

KN.fit(X_train, Y_train)
X_test = sc.fit_transform(X_test)
Y_pred = KN.predict(X_test)

print(f1_score(Y_test, Y_pred))
print(accuracy_score(Y_test, Y_pred))
import matplotlib.pyplot as plt
plt.figure()
data["Outcome"].hist(bins=15, color = 'mediumseagreen')
plt.show()

import seaborn as sns
cmap = sns.cubehelix_palette(as_cmap=True)
f, ax = plt.subplots()
points = ax.scatter(X_test[:, 4], X_test[:, 6], c = Y_pred, s=50, cmap=cmap)
```

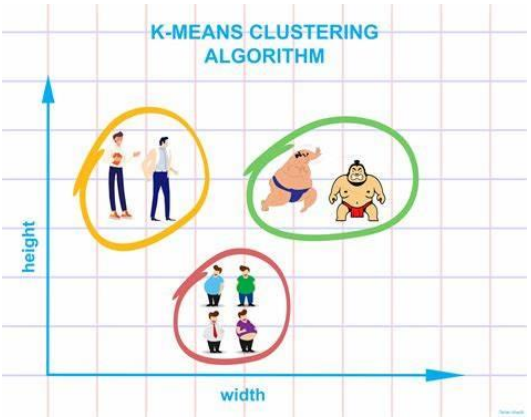
```
f.colorbar(points)
plt.show()

sns.pairplot(data = data, hue = 'Outcome', palette = ['Red', 'Blue', 'limegreen'])
plt.show()

h = 0.2
x_min, x_max = X_train[:,0].min() - .5, X_train[:,0].max() + .5
y_min, y_max = X_train[:,1].min() - .5, X_train[:,1].max() + .5
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
Kn = KNeighborsClassifier()
X = data.iloc[:,[4, 6]].values
Y = data.Outcome
Y = Y.values
Y = Y.reshape(308,1)
Kn.fit(X, Y)
Z = Kn.predict(np.c_[xx.ravel(), yy.ravel()])

import pylab as pl
# Put the result into a color plot
Z = Z.reshape(xx.shape)
pl.figure(1, figsize=(4, 3))
pl.set_cmap(pl.cm.Paired)
pl.pcolormesh(xx, yy, Z)

# Plot also the training points
pl.scatter(X[:,0], X[:,1],c=Y )
pl.xlabel('Glucose')
pl.ylabel('DiabetesPedigreeFunction')
##
##pl.xlim(xx.min(), xx.max())
##pl.ylim(yy.min(), yy.max())
##pl.xticks(())
##pl.yticks(())
##plt.show()
pl.show()
```

DATASET DIABETES
OUTPUT




```
- RESTART: D:\Pandas\KNNAlgo.py
0.6111111111111112
0.7272727272727273
```

K-MEANS CLUSTERING



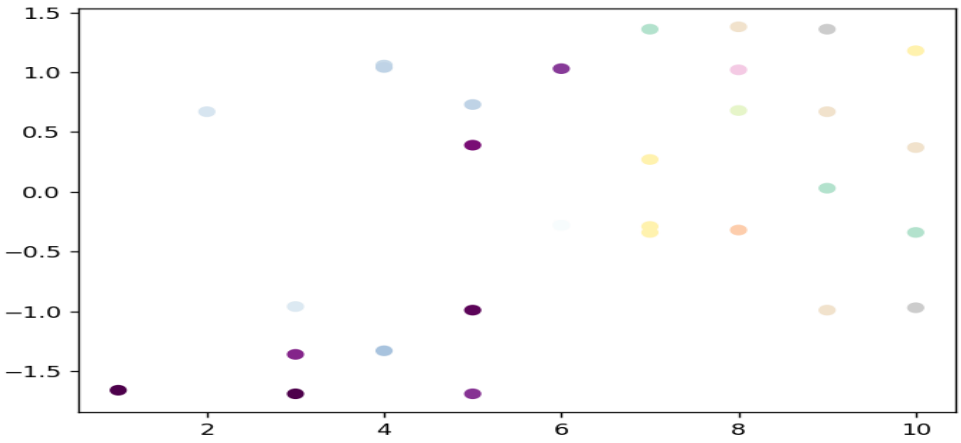
It is an unsupervised algorithm.
K-means clustering aims to divide a set of observations into *k* clusters. Here's how it works:

1. **Objective**:
    - Given a set of *n* observations (data points), we want to group them into *k* clusters.
    - Each observation belongs to the cluster with the nearest mean (centroid).
2. **Algorithm Steps**:
    - Initialize *k* cluster centroids randomly.
    - Assign each data point to the nearest centroid.
    - Update centroids by computing the mean of points in each cluster.
    - Repeat the assignment and update steps until convergence.
3. **Key Concepts**:
    - **Centroid**: The mean of data points within a cluster.
    - **Voronoi Cells**: The partitioning of data space based on centroids.
    - **Within-Cluster Sum of Squares (WCSS)**: Minimized by k-means.

CODE

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans, k_means

data = pd.read_csv("C:\\Users\\soban\\Downloads\\3.12. Example.csv")
km = KMeans(n_clusters = 2, random_state = 0, n_init = "auto")
label = km.fit_predict(data)

centre = km.cluster_centers_

label0 = data[label == 0]
label1 = data[label == 1]

color = np.random.randint(10, 100, len(label0))
colors = np.random.randint(10, 100, len(label1))
plt.scatter(label0.iloc[:,0], label0.iloc[:,1], c = color, cmap = "Pastel2")
plt.scatter(label1.iloc[:,0], label1.iloc[:,1], c = colors, cmap = "BuPu")
plt.show()
print(abs(km.score(data)))
```

OUTPUT




```
74.54254375
```

INTRODUCTION TO PYCARRET

1. INTRODUCTION

- **PyCaret** is an open-source, low-code machine learning library designed to simplify common tasks in a machine learning project.

- It's a Python version of the popular **Caret** package in R.
- With just a few lines of code, you can evaluate, compare, and tune machine learning models.

2. **Key Features**:
   - **Automated Workflow**: PyCaret automates steps like data transformation, model evaluation, and hyperparameter tuning.
   - **Productivity Boost**: Achieve more with less code, making it great for both seasoned data scientists and beginners.
   - **Model Comparison**: Easily compare standard machine learning algorithms.
   - **Hyperparameter Tuning**: Fine-tune model performance effortlessly.
3. **Use Cases**:
   - **Classification** and **Regression**: PyCaret works well for both.
   - **Text Classification**, **Image Recognition**, and more.
4. **Installation**:
   - You can install PyCaret using pip:
   - pip install pycaret
5. **Getting Started**:
   - Define data transformations with setup ().
   - Compare models using compare_models().
   - Tune hyperparameters with tune_model().

CODE

```
from pycaret.classification import *
import pandas as pd
from sklearn.model_selection import train_test_split
from pycaret.datasets import get_data

data = get_data('diabetes')
train_data, test_data = train_test_split(data, test_size = 0.3, random_state = 42)
new = setup(data = train_data, target = 'Class variable')

compare_models()
best_model = automl(optimize = 'Accuracy')
print(best_model)
tuned = tune_model(best_model)
print(tuned)

predict = predict_model(tuned, data = test_data)
print(predict)
```

```
from sklearn.metrics import accuracy_score
accuracy_score(test_data['Class variable'], predict['Class variable'])
```

OUTPUT

	Number of times pregnant	Plasma glucose concentration a 2 hours in an oral glucose tolerance test	Diastolic blood pressure (mm Hg)	Triceps skin fold thickness (mm)	2-Hour serum insulin (mu U/ml)	Body mass index (weight in kg/(height in m)^2)	Diabetes pedigree function	Age (years)	Class variable
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

	Description	Value
0	Session id	6832
1	Target	Class variable
2	Target type	Binary
3	Original data shape	(537, 9)
4	Transformed data shape	(537, 9)
5	Transformed train set shape	(375, 9)
6	Transformed test set shape	(162, 9)
7	Numeric features	8
8	Preprocess	True
9	Imputation type	simple
10	Numeric imputation	mean
11	Categorical imputation	mode
12	Fold Generator	StratifiedKFold
13	Fold Number	10
14	CPU Jobs	-1
15	Use GPU	False
16	Log Experiment	False
17	Experiment Name	clf-default-name
18	USI	4b29

MODEL SELECTION

	Model	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC	TT (Sec)
et	Extra Trees Classifier	0.7809	0.8484	0.5571	0.7475	0.6352	0.4854	0.4974	
lr	Logistic Regression	0.7782	0.8249	0.5731	0.7318	0.6322	0.4811	0.4943	
ridge	Ridge Classifier	0.7782	0.8293	0.5577	0.7461	0.6245	0.4769	0.4945	
lda	Linear Discriminant Analysis	0.7755	0.8293	0.5654	0.7270	0.6243	0.4730	0.4866	
ada	Ada Boost Classifier	0.7679	0.8161	0.5962	0.6952	0.6380	0.4700	0.4756	
rf	Random Forest Classifier	0.7622	0.8273	0.5566	0.7074	0.6143	0.4484	0.4603	
nb	Naive Bayes	0.7516	0.7971	0.6033	0.6587	0.6272	0.4425	0.4452	
xgboost	Extreme Gradient Boosting	0.7465	0.8033	0.6027	0.6639	0.6235	0.4346	0.4420	
lightgbm	Light Gradient Boosting Machine	0.7439	0.7959	0.5500	0.6711	0.5989	0.4146	0.4230	
gbc	Gradient Boosting Classifier	0.7383	0.8352	0.5808	0.6471	0.6044	0.4120	0.4189	
qda	Quadratic Discriminant Analysis	0.7328	0.8086	0.5264	0.6467	0.5696	0.3833	0.3923	
knn	K Neighbors Classifier	0.7149	0.7484	0.5055	0.6317	0.5449	0.3457	0.3593	
dt	Decision Tree Classifier	0.7147	0.6961	0.6341	0.5916	0.6081	0.3855	0.3890	
dummy	Dummy Classifier	0.6506	0.5000	0.0000	0.0000	0.0000	0.0000	0.0000	
svm	SVM - Linear Kernel	0.6265	0.5448	0.1923	0.4358	0.2210	0.0554	0.0746	

```
ExtraTreesClassifier(bootstrap=False, ccp_alpha=0.0, class_weight=None,
                     criterion='gini', max_depth=None, max_features='sqrt',
                     max_leaf_nodes=None, max_samples=None,
                     min_impurity_decrease=0.0, min_samples_leaf=1,
                     min_samples_split=2, min_weight_fraction_leaf=0.0,
                     monotonic_cst=None, n_estimators=100, n_jobs=-1,
                     oob_score=False, random_state=6832, verbose=0,
                     warm_start=False)
```

Fold	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	0.7632	0.9256	0.3571	1.0000	0.5263	0.4124	0.5096
1	0.8421	0.8892	0.5385	1.0000	0.7000	0.6055	0.6590
2	0.8684	0.9415	0.6923	0.9000	0.7826	0.6906	0.7028
3	0.8158	0.8492	0.6154	0.8000	0.6957	0.5668	0.5768
4	0.7895	0.8646	0.6154	0.7273	0.6667	0.5144	0.5182
5	0.7027	0.6635	0.3846	0.6250	0.4762	0.2847	0.3011
6	0.7568	0.7917	0.3846	0.8333	0.5263	0.3912	0.4442
7	0.7568	0.7436	0.3077	1.0000	0.4706	0.3657	0.4730
8	0.7297	0.8910	0.3846	0.7143	0.5000	0.3369	0.3672
9	0.6757	0.7340	0.3077	0.5714	0.4000	0.2043	0.2227
Mean	0.7701	0.8294	0.4588	0.8171	0.5744	0.4372	0.4775

```
Fitting 10 folds for each of 10 candidates, totalling 100 fits
Original model was better than the tuned model, hence it will be returned. NOTE: The display metrics are for the tuned model (not the original o
ExtraTreesClassifier(bootstrap=False, ccp_alpha=0.0, class_weight=None,
                     criterion='gini', max_depth=None, max_features='sqrt',
                     max_leaf_nodes=None, max_samples=None,
                     min_impurity_decrease=0.0, min_samples_leaf=1,
                     min_samples_split=2, min_weight_fraction_leaf=0.0,
                     monotonic_cst=None, n_estimators=100, n_jobs=-1,
                     oob_score=False, random_state=6832, verbose=0,
                     warm_start=False)
```

```
     prediction_label  prediction_score
668                 0              0.69
324                 0              0.79
624                 0              0.97
690                 0              0.75
473                 1              0.53
..                ...               ...
619                 0              0.51
198                 0              0.64
538                 0              0.57
329                 0              0.78
302                 0              0.76

[231 rows x 11 columns]
1.0
```

1. NEURAL NETWORKS

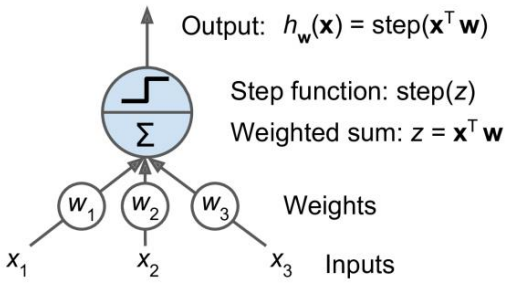
- A **neural network** (also known as an artificial neural network or ANN) is a subset of machine learning models designed to mimic the structure and functionality of a biological brain.
- It consists of interconnected nodes (artificial neurons) organized in layers, with weighted connections that transmit and process data.

2. **Components of a Neural Network**:
    - **Neurons**: These receive inputs and apply activation functions. Neurons are governed by thresholds and biases.
    - **Connections**: Weighted connections regulate information transfer between neurons.
    - **Weights and Biases**: These parameters adjust the strength of connections and neuron activations.
    - **Propagation Functions**: Determine how signals propagate through the network.
    - **Learning Rule**: Algorithms update weights during training to improve model performance.
3. **Inspiration from the Human Brain**:
    - Neural networks draw inspiration from the layered architecture of the human visual cortex.
    - Key similarities include hierarchical structures, local connectivity, and translation invariance.
    - Just like our brains process information, neural networks learn patterns and make decisions.

PERCEPTRON

1. **What Is a Perceptron?**

- A **perceptron** is an artificial neuron, often considered the simplest form of a neural network.
- It serves as a model for a single neuron that can make binary classification decisions.
- Given an input represented by a vector of numbers, the perceptron decides whether it belongs to one class or another.



2. **How It Works**:

- The perceptron computes a weighted sum of its input features.
- If the sum exceeds a certain threshold (bias), it activates and produces an output (usually 1 or 0).
- Mathematically, the output can be expressed as:

$$\text{heaviside}\,(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \qquad \text{sgn}\,(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

Otherwise,
where:

- (wi) represents the weight associated with input feature (xi).

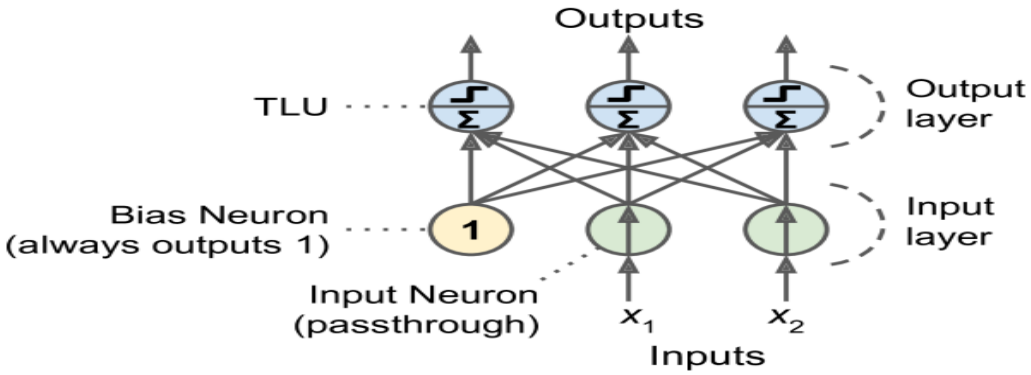
- (b) is the bias term.

1. **Learning Process**:

- During training, the perceptron adjusts its weights based on the error between predicted and actual labels.
- It uses a learning rate to update weights incrementally.
- The process continues until convergence or a predefined number of iterations.

2. **Applications**:

- Perceptron's were historically used for simple linear separations.
- While limited individually, they form the basis for more complex neural networks.



The neuron is shown by:

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{XW} + \mathbf{b})$$

Next weights are:

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta\left(y_j - \hat{y}_j\right)x_i$$

CODE

```
import numpy as np
import pandas as pd

data = pd.read_csv("C:\\Users\\soban\\OneDrive - National University of Sciences & Technology\\Desktop\\Pandas\\pre.csv")
X = data.iloc[:,:-1].values
Y = data.iloc[:, -1].values
Y = Y.reshape(Y.size, 1)

def sigmoid(x):
  return 1/(1 + np.exp(-x))
```

```
def sigder(x):
  return x * (1 - x)

np.random.seed(1)
synaptic_weights = 2 * np.random.random((3, 1)) - 1

print("Synaptic weights: ", synaptic_weights)

for iterations in range(100000):
  input_layer = X
  output = sigmoid(np.dot(X, synaptic_weights))
  error = output - Y
  adjustments = error * sigder(Y)
  synaptic_weights += np.dot(X.T, adjustments)

print("Synaptic Weights: " , synaptic_weights)
print("Output after weights: ", output)
```

OUTPUT

```
Synaptic weights:   [[-0.16595599]
 [ 0.44064899]
 [-0.99977125]]
Synaptic Weights:   [[-0.16595599]
 [ 0.44064899]
 [-0.99977125]]
Output after weights:   [[0.5       ]
 [0.45860596]
 [0.5       ]
 [0.56824466]
 [0.60841366]
 [0.56824466]
 [0.36375058]
 [0.23762817]
 [0.23762817]
 [0.2689864 ]
 [0.2689864 ]
 [0.45860596]
 [0.36375058]
 [0.56824466]
 [0.60841366]
 [0.60841366]
 [0.60841366]
 [0.5       ]
 [0.5       ]
 [0.5       ]
 [0.23762817]
 [0.23762817]
 [0.45860596]
 [0.23762817]]
```

CODE

```
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron
from sklearn.model_selection import train_test_split
from sklearn.metrics import *
import warnings
```

```
warnings.filterwarnings('ignore')

iris = load_iris()
X = iris.data

Y = (iris.target == 0).astype(int)
Y = Y.reshape(Y.size, 1)

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.2, random_state = 42)
per = Perceptron(random_state = 42)
per.fit(X_train, Y_train)

Y_pred = per.predict(X_test)
print(accuracy_score(Y_test, Y_pred))
```

OUTPUT

```
1.0
```

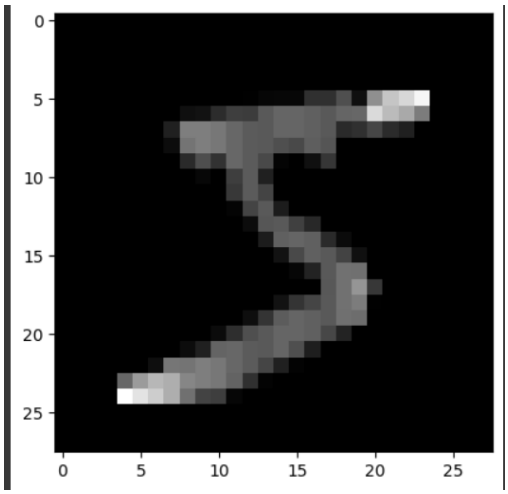
MULTILAYER PERCEPTRON

1. **What Is a Multilayer Perceptron (MLP)?**:
    - An MLP is a type of **feedforward neural network**.
    - It consists of fully connected neurons organized in layers: input layer, hidden layers, and an output layer.
    - MLPs are notable for their ability to learn complex patterns in data, even when the data is not linearly separable.
2. **Key Characteristics**:
    - **Architecture**: Input layer receives data, hidden layers process it, and the output layer produces predictions.
    - **Activation Functions**: Nonlinear activation functions (e.g., sigmoid, ReLU) allow MLPs to model complex relationships.
    - **Training**: Backpropagation is used to adjust weights during training.
3. **Applications**:
    - **Classification**: MLPs can classify data into different classes (e.g., image recognition, sentiment analysis).
    - **Regression**: They can predict continuous values (e.g., stock prices, house prices).

CODE

```
import tensorflow as tf
from tensorflow import keras
from keras.layers import *
```

```
from keras.datasets import mnist

(X, Y), (X_test, Y_test) = mnist.load_data()
import matplotlib.pyplot as plt
plt.imshow(X[0], cmap = 'gray')
```




```
X = keras.utils.normalize(X, axis = 1)
X_test = keras.utils.normalize(X_test, axis = 1)
model = tf.keras.Sequential()

model.add(Flatten(input_shape = [28, 28]))
model.add(Dense(300, activation = 'relu'))
model.add(Dense(100, activation = 'relu'))
model.add(Dense(10, activation = 'softmax'))
model.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_2 (Flatten)         (None, 784)               0

 dense_4 (Dense)             (None, 300)               235500

 dense_5 (Dense)             (None, 100)               30100

 dense_6 (Dense)             (None, 10)                1010

=================================================================
Total params: 266610 (1.02 MB)
Trainable params: 266610 (1.02 MB)
Non-trainable params: 0 (0.00 Byte)
```

```
X_val, X_train = X[:5000], X[5000:]
Y_val, Y_train = Y[:5000], Y[5000:]
model.get_weights()
```

```
[array([[ 0.0130581 ,  0.03443112,  0.03933656, ..., -0.03892207,
          0.00838364,  0.06539334],
        [ 0.03131286, -0.00276475,  0.02364051, ..., -0.02803748,
         -0.03218867,  0.04096994],
        [ 0.05442974, -0.06618492,  0.04353184, ...,  0.03520033,
          0.01009665,  0.02423865],
        ...,
        [-0.03414622, -0.03236447, -0.02776096, ...,  0.07172225,
          0.03424867,  0.0504488 ],
        [ 0.07184358,  0.06509243,  0.05548   , ..., -0.0150088 ,
         -0.01074377,  0.03366581],
        [ 0.01011691,  0.00447849, -0.06427854, ..., -0.06457159,
          0.0194876 ,  0.02097178]], dtype=float32),
 array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32),
```

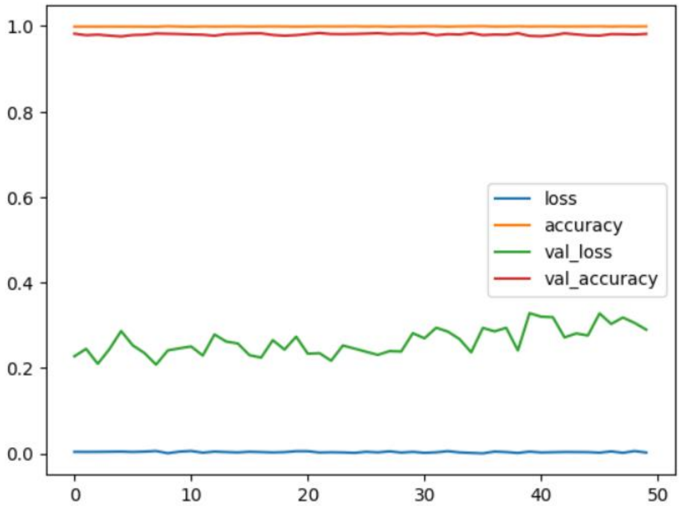
model.compile(loss ="sparse_categorical_crossentropy", optimizer = 'adam', metrics = ["accuracy"])

history = model.fit(X_train, Y_train, epochs = 50, validation_data = (X_val, Y_val))

```
Epoch 42/50
1719/1719 [==============================] - 14s 8ms/step - loss: 0.0032 - accuracy: 0.9993 - val_loss: 0.3190 - val_accuracy: 0.9786
Epoch 43/50
1719/1719 [==============================] - 20s 12ms/step - loss: 0.0038 - accuracy: 0.9992 - val_loss: 0.2717 - val_accuracy: 0.9830
Epoch 44/50
1719/1719 [==============================] - 18s 10ms/step - loss: 0.0036 - accuracy: 0.9993 - val_loss: 0.2811 - val_accuracy: 0.9806
Epoch 45/50
1719/1719 [==============================] - 18s 10ms/step - loss: 0.0034 - accuracy: 0.9993 - val_loss: 0.2762 - val_accuracy: 0.9782
Epoch 46/50
1719/1719 [==============================] - 17s 10ms/step - loss: 0.0021 - accuracy: 0.9996 - val_loss: 0.3279 - val_accuracy: 0.9776
Epoch 47/50
1719/1719 [==============================] - 18s 10ms/step - loss: 0.0052 - accuracy: 0.9990 - val_loss: 0.3032 - val_accuracy: 0.9812
Epoch 48/50
1719/1719 [==============================] - 14s 8ms/step - loss: 0.0018 - accuracy: 0.9995 - val_loss: 0.3184 - val_accuracy: 0.9810
Epoch 49/50
1719/1719 [==============================] - 12s 7ms/step - loss: 0.0060 - accuracy: 0.9991 - val_loss: 0.3057 - val_accuracy: 0.9802
Epoch 50/50
1719/1719 [==============================] - 14s 8ms/step - loss: 0.0024 - accuracy: 0.9995 - val_loss: 0.2899 - val_accuracy: 0.9818
```

import pandas as pd

pd.DataFrame(history.history).plot()



model.evaluate(X_test, Y_test)

```
313/313 [==============================] - 1s 2ms/step - loss: 0.3105 - accuracy: 0.9802
[0.310527503490448, 0.9801999926567078]
```

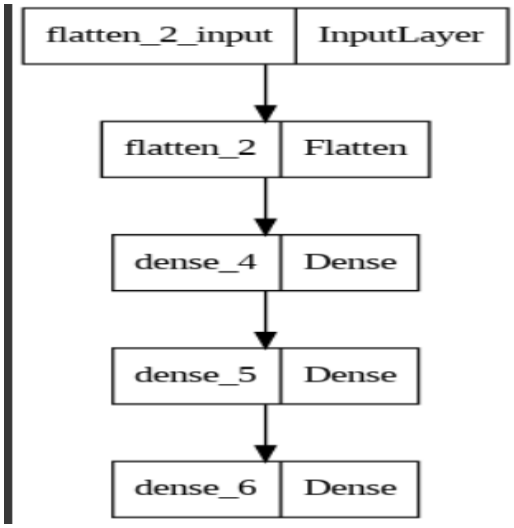
```
Y_proba = model.predict(X_test[:10])
print(Y_proba.round(2))
```

```
[[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]
```

```
import numpy as np
Y_pred = np.argmax(Y_proba, axis = 1)
print(Y_pred)
```

```
[7 2 1 0 4 1 4 9 5 9]
```

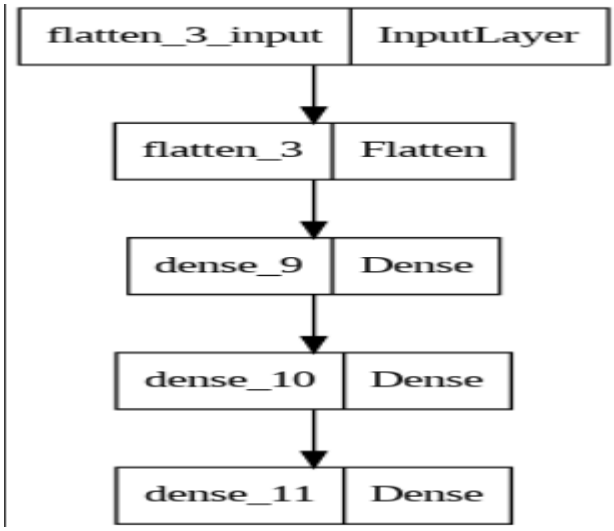
```
class_names = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
np.array(class_names)[Y_pred]
keras.utils.plot_model(model)
```



USING FASHION_MNIST DATASET

```
import tensorflow as tf
from tensorflow import keras
from keras.datasets import fashion_mnist
(X_t, Y_t), (X_test, Y_test) = fashion_mnist.load_data()

model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape = [28, 28]))
model.add(keras.layers.Dense(300, activation = 'relu'))
model.add(keras.layers.Dense(100, activation= 'relu'))
model.add(keras.layers.Dense(10, activation='softmax'))
```

keras.utils.plot_model(model)



X_t = keras.utils.normalize(X_t, axis = 1)
X_test = keras.utils.normalize(X_test, axis = 1)
print(X_t.shape)

```
(60000, 28, 28)
```

model.layers

X_valid, X_train = X_t[:5000], X_t[5000:]
Y_valid, Y_train = Y_t[:5000], Y_t[5000:]

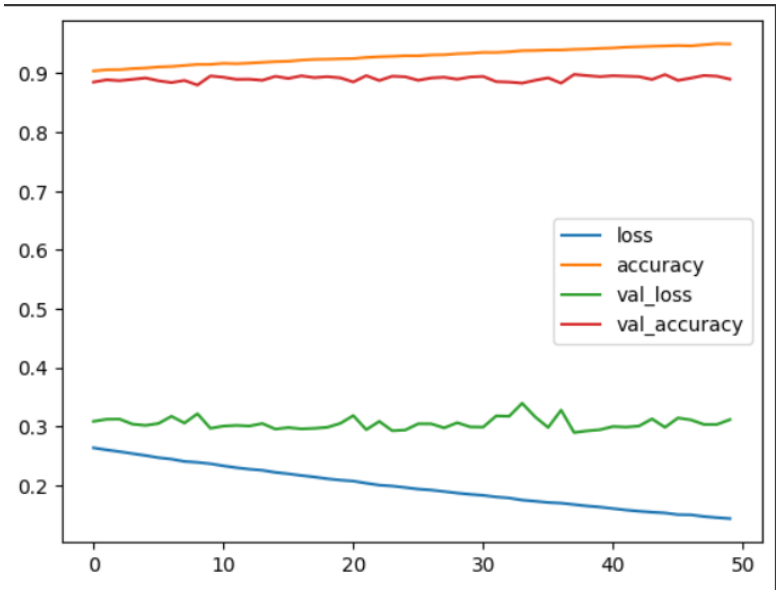
print(X_train.shape)
print(X_valid.shape)

```
(55000, 28, 28)
(5000, 28, 28)
```

weights = model.get_weights()
model.compile(loss = "sparse_categorical_crossentropy", optimizer = "sgd", metrics = ["accuracy"])
history = model.fit(X_train, Y_train, epochs = 50, validation_data = (X_valid, Y_valid))

```
Epoch 42/50
1719/1719 [==============================] - 10s 6ms/step - loss: 0.1578 - accuracy: 0.9446 - val_loss: 0.2988 - val_accuracy: 0.8952
Epoch 43/50
1719/1719 [==============================] - 11s 6ms/step - loss: 0.1558 - accuracy: 0.9452 - val_loss: 0.3007 - val_accuracy: 0.8944
Epoch 44/50
1719/1719 [==============================] - 9s 5ms/step - loss: 0.1542 - accuracy: 0.9460 - val_loss: 0.3127 - val_accuracy: 0.8896
Epoch 45/50
1719/1719 [==============================] - 10s 6ms/step - loss: 0.1529 - accuracy: 0.9466 - val_loss: 0.2984 - val_accuracy: 0.8980
Epoch 46/50
1719/1719 [==============================] - 9s 5ms/step - loss: 0.1500 - accuracy: 0.9472 - val_loss: 0.3144 - val_accuracy: 0.8880
Epoch 47/50
1719/1719 [==============================] - 9s 5ms/step - loss: 0.1497 - accuracy: 0.9467 - val_loss: 0.3110 - val_accuracy: 0.8920
Epoch 48/50
1719/1719 [==============================] - 11s 6ms/step - loss: 0.1468 - accuracy: 0.9487 - val_loss: 0.3033 - val_accuracy: 0.8964
Epoch 49/50
1719/1719 [==============================] - 10s 6ms/step - loss: 0.1449 - accuracy: 0.9505 - val_loss: 0.3033 - val_accuracy: 0.8952
Epoch 50/50
1719/1719 [==============================] - 8s 5ms/step - loss: 0.1433 - accuracy: 0.9500 - val_loss: 0.3115 - val_accuracy: 0.8902
```

import pandas as pd
pd.DataFrame(history.history).plot()



model.evaluate(X_test, Y_test)

```
313/313 [==============================] - 1s 3ms/step - loss: 0.3434 - accuracy: 0.8850
[0.343433141708374, 0.8849999904632568]
```

Y_proba = model.predict(X_test[:10])

print(Y_proba.round(2))

```
[[0.   0.   0.   0.   0.   0.   0.   0.   1.   ]
 [0.   0.   0.99 0.   0.01 0.   0.   0.   0.   ]
 [0.   1.   0.   0.   0.   0.   0.   0.   0.   ]
 [0.   1.   0.   0.   0.   0.   0.   0.   0.   ]
 [0.29 0.   0.   0.   0.01 0.   0.7  0.   0.   ]
 [0.   1.   0.   0.   0.   0.   0.   0.   0.   ]
 [0.   0.   0.01 0.   0.99 0.   0.   0.   0.   ]
 [0.   0.   0.01 0.   0.   0.   0.99 0.   0.   ]
 [0.   0.   0.   0.   0.   1.   0.   0.   0.   ]
 [0.   0.   0.   0.   0.   0.   0.   1.   0.   ]]
```

class_names = ["T-shirt/top", "Trouser", "Pullover", "Shirt", "Coat",
"Sandal", "Sneaker", "Bag", "Ankle boot", "Boot"]

print(class_names[Y_train[0]])

```
Coat
```

import numpy as np

Y_pred = np.argmax(Y_proba, axis = 1)

print (Y_pred)

```
[9 2 1 1 6 1 4 6 5 7]
```

np.array(class_names)[Y_pred]

```
array(['Boot', 'Pullover', 'Trouser', 'Trouser', 'Sneaker', 'Trouser',
       'Coat', 'Sneaker', 'Sandal', 'Bag'], dtype='<U11')
```

import matplotlib.pyplot as plt

some = X_t[3]

plt.imshow(some, cmap = 'binary', interpolation = 'bilinear')

plt.axis("off")

REGRESSION MLPS

1. **What Are Multilayer Perceptron's (MLPs)?**
    - MLPs break the restriction of linear separability.
    - They use a more robust and complex architecture to learn regression and classification models for challenging datasets.
    - Unlike simple perceptron, MLPs can handle non-linear relationships.
2. **Use Cases for MLPs**:
    - **Regression**: MLPs predict real-valued quantities given input features.
    - **Classification**: They assign class labels to inputs.
    - Often used with tabular data (e.g., CSV files or spreadsheets)

CODE

```
import pandas as pd
import numpy as np
data = pd.read_csv('/content/Housing.csv')
data.head()
```

	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterheating	airconditioning	parking	prefarea	furnishingstatus
0	13300000	7420	4	2	3	yes	no	no	no	yes	2	yes	furnished
1	12250000	8960	4	4	4	yes	no	no	no	yes	3	no	furnished
2	12250000	9960	3	2	2	yes	no	yes	no	no	2	yes	semi-furnished
3	12215000	7500	4	2	2	yes	no	yes	no	yes	3	yes	furnished
4	11410000	7420	4	1	2	yes	yes	yes	no	yes	2	no	furnished

```
data.mainroad = data.mainroad.map({'yes': 1, 'no': 0})
data.guestroom = data.guestroom.map({'yes': 1, 'no': 0})
data.basement = data.basement.map({'yes': 1, 'no': 0})
data.hotwaterheating = data.hotwaterheating.map({'yes': 1, 'no': 0})
data.airconditioning = data.airconditioning.map({'yes': 1, 'no': 0})
data.prefarea = data.prefarea.map({'yes': 1, 'no': 0})
data.furnishingstatus = data.furnishingstatus.map({'furnished': 2, 'semi-furnished': 1,
'unfurnished': 0})
```

```
data.head()
data.tail()
```

	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterheating	airconditioning	parking	prefarea	furnishingstatus
540	1820000	3000	2	1	1	1	0	1	0	0	2	0	0
541	1767150	2400	3	1	1	0	0	0	0	0	0	0	1
542	1750000	3620	2	1	1	1	0	0	0	0	0	0	0
543	1750000	2910	3	1	1	0	0	0	0	0	0	0	2
544	1750000	3850	3	1	2	1	0	0	0	0	0	0	0

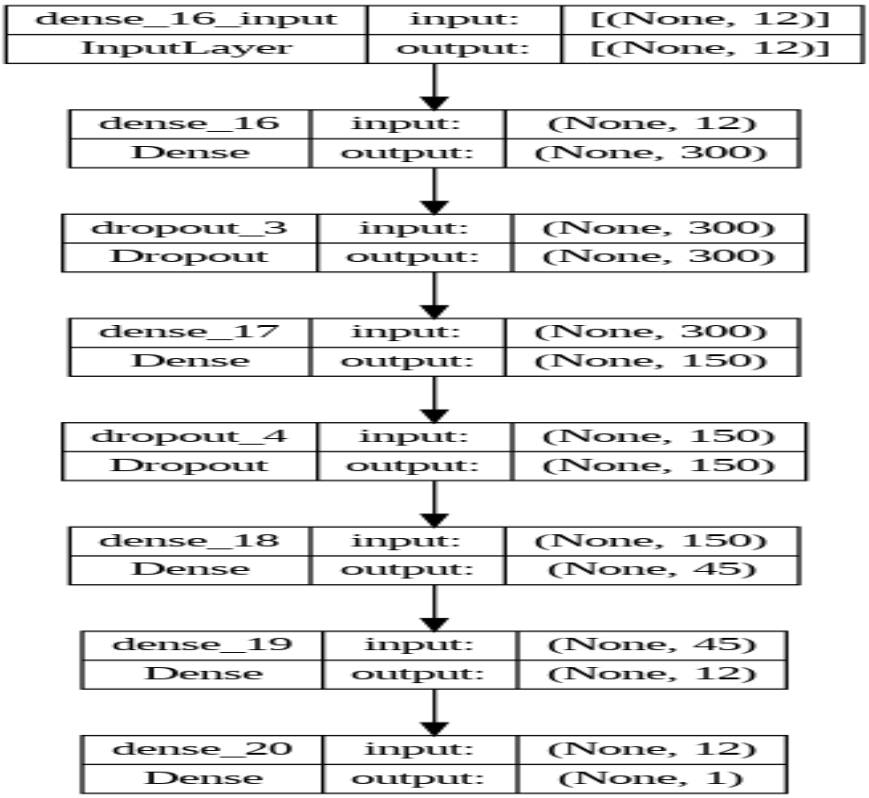
```
Y = data.price
Y = Y.values
Y = Y.reshape(Y.size, 1)
X = data.drop(['price'], axis = 1)
X = X.values
print(X.shape)
print(Y.shape)
```

```
(545, 12)
(545, 1)
```

```
from sklearn.model_selection import train_test_split
X_train, X_temp, Y_train, Y_temp = train_test_split(X, Y, test_size = 0.4, random_state =
42)
X_valid, X_test, Y_valid, Y_test = train_test_split(X_temp, Y_temp, test_size = 0.4,
random_state = 42)
import tensorflow as tf
from tensorflow import keras
from keras.models import Sequential
from keras.layers import *
from keras import regularizers
model = Sequential()
model.add(Dense(300, activation = 'leaky_relu', input_shape =
(X.shape[1],),kernel_regularizer = keras.regularizers.l2(0.01)))
model.add(Dropout(0.5))
model.add(Dense(150, activation = 'leaky_relu', kernel_regularizer =
keras.regularizers.l2(0.01)))
model.add(Dropout(0.5))
model.add(Dense(45, activation = 'leaky_relu', kernel_regularizer =
keras.regularizers.l2(0.01)))
model.add(Dense(12, activation = 'relu'))
model.add(Dense(1))
model.summary()
```

```
Model: "sequential_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_16 (Dense)            (None, 300)               3900

 dropout_3 (Dropout)         (None, 300)               0

 dense_17 (Dense)            (None, 150)               45150

 dropout_4 (Dropout)         (None, 150)               0

 dense_18 (Dense)            (None, 45)                6795

 dense_19 (Dense)            (None, 12)                552

 dense_20 (Dense)            (None, 1)                 13

=================================================================
Total params: 56410 (220.35 KB)
Trainable params: 56410 (220.35 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
from sklearn.preprocessing import StandardScaler
Xscaler = StandardScaler()
Yscaler = StandardScaler()
X_train = Xscaler.fit_transform(X_train)
X_test = Xscaler.transform(X_test)
X_valid = Xscaler.transform(X_valid)
Y_train = Yscaler.fit_transform(Y_train)
Y_test = Yscaler.transform(Y_test)
Y_valid = Yscaler.transform(Y_valid)
keras.utils.plot_model(model, show_shapes = True)
```



```
model.compile(loss = 'mean_squared_error', optimizer = 'adam', metrics =
['mean_squared_error'])
history = model.fit(X_train, Y_train, epochs = 100, validation_data = (X_valid, Y_valid))
```

```
Epoch 95/100
11/11 [==============================] - 0s 12ms/step - loss: 0.3514 - mean_squared_error: 0.2241 - val_loss: 0.6161 - val_mean_squared_error: 0.4897
Epoch 96/100
11/11 [==============================] - 0s 10ms/step - loss: 0.3656 - mean_squared_error: 0.2409 - val_loss: 0.6398 - val_mean_squared_error: 0.5171
Epoch 97/100
11/11 [==============================] - 0s 12ms/step - loss: 0.3243 - mean_squared_error: 0.2015 - val_loss: 0.6224 - val_mean_squared_error: 0.5002
Epoch 98/100
11/11 [==============================] - 0s 12ms/step - loss: 0.3529 - mean_squared_error: 0.2319 - val_loss: 0.6616 - val_mean_squared_error: 0.5421
Epoch 99/100
11/11 [==============================] - 0s 13ms/step - loss: 0.3466 - mean_squared_error: 0.2281 - val_loss: 0.6451 - val_mean_squared_error: 0.5264
Epoch 100/100
11/11 [==============================] - 0s 10ms/step - loss: 0.3319 - mean_squared_error: 0.2124 - val_loss: 0.6147 - val_mean_squared_error: 0.4959
```

```
pd.DataFrame(history.history).plot()
test_loss, test_mae = model.evaluate(X_test, Y_test)
Y_proba = model.predict(X_test)
print(Y_proba)
print(test_mae)
```

```
[-0.08586153]
[ 1.0133729 ]
[ 0.30049896]
[ 0.01353296]
[-0.63888854]
[-0.5198868 ]
[ 0.65720487]
[ 0.35652363]
[-0.47991788]
[-0.27156413]
[-0.6673649 ]
[-0.9786795 ]
[ 0.0888699 ]
[-0.22278014]
[-0.9251282 ]
[-0.9462719 ]
[ 0.09946349]
[ 0.25591356]
[-0.8968935 ]
[ 0.38566393]
[ 1.4673477 ]
[ 1.7812635 ]
```

```
from sklearn.metrics import r2_score
r2_score (Y_test, Y_proba)
```

```
0.6115105044823269
```

DATA LOADER

1. **Batch Size**:
    - The **batch size** refers to the number of data samples processed together in each iteration during training.
    - When you create a data loader, you specify the batch size, and the loader divides the dataset into batches.
    - Larger batch sizes can lead to faster training but require more memory.
    - Smaller batch sizes provide more frequent updates to model weights but may slow down training.
    - It's essential to find a balance based on your available resources and model requirements.
2. **Data Loader**:

- A **data loader** is responsible for managing data access during training.
- It wraps an iterable around a dataset (such as images, text, or other features).
- The data loader handles operations like shuffling, data augmentation, and batching.
- When you iterate over a data loader, it provides batches of samples for model training.
  Steps per Epoch = Len (Data Loader) = Total Number of Examples/ Batch Size

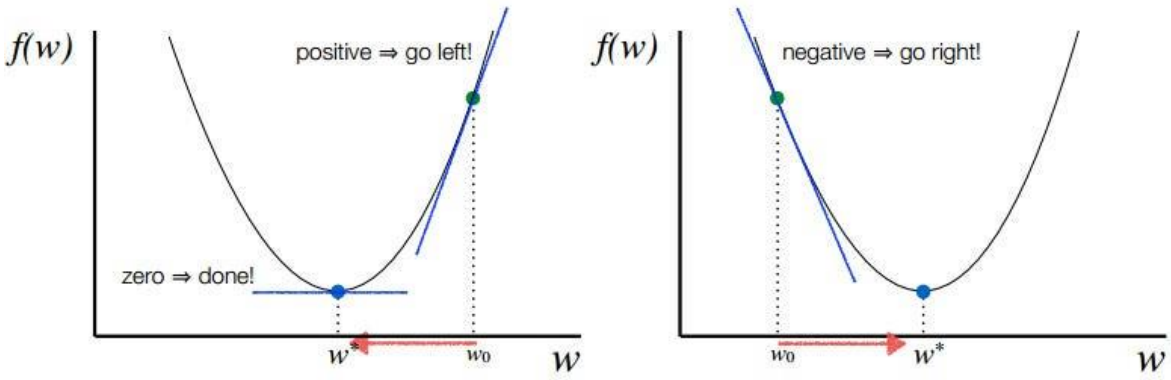
RANDOM STATE

In machine learning and data analysis, the term "random state" refers to a parameter used to initialize the random number generator. It ensures that the same sequence of random numbers is generated each time you run an algorithm or perform a specific operation. Here are some key points:

1. **Reproducibility**:
   - Setting a fixed random state allows you to reproduce the same results across different runs.
   - When you initialize a model (e.g., a neural network, decision tree, or k-means clustering), specifying the random state ensures consistency.
2. **Randomness and Variability**:
   - Many algorithms involve randomness (e.g., initializing weights in neural networks, shuffling data).
   - By setting the random state, you control this randomness, making experiments more predictable.

OPTIMIZATIONS IN DEEP LEARNING
GRADIENT DESCENT

This is one of the oldest and the most common optimizer used in neural networks, best for the cases where the data is arranged in a way that it possesses a convex optimization problem.



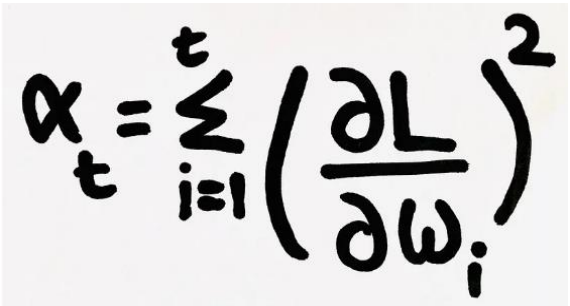
It will try to find the least cost function value by updating the weights of your learning algorithm and will come up with the best-suited parameter values corresponding to the Global Minima.
This is done by moving down the hill with a negative slope, increasing the older weight, and positive slope reducing the older weight.

STOCHASTIC GRADIENT DESCENT

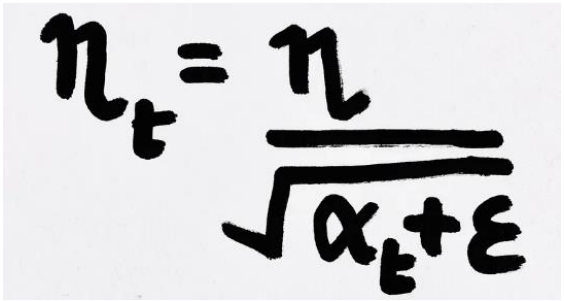
This is another variant of the Gradient Descent optimizer with an additional capability of working with the data with a non-convex optimization problem. The problem with such data is that the cost function results to rest at the local minima which are not suitable for your learning algorithm.
Rather than going for batch processing, this optimizer focuses on performing one update at a time. It is therefore usually much faster, also the cost function minimizes after each iteration (EPOCH). It performs frequent updates with a high variance that causes the objective function (cost function) to fluctuate heavily. Due to which it makes the gradient to jump to a potential Global Minima.

ADAGRAD

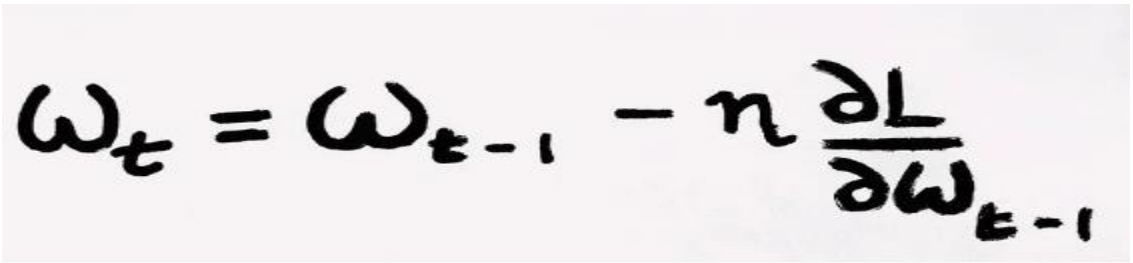
This is the Adaptive Gradient optimization algorithm, where the learning rate plays an important role in determining the updated parameter values. Unlike Stochastic Gradient descent, this optimizer uses a different learning rate for each iteration (EPOCH) rather than using the same learning rate for determining all the parameters.
Thus, it performs smaller updates (lower learning rates) for the weights corresponding to the high-frequency features and bigger updates (higher learning rates) for the weights corresponding to the low-frequency features, which in turn helps in better performance with higher accuracy. Adagrad is well-suited for dealing with sparse data.
So, at each iteration, first the alpha at time t will be calculated and as the iterations increase the value of t increases, and thus alpha t will start increasing.



$$\alpha_t = \sum_{i=1}^{t} \left(\frac{\partial L}{\partial \omega_i}\right)^2$$



$$\eta_t = \frac{\eta}{\sqrt{\alpha_t + \varepsilon}}$$



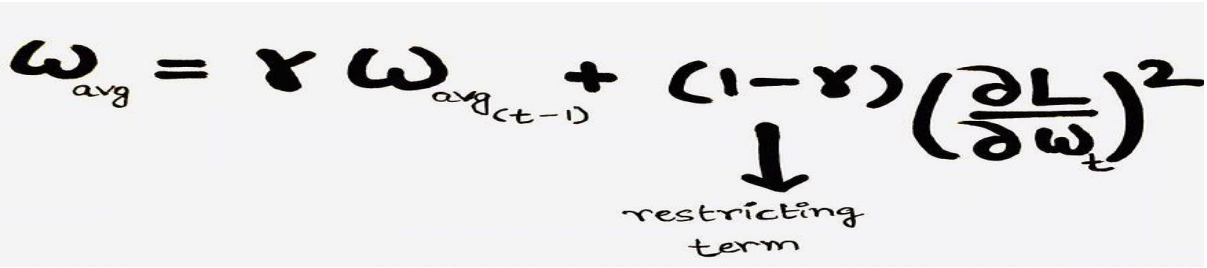
$$\omega_t = \omega_{t-1} - \eta \frac{\partial L}{\partial \omega_{t-1}}$$

ADADELTA

This is an extension of the Adaptive Gradient optimizer, taking care of its aggressive nature of reducing the learning rate infinitesimally. Here instead of using the previous squared gradients, the sum of gradients is defined as a reducing weighted average of all past squared gradients (weighted averages) this restricts the learning rate to reduce to a very small value.

The formula for the new weight remains the same as in Adagrad. However, there are some changes in determining the learning rate at time step t for each iteration.

At each iteration, first the weighted average is calculated. Where we have the restricting term(gamma = 0.95) which helps in avoiding the problem of *Vanishing Gradient*.



RMSPROP

Both the optimizing algorithms, RMSprop (Root Mean Square Propagation) and Adadelta were developed around the same time, for the same purpose to resolve Adagrad's problem of destructive learning rates. However, both use the same method which utilizes an Exponential Weighted Average to determine the learning rate at time t for each iteration.

RMSprop is an adaptive learning rate method proposed by Geoffrey Hinton, which appropriately divides the learning rate by an exponentially weighted average of squared gradients. It is suggested to set gamma at 0.95, as it has been showing good results for most of the cases.

ADAM

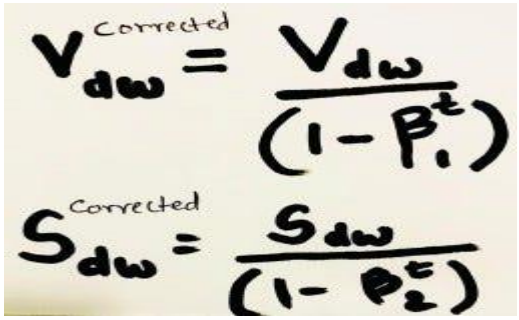
This is the Adaptive Moment Estimation algorithm which also works on the method of computing adaptive learning rates for each parameter at every iteration. It uses a combination of *Gradient Descent with Momentum* and RMSprop to determine the parameter values.

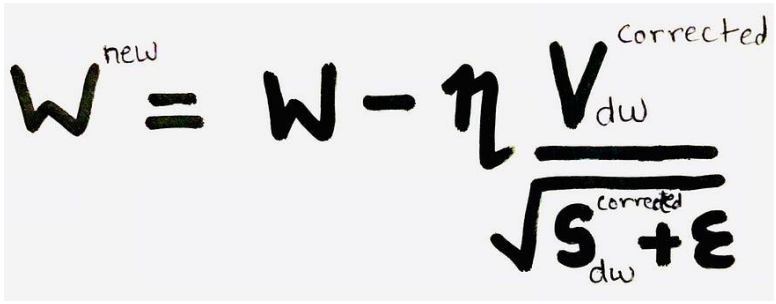
When introducing the algorithm, there was a list of attractive benefits of using Adam on non-convex optimization problems which made it the most commonly used optimizer.

It comes with several advantages combining the benefits of both Gradient with Momentum and RMSProp like low memory requirements, appropriate for non-stationary objectives, works best with large data and parameters with efficient computation. This works using the same methodology of adaptive learning rate in addition to storing an exponential weighted average of the past squared derivative of loss with respect to the weight at time t-1.
It comes with several parameters, which are β1, β2, and ε (epsilon). Where β1 and β2 are the initial restricting parameters for Momentum and RMSprop respectively. Here, β1 corresponds to the first moment and β2 corresponds to the second moment.

$$VdW = \beta1 \times VdW + (1-\beta1) \times dW \text{ — — GD with Momentum (1st)}$$

$$SdW = \beta2 \times SdW + (1-\beta2) \times dW^2 \text{ — — RMSprop (2nd)}$$





CROSS VALIDATION

Cross-validation is a technique used in machine learning to evaluate the performance of a model on unseen data. Here's how it works:

1. **Purpose**:
    - Cross-validation assesses how well a model generalizes to independent data (not used during training).
    - It helps detect issues like overfitting or selection bias.
2. **Process**:
    - Divide the available data into multiple folds or subsets.
    - Use one-fold as a validation set and train the model on the remaining folds.
    - Repeat this process, rotating the validation set, to get a robust estimate of model performance.
3. **Benefits**:

- Provides a more accurate assessment of how the model will perform on new data.
- Helps choose the best model or tune hyperparameters.

GETTING STARTED WITH KERAS & TENSORFLOW

Keras, a high-level, user-friendly API for building and training neural networks. Here are the key points:

1. **What Is Keras?**
   - Keras is an open-source library built in Python that runs on top of TensorFlow.
   - It provides a user-friendly interface for creating, training, and deploying deep learning models.
   - Designed for fast experimentation and iteration, Keras simplifies working with neural networks.
2. **Features**:
   - **Modularity**: Keras allows you to build models by stacking layers, making it easy to create complex architectures.
   - **Minimal Code**: You can achieve powerful results with minimal code.
   - **Extensibility**: Keras is easy to extend and customize for specific use cases.
   - **Focus on Deep Learning**: It covers every step of the machine learning workflow, from data processing to deployment.

TensorFlow is a free and open-source software library developed by the Google Brain team. It's widely used for machine learning and artificial intelligence tasks. Here are the key points:

1. **Deep Learning Focus**:
   - TensorFlow specializes in training and inference of deep neural networks.
   - It's particularly powerful for building and deploying complex models.
2. **Applications**:
   - You can use TensorFlow for various AI tasks:
     - **Image and Speech Recognition**: Identify objects in images or transcribe speech.
     - **Natural Language Processing (NLP)**: Analyse and generate human language.
     - **Predictive Modelling**: Make predictions based on data.
3. **Ecosystem**:
   - TensorFlow provides an extensive ecosystem:
     - **TensorFlow.js**: Create machine learning models in JavaScript for web applications.

- **TensorFlow Lite**: Deploy models on mobile devices, edge devices, and microcontrollers.
- **TensorFlow Serving**: Run ML models at scale in production environments.

ACTIVATION FUNCTIONS IN DEEP LEARNING

Activation functions are mathematical operations applied to the outputs of individual neurons in a neural network. These functions introduce nonlinearity, allowing the network to capture intricate patterns and make nonlinear transformations from input to output. Without activation functions, a neural network would be limited to linear mappings, rendering it incapable of representing and learning complex relationships in data.

Types of Activation Functions

The universe of activation functions is rich and diverse, each type possessing unique characteristics and applications.

1. Sigmoid Activation

The sigmoid activation function, defined as ***f(x) = 1 / (1 + exp(-x))***, compresses input values into a range between 0 and 1. This function is often employed in the output layer of binary classification problems, as it yields outputs that resemble probabilities.

```
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

2. Hyperbolic Tangent (Tanh) Activation

Tanh activation, characterized by ***f(x) = (exp(x) - exp(-x)) / (exp(x) + exp(-x))***, maps input values to the range [-1, 1]. It is frequently used within hidden layers of neural networks and can help alleviate the vanishing gradient problem when compared to sigmoid.

3. Rectified Linear Unit (ReLU) Activation

ReLU, defined as ***f(x) = max (0, x)***, stands as one of the most popular activation functions. It introduces sparsity by setting negative values to zero, making it computationally efficient and well-suited for deep networks. However, it is not without its shortcomings, such as the dying ReLU problem.

4. Leaky ReLU Activation

Leaky ReLU, an enhancement over standard ReLU, allows a small gradient for negative values. It is defined as ***f(x) = max (alpha * x, x)***, where alpha is a small positive constant. Leaky ReLU is a useful choice when confronted with the dying ReLU problem.

5. Exponential Linear Unit (ELU) Activation

ELU, characterized by ***f(x) = x if x > 0 else alpha * (exp(x) - 1)***, combines the strengths of ReLU and Leaky ReLU while mitigating the dying ReLU problem. It particularly shines when the network needs to capture both positive and negative values.

6. Swish Activation

Swish, proposed by Google's research team, is defined as ***f(x) = x * sigmoid(x)***. It combines the advantages of ReLU's computational efficiency with a smoother, non-monotonic behaviour, potentially leading to improved performance.

7. Parametric ReLU (PReLU) Activation

Parametric ReLU, an extension of Leaky ReLU, allows the learning of the alpha parameter instead of setting it manually. It is expressed as ***f(x) = max (alpha * x, x)***, with alpha being a learnable parameter.

8. Randomized Leaky ReLU (RReLU) Activation

Randomized Leaky ReLU (RReLU) is a variation of Leaky ReLU where the ***alpha*** parameter is randomly selected from a uniform distribution during training. This randomness can act as a regularization technique.

9. Parametric Exponential Linear Unit (PELU) Activation

Parametric Exponential Linear Unit (PELU) extends ELU by allowing the learning of the alpha parameter. It is defined as ***f(x) = x if x > 0 else alpha * (exp(x) - 1)*** with alpha being a learnable parameter.

10. Softmax Activation

The softmax activation function is primarily used in the output layer of multi-class classification problems. It converts a vector of real numbers into a probability distribution over multiple class.

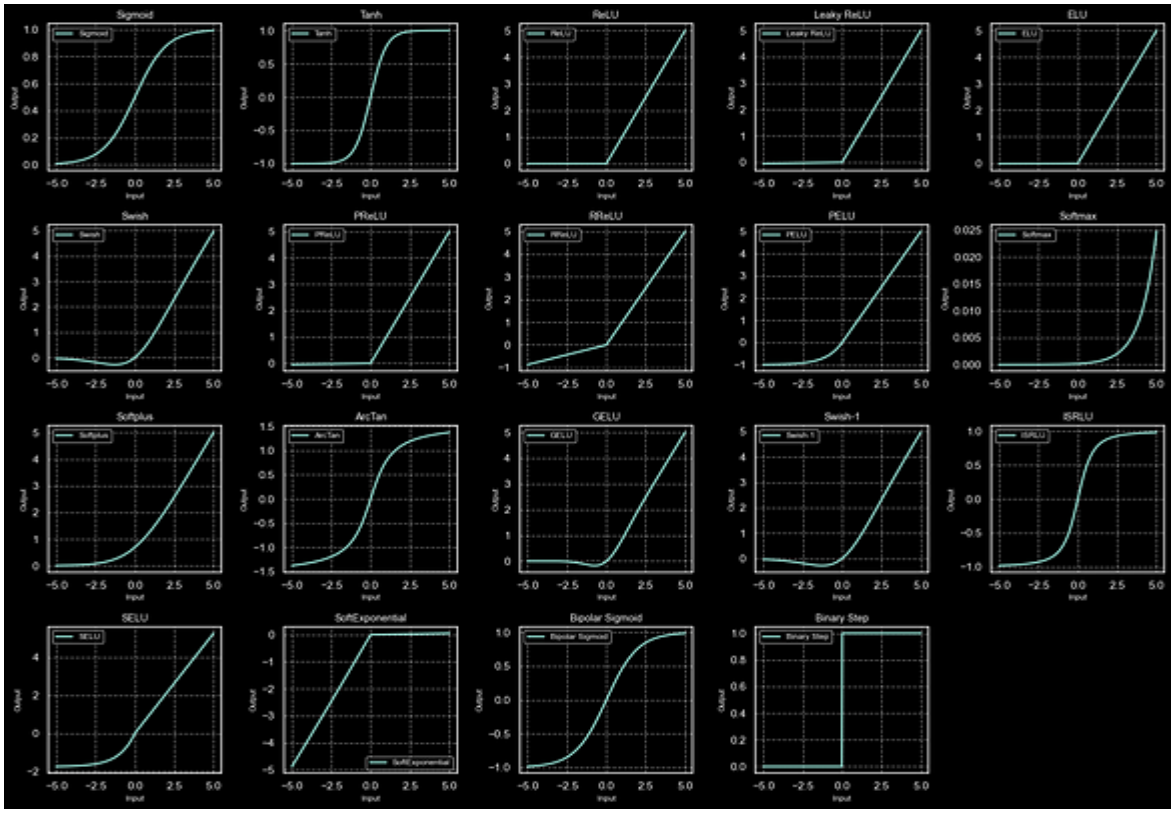
```
import numpy as np

def softmax(x):
    exp_x = np.exp(x - np.max(x))  # To prevent numerical instability
    return exp_x / np.sum(exp_x, axis=0)
```

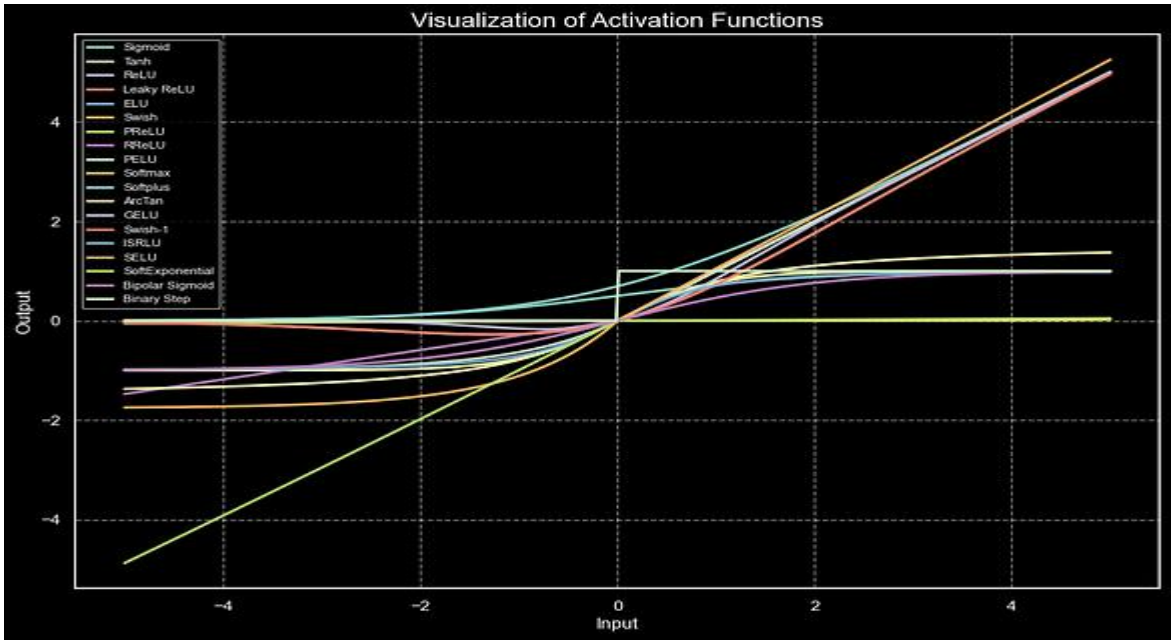
11. Gaussian Error Linear Unit (GELU) Activation

The GELU activation function, popularized by its use in Transformers, is defined as a smooth approximation of the rectifier function with Gaussian noise. Its formula is complex ***GELU(x) = 0.5 * x * (1 + tanh(sqrt(2/pi) * (x + 0.044715 * x^3)))***

12. Scaled Exponential Linear Unit (SELU) Activation

SELU is an activation function designed to ensure that neural networks automatically normalize their activations, ***SELU(x) = λ * {x, if x > 0; α * (e^x — 1), if x <= 0 }.*** It is defined as a piecewise function.





Choosing the Right Activation Function

Selecting the appropriate activation function is a critical decision in the design of neural networks. The choice should be based on several factors, including the nature of your task, the architecture of your network, and the characteristics of your data. Below, we provide a comprehensive overview of different activation functions and their recommended use cases:

1. Sigmoid: Sigmoid activation is well-suited for binary classification problems where you need outputs that resemble probabilities. It squashes input values into the range between 0 and 1, making it ideal for problems with two distinct classes.

2. Tanh (Hyperbolic Tangent): Tanh is an excellent choice for hidden layers, especially when your input data is centered around zero (mean-zero data). It maps input values to the range [-1, 1], which helps mitigate the vanishing gradient problem and is often preferred in recurrent neural networks (RNNs).
3. ReLU (Rectified Linear Unit): ReLU is a widely used activation function and serves as a good default choice for most situations. It introduces sparsity by setting negative values to zero, making it computationally efficient. However, it may lead to dead neurons during training, so it's crucial to monitor its performance.
4. Leaky ReLU: Leaky ReLU is a variant of ReLU and is employed when the standard ReLU causes neurons to become inactive. It allows a small gradient for negative values, preventing the issue of dead neurons. It's a recommended alternative to standard ReLU.
5. ELU (Exponential Linear Unit): ELU is valuable when you want the network to capture both positive and negative values within the hidden layers. It addresses the dying ReLU problem and can lead to faster convergence during training.
6. Swish: Swish is an activation function worth experimenting with, as it combines the computational efficiency of ReLU with a smoother, non-monotonic behavior. It has shown potential performance improvements in some architectures.
7. PReLU (Parametric ReLU): PReLU extends Leaky ReLU by allowing each neuron to learn its optimal alpha parameter. This can be beneficial when you want the network to adapt its activation function during training.
8. RReLU (Randomized Leaky ReLU): RReLU introduces randomness as a form of regularization during training. It can help prevent overfitting and enhance the generalization ability of the network.
9. PELU (Parametric Exponential Linear Unit): PELU extends ELU by enabling neurons to learn their alpha parameter. This flexibility can be advantageous in various scenarios, allowing the network to adapt to the data.
10. Softmax: Softmax activation is essential for multi-class classification problems in the output layer. It transforms a vector of real numbers into a probability distribution over multiple classes, enabling the network to make class predictions.
11. Softplus: Softplus is a smooth approximation of ReLU and can be helpful when you need a smooth activation function with continuous and differentiable derivatives.
12. ArcTan: ArcTan squashes input values to a limited range between $-\pi/2$ and $\pi/2$. It can be suitable for specific applications where you need to restrict the output within this range.
13. GELU (Gaussian Error Linear Unit): GELU is popular in transformer models and combines a smooth function with Gaussian noise, potentially leading to improved model performance.
14. Swish-1: Swish-1 is a variant of Swish with a division operation. It offers a different activation profile compared to standard Swish and is worth considering in experimentation.

15. ISRLU (Inverse Square Root Linear Unit): ISRLU is a smooth alternative to ReLU that can be helpful when you want to maintain smooth gradients throughout the network.
16. SELU (Scaled Exponential Linear Unit): SELU encourages automatic activation normalization and can lead to better training performance, especially in deep neural networks.
17. SoftExponential: SoftExponential introduces nonlinearity with a learnable parameter, allowing the network to adapt to specific data distributions.
18. Bipolar Sigmoid: Bipolar Sigmoid maps inputs to the range between -1 and 1, which can be beneficial when you want to model data with positive and negative values.
19. Binary Step: Binary Step is the simplest activation function, providing binary outputs based on a specified threshold. It's suitable for binary decision problems.

LOSSES IN DEEP LEARNING

A **loss function** is a function that **compares** the target and predicted output values; measures how well the neural network models the training data. When training, we aim to minimize this loss between the predicted and target outputs.
The **hyperparameters** are adjusted to minimize the average loss — we find the weights, *wT*, and biases, *b*, that minimize the value of *J* (average loss).

$$J(w^T, b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

Types of Loss Functions
In supervised learning, there are two main types of loss functions — these correlate to the 2 major types of neural networks: regression and classification loss functions

1. Regression Loss Functions — used in regression neural networks; given an input value, the model predicts a corresponding output value (rather than pre-selected labels); Ex. Mean Squared Error, Mean Absolute Error
2. Classification Loss Functions — used in classification neural networks; given an input, the neural network produces a vector of probabilities of the input belonging to various pre-set categories — can then select the category with the highest probability of belonging; Ex. Binary Cross-Entropy, Categorical Cross-Entropy

Mean Squared Error (MSE)
One of the most popular loss functions, MSE finds the average of the squared differences between the target and the predicted outputs.

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (y^{(i)} - \hat{y}^{(i)})^2$$

Mean Absolute Error (MAE)

MAE finds the average of the absolute differences between the target and the predicted outputs.

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |y^{(i)} - \hat{y}^{(i)}|$$

This loss function is used as an alternative to MSE in some cases. As mentioned previously, MSE is highly sensitive to outliers, which can dramatically affect the loss because the distance is squared. MAE is used in cases when the training data has many outliers to mitigate this.

It also has some disadvantages; as the average distance approaches 0, gradient descent optimization will not work, as the function's derivative at 0 is undefined (which will result in an error, as it is impossible to divide by 0).

Because of this, a loss function called a **Huber Loss** was developed, which has the advantages of both MSE and MAE.

$$Huber\ Loss = \begin{cases} \frac{1}{n}\sum_{i=1}^{n} (y^{(i)} - \hat{y}^{(i)})^2 & |y^{(i)} - \hat{y}^{(i)}| \leq \delta \\ \frac{1}{n}\sum_{i=1}^{n} \delta(|y^{(i)} - \hat{y}^{(i)}| - \frac{1}{2}\delta) & |y^{(i)} - \hat{y}^{(i)}| > \delta \end{cases}$$

Binary Cross-Entropy/Log Loss

This is the loss function used in binary classification models — where the model takes in an input and must classify it into one of two pre-set categories.

$$CE\ Loss = \frac{1}{n}\sum_{i=1}^{N} - (y_i \cdot log(p_i) + (1 - y_i) \cdot log(1 - p_i)$$

Classification neural networks work by outputting a vector of probabilities — the probability that the given input fits into each of the pre-set categories; then selecting the category with the highest probability as the final output.

In binary classification, there are only two possible actual values of y — 0 or 1. Thus, to accurately determine loss between the actual and predicted values, it needs to compare the actual value (0 or 1) with the probability that the input aligns with that category (*p(i)* = probability that the category is 1; 1 — *p(i)* = probability that the category is 0)

Categorical Cross-Entropy Loss

In cases where the number of classes is greater than two, we utilize categorical cross-entropy — this follows a very similar process to binary cross-entropy.

$$CE\ Loss = -\frac{1}{n}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij} \cdot log(p_{ij})$$

Binary cross-entropy is a special case of categorical cross-entropy, where $M = 2$ — the number of categories is 2.

SPARSED CATEGORICAL CROSSENTROPY

Label encoded data instead of one hot encoded data, multiclass classification.

BATCH NORMALIZATION

1. **What Is Batch Normalization?**
    - **Batch normalization** (also known as batch norm) is a technique used to enhance the training of artificial neural networks.
    - It normalizes the inputs to each layer by re-centring and re-scaling them within mini batches during training.
    - Proposed by Sergey Ioffe and Christian Szegedy in 2015, batch normalization aims to make training faster and more stable.
2. **Why Is It Needed?**
    - **Internal Covariate Shift**: During training, the distribution of inputs to each layer changes due to parameter updates and input randomness. This phenomenon is called internal covariate shift.
    - Batch normalization mitigates this shift by standardizing the inputs, making training more efficient.
3. **How It Works**:
    - For each mini batch during training:
        - Compute the mean and variance of the batch.
        - Normalize the inputs by subtracting the mean and dividing by the standard deviation.
        - Scale and shift the normalized values using learnable parameters (gamma and beta).
    - The normalized inputs are then used for further computations within the network.
4. **Benefits**:

- **Faster Training**: Batch normalization allows higher learning rates without gradient issues.
- **Regularization**: It acts as a regularizes, improving generalization.
- **Stabilized Learning**: Helps prevent vanishing or exploding gradients.
- **Smoothing Objective Function**: Some argue that it smooths the loss landscape, leading to better performance.

INTRODUCTION TO SCIPY

SciPy is a powerful scientific computation library built on top of NumPy. Here are the key points:

1. **What Is SciPy?**
   - **SciPy** stands for Scientific Python.
   - It provides utility functions for optimization, statistics, signal processing, and more.
   - Created by Travis Oliphant (the creator of NumPy), SciPy enhances NumPy by adding specialized functions frequently used in data science.
2. **Features**:
   - **Mathematical Algorithms**: SciPy includes a wide range of mathematical routines.
   - **High-Level Commands**: It simplifies complex tasks with user-friendly functions.
   - **Data Manipulation**: Use SciPy for data processing, visualization, and analysis.

INTRODUCTION TO OPENCV

OpenCV (Open-Source Computer Vision Library) is a powerful open-source software library for computer vision and machine learning. Developed by Intel, it provides a common infrastructure for various computer vision applications. Here are the key points:

1. **Functionality**:
   - OpenCV accelerates real-time machine perception tasks, including image recognition, object detection, and video processing.

- It offers over 2500 optimized algorithms, ranging from classical methods (like Support Vector Machines and K-Nearest Neighbours) to cutting-edge deep learning techniques.

2. **Applications**:
    - Detect and recognize faces.
    - Identify objects in images.
    - Analyse video streams.
    - Extract 3D models from stereo cameras.
    - Stitch images together for panoramic views.
    - And much more!
3. **Community and Usage**:
    - OpenCV has a large user community, with over 47 thousand contributors.
    - It's employed by well-established companies (Google, Microsoft, Intel) and startups alike.
    - Use cases span from surveillance video analysis to interactive art and rapid face detection.

OBJECT DETECTION USING OPENCV

The purpose is to detect a simple object detection script using OpenCV in Python. This script captures video from your webcam, applies colour-based masking, and displays the results.

1. **Setting Up the Environment**:
    - You initialize the webcam capture using cv2.VideoCapture(0).
    - Create a window named "Object_Detection" to display the results.
    - Set up trackbars to adjust threshold values and color ranges.
2. **Processing the Video Stream**:
    - Read frames from the webcam (cao.read()).
    - Resize the frame for display and flip it horizontally.
    - Convert the frame to HSV color space (cv2.cvtColor(img, cv2.COLOR_BGR2HSV)).
3. **Color-Based Masking**:

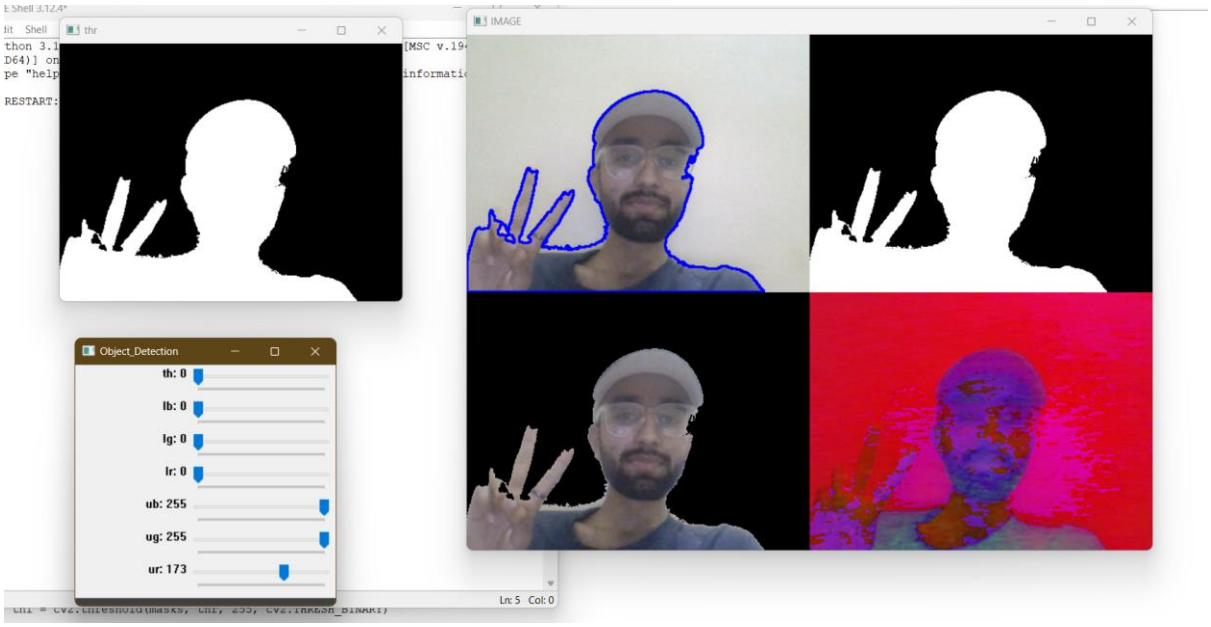
- Define lower (lo) and upper (up) bounds for the colour range you want to detect.
- Create a binary mask (masks) by thresholding the HSV image based on these bounds.
- Apply the mask to the original frame to get the masked result (res).

4. **Contour Detection**:
   - Find contours in the threshold image (thi) using cv2.findContours().
   - Draw the contours on the original frame (img).
5. **Display the Results**:
   - Stack the original frame, mask, and other intermediate images horizontally and vertically.
   - Show the combined result in the “IMAGE” window.

CODE

```
import cv2
import numpy as np
cao = cv2.VideoCapture(0)
def python(x):
  pass
cv2.namedWindow('Object_Detection')
cv2.createTrackbar('th', 'Object_Detection', 0 ,255, python)
cv2.createTrackbar('lb', 'Object_Detection', 0 ,255, python)
cv2.createTrackbar('lg', 'Object_Detection', 0 ,255, python)
cv2.createTrackbar('lr', 'Object_Detection', 0 ,255, python)
cv2.createTrackbar('ub', 'Object_Detection', 255 ,255, python)
cv2.createTrackbar('ug', 'Object_Detection', 255 ,255, python)
cv2.createTrackbar('ur', 'Object_Detection', 255 ,255, python)
while cao.isOpened():
  r, img = cao.read()
  if r == True:
    img = cv2.resize(img, (400, 300))
    img = cv2.flip(img, 1)
    hsv_img = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
    thr = cv2.getTrackbarPos('th', 'Object_Detection')
    Lb = cv2.getTrackbarPos('lb', 'Object_Detection')
    Lg = cv2.getTrackbarPos('lg', 'Object_Detection')
    Lr = cv2.getTrackbarPos('lr', 'Object_Detection')
```

```
      Ub = cv2.getTrackbarPos('ub', 'Object_Detection')
       Ug = cv2.getTrackbarPos('ug', 'Object_Detection')
       Ur = cv2.getTrackbarPos('ur', 'Object_Detection')
       lo = np.array([Lb, Lg, Lr])
       up = np.array([Ub, Ug, Ur])
       masks = cv2.inRange(hsv_img, lo, up)
       res = cv2.bitwise_and(img, img, mask = masks)
       fr = cv2.bitwise_not(res)
       r, thi = cv2.threshold(masks, thr, 255, cv2.THRESH_BINARY)
       c, h = cv2.findContours(thi, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
       cv2.drawContours(img, c, -1, (255,0,0), 2)
       cv2.imshow('thr', thi)
       img = np.array(img)
       hsv_img = np.array(hsv_img)
       masks = np.stack((masks,)*3, axis = -1)
       res = np.array(res)
       h = np.hstack((img, masks))
       g = np.hstack((res, hsv_img))
       r = np.vstack((h,g))
       cv2.imshow('IMAGE', r)
       if cv2.waitKey(25) & 0xFF == ord('p'):
          break
    else:
       break
cao.release()
cv2.destroyAllWindows()
```



EYE & SMILE DETECTION USING OPENCV

This is real-time eye & smile detection application using OpenCV and the Mediapipe library! Let's break it down:

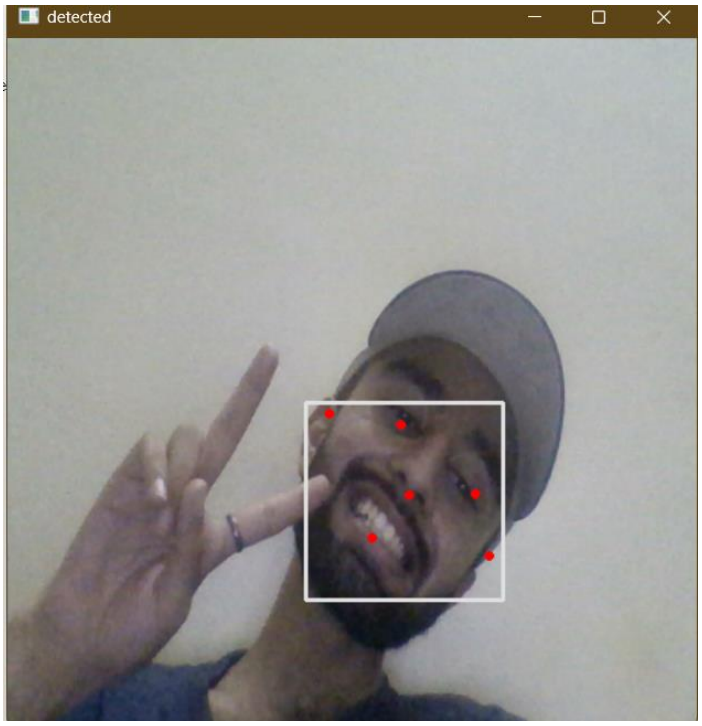
1. You're capturing video from the default camera (usually the webcam) using cv2.VideoCapture(0).
2. Inside the loop, you read frames from the camera with cao.read().
3. You flip the frame horizontally using cv2.flip(f, 1) to make it more intuitive.
4. The frame is resized to 500x500 pixels.
5. You convert the frame from BGR to RGB using cv2.cvtColor(f, cv2.COLOR_BGR2RGB) because Mediapipe works with RGB images.
6. The face detection model processes the frame, and the results are stored in res.
7. If any face detections are found (res.detections), you draw bounding boxes around the detected faces using mp_draw.draw_detection(f, cr).
8. The processed frame is converted back to BGR format.
9. The frame is displayed in a window named "detected".

CODE

```
import cv2
import mediapipe as mp
import warnings
warnings.filterwarnings('ignore')
mp_face = mp.solutions.face_detection
mp_draw = mp.solutions.drawing_utils
face_det = mp_face.FaceDetection(min_detection_confidence = 1, model_selection = 0)
cao = cv2.VideoCapture(0)
while cao.isOpened():
  r, f = cao.read()
  if r == True:
    f = cv2.flip(f, 1)
    f = cv2.resize(f, (500, 500))
    f = cv2.cvtColor(f, cv2.COLOR_BGR2RGB)
    res = face_det.process(f)
    f = cv2.cvtColor(f, cv2.COLOR_RGB2BGR)

    if res.detections:
      for cr in res.detections:
        mp_draw.draw_detection(f, cr)
    cv2.imshow('detected', f)
    if cv2.waitKey(20) & 0xFF == ord('p'):
```

```
        break
    else:
        break
cao.release()
cv2.destroyAllWindows()
```

OUTPUT



FACE DETECTION USING OPENCV

This is a real-time face detection application using OpenCV's Haar Cascade classifier.

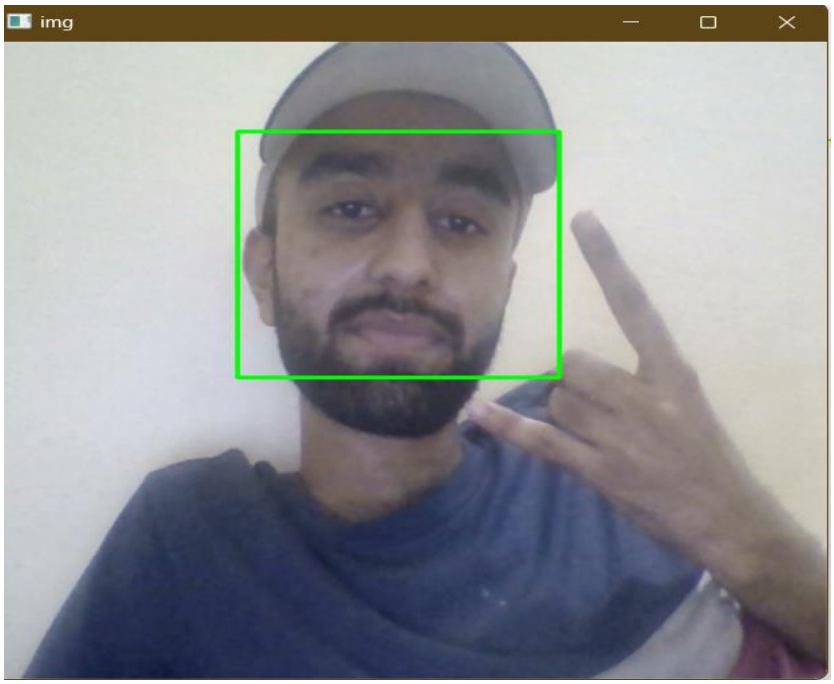
1. We are capturing video from the default camera (usually the webcam) using cv2.VideoCapture(0).
2. Inside the loop, the read frames from the camera with cao.read().
3. The flip the frame horizontally using cv2.flip(f, 1) to make it more intuitive.
4. The frame is resized to 500x500 pixels.
5. Now, convert the frame to grayscale using cv2.cvtColor(f, cv2.COLOR_BGR2GRAY) because Haar Cascade works with grayscale images.
6. To load the Haar Cascade classifier for frontal face detection using cv2.CascadeClassifier.
7. The detectMultiScale function detects faces in the grayscale frame. The parameters 1.2 and 4 control the scale factor and minimum neighbours, respectively.
8. For each detected face, you draw a green rectangle around it using cv2.rectangle(f, (x, y), (x + w, y + h), (0, 255, 0), 2, 3).
9. The processed frame is displayed in a window named "img".

10. If the ‘p’ key is pressed, the loop breaks.

CODE

```
import cv2
cao = cv2.VideoCapture(0)
while cao.isOpened():
  r,f = cao.read()
  if r == True:
    f = cv2.flip(f, 1)
    f = cv2.resize(f, (500, 500))
    gray = cv2.cvtColor(f, cv2.COLOR_BGR2GRAY)
    fil =
cv2.CascadeClassifier(r"C:\Users\soban\AppData\Local\Programs\Python\Python312\
Lib\site-packages\cv2\data\haarcascade_frontalface_default.xml")
    d = fil.detectMultiScale(gray, 1.2, 4)
    for (x, y, w, h) in d:
      cv2.rectangle(f, (x, y), (x + w, y + h), (0, 255, 0), 2, 3)
    cv2.imshow('img', f)
    if cv2.waitKey(25) & 0xFF == ord('p'):
      break
  else:
    break
cao.release()
cv2.destroyAllWindows()
```

OUTPUT



HAND GESTURE RECOGNITION

Face Detection using Mediapipe and OpenCV:

- The code captures video from the default camera (webcam).
- It processes each frame using the Mediapipe face detection model.
- Detected faces are highlighted with bounding boxes.
- Press ‘p’ to exit the loop.

Face Detection using Haar Cascade Classifier:

- The code captures video from the default camera.
- It converts frames to grayscale.
- Uses Haar Cascade classifier for frontal face detection.
- Detected faces are outlined with green rectangles.
- Press ‘p’ to exit the loop.

Hand Detection using Mediapipe:

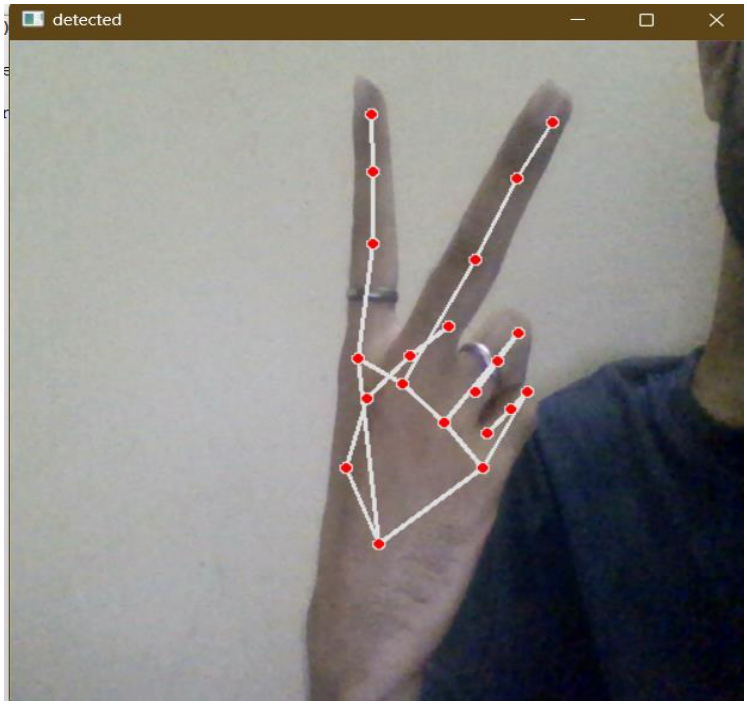
- The code captures video from the default camera.
- It processes each frame using the Mediapipe hand detection model.
- Detected hand landmarks are visualized with connections.
- Press ‘p’ to exit the loop.

CODE

```
import cv2
import mediapipe as mp
import warnings
warnings.filterwarnings('ignore')
mp_hand = mp.solutions.hands
mp_draw = mp.solutions.drawing_utils
det = mp_hand.Hands(static_image_mode = True, max_num_hands = 3,
min_detection_confidence = 0.5, min_tracking_confidence = 0.5)
cao = cv2.VideoCapture(0)
while cao.isOpened():
  r, f = cao.read()
  if r == True:
    f = cv2.flip(f, 1)
    f = cv2.resize(f, (500, 500))
    f = cv2.cvtColor(f, cv2.COLOR_BGR2RGB)
    res = det.process(f)
    f = cv2.cvtColor(f, cv2.COLOR_RGB2BGR)
    if res.multi_hand_landmarks:
      for hand_landmarks in res.multi_hand_landmarks:
        mp_draw.draw_landmarks(f, hand_landmarks,
mp_hand.HAND_CONNECTIONS)
    cv2.imshow('detected', f)
```

```
    if cv2.waitKey(20) & 0xFF == ord('p'):
        break
  else:
    break
cao.release()
cv2.destroyAllWindows()
```

OUTPUT



OBJECT DETECTION FROM VIDEO

Implementing object tracking using the CamShift algorithm with OpenCV. Let's break it down:

1. **Initialization**:
    - You load a video file using cv2.VideoCapture.
    - Define a region of interest (ROI) within the frame using (x, y, w, h) coordinates.
    - Extract the ROI from the first frame and convert it to HSV color space.
    - Create a mask based on HSV values to focus on specific colors within the ROI.
    - Calculate the histogram of the ROI in the hue channel.
    - Normalize the histogram.
2. **Tracking Loop**:
    - Continuously read frames from the video.
    - Convert each frame to HSV color space.

- Calculate the back projection of the hue channel using the ROI histogram.
- Apply the CamShift algorithm to find the new position of the tracked object.
- Update the tracking window coordinates (x, y, w, h).
- Draw a rectangle around the tracked object in the frame.
- Display the processed frame in a window named “TRACKING.”

3. **Exiting the Loop**:
   - Press the ‘p’ key to break out of the loop.
   - Release the video capture and close all windows.

CODE

```
import cv2
import numpy as np
cao =
cv2.VideoCapture(r"C:\Users\soban\Downloads\Tom_and_jerry_funny_video_😂😂_ll_
when_your_friend_running_in_school__#memes_#tom_#funny_#funnyvideo(480p).mp4
")
r, f = cao.read()
x, y ,w, h = [3, 328, 232, 300]
t = (x, y, w, h)
roi = f[y:y+h, x:x+w]
hsv_roi = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)
mask = cv2.inRange(hsv_roi, np.array((0., 60., 32.)), np.array((180., 255., 255.)))
roi_hist = cv2.calcHist([hsv_roi], [0], mask, [180], [0, 180])
cv2.normalize(roi_hist, roi_hist, 0, 255, cv2.NORM_MINMAX)
tr = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 1)
while True:
  r, f = cao.read()
  if r == True:
    hsv_f = cv2.cvtColor(f, cv2.COLOR_BGR2HSV)
    d = cv2.calcBackProject([hsv_f], [0], roi_hist, [0, 180], 1)
    r, tp = cv2.CamShift(d, t, tr)
    x, y, w, h = tp
    final = cv2.rectangle(f, (x,y), (x+w, y+h), (0,0,255), 4)
    cv2.imshow('TRACKING', final)
    if cv2.waitKey(20) & 0xFF == ord('p'):
      break
  else:
    break
cao.release()
cv2.destroyAllWindows()
```

OUTPUT:

PERSON DETECTION FROM FRAME

The template matching to find a specific image within another image. Let's break it down:

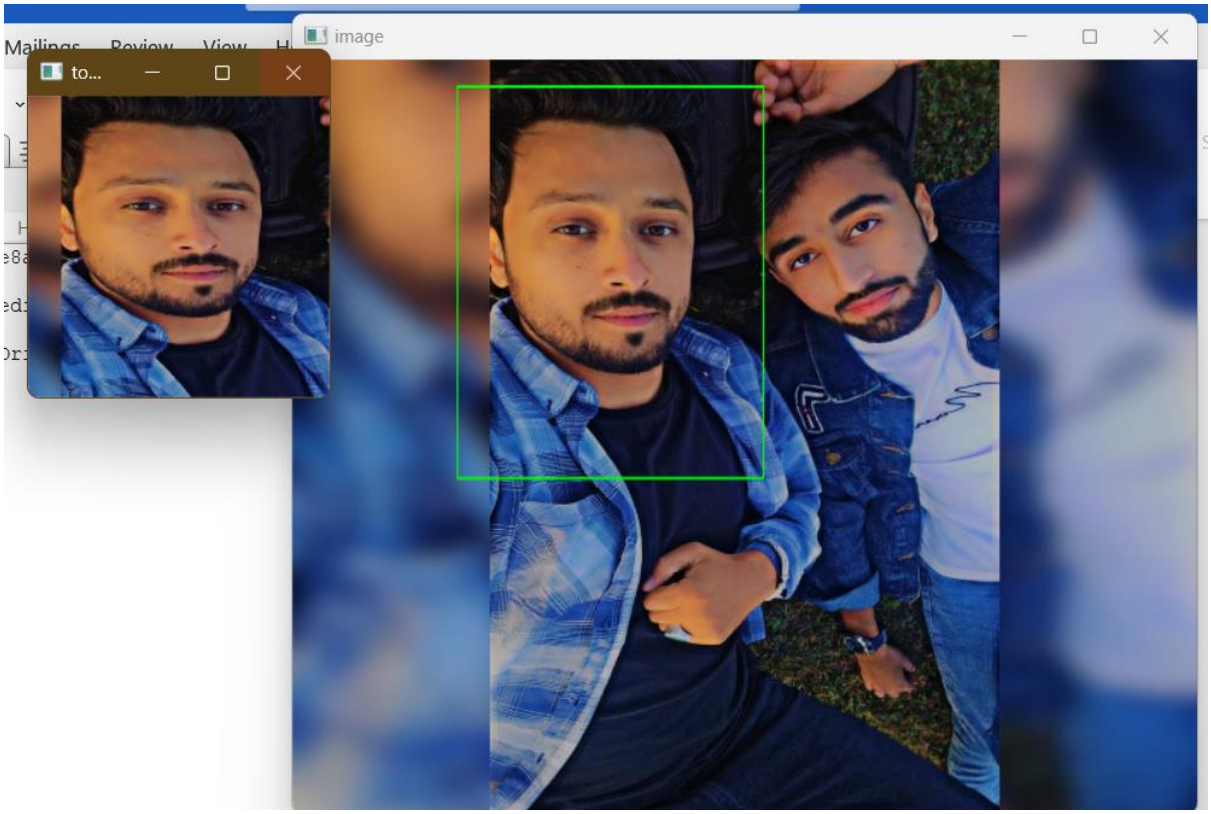
1. **Image Preparation**:
    - You load two images: temp (the template image) and org (the original image).
    - Convert both images to grayscale using cv2.cvtColor(..., cv2.COLOR_BGR2GRAY).
2. **Template Matching**:
    - Calculate the correlation between the template and the original image using cv2.matchTemplate(...).
    - Set a threshold (e.g., thr = 0.99) to determine matches.
    - Find the locations where the correlation exceeds the threshold using np.where(res >= thr).
3. **Drawing Rectangles**:
    - For each match, draw a green rectangle around the detected region in the original image using cv2.rectangle(...).
4. **Display Results**:
    - Resize the original image (org) and display it in a window named "image."
    - Resize the template image (temp) and display it in a window named "to be found."

CODE

```
import cv2
```

```
import numpy as np
temp = cv2.imread(r"C:\Users\soban\Downloads\new.jpg")
org = cv2.imread(r"C:\Users\soban\Downloads\IMG-20240626-WA0186.jpg")
gry = cv2.cvtColor(temp, cv2.COLOR_BGR2GRAY)
gry1 = cv2.cvtColor(org, cv2.COLOR_BGR2GRAY)
w, h = gry.shape[::-1]
res = cv2.matchTemplate(gry1, gry, cv2.TM_CCORR_NORMED)
thr = 0.99
l = np.where(res >= thr)
for i in zip(*l[::-1]):
    cv2.rectangle(org, i, (i[0]+w, i[1]+h), (0, 255, 0), 0)
org = cv2.resize(org, (600, 500))
cv2.imshow('image', org)
temp = cv2.resize(temp, (200, 200))
cv2.imshow('to be found', temp)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

OUTPUT



DEEP NEURAL NETWORKS

1. **What Are Deep Neural Networks?**

- A **deep neural network** (DNN) is a type of **artificial neural network (ANN)** with multiple layers between its input and output layers[1][2].
- These networks consist of interconnected **neurons**, each performing computations on input data.
- The term "deep" refers to the **number of layers** through which the data is transformed.

2. **Why "Deep"?**
   - Unlike traditional neural networks, which typically have only one or two hidden layers, DNNs have **multiple hidden layers**.
   - These layers allow DNNs to learn increasingly **abstract and composite representations** of the input data.
   - For example, in an image recognition model:
     - The first layer might identify basic shapes (like lines and circles).
     - Subsequent layers could encode more complex features (edges, noses, eyes, etc.).
     - The final layers recognize high-level patterns (faces, objects, etc.).

CONVOLUTIONAL NEURAL NETWORKS (CNN)

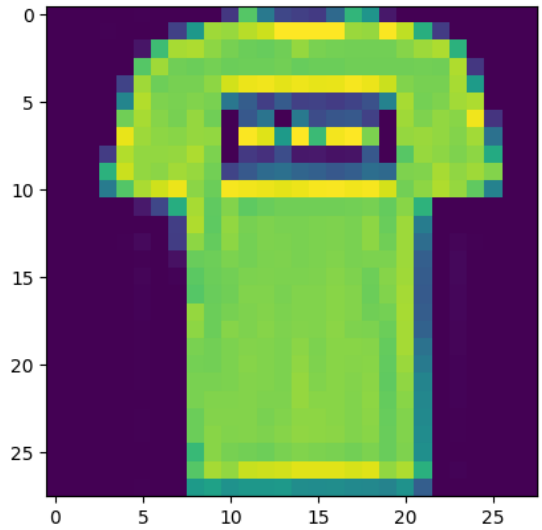
Certainly! Let's delve into the fascinating world of **Convolutional Neural Networks (CNNs)**.

1. **What Are CNNs?**
   - A **Convolutional Neural Network** (CNN) is a type of deep learning neural network architecture commonly used in **Computer Vision**.
   - CNNs excel at tasks like **image classification**, **object detection**, and **feature extraction**.
   - They are particularly effective for finding patterns in images to recognize objects.
2. **Key Components of CNNs**:
   - **Input Layer**: Receives the raw data (e.g., an image).
   - **Convolutional Layers**:
     - Apply filters (also known as kernels) to the input image to extract features.
     - Learn local patterns, edges, and textures.
   - **Pooling Layers**:
     - Downsample feature maps to reduce computation.
     - Common pooling methods include max-pooling and average-pooling.
   - **Fully Connected Layers**:
     - Also known as dense layers.
     - Combine features from previous layers for final prediction.
     - Often followed by a softmax layer for classification.
3. **Why CNNs?**

- CNNs capture spatial hierarchies and patterns effectively.
- They automatically learn relevant features from raw pixel data.
- Widely used in tasks like image recognition, segmentation, and style transfer.

CODE

```
import tensorflow as tf
from tensorflow import keras
from keras.layers import *
from keras.datasets import fashion_mnist
from keras.models import Sequential
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

(X_train, Y_train), (X_test, Y_test) = fashion_mnist.load_data()
plt.imshow(X_train[1])
```




```
model = Sequential()
from functools import partial
DefaultConv2D = partial(keras.layers.Conv2D, kernel_size=3, activation='relu', padding = "SAME")
model.add(DefaultConv2D(filters=64, kernel_size=7, input_shape=[28, 28, 1]))
model.add(keras.layers.MaxPooling2D(pool_size=2))
model.add(DefaultConv2D(filters=128))
model.add(DefaultConv2D(filters=128))
model.add(keras.layers.MaxPooling2D(pool_size=2))
model.add(DefaultConv2D(filters=256))
model.add(DefaultConv2D(filters=256))
model.add(keras.layers.MaxPooling2D(pool_size=2))
model.add(keras.layers.Flatten())
```

```
model.add(Dense(units=128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(units=64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(units=10, activation='softmax'))
```

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 64)        3200

max_pooling2d (MaxPooling2   (None, 14, 14, 64)        0
D)

conv2d_1 (Conv2D)            (None, 14, 14, 128)       73856

conv2d_2 (Conv2D)            (None, 14, 14, 128)       147584

max_pooling2d_1 (MaxPoolin   (None, 7, 7, 128)         0
g2D)

conv2d_3 (Conv2D)            (None, 7, 7, 256)         295168

conv2d_4 (Conv2D)            (None, 7, 7, 256)         590080

max_pooling2d_2 (MaxPoolin   (None, 3, 3, 256)         0
g2D)

flatten (Flatten)            (None, 2304)              0

dense (Dense)                (None, 128)               295040

dropout (Dropout)            (None, 128)               0

dense_1 (Dense)              (None, 64)                8256

dropout_1 (Dropout)          (None, 64)                0

dense_2 (Dense)              (None, 10)                650
```

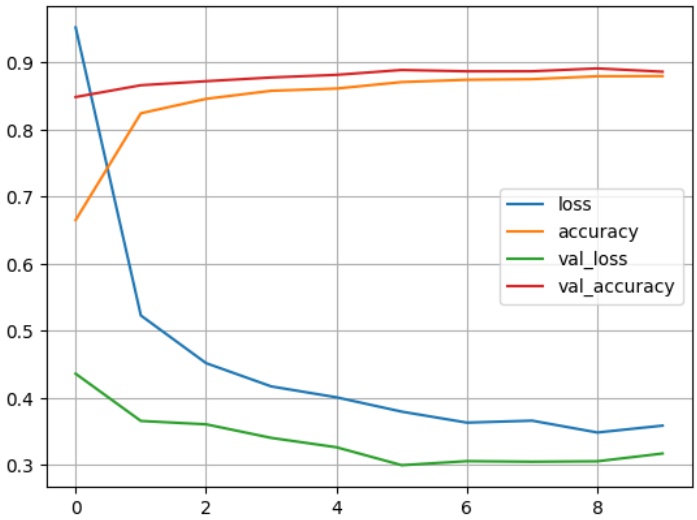
```
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam",
metrics=["accuracy"])
```

```
from keras.callbacks import EarlyStopping
es = EarlyStopping(monitor='accuracy', mode='max', verbose=1, patience=5)
history = model.fit(X_train, Y_train, epochs=10, validation_split = 0.2, callbacks = [es])
```

```
Epoch 4/10
1500/1500 [==============================] - 10s 7ms/step - loss: 0.4171 - accuracy: 0.8581 - val_loss: 0.3403 - val_accuracy: 0.8779
Epoch 5/10
1500/1500 [==============================] - 10s 7ms/step - loss: 0.4009 - accuracy: 0.8614 - val_loss: 0.3264 - val_accuracy: 0.8818
Epoch 6/10
1500/1500 [==============================] - 11s 7ms/step - loss: 0.3795 - accuracy: 0.8711 - val_loss: 0.2996 - val_accuracy: 0.8891
Epoch 7/10
1500/1500 [==============================] - 11s 7ms/step - loss: 0.3631 - accuracy: 0.8745 - val_loss: 0.3056 - val_accuracy: 0.8871
Epoch 8/10
1500/1500 [==============================] - 11s 7ms/step - loss: 0.3660 - accuracy: 0.8754 - val_loss: 0.3048 - val_accuracy: 0.8872
Epoch 9/10
1500/1500 [==============================] - 10s 7ms/step - loss: 0.3484 - accuracy: 0.8797 - val_loss: 0.3054 - val_accuracy: 0.8914
Epoch 10/10
1500/1500 [==============================] - 11s 7ms/step - loss: 0.3586 - accuracy: 0.8799 - val_loss: 0.3170 - val_accuracy: 0.8865
```

```
pd.DataFrame(history.history).plot()
```

```
plt.grid(True)
plt.show()
```




```
model.evaluate(X_test, Y_test)
proba_y = model.predict(X_test[:10])
pred_y = np.argmax(proba_y, axis=1)

print(pred_y)
```

```
313/313 [==============================] - 1s 4ms/step - loss: 0.3472 - accuracy: 0.8808
1/1 [==============================] - 0s 25ms/step
[9 2 1 1 6 1 4 6 5 7]
```

LENET-5

Certainly! The **LeNet-5 architecture** is a classic convolutional neural network (CNN) that played a pivotal role in demonstrating the effectiveness of deep learning for image recognition tasks. Let's explore its key components:

1. **Input Layer**:
   - Input size: 32×32 pixels.
   - Larger than the largest character in the database (usually 20×20 pixels).
   - Normalization: Pixel values are normalized to accelerate learning.
2. **Layer C1 (Convolutional Layer)**:
   - 6 feature maps.
   - Each unit connected to a 5×5 neighborhood in the input.
   - Prevents boundary effects.
   - Parameters: 156 trainable parameters and 117,600 connections.
3. **Layer S2 (Subsampling Layer)**:
   - 6 feature maps.
   - Size: 14×14 (each unit connected to a 2×2 neighborhood in C1).
   - Partial connectivity to break symmetry.
   - Parameters: 12 trainable parameters and 5,880 connections.

4. **Layer C3 (Convolutional Layer)**:
    - 16 feature maps.
    - Connections to several 5×5 neighborhoods in a subset of S2's feature maps.
    - Partial connectivity to learn different features.
    - Parameters: 1,516 trainable parameters and 151,600 connections.
5. **Fully Connected Layers**:
    - Two fully connected layers after convolution and pooling.
    - Finally, a softmax classifier for classification.

IMPLEMENTATION

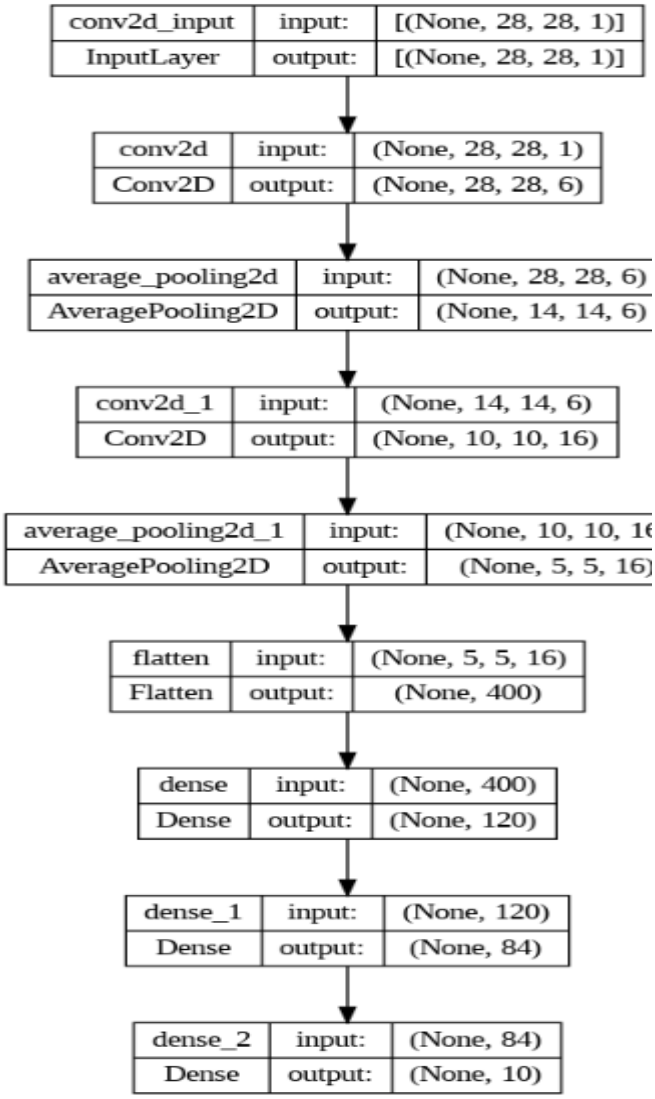
```
import tensorflow as tf
from tensorflow import keras
from keras import *
from keras.datasets import mnist
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
X_train.shape
```

```
(60000, 28, 28)
```

```
model = Sequential()
from keras.layers import *
model.add(Conv2D(filters = 6, kernel_size = 5, strides = 1, padding = 'same', activation = 'tanh', input_shape = (28, 28, 1)))
model.add(AveragePooling2D(pool_size = 2))
model.add(Conv2D(filters = 16, kernel_size = 5, strides = 1, activation = 'tanh', padding = 'valid'))
model.add(AveragePooling2D(pool_size = 2, strides = 2))
model.add(Flatten())
model.add(Dense(units = 120, activation = 'tanh'))
model.add(Dense(units = 84, activation = 'tanh'))
model.add(Dense(units = 10, activation = 'softmax'))
model.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 6)         156

 average_pooling2d (Average  (None, 14, 14, 6)         0
 Pooling2D)

 conv2d_1 (Conv2D)           (None, 10, 10, 16)        2416

 average_pooling2d_1 (Avera  (None, 5, 5, 16)          0
 gePooling2D)

 flatten (Flatten)           (None, 400)               0

 dense (Dense)               (None, 120)               48120

 dense_1 (Dense)             (None, 84)                10164

 dense_2 (Dense)             (None, 10)                850

=================================================================
Total params: 61706 (241.04 KB)
Trainable params: 61706 (241.04 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

keras.utils.plot_model(model, show_shapes = True)



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

from keras.callbacks import EarlyStopping

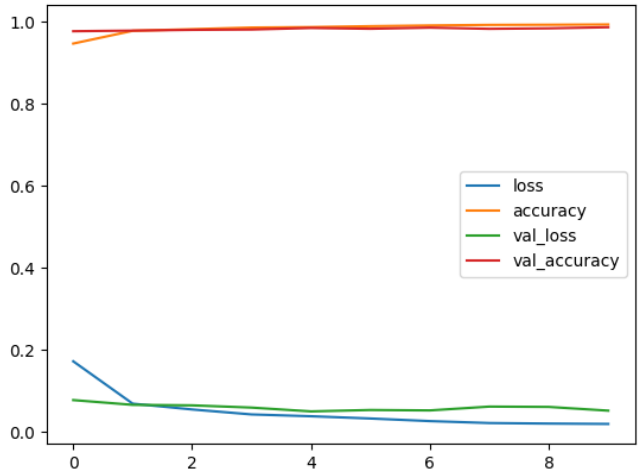
es = EarlyStopping(monitor = 'accuracy', mode = 'max', verbose = 1, patience = 5)

history = model.fit(X_train, Y_train, epochs = 10, validation_split = 0.2, callbacks = [es])

```
Epoch 1/10
1500/1500 [==============================] - 38s 24ms/step - loss: 0.1726 - accuracy: 0.9473 - val_loss: 0.0781 - val_accuracy: 0.9773
Epoch 2/10
1500/1500 [==============================] - 32s 22ms/step - loss: 0.0694 - accuracy: 0.9784 - val_loss: 0.0665 - val_accuracy: 0.9788
Epoch 3/10
1500/1500 [==============================] - 34s 23ms/step - loss: 0.0553 - accuracy: 0.9823 - val_loss: 0.0653 - val_accuracy: 0.9807
Epoch 4/10
1500/1500 [==============================] - 35s 23ms/step - loss: 0.0431 - accuracy: 0.9863 - val_loss: 0.0598 - val_accuracy: 0.9815
Epoch 5/10
1500/1500 [==============================] - 34s 23ms/step - loss: 0.0387 - accuracy: 0.9874 - val_loss: 0.0505 - val_accuracy: 0.9852
Epoch 6/10
1500/1500 [==============================] - 34s 23ms/step - loss: 0.0332 - accuracy: 0.9897 - val_loss: 0.0540 - val_accuracy: 0.9833
Epoch 7/10
1500/1500 [==============================] - 34s 22ms/step - loss: 0.0269 - accuracy: 0.9914 - val_loss: 0.0530 - val_accuracy: 0.9861
Epoch 8/10
1500/1500 [==============================] - 32s 22ms/step - loss: 0.0222 - accuracy: 0.9927 - val_loss: 0.0622 - val_accuracy: 0.9832
Epoch 9/10
1500/1500 [==============================] - 35s 23ms/step - loss: 0.0208 - accuracy: 0.9931 - val_loss: 0.0614 - val_accuracy: 0.9844
Epoch 10/10
1500/1500 [==============================] - 35s 23ms/step - loss: 0.0200 - accuracy: 0.9938 - val_loss: 0.0524 - val_accuracy: 0.9871
```

import pandas as pd

```
pd.DataFrame(history.history).plot()
```




```
loss, entropy = model.evaluate(X_test, Y_test)
print(loss)
print(entropy)
```

```
313/313 [==============================] - 3s 8ms/step - loss: 0.0378 - accuracy: 0.9890
0.037788670510053635
0.9890000224113464
```

```
Y_proba = model.predict(X_test[:10]).round(2)
import numpy as np
Y_pred = np.argmax(Y_proba, axis = 1)
print(Y_pred)
```

```
[7 2 1 0 4 1 4 9 5 9]
```

ALEXNET

Certainly! Let's explore the **AlexNet architecture**, a groundbreaking convolutional neural network (CNN) that significantly influenced the field of deep learning:

1. **Historical Context**:
    - AlexNet was designed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton.
    - It was submitted to the **ImageNet Large Scale Visual Recognition Challenge** in 2012.
    - The network achieved remarkable performance, outperforming other models by a significant margin.
2. **Key Components**:
    - **Input Size**: AlexNet takes RGB images of size **227x227x3** as input.
    - **Architecture**:
        - **Convolutional Layers**:
            - Five convolutional layers with varying filter sizes (e.g., 11x11, 5x5).
            - Non-linearity introduced by the ReLU activation function.

        - **Max Pooling Layers**:
            - Three max-pooling layers to downsample feature maps.
        - **Fully Connected Layers**:
            - Three fully connected layers (also known as dense layers).
            - The final layer produces probabilities for 1000 object categories (ImageNet classes).
        - **Softmax Output Layer**:
            - Computes class probabilities for classification.
3. **Achievements**:
    - AlexNet demonstrated that **deep architectures** could significantly improve performance.
    - It was computationally expensive but feasible due to GPU utilization during training.
    - The top 5 error rate was reduced by more than 10.8 percentage points compared to the runner-up.
4. **Legacy**:
    - While AlexNet is no longer state-of-the-art, it paved the way for subsequent deep learning models.
    - Its impact on computer vision and neural network research remains significant.

CODE

```
import tensorflow as tf
import numpy as np
from tensorflow import keras
from keras.models import Sequential
from keras.layers import *
from keras.datasets import mnist
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
X_train = X_train.reshape(-1, 28, 28, 1).astype('float32') / 255.0
X_test = X_test.reshape(-1, 28, 28, 1).astype('float32') / 255.0
print(X_train.shape)
```

```
(60000, 28, 28, 1)
```

```
model = Sequential()
model.add(Conv2D(filters = 96, kernel_size = 11, input_shape = (227, 227, 3), strides =
4, activation = 'relu'))
model.add(MaxPooling2D(pool_size = 3, strides = 2))
model.add(Conv2D(filters = 256, kernel_size = 5, strides = 1, padding = 'same',
activation = 'relu'))
model.add(MaxPooling2D(pool_size = 2, strides = 2))
```

```
model.add(Conv2D(filters = 384, kernel_size = 3, strides = 1, activation = 'relu', padding = 'same' ))
model.add(Conv2D(filters = 384, kernel_size = 3, strides = 1, activation = 'relu', padding = 'same' ))
model.add(Conv2D(filters = 384, kernel_size = 3, strides = 1, activation = 'relu', padding = 'same' ))
model.add(Flatten())
model.add(Dense(4096, activation = 'relu'))
model.add(Dropout(0.4))
model.add(Dense(4096, activation = 'relu'))
model.add(Dropout(0.4))
model.add(Dense(1000, activation = 'relu'))
model.add(Dropout(0.4))
model.add(Dense(10, activation = 'softmax'))
```

```
model.summary()
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 55, 55, 96)	34,944
max_pooling2d (MaxPooling2D)	(None, 27, 27, 96)	0
conv2d_1 (Conv2D)	(None, 27, 27, 256)	614,656
max_pooling2d_1 (MaxPooling2D)	(None, 13, 13, 256)	0
conv2d_2 (Conv2D)	(None, 13, 13, 384)	885,120
conv2d_3 (Conv2D)	(None, 13, 13, 384)	1,327,488
conv2d_4 (Conv2D)	(None, 13, 13, 384)	1,327,488
flatten (Flatten)	(None, 64896)	0
dense (Dense)	(None, 4096)	265,818,112
dropout (Dropout)	(None, 4096)	0
dense_1 (Dense)	(None, 4096)	16,781,312
dropout_1 (Dropout)	(None, 4096)	0
dense_2 (Dense)	(None, 1000)	4,097,000
dropout_2 (Dropout)	(None, 1000)	0
dense_3 (Dense)	(None, 10)	10,010

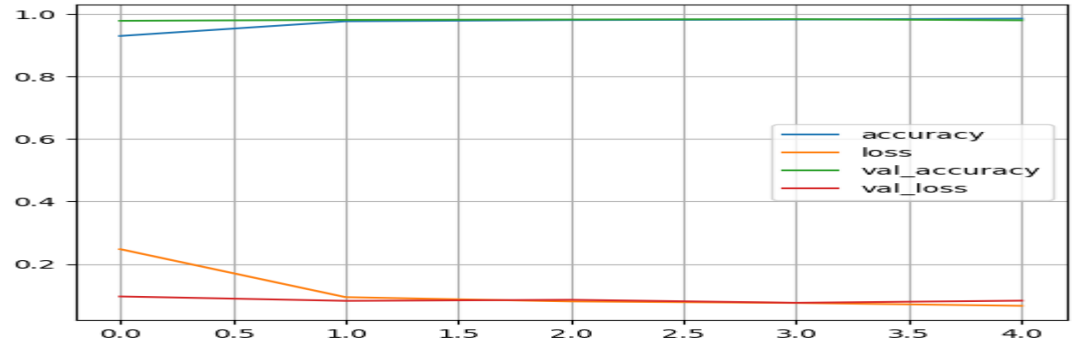
Total params: 290,896,130 (1.08 GB)
Trainable params: 290,896,130 (1.08 GB)
Non-trainable params: 0 (0.00 B)

```
model.compile(optimizer = 'adam', loss = 'sparse_categorical_crossentropy', metrics = ['accuracy'])
batch_size = 32
import cv2
def resizeimg(img):
  resize_img = cv2.resize(img, (227, 227))
```

```
 resize_img = np.stack((resize_img,)*3, axis = -1)
 return resize_img

def data(X, Y, batch_size):
 while True:

  for start in range(0, len(X), batch_size):
   end = min(start + batch_size, len(X_train))
   batch_X = X[start:end]
   batch_Y = Y[start:end]
   batch_X = np.array([resizeimg(img) for img in batch_X])
   yield batch_X, batch_Y

train = data(X_train[10000:], Y_train[10000:], batch_size)
test = data(X_test, Y_test, batch_size)
steps_per_epoch = len(X_train)//batch_size
validation_steps = len(X_test)//batch_size
valid = data(X_train[:10000], Y_train[:10000], batch_size)

from tensorflow.keras.callbacks import EarlyStopping
es = EarlyStopping(monitor = 'accuracy', patience = 3, verbose = 1, mode = 'auto')
history = model.fit(train, steps_per_epoch = steps_per_epoch, epochs =
5,  validation_steps = validation_steps, validation_data = valid, callbacks = [es])
```

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 218s 109ms/step - accuracy: 0.8297 - loss: 0.6063 - val_accuracy: 0.9779 - val_loss: 0.0962
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 197s 105ms/step - accuracy: 0.9761 - loss: 0.0947 - val_accuracy: 0.9807 - val_loss: 0.0822
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 196s 104ms/step - accuracy: 0.9783 - loss: 0.0895 - val_accuracy: 0.9819 - val_loss: 0.0856
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 195s 104ms/step - accuracy: 0.9814 - loss: 0.0802 - val_accuracy: 0.9831 - val_loss: 0.0760
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 194s 103ms/step - accuracy: 0.9848 - loss: 0.0710 - val_accuracy: 0.9795 - val_loss: 0.0828
```

```
import pandas as pd
import matplotlib.pyplot as plt
pd.DataFrame(history.history).plot()
plt.grid(True)
plt.show()
```



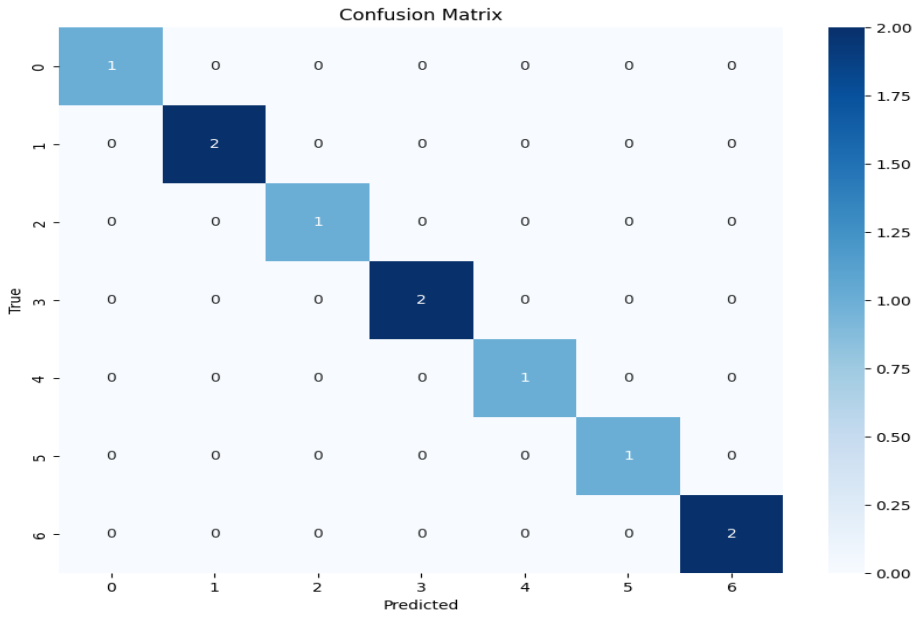
```
X_test, Y_test = next(test)
print(model.evaluate(X_test, Y_test))
proba_y = model.predict(X_test[:10])
pred_y = np.argmax(proba_y, axis=1)

print(pred_y)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 65ms/step - accuracy: 0.9688 - loss: 0.1575
[0.15746571123600006, 0.96875]
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step
[7 2 1 0 4 1 4 9 5 9]
```

```
from sklearn.metrics import confusion_matrix
conf = confusion_matrix(pred_y, Y_test[:10])
print(conf)
```

```
[[1 0 0 0 0 0 0]
 [0 2 0 0 0 0 0]
 [0 0 1 0 0 0 0]
 [0 0 0 2 0 0 0]
 [0 0 0 0 1 0 0]
 [0 0 0 0 0 1 0]
 [0 0 0 0 0 0 2]]
```

```
import seaborn as sns
plt.figure(figsize=(10, 8))
sns.heatmap(conf, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```



DATA AUGMENTATION

Data augmentation is a technique used to artificially increase the size and diversity of a training dataset by creating modified copies of existing data. It plays a crucial role in improving machine learning models, especially in scenarios where the initial training set is limited or prone to overfitting. Here are some key points about data augmentation:

1. **Purpose of Data Augmentation**:
    - **Preventing Overfitting**: By introducing variations in the training data, data augmentation helps prevent models from memorizing specific examples and generalizes better to unseen data.
    - **Increasing Model Accuracy**: A larger and more diverse dataset often leads to improved model performance.
    - **Reducing Labelling Costs**: Instead of manually collecting new data, we can create synthetic variations of existing data.
2. **Techniques for Data Augmentation**:
    - **Image Augmentation** (commonly used in computer vision):
        - Geometric transformations: Randomly flip, crop, rotate, stretch, and zoom images.
        - Color space transformations: Randomly adjust RGB color channels, contrast, and brightness.
        - Kernel filters: Randomly change image sharpness or apply blurring.
    - **Audio Data Augmentation**:
        - Noise injection: Add Gaussian or random noise to audio data.
        - Shifting: Shift audio left or right by random seconds.
        - Changing speed: Stretch time series by a fixed rate.
        - Changing pitch: Randomly alter the pitch of audio.
    - **Text Data Augmentation**:
        - Word or sentence shuffling: Randomly change the position of words or sentences.
        - Word replacement: Replace words with synonyms.
        - Syntax-tree manipulation: Paraphrase sentences using the same words.
        - Random word insertion and deletion.
3. **Limitations and Considerations**:
    - **Biases**: Augmented data inherits biases from the original dataset.
    - **Quality Assurance**: Ensuring the quality of augmented data can be expensive.
    - **Advanced Applications**: Generating high-resolution images using techniques like GANs can be challenging.

To apply data augmentation in two ways:

- Use the Keras preprocessing layers, such as tf.keras.layers.Resizing, tf.keras.layers.Rescaling, tf.keras.layers.RandomFlip, and tf.keras.layers.RandomRotation.
- Use the tf.image methods, such as tf.image.flip_left_right, tf.image.rgb_to_grayscale, tf.image.adjust_brightness, tf.image.central_crop, and tf.image.stateless_random*.

IMPLEMENTATION OF ALEXNET ON MNIST DATASET USING DATA AUGMENTATION

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import tensorflow_datasets as tfds
from keras.models import Sequential
from keras.layers import *

(train, test, valid), info = tfds.load('mnist', split = ['train[:80%]', 'train[80%:90%]',
'train[90%:]'], as_supervised = True, with_info = True)
resize = Sequential()
resize.add(Resizing(227,227))
resize.add(Rescaling(1./255))
train = train.map(lambda x, y: (resize(x, training = True), y))
test = test.map(lambda x, y: (resize(x, training = True), y))
valid = valid.map(lambda x, y: (resize(x, training = True), y))
train = train.map(lambda x, y: (tf.image.grayscale_to_rgb(x), y))
test = test.map(lambda x, y: (tf.image.grayscale_to_rgb(x), y))
valid = valid.map(lambda x, y: (tf.image.grayscale_to_rgb(x), y))
train = train.batch(32)
test = test.batch(32)
valid = valid.batch(32)
for bi, bl in train:
 for img, label in zip(bi, bl):
  print(img.shape)
  break
 break
```

```
(227, 227, 3)
```

```
model = Sequential()
model.add(Conv2D(filters = 96, kernel_size = 11, input_shape = (227, 227, 3), strides =
4, activation = 'relu'))
model.add(MaxPooling2D(pool_size = 3, strides = 2))
```

```
model.add(Conv2D(filters = 256, kernel_size = 5, strides = 1, padding = 'same', activation = 'relu'))
model.add(MaxPooling2D(pool_size = 2, strides = 2))
model.add(Conv2D(filters = 384, kernel_size = 3, strides = 1, activation = 'relu', padding = 'same' ))
model.add(Conv2D(filters = 384, kernel_size = 3, strides = 1, activation = 'relu', padding = 'same' ))
model.add(Conv2D(filters = 384, kernel_size = 3, strides = 1, activation = 'relu', padding = 'same' ))
model.add(Flatten())
model.add(Dense(4096, activation = 'relu'))
model.add(Dropout(0.4))
model.add(Dense(4096, activation = 'relu'))
model.add(Dropout(0.4))
model.add(Dense(1000, activation = 'relu'))
model.add(Dropout(0.4))
model.add(Dense(10, activation = 'softmax'))
```

```
model.summary()
```

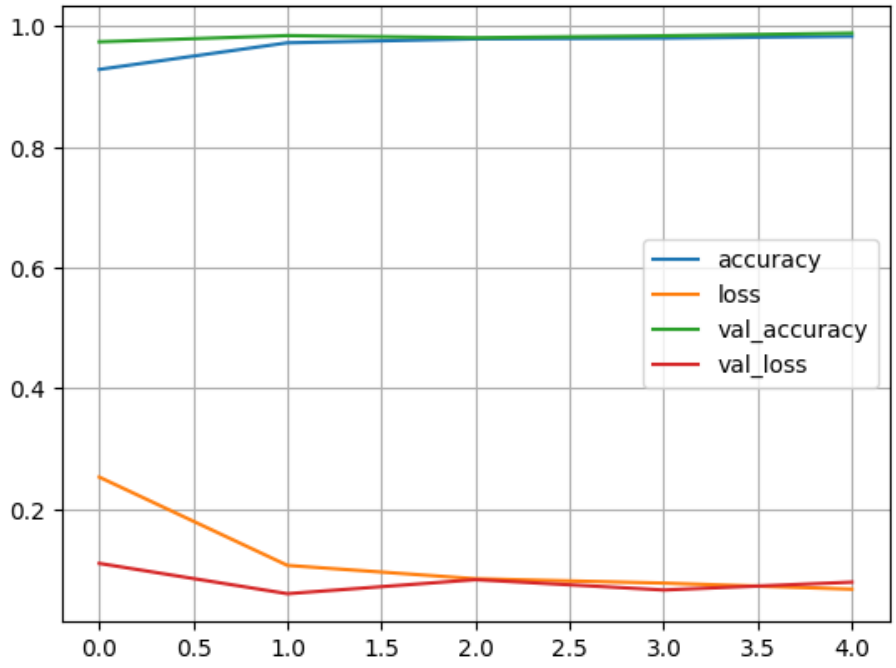
Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 55, 55, 96)	34,944
max_pooling2d (MaxPooling2D)	(None, 27, 27, 96)	0
conv2d_1 (Conv2D)	(None, 27, 27, 256)	614,656
max_pooling2d_1 (MaxPooling2D)	(None, 13, 13, 256)	0
conv2d_2 (Conv2D)	(None, 13, 13, 384)	885,120
conv2d_3 (Conv2D)	(None, 13, 13, 384)	1,327,488
conv2d_4 (Conv2D)	(None, 13, 13, 384)	1,327,488
flatten (Flatten)	(None, 64896)	0
dense (Dense)	(None, 4096)	265,818,112
dropout (Dropout)	(None, 4096)	0
dense_1 (Dense)	(None, 4096)	16,781,312
dropout_1 (Dropout)	(None, 4096)	0
dense_2 (Dense)	(None, 1000)	4,097,000
dropout_2 (Dropout)	(None, 1000)	0
dense_3 (Dense)	(None, 10)	10,010

Total params: 290,896,130 (1.08 GB)

```
model.compile(optimizer = 'adam', loss = 'sparse_categorical_crossentropy', metrics = ['accuracy'])
from tensorflow.keras.callbacks import EarlyStopping
es = EarlyStopping(monitor = 'accuracy', patience = 3, verbose = 1, mode = 'auto')
history = model.fit(train, epochs = 5, validation_data = valid, callbacks = [es])
```

```
Epoch 1/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 168s 103ms/step - accuracy: 0.8286 - loss: 0.6165 - val_accuracy: 0.9743 - val_loss: 0.1099
Epoch 2/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 190s 102ms/step - accuracy: 0.9707 - loss: 0.1135 - val_accuracy: 0.9847 - val_loss: 0.0596
Epoch 3/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 196s 98ms/step - accuracy: 0.9778 - loss: 0.0914 - val_accuracy: 0.9810 - val_loss: 0.0829
Epoch 4/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 202s 99ms/step - accuracy: 0.9803 - loss: 0.0796 - val_accuracy: 0.9843 - val_loss: 0.0658
Epoch 5/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 153s 102ms/step - accuracy: 0.9827 - loss: 0.0683 - val_accuracy: 0.9887 - val_loss: 0.0788
```

```
import pandas as pd
import matplotlib.pyplot as plt
pd.DataFrame(history.history).plot()
plt.grid(True)
plt.show()
```




```
X_test, Y_test = next(iter(test))
print(X_test.shape)
print(Y_test.shape)
```

```
(32, 227, 227, 3)
(32,)
```

```
print(model.evaluate(test))
proba_y = model.predict(X_test[:10])
pred_y = np.argmax(proba_y, axis=1)
```

print(pred_y)

```
188/188 ━━━━━━━━━━━━━━━━━━━━ 8s 43ms/step - accuracy: 0.9881 - loss: 0.0682
[0.07366809248924255, 0.9856666922569275]
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step
[8 5 3 9 5 8 8 1 0 0]
```

from sklearn.metrics import confusion_matrix

conf = confusion_matrix(pred_y, Y_test[:10])

print(conf)

```
[[2 0 0 0 0 0]
 [0 1 0 0 0 0]
 [0 0 1 0 0 0]
 [0 0 0 2 0 0]
 [0 0 0 0 3 0]
 [0 0 0 0 0 1]]
```

import seaborn as sns

plt.figure(figsize=(10, 8))

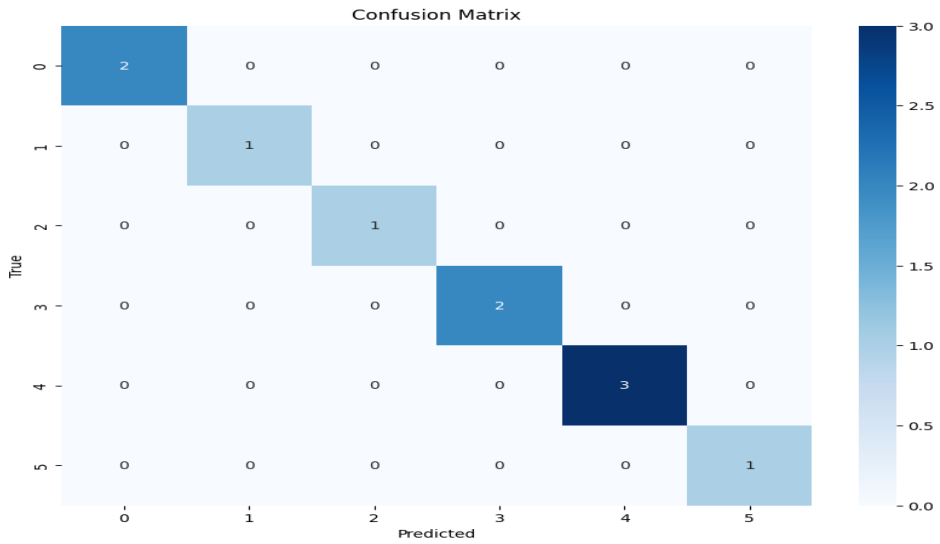
sns.heatmap(conf, annot=True, fmt='d', cmap='Blues')

plt.xlabel('Predicted')

plt.ylabel('True')

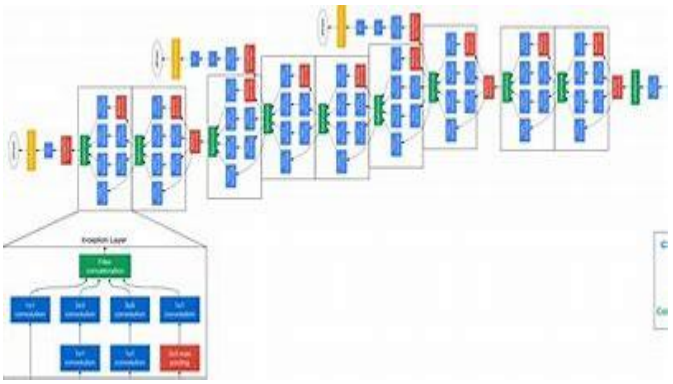
plt.title('Confusion Matrix')

plt.show()



GOOGLENET

GoogLeNet, also known as **Inception-v1**, stands as a landmark achievement in the realm of convolutional neural networks (CNNs). Unveiled by researchers at Google in 2014, it introduced a novel approach to designing deep networks that not only achieved exceptional accuracy but also demonstrated remarkable computational efficiency.



Here are the key points about GoogLeNet:

1. **Inception Modules**:
   - GoogLeNet is particularly well-known for its use of **Inception modules** as building blocks.
   - These modules allow the network to choose between multiple convolutional filter sizes (1×1, 3×3, and 5×5) within a single layer.
   - The outputs from these filters are then concatenated, creating a richer representation.
2. **Efficiency and Depth**:
   - GoogLeNet had 22 layers, making it deeper than previous models.
   - It demonstrated that depth and width in a neural network could be increased without exploding computations.
   - The inception module creatively solved the vanishing gradient problem by combining multiple convolution filters.
3. **Historical Context**:
   - Before GoogLeNet, AlexNet (2012) brought deep learning to the forefront of AI research.
   - GoogLeNet surpassed previous benchmarks, achieving a top 5 error rate of 6.7% in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC).
   - Its efficiency and accuracy set a new standard for CNNs.

CODE

```
from keras.models import Model
from keras.layers import Input, Conv2D, MaxPooling2D, AveragePooling2D, Flatten,
GlobalAveragePooling2D, Dense, Dropout
from keras.layers import concatenate

from tensorflow.keras.datasets import mnist
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()

import numpy as np
```

```
X_train = X_train.reshape(-1, 28, 28, 1).astype('float32') / 255.0
X_test = X_test.reshape(-1, 28, 28, 1).astype('float32') / 255.0

print(X_train.shape)


def Inception_block(input_layer, f1, f2_conv1, f2_conv3, f3_conv1, f3_conv5, f4):

 # 1st path:
 path1 = Conv2D(filters=f1, kernel_size = (1,1), padding = 'same', activation =
'relu')(input_layer)

 # 2nd path
 path2 = Conv2D(filters = f2_conv1, kernel_size = (1,1), padding = 'same', activation =
'relu')(input_layer)
 path2 = Conv2D(filters = f2_conv3, kernel_size = (3,3), padding = 'same', activation =
'relu')(path2)

 # 3rd path
 path3 = Conv2D(filters = f3_conv1, kernel_size = (1,1), padding = 'same', activation =
'relu')(input_layer)
 path3 = Conv2D(filters = f3_conv5, kernel_size = (5,5), padding = 'same', activation =
'relu')(path3)

 # 4th path
 path4 = MaxPooling2D((3,3), strides= (1,1), padding = 'same')(input_layer)
 path4 = Conv2D(filters = f4, kernel_size = (1,1), padding = 'same', activation =
'relu')(path4)

 output_layer = concatenate([path1, path2, path3, path4], axis = -1)

 return output_layer

def GoogLeNet():
 # input layer
 input_layer = Input(shape = (224, 224, 3))

 # convolutional layer: filters = 64, kernel_size = (7,7), strides = 2
 X = Conv2D(filters = 64, kernel_size = (7,7), strides = 2, padding = 'valid', activation =
'relu')(input_layer)
```

```
  # max-pooling layer: pool_size = (3,3), strides = 2
  X = MaxPooling2D(pool_size = (3,3), strides = 2)(X)

  # convolutional layer: filters = 64, strides = 1
  X = Conv2D(filters = 64, kernel_size = (1,1), strides = 1, padding = 'same', activation =
'relu')(X)

  # convolutional layer: filters = 192, kernel_size = (3,3)
  X = Conv2D(filters = 192, kernel_size = (3,3), padding = 'same', activation = 'relu')(X)

  # max-pooling layer: pool_size = (3,3), strides = 2
  X = MaxPooling2D(pool_size= (3,3), strides = 2)(X)

  # 1st Inception block
  X = Inception_block(X, f1 = 64, f2_conv1 = 96, f2_conv3 = 128, f3_conv1 = 16, f3_conv5
= 32, f4 = 32)

  # 2nd Inception block
  X = Inception_block(X, f1 = 128, f2_conv1 = 128, f2_conv3 = 192, f3_conv1 = 32,
f3_conv5 = 96, f4 = 64)

  # max-pooling layer: pool_size = (3,3), strides = 2
  X = MaxPooling2D(pool_size= (3,3), strides = 2)(X)

  # 3rd Inception block
  X = Inception_block(X, f1 = 192, f2_conv1 = 96, f2_conv3 = 208, f3_conv1 = 16,
f3_conv5 = 48, f4 = 64)

  # Extra network 1:
  X1 = AveragePooling2D(pool_size = (5,5), strides = 3)(X)
  X1 = Conv2D(filters = 128, kernel_size = (1,1), padding = 'same', activation = 'relu')(X1)
  X1 = Flatten()(X1)
  X1 = Dense(1024, activation = 'relu')(X1)
  X1 = Dropout(0.7)(X1)
  X1 = Dense(5, activation = 'softmax')(X1)



  # 4th Inception block
  X = Inception_block(X, f1 = 160, f2_conv1 = 112, f2_conv3 = 224, f3_conv1 = 24,
f3_conv5 = 64, f4 = 64)
```

```
  # 5th Inception block
  X = Inception_block(X, f1 = 128, f2_conv1 = 128, f2_conv3 = 256, f3_conv1 = 24,
f3_conv5 = 64, f4 = 64)

  # 6th Inception block
  X = Inception_block(X, f1 = 112, f2_conv1 = 144, f2_conv3 = 288, f3_conv1 = 32,
f3_conv5 = 64, f4 = 64)

  # Extra network 2:
  X2 = AveragePooling2D(pool_size = (5,5), strides = 3)(X)
  X2 = Conv2D(filters = 128, kernel_size = (1,1), padding = 'same', activation = 'relu')(X2)
  X2 = Flatten()(X2)
  X2 = Dense(1024, activation = 'relu')(X2)
  X2 = Dropout(0.7)(X2)
  X2 = Dense(1000, activation = 'softmax')(X2)



  # 7th Inception block
  X = Inception_block(X, f1 = 256, f2_conv1 = 160, f2_conv3 = 320, f3_conv1 = 32,
                f3_conv5 = 128, f4 = 128)

  # max-pooling layer: pool_size = (3,3), strides = 2
  X = MaxPooling2D(pool_size = (3,3), strides = 2)(X)

  # 8th Inception block
  X = Inception_block(X, f1 = 256, f2_conv1 = 160, f2_conv3 = 320, f3_conv1 = 32,
f3_conv5 = 128, f4 = 128)

  # 9th Inception block
  X = Inception_block(X, f1 = 384, f2_conv1 = 192, f2_conv3 = 384, f3_conv1 = 48,
f3_conv5 = 128, f4 = 128)

  # Global Average pooling layer
  X = GlobalAveragePooling2D(name = 'GAPL')(X)

  # Dropoutlayer
  X = Dropout(0.4)(X)

  # output layer
  X = Dense(10, activation = 'softmax')(X)
```

```
 # model
 model = Model(input_layer, [X, X1, X2], name = 'GoogLeNet')

 return model

model = GoogLeNet()

model.summary()
```

Model: "GoogLeNet"

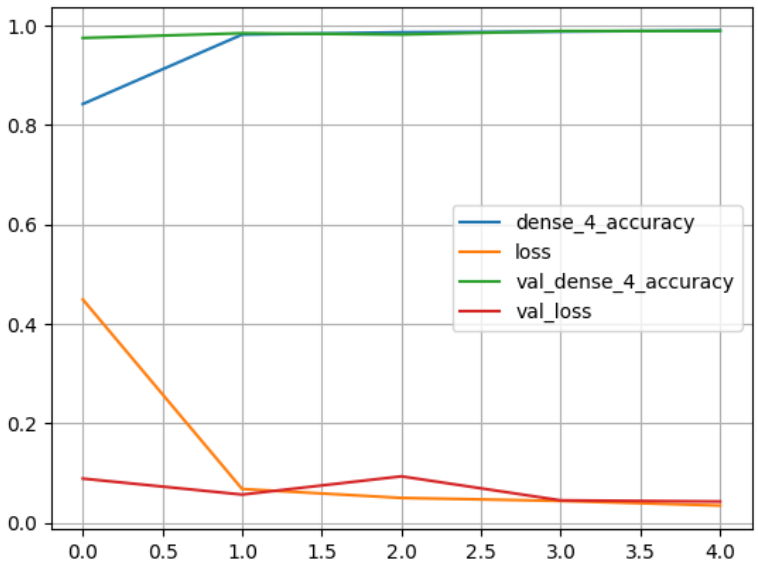
Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 224, 224, 3)	0	-
conv2d (Conv2D)	(None, 109, 109, 64)	9,472	input_layer[0][0]
max_pooling2d (MaxPooling2D)	(None, 54, 54, 64)	0	conv2d[0][0]
conv2d_1 (Conv2D)	(None, 54, 54, 64)	4,160	max_pooling2d[0][0]
conv2d_2 (Conv2D)	(None, 54, 54, 192)	110,784	conv2d_1[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 26, 26, 192)	0	conv2d_2[0][0]
conv2d_4 (Conv2D)	(None, 26, 26, 96)	18,528	max_pooling2d_1[0][0]
conv2d_6 (Conv2D)	(None, 26, 26, 16)	3,088	max_pooling2d_1[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 26, 26, 192)	0	max_pooling2d_1[0][0]
conv2d_3 (Conv2D)	(None, 26, 26, 64)	12,352	max_pooling2d_1[0][0]
conv2d_5 (Conv2D)	(None, 26, 26, 128)	110,720	conv2d_4[0][0]
conv2d_7 (Conv2D)	(None, 26, 26, 32)	12,832	conv2d_6[0][0]
conv2d_8 (Conv2D)	(None, 26, 26, 32)	6,176	max_pooling2d_2[0][0]
concatenate (Concatenate)	(None, 26, 26, 256)	0	conv2d_3[0][0], conv2d_5[0][0], conv2d_7[0][0], conv2d_8[0][0]
conv2d_10 (Conv2D)	(None, 26, 26, 128)	32,896	concatenate[0][0]
conv2d_12 (Conv2D)	(None, 26, 26, 32)	8,224	concatenate[0][0]
max_pooling2d_3 (MaxPooling2D)	(None, 26, 26, 256)	0	concatenate[0][0]
conv2d_9 (Conv2D)	(None, 26, 26, 128)	32,896	concatenate[0][0]
conv2d_11 (Conv2D)	(None, 26, 26, 192)	221,376	conv2d_10[0][0]
conv2d_13 (Conv2D)	(None, 26, 26, 96)	76,896	conv2d_12[0][0]
conv2d_14 (Conv2D)	(None, 26, 26, 64)	16,448	max_pooling2d_3[0][0]
concatenate_1 (Concatenate)	(None, 26, 26, 480)	0	conv2d_9[0][0], conv2d_11[0][0], conv2d_13[0][0], conv2d_14[0][0]
max_pooling2d_4 (MaxPooling2D)	(None, 12, 12, 480)	0	concatenate_1[0][0]
conv2d_16 (Conv2D)	(None, 12, 12, 96)	46,176	max_pooling2d_4[0][0]

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

batch_size =32
```

```
import cv2
def resizeimg(img):
 resize_img = cv2.resize(img, (224, 224))
 resize_img = np.stack((resize_img,)*3, axis = -1)
 return resize_img

def data(X, Y, batch_size):
 while True:

  for start in range(0, len(X), batch_size):
   end = min(start + batch_size, len(X_train))
   batch_X = X[start:end]
   batch_Y = Y[start:end]
   batch_X = np.array([resizeimg(img) for img in batch_X])
   yield batch_X, batch_Y

train = data(X_train[10000:], Y_train[10000:], batch_size)
test = data(X_test, Y_test, batch_size)
valid = data(X_train[:10000], Y_train[:10000], batch_size)
steps_per_epoch = len(X_train)//batch_size
validation_steps = len(X_train[:10000])//batch_size

import warnings
warnings.filterwarnings('ignore')

from tensorflow.keras.callbacks import EarlyStopping
es = EarlyStopping(monitor = 'dense_4_accuracy', patience = 3, verbose = 1, mode =
'max')
history = model.fit(train, steps_per_epoch = steps_per_epoch, epochs = 5,
validation_data = valid, validation_steps = validation_steps, callbacks = [es])
```

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 264s 110ms/step - dense_4_accuracy: 0.6166 - loss: 1.0326 - val_dense_4_accuracy: 0.9743 - val_loss: 0.0887
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 182s 97ms/step - dense_4_accuracy: 0.9794 - loss: 0.0747 - val_dense_4_accuracy: 0.9838 - val_loss: 0.0566
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 178s 95ms/step - dense_4_accuracy: 0.9840 - loss: 0.0571 - val_dense_4_accuracy: 0.9811 - val_loss: 0.0933
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 178s 95ms/step - dense_4_accuracy: 0.9862 - loss: 0.0485 - val_dense_4_accuracy: 0.9882 - val_loss: 0.0450
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 177s 94ms/step - dense_4_accuracy: 0.9893 - loss: 0.0356 - val_dense_4_accuracy: 0.9884 - val_loss: 0.0427
```

```
import pandas as pd
import matplotlib.pyplot as plt
pd.DataFrame(history.history).plot()
plt.grid(True)
plt.show()
```




```
X_test, Y_test = next(test)

print(model.evaluate(X_test, Y_test))
print(model.evaluate(X_test, Y_test))
proba_y = model.predict(X_test[:10])

proba_y[0]
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 64ms/step - dense_4_accuracy: 1.0000 - loss: 0.0114
[0.011410207487642765, 1.0]
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 62ms/step - dense_4_accuracy: 1.0000 - loss: 0.0114
[0.011410207487642765, 1.0]
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step
```

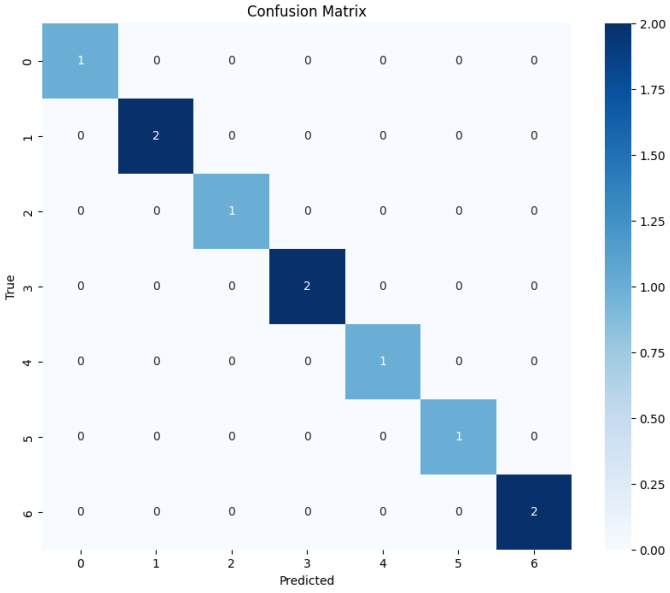
```
pred_y = np.argmax(proba_y[0], axis=1)

print(pred_y)
```

```
[7 2 1 0 4 1 4 9 5 9]
```

```
from sklearn.metrics import confusion_matrix
conf = confusion_matrix(pred_y, Y_test[:10])
print(conf)
```

```
[[1 0 0 0 0 0 0]
 [0 2 0 0 0 0 0]
 [0 0 1 0 0 0 0]
 [0 0 0 2 0 0 0]
 [0 0 0 0 1 0 0]
 [0 0 0 0 0 1 0]
 [0 0 0 0 0 0 2]]
```

```
import seaborn as sns
plt.figure(figsize=(10, 8))
sns.heatmap(conf, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```



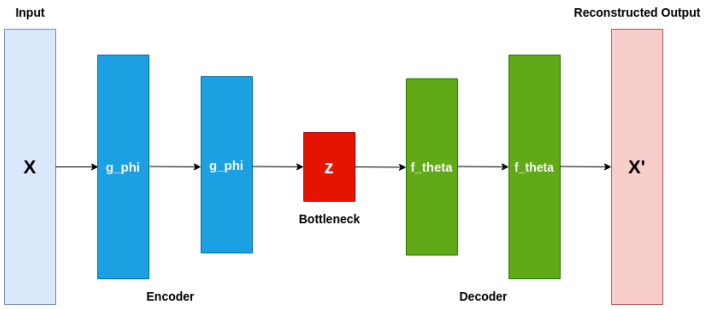
AUTOENCODERS

Autoencoders, a fascinating class of neural networks used for unsupervised learning.

1. **What Are Autoencoders?**
    - An **autoencoder** is an artificial neural network designed to learn efficient representations (encodings) of unlabelled data.
    - It consists of two main parts:
        - **Encoder**: Transforms the input data into a compact representation (latent space).
        - **Decoder**: Reconstructs the input data from the encoded representation.
    - Autoencoders aim to capture essential features while minimizing information loss.
2. **Purpose and Applications**:
    - **Dimensionality Reduction**: Autoencoders are used for reducing the dimensionality of data.
    - **Feature Learning**: They learn useful features for subsequent tasks (e.g., classification).
    - **Generative Models**: Variants like variational autoencoders (VAEs) can generate new data like the training samples.

3. **Types of Autoencoders**:
    - **Regularized Autoencoders**:
        - **Sparse Autoencoders**: Encourage sparsity in the encoded representation.
        - **Denoising Autoencoders**: Train on noisy input to improve robustness.
        - **Contractive Autoencoders**: Penalize sensitivity to small input perturbations.
    - **Variational Autoencoders (VAEs)**:
        - Probabilistic approach that models the latent space as a distribution.
        - Enables generative capabilities.
4. **Use Cases**:
    - **Facial Recognition**: Learning compact face representations.
    - **Anomaly Detection**: Identifying unusual patterns.
    - **Word Embeddings**: Capturing word meanings.
    - **Data Compression**: Efficiently storing data.

Autoencoders are neural network-based models that are used for unsupervised learning purposes to discover underlying correlations among data and represent data in a smaller dimension. The autoencoders frame unsupervised learning problems as supervised learning problems to train a neural network model. The input only is passed a the output. The input is squeezed down o a lower encoded representation using an encoder network, then a decoder network decodes the encoding to recreate back the input.



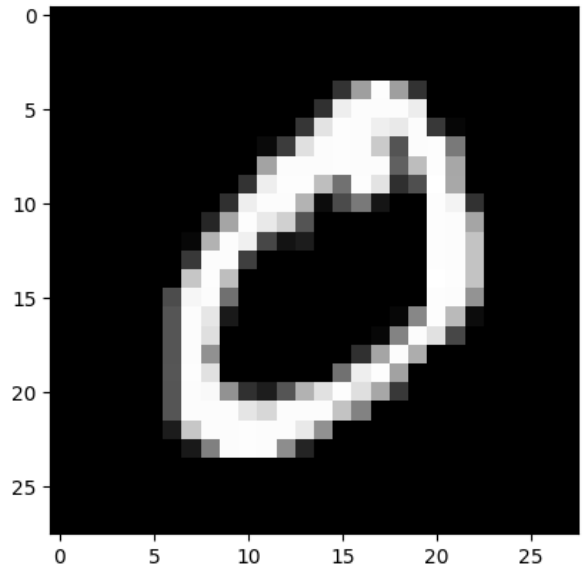
The encoding produced by the encoder layer has a lower-dimensional representation of the data and shows several interesting complex relationships among data.

An Autoencoder has the following parts:

1. **Encoder:** The encoder is the part of the network which takes in the input and produces a lower Dimensional encoding
2. **Bottleneck:** It is the lower dimensional hidden layer where the encoding is produced. The bottleneck layer has a lower number of nodes and the number of nodes in the bottleneck layer also gives the dimension of the encoding of the input.
3. **Decoder:** The decoder takes in the encoding and recreates back the input.

UNDERCOMPLETE AUTOENCODER

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import *
from tensorflow.keras.datasets import mnist
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
X_train = X_train.astype('float32')/255.0
X_test = X_test.astype('float32')/255.0
plt.imshow(X_train[1], cmap='gray')
plt.show()
```




```
X_valid = X_train[:10000]
X_train = X_train[10000:]
from tensorflow.keras.models import Sequential
autoencoder = Sequential()
autoencoder.add(Dense(32, activation='relu', input_shape = (784,)))
autoencoder.add(Dense(784, activation='sigmoid'))
encoder = Sequential()
encoder.add(Dense(32, activation='relu', input_shape = (784,)))
Decoder = Sequential()
Decoder.add(Dense(784, activation='sigmoid', input_shape = (32,)))
autoencoder.summary()
encoder.summary()
Decoder.summary()
```

```
| Layer (type)                    | Output Shape              |            Param # |
|---------------------------------|---------------------------|--------------------|
| dense (Dense)                   | (None, 32)                |             25,120 |
| dense_1 (Dense)                 | (None, 784)               |             25,872 |

 Total params: 50,992 (199.19 KB)
 Trainable params: 50,992 (199.19 KB)
 Non-trainable params: 0 (0.00 B)
Model: "sequential_1"

| Layer (type)                    | Output Shape              |            Param # |
|---------------------------------|---------------------------|--------------------|
| dense_2 (Dense)                 | (None, 32)                |             25,120 |

 Total params: 25,120 (98.12 KB)
 Trainable params: 25,120 (98.12 KB)
 Non-trainable params: 0 (0.00 B)
Model: "sequential_2"

| Layer (type)                    | Output Shape              |            Param # |
|---------------------------------|---------------------------|--------------------|
| dense_3 (Dense)                 | (None, 784)               |             25,872 |

 Total params: 25,872 (101.06 KB)
 Trainable params: 25,872 (101.06 KB)
 Non-trainable params: 0 (0.00 B)
```

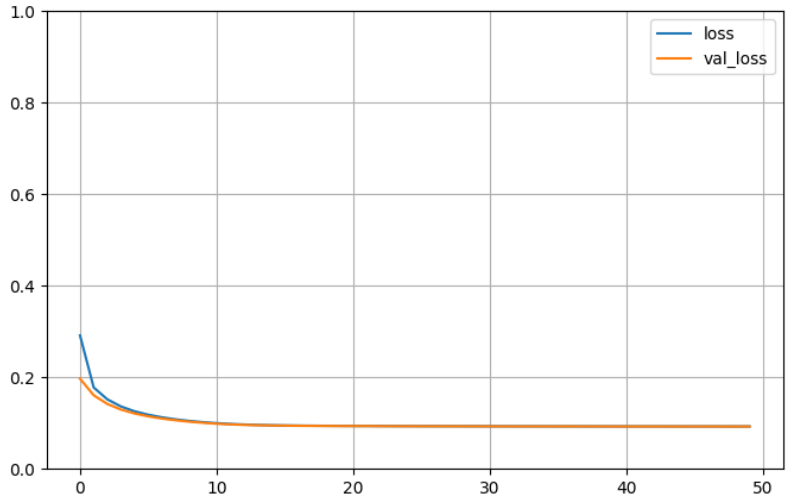
```
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
encoder.compile(optimizer='adam', loss='binary_crossentropy')
Decoder.compile(optimizer='adam', loss='binary_crossentropy')
from tensorflow.keras.callbacks import EarlyStopping

early_stopping = EarlyStopping(patience = 3, mode = 'min', monitor = 'loss')
history = autoencoder.fit(X_train, X_train, epochs = 50, batch_size = 256, shuffle = True,
validation_data = (X_valid, X_valid), callbacks = [early_stopping])
```

```
196/196 ━━━━━━━━━━━━━━━━━━━━ 4s 20ms/step - loss: 0.0930 - val_loss: 0.0929
Epoch 40/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 10ms/step - loss: 0.0931 - val_loss: 0.0928
Epoch 41/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step - loss: 0.0927 - val_loss: 0.0928
Epoch 42/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 10ms/step - loss: 0.0930 - val_loss: 0.0927
Epoch 43/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 3s 14ms/step - loss: 0.0930 - val_loss: 0.0928
Epoch 44/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - loss: 0.0928 - val_loss: 0.0927
Epoch 45/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - loss: 0.0929 - val_loss: 0.0927
Epoch 46/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 10ms/step - loss: 0.0928 - val_loss: 0.0927
Epoch 47/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step - loss: 0.0928 - val_loss: 0.0928
Epoch 48/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 3s 15ms/step - loss: 0.0928 - val_loss: 0.0927
Epoch 49/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 0.0926 - val_loss: 0.0927
Epoch 50/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.0927 - val_loss: 0.0926
```

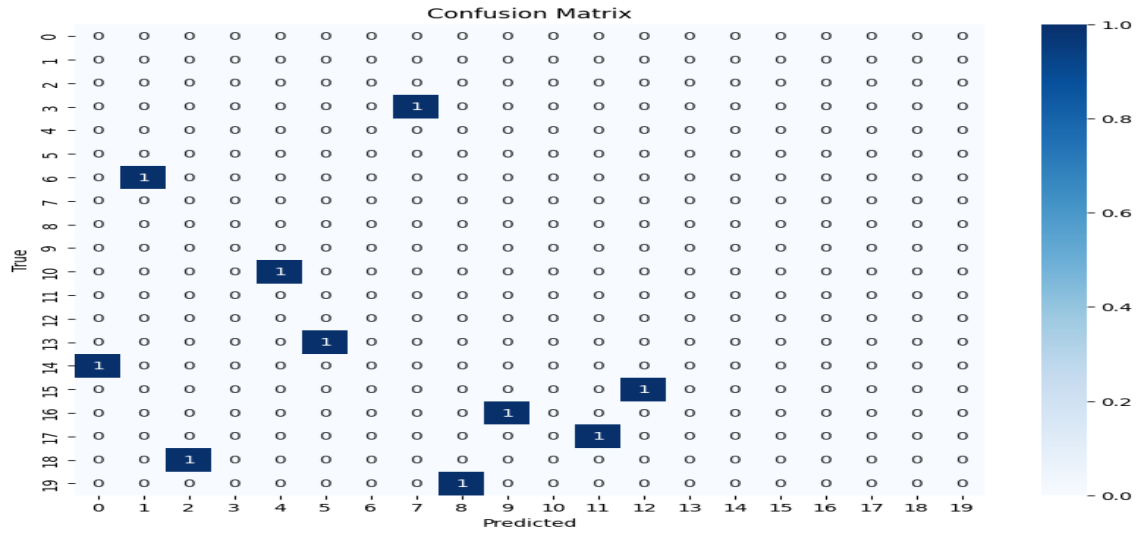
```
loss = autoencoder.evaluate(X_test, X_test)
print(loss)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 0.0922
0.09157735109329224
```

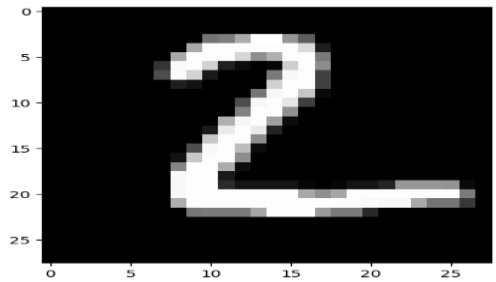
```
y = np.argmax(y_proba, axis=1)
y_true = np.argmax(X_test[:10], axis=1)
pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()
```



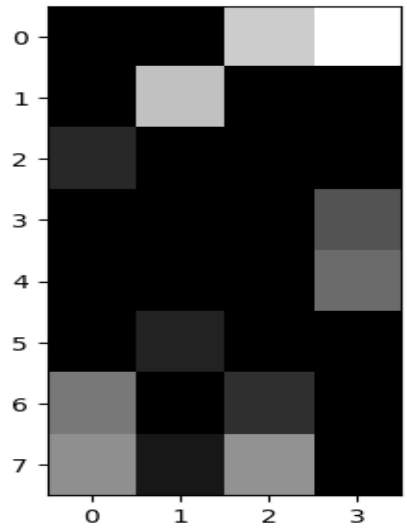

```
import sklearn
from sklearn.metrics import confusion_matrix
conf = confusion_matrix(y, y_true)
import seaborn as sns
plt.figure(figsize=(10, 8))
sns.heatmap(conf, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```



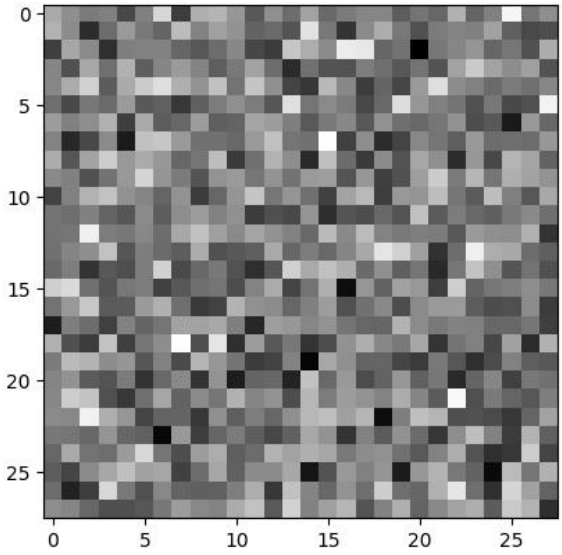

```
plt.imshow(X_test[1].reshape(28, 28), cmap = 'gray')
```



```
y = encoder.predict(X_test[1].reshape(1,-1))
plt.imshow(y.reshape(8,4), cmap='gray')
plt.show()
```




```
y1 = Decoder.predict(y)
plt.imshow(y1.reshape(28,28), cmap='gray')
plt.show()
```



COMPLEX UNDERCOMPLETE AUTOENCODER

CODE

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import *
from tensorflow.keras.datasets import mnist
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
X_train = X_train.astype('float32')/255.0
X_test = X_test.astype('float32')/255.0
X_valid = X_train[:10000]
X_train = X_train[10000:]
X_train = X_train.reshape(len(X_train), X_train[1].size)
X_valid = X_valid.reshape(len(X_valid), X_valid[1].size)
X_test = X_test.reshape(len(X_test), X_test[1].size)
from tensorflow.keras.models import Model

input_l=Input(shape=(784,))
encoding_1=Dense(256, activation='relu')(input_l)
encoding_2=Dense(128, activation='relu')(encoding_1)
bottleneck=Dense(32, activation='relu')(encoding_2)
decoding_1=Dense(128, activation='relu')(bottleneck)
decoding_2=Dense(256, activation='relu')(decoding_1)
output_l=Dense(784, activation='sigmoid')(decoding_2)
autoencoder=Model(inputs=[input_l],outputs=[output_l])
encoder=Model(inputs=[input_l],outputs=[bottleneck])
encoded_input=Input(shape=(32,))
decoded_layer_1=autoencoder.layers[-3](encoded_input)
decoded_layer_2=autoencoder.layers[-2](decoded_layer_1)
decoded=autoencoder.layers[-1](decoded_layer_2)
decoder=Model(inputs=[encoded_input],outputs=[decoded])
autoencoder.summary()
```

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 784)	0
dense (Dense)	(None, 256)	200,960
dense_1 (Dense)	(None, 128)	32,896
dense_2 (Dense)	(None, 32)	4,128
dense_3 (Dense)	(None, 128)	4,224
dense_4 (Dense)	(None, 256)	33,024
dense_5 (Dense)	(None, 784)	201,488

Total params: 476,720 (1.82 MB)
Trainable params: 476,720 (1.82 MB)
Non-trainable params: 0 (0.00 B)

```
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
history = autoencoder.fit(X_train, X_train, epochs=50, batch_size=256, shuffle=True,
validation_data=(X_test, X_test))
```

```
Epoch 46/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 10s 26ms/step - loss: 0.0806 - val_loss: 0.0809
Epoch 47/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 6s 30ms/step - loss: 0.0805 - val_loss: 0.0812
Epoch 48/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 9s 26ms/step - loss: 0.0804 - val_loss: 0.0811
Epoch 49/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 7s 34ms/step - loss: 0.0802 - val_loss: 0.0807
Epoch 50/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 9s 28ms/step - loss: 0.0802 - val_loss: 0.0805
```

loss = autoencoder.evaluate(X_test, X_test)

print(loss)

```
313/313 ━━━━━━━━━━━━━━━━━━━━
0.08050917088985443
```

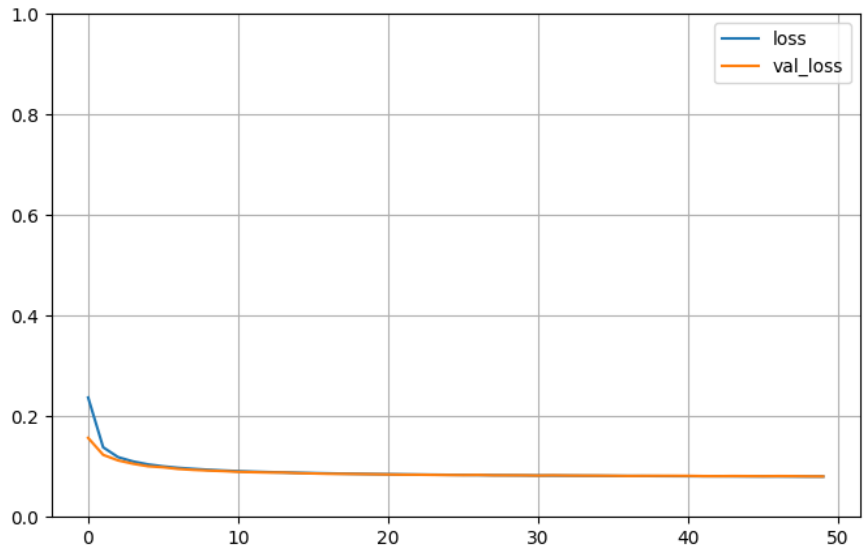
y_proba = autoencoder.predict(X_test[0:1])

pd.DataFrame(history.history).plot(figsize=(8, 5))

plt.grid(True)

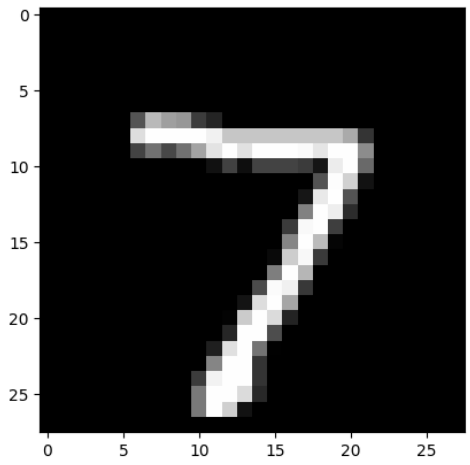
plt.gca().set_ylim(0, 1)

plt.show()

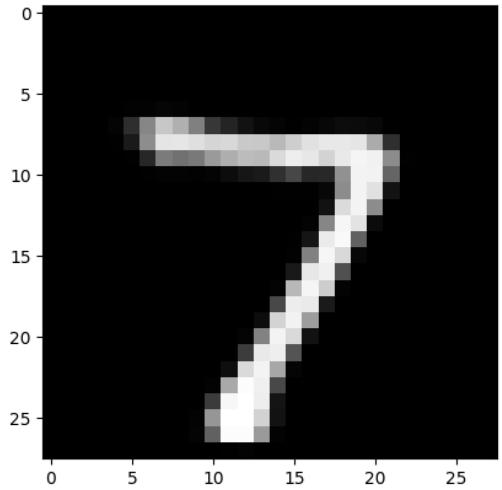


plt.imshow(X_test[0:1].reshape(28,28), cmap='gray')

plt.show()



```
plt.imshow(y_proba.reshape(28,28), cmap='gray')
plt.show()
```



SPARSE AUTOENCODERS

CODE

```
import tensorflow as tf
import tensorflow.keras as keras
from tensorflow.keras.layers import *
from tensorflow.keras.models import Sequential
from tensorflow.keras.regularizers import l1
from tensorflow.keras.datasets import mnist

(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
X_valid = X_train[50000:]
Y_valid = Y_train[50000:]
X_train = X_train[:50000]
Y_train = Y_train[:50000]
X_train, X_valid, X_test = X_train.reshape(-1, X_train[0].size), X_valid.reshape(-1,
X_valid[0].size), X_test.reshape(-1, X_test[0].size)
X_train = X_train.astype('float32') / 255
X_valid = X_valid.astype('float32') / 255
X_test = X_test.astype('float32') / 255
from tensorflow.keras.models import Model
input_l=Input(shape=(784,))
encoding_1=Dense(256, activation='relu', activity_regularizer=l1(0.001))(input_l)
bottleneck=Dense(32, activation='relu', activity_regularizer=l1(0.001))(encoding_1)
decoding_1=Dense(256, activation='relu', activity_regularizer=l1(0.001))(bottleneck)
output_l=Dense(784, activation='sigmoid')(decoding_1)
autoencoder=Model(inputs=[input_l],outputs=[output_l])
```

```
encoder=Model(inputs=[input_l],outputs=[bottleneck])
encoded_input=Input(shape=(32,))
decoded_layer_2=autoencoder.layers[-2](encoded_input)
decoded=autoencoder.layers[-1](decoded_layer_2)
decoder=Model(inputs=[encoded_input],outputs=[decoded])
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.summary()
```

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 784)	0
dense_4 (Dense)	(None, 256)	200,960
dense_5 (Dense)	(None, 32)	8,224
dense_6 (Dense)	(None, 256)	8,448
dense_7 (Dense)	(None, 784)	201,488

Total params: 419,120 (1.60 MB)
Trainable params: 419,120 (1.60 MB)
Non-trainable params: 0 (0.00 B)

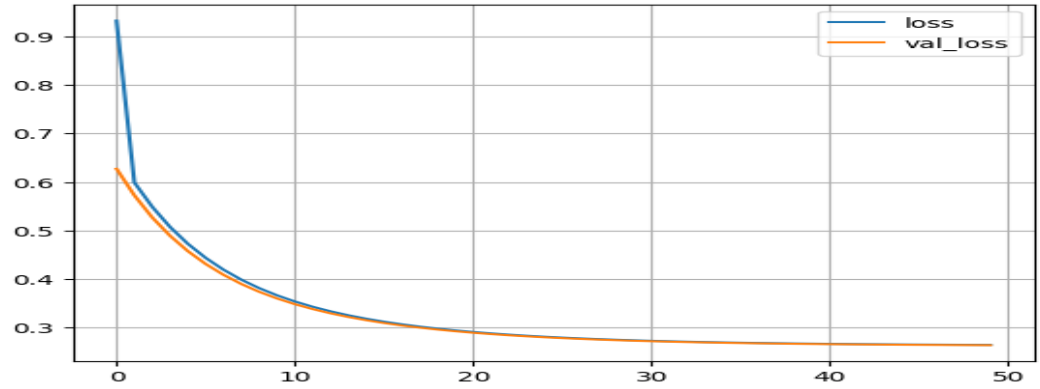
```
autoencoder.compile(optimizer = 'adam', loss = 'binary_crossentropy')
history = autoencoder.fit(X_train, X_train,
        epochs=50,
        batch_size=256,
        shuffle=True,
        validation_data=(X_test, X_test))
```

```
Epoch 47/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 5s 25ms/step - loss: 0.2650 - val_loss: 0.2646
Epoch 48/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 5s 24ms/step - loss: 0.2648 - val_loss: 0.2645
Epoch 49/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 6s 32ms/step - loss: 0.2650 - val_loss: 0.2643
Epoch 50/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 5s 25ms/step - loss: 0.2651 - val_loss: 0.2641
```

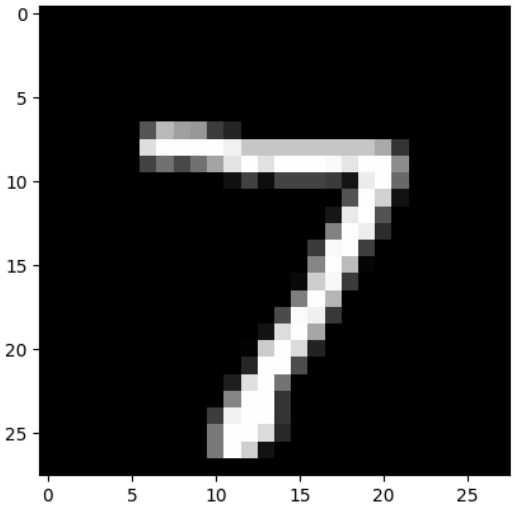
```
loss = autoencoder.evaluate(X_test, X_test)
print(loss)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.2584
0.26406943798065186
```

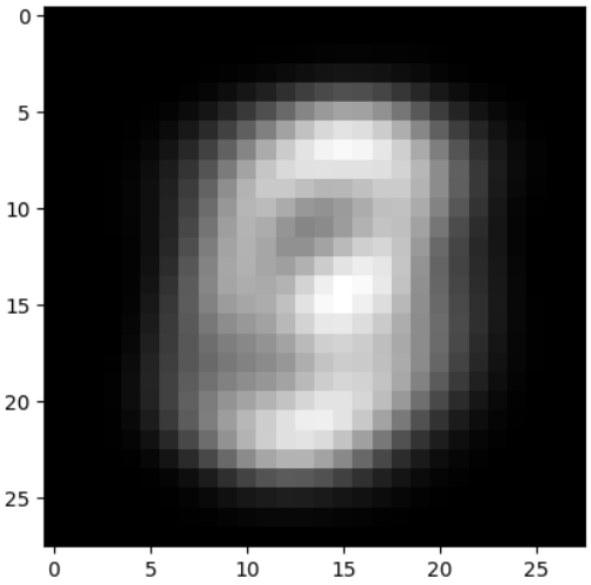
```
import pandas as pd
pd.DataFrame(history.history).plot()
import matplotlib.pyplot as plt
plt.grid(True)
plt.show()
```




```
y = autoencoder.predict(X_test[0].reshape(1, -1))

plt.imshow(X_test[0].reshape(28, 28), cmap = 'gray')
plt.show()
```




```
plt.imshow(y.reshape(28, 28), cmap = 'gray')
plt.show()
```



DENOISING AUTOENCODER

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.layers import *
from tensorflow.keras.models import Sequential
from tensorflow.keras.datasets import fashion_mnist
import numpy as np
import matplotlib.pyplot as plt

fashion_mnist = keras.datasets.fashion_mnist
(train, _), (test, _) = fashion_mnist.load_data()
train = train[10000:]
valid = train[:10000]
r = Sequential()
r.add(Rescaling(1/255.0))
train = r(train)
valid = r(valid)
test = r(test)
train = np.asarray(train).astype('float32')
valid = np.asarray(valid).astype('float32')
test = np.asarray(test).astype('float32')
train = train.reshape(-1, 28, 28, 1)
valid = valid.reshape(-1, 28, 28, 1)
test = test.reshape(-1, 28, 28, 1)
print(train.shape)
autoencoder = Sequential()
autoencoder.add(Conv2D(32, (3,3), activation='relu',padding='same', input_shape = (28,
28, 1)))
autoencoder.add(MaxPooling2D((2,2), padding='same'))
autoencoder.add(Conv2D(16, (3,3), activation='relu',padding='same'))
autoencoder.add(MaxPooling2D((2,2), padding='same'))
autoencoder.add(Conv2D(8, (3,3), activation='relu',padding='same'))
autoencoder.add(MaxPooling2D((2,2), padding='same'))
autoencoder.add(Conv2D(8, (3,3), activation='relu', padding='same'))
autoencoder.add(UpSampling2D((2,2)))
autoencoder.add(Conv2D(16, (3,3), activation='relu', padding='same'))
autoencoder.add(UpSampling2D((2,2)))
autoencoder.add(Conv2D(32, (3,3), activation='relu'))
autoencoder.add(UpSampling2D((2,2)))
autoencoder.add(Conv2D(1, (3,3), activation='sigmoid', padding='same'))
```

autoencoder.summary()

Model: "sequential_2"

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 28, 28, 32)	320
max_pooling2d_1 (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_2 (Conv2D)	(None, 14, 14, 16)	4,624
max_pooling2d_2 (MaxPooling2D)	(None, 7, 7, 16)	0
conv2d_3 (Conv2D)	(None, 7, 7, 8)	1,160
max_pooling2d_3 (MaxPooling2D)	(None, 4, 4, 8)	0
conv2d_4 (Conv2D)	(None, 4, 4, 8)	584
up_sampling2d (UpSampling2D)	(None, 8, 8, 8)	0
conv2d_5 (Conv2D)	(None, 8, 8, 16)	1,168
up_sampling2d_1 (UpSampling2D)	(None, 16, 16, 16)	0
conv2d_6 (Conv2D)	(None, 14, 14, 32)	4,640
up_sampling2d_2 (UpSampling2D)	(None, 28, 28, 32)	0
conv2d_7 (Conv2D)	(None, 28, 28, 1)	289

Total params: 12,785 (49.94 KB)
Trainable params: 12,785 (49.94 KB)
Non-trainable params: 0 (0.00 B)

autoencoder.compile(optimizer = 'adam', loss = 'binary_crossentropy')

from tensorflow.keras.callbacks import EarlyStopping

early_stopping = EarlyStopping(monitor = 'val_loss', patience = 5)

history = autoencoder.fit(train, train, epochs = 50, validation_data = (valid, valid),
callbacks = [early_stopping])

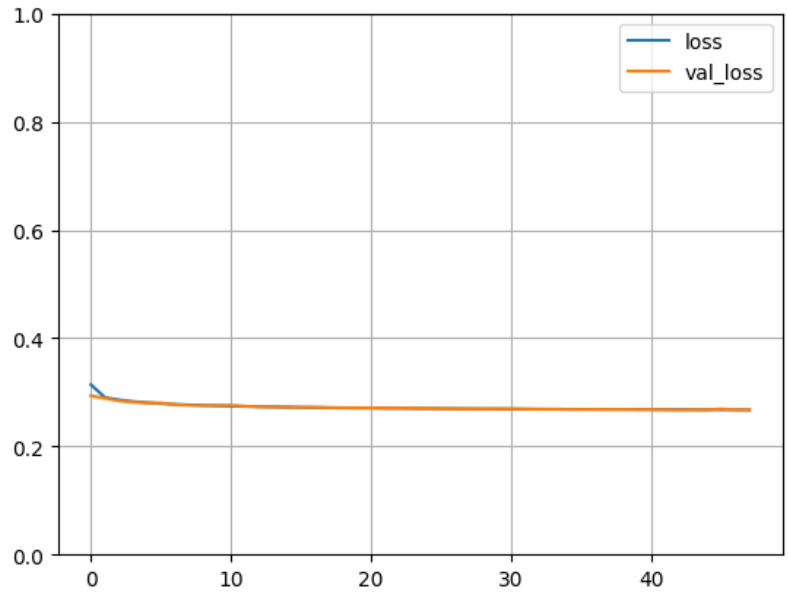
```
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 139s 69ms/step - loss: 0.2674 - val_loss: 0.2681
Epoch 42/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 142s 69ms/step - loss: 0.2670 - val_loss: 0.2678
Epoch 43/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 142s 69ms/step - loss: 0.2681 - val_loss: 0.2676
Epoch 44/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 112s 72ms/step - loss: 0.2673 - val_loss: 0.2677
Epoch 45/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 140s 71ms/step - loss: 0.2679 - val_loss: 0.2677
Epoch 46/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 141s 70ms/step - loss: 0.2674 - val_loss: 0.2693
Epoch 47/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 141s 70ms/step - loss: 0.2676 - val_loss: 0.2678
Epoch 48/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 142s 70ms/step - loss: 0.2671 - val_loss: 0.2678
```

loss = autoencoder.evaluate(test, test)

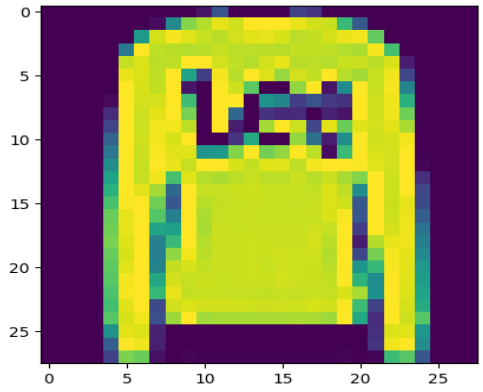
print(loss)

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 6s 18ms/step - loss: 0.2694
0.26953133940696716
```

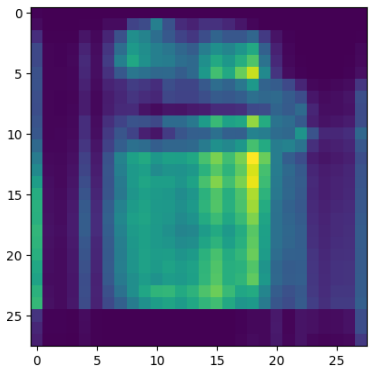
```
import pandas as pd
pd.DataFrame(history.history).plot()
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()
```




```
y = autoencoder.predict(test[1])
plt.imshow(test[1])
plt.show()
```




```
plt.imshow(y[:, :, 0, 0])
plt.show()
```



OPTIMIZING THE AUTOENCODERS

To make an autoencoder that can learn and extract representations and also reconstruct the input, we need to:

Increase model capacity,

Increase information bottleneck capacity,

Pros:

Since the Undercomplete autoencoders maximize the probability distribution, they do not need a regularization function.

Cons:

Undercomplete autoencoders are not versatile and they tend to overfit. One of the reasons why it overfits is because it is a simple model with a limited capacity which does not allow it to be flexible.

Regularised autoencoders are designed based on data complexity, and they address the problems of Undercomplete autoencoders. The encoder and decoder, along with the information bottleneck, can have a higher capacity. This makes them more flexible and powerful.

Regularised autoencoders use a loss function for properties like:

The ability to reconstruct the output from the input through approximation.

The sparsity of representation.

The smallness of the derivative of the representation.

Robustness to noise, outliers, or missing inputs.

ADVERSIAL ATTACKS

Adversarial attacks refer to deliberate attempts by malicious actors to subvert the functionality of machine learning models. These attacks exploit vulnerabilities in the models, causing them to make incorrect or unintended predictions or decisions. They can take various forms:

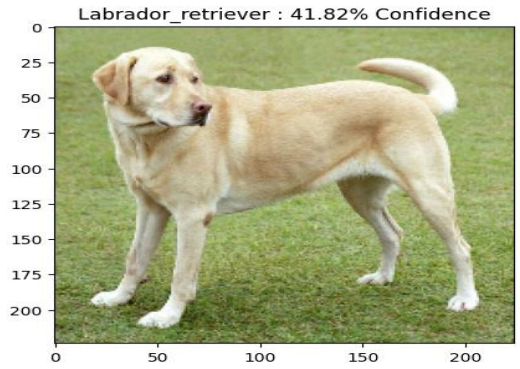
1. **Evasion Attacks**:
    - Adversaries manipulate input data to deceive the model during prediction.
    - Examples include adding subtle perturbations to images or text to mislead the classifier.
2. **Data Poisoning Attacks**:
    - Adversaries inject malicious data into the training set.
    - The poisoned data biases the model's learning process, leading to incorrect predictions.
3. **Byzantine Attacks**:

- In distributed systems, Byzantine adversaries intentionally provide incorrect information.
- These attacks disrupt consensus algorithms or decision-making processes.

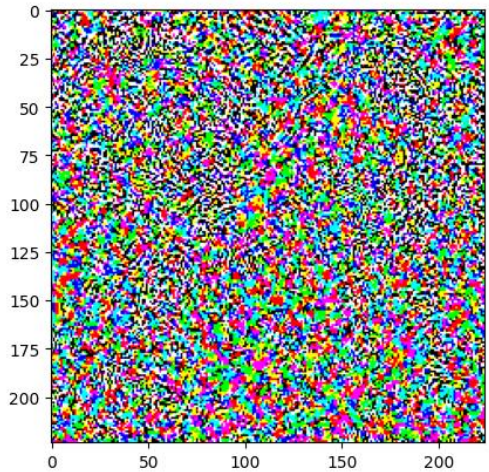
4. **Model Extraction**:
   - Adversaries create surrogate models to approximate the target model.
   - These surrogates are then used to infer sensitive information about the original model.

CODE

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

pretrained_model = tf.keras.applications.MobileNetV2(include_top=True,
weights='imagenet')
pretrained_model.trainable = False
decode_predictions = tf.keras.applications.mobilenet_v2.decode_predictions
def preprocess(image):
  image = tf.cast(image, tf.float32)
  image = tf.image.resize(image, (224, 224))
  image = tf.keras.applications.mobilenet_v2.preprocess_input(image)
  image = image[None, ...]
  return image

def get_imagenet_label(probs):
  return decode_predictions(probs, top=1)[0][0]

image_path = tf.keras.utils.get_file('YellowLabradorLooking_new.jpg',
'https://storage.googleapis.com/download.tensorflow.org/example_images/YellowLabr
adorLooking_new.jpg')
image_raw = tf.io.read_file(image_path)
image = tf.image.decode_image(image_raw)

image = preprocess(image)
image_probs = pretrained_model.predict(image)

plt.figure()
plt.imshow(image[0] * 0.5 + 0.5)
_, image_class, class_confidence = get_imagenet_label(image_probs)
plt.title('{} : {:.2f}% Confidence'.format(image_class, class_confidence*100))
plt.show()
```

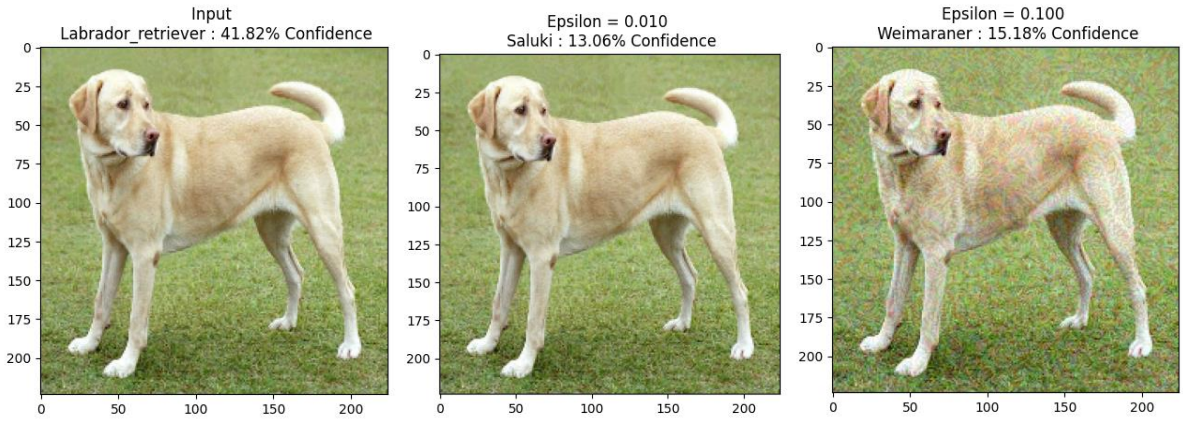



```
loss_object = tf.keras.losses.CategoricalCrossentropy()

def create_adversarial_pattern(input_image, input_label):
  with tf.GradientTape() as tape:
    tape.watch(input_image)
    prediction = pretrained_model(input_image)
    loss = loss_object(input_label, prediction)

  # Get the gradients of the loss w.r.t to the input image.
  gradient = tape.gradient(loss, input_image)
  # Get the sign of the gradients to create the perturbation
  signed_grad = tf.sign(gradient)
  return signed_grad

labrador_retriever_index = 208
label = tf.one_hot(labrador_retriever_index, image_probs.shape[-1])
label = tf.reshape(label, (1, image_probs.shape[-1]))

perturbations = create_adversarial_pattern(image, label)
plt.imshow(perturbations[0] * 0.5 + 0.5);
```




```
def display_images(image, description):
```

```
  _, label, confidence = get_imagenet_label(pretrained_model.predict(image))
  plt.figure()
  plt.imshow(image[0]*0.5+0.5)
  plt.title('{} \n {} : {:.2f}% Confidence'.format(description,
                                    label, confidence*100))
  plt.show()
epsilons = [0, 0.01, 0.1, 0.15]
descriptions = [('Epsilon = {:0.3f}'.format(eps) if eps else 'Input')
          for eps in epsilons]

for i, eps in enumerate(epsilons):
  adv_x = image + eps*perturbations
  adv_x = tf.clip_by_value(adv_x, -1, 1)
  display_images(adv_x, descriptions[i])
```



CASE STUDY FOR ECG SIGNAL ANALYSIS

- The project titled **"Detecting Anomaly in ECG Data Using AutoEncoder with TensorFlow"** aims to enhance cardiac health monitoring through real-time anomaly detection in ECG signals.
- Data is trained on normal signal and verified for anomalous.
- Key components of the project:
    - **UnderComplete deep Autoencoder**: Utilizes an architecture for sequence modelling and an autoencoder for feature extraction.
    - **TensorFlow Framework**: Employs TensorFlow for both training and evaluating the model.
    - **Objective**: Develop a model capable of accurately identifying irregularities in ECG data as they occur.

Procedures:

1. **Data Preprocessing**:
    - Load ECG data from a CSV file.
    - Standardize the data using StandardScaler.
    - Split the data into training and test sets.

2. **Autoencoder Model**:
    - Create an autoencoder model with an encoder and a decoder.
    - The encoder reduces the input dimensionality, capturing essential features.
    - The decoder reconstructs the input from the encoded representation.
    - Train the model using normal ECG data.
3. **Training and Validation**:
    - Monitor training and validation loss to assess model performance.
    - Visualize the reconstruction of normal ECG signals.
4. **Anomaly Detection**:
    - Compute the mean absolute error (MAE) between reconstructed and original normal ECG data.
    - Set a threshold based on the MAE distribution.
    - Evaluate the model's performance on anomalous ECG data.
5. **Results and Insights**:
    - The model successfully detects anomalies in ECG signals.
    - Anomalies can be identified based on the MAE exceeding the threshold.

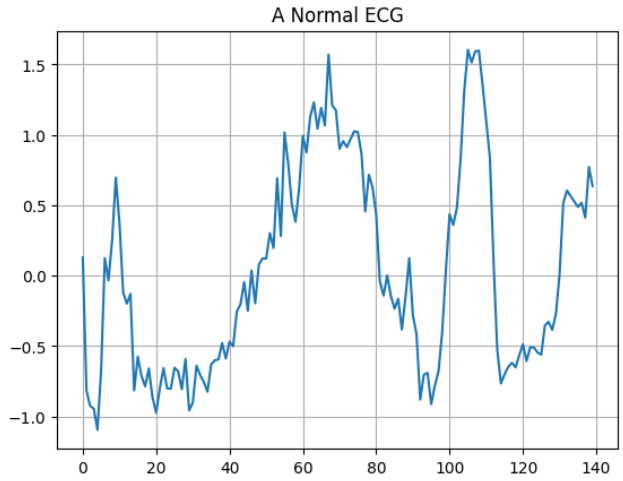
CODE

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.layers import *
from tensorflow.keras.models import Sequential
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
dataframe =
pd.read_csv('http://storage.googleapis.com/download.tensorflow.org/data/ecg.csv',
header=None)
raw_data = dataframe.values
dataframe.head()
```

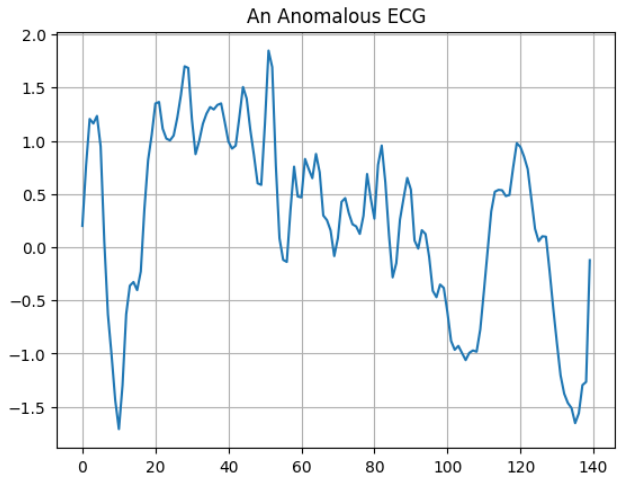
	0	1	2	3	4	5	6	7	8	9	...	131	132	
0	-0.112522	-2.827204	-3.773897	-4.349751	-4.376041	-3.474986	-2.181408	-1.818286	-1.250522	-0.477492	...	0.792168	0.933541	0.
1	-1.100878	-3.996840	-4.285843	-4.506579	-4.022377	-3.234368	-1.566126	-0.992258	-0.754680	0.042321	...	0.538356	0.656881	0.
2	-0.567088	-2.593450	-3.874230	-4.584095	-4.187449	-3.151462	-1.742940	-1.490659	-1.183580	-0.394229	...	0.886073	0.531452	0.
3	0.490473	-1.914407	-3.616364	-4.318823	-4.268016	-3.881110	-2.993280	-1.671131	-1.333884	-0.965629	...	0.350816	0.499111	0.
4	0.800232	-0.874252	-2.384761	-3.973292	-4.338224	-3.802422	-2.534510	-1.783423	-1.594450	-0.753199	...	1.148884	0.958434	1.

```
X = raw_data[:, 0:-1]
Y = raw_data[:, -1]
from sklearn.preprocessing import StandardScaler
Scale = StandardScaler()
X = Scale.fit_transform(X)
```

```
X = tf.cast(X, tf.float32)
Y = Y.astype(bool)
normal_data = X[Y]
anomalous_data = X[~Y]
plt.grid()
plt.plot(np.arange(140), normal_data[0])
plt.title("A Normal ECG")
plt.show()
```




```
plt.grid()
plt.plot(np.arange(140), anomalous_data[0])
plt.title("An Anomalous ECG")
plt.show()
```

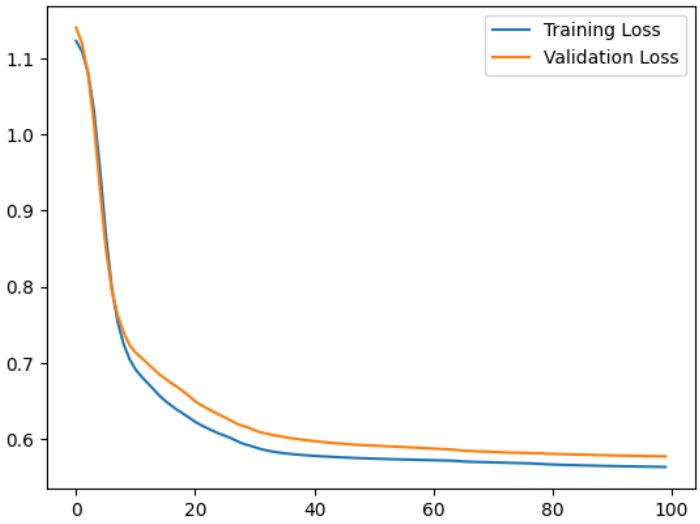



```
model = Sequential()
model.add(layers.Dense(32, activation="relu"))
model.add(layers.Dense(16, activation="relu"))
model.add(layers.Dense(8, activation="relu"))
model.add(layers.Dense(16, activation="relu"))
model.add(layers.Dense(32, activation="relu"))
model.add(layers.Dense(140, activation="sigmoid"))
```

```
model.compile(optimizer = 'adam', loss = 'mse')
X_valid = X[:100]
X_test = X[100:200]
history = model.fit(normal_data, normal_data, epochs = 100, batch_size = 512,
validation_data = (X_valid, X_valid))
```

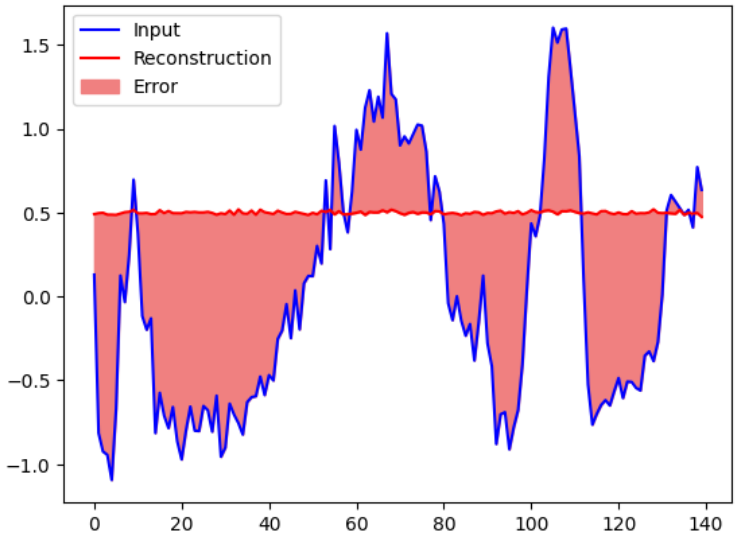
```
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: 0.5601 - val_loss: 0.5780
Epoch 96/100
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - loss: 0.5624 - val_loss: 0.5779
Epoch 97/100
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - loss: 0.5634 - val_loss: 0.5776
Epoch 98/100
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - loss: 0.5641 - val_loss: 0.5777
Epoch 99/100
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: 0.5755 - val_loss: 0.5774
Epoch 100/100
6/6 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: 0.5502 - val_loss: 0.5774
```

```
plt.plot(history.history["loss"], label="Training Loss")
plt.plot(history.history["val_loss"], label="Validation Loss")
plt.legend()
plt.show()
```

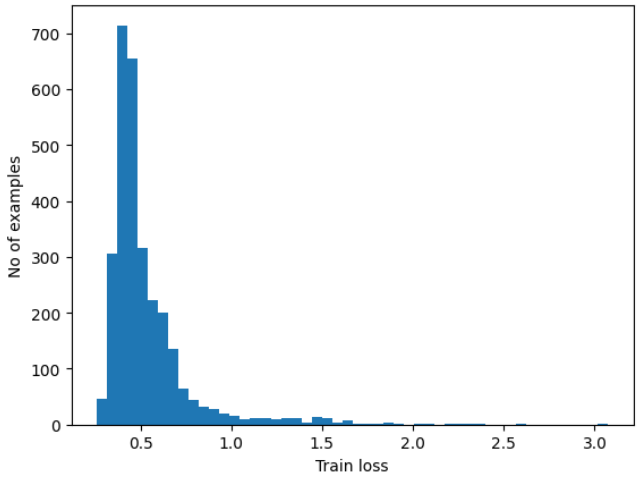



```
encode = Sequential()
encode.add(layers.Dense(32, activation="relu"))
encode.add(layers.Dense(16, activation="relu"))
encode.add(layers.Dense(8, activation="relu"))

decode = Sequential()
decode.add(layers.Dense(16, activation="relu"))
decode.add(layers.Dense(32, activation="relu"))
decode.add(layers.Dense(140, activation="sigmoid"))
X_normal = normal_data[:100]
e = encode(X_normal).numpy()
d = decode(e).numpy()
plt.plot(normal_data[0], 'b')
```

```
plt.plot(d[0], 'r')
plt.fill_between(np.arange(140), d[0], normal_data[0], color='lightcoral')
plt.legend(labels=["Input", "Reconstruction", "Error"])
plt.show()
```




```
reconstructions = model.predict(normal_data)
train_loss = tf.keras.losses.mae(reconstructions, normal_data)

plt.hist(train_loss[None,:], bins=50)
plt.xlabel("Train loss")
plt.ylabel("No of examples")
plt.show()
```



IMPROVING THE MODEL ACCURACY & GENERATION

Convert the model from sequential to Model

```
import pandas as pd# Download the dataset
```

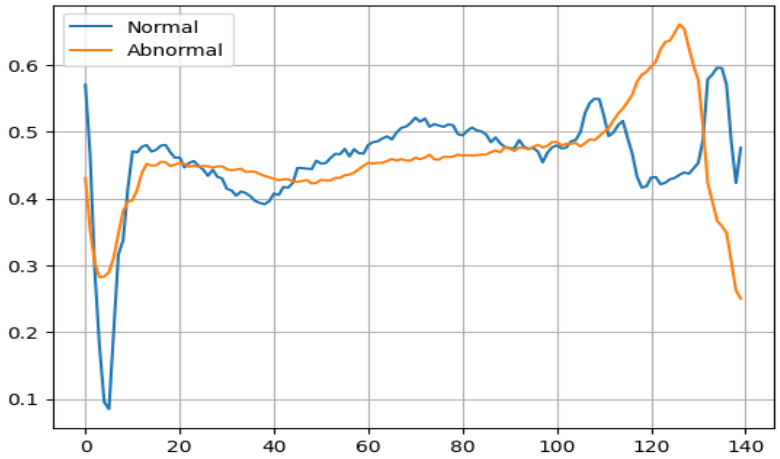
```
dataframe =
pd.read_csv('http://storage.googleapis.com/download.tensorflow.org/data/ecg.csv',
header=None)
raw_data = dataframe.values
dataframe.head()
```

	0	1	2	3	4	5	6	7	8	9	...	131	132	133	134	13
0	-0.112522	-2.827204	-3.773897	-4.349751	-4.376041	-3.474986	-2.181408	-1.818286	-1.250522	-0.477492	...	0.792168	0.933541	0.796958	0.578621	0.25774
1	-1.100878	-3.996840	-4.285843	-4.506579	-4.022377	-3.234368	-1.566126	-0.992258	-0.754680	0.042321	...	0.538356	0.656881	0.787490	0.724046	0.55578
2	-0.567088	-2.593450	-3.874230	-4.584095	-4.187449	-3.151462	-1.742940	-1.490659	-1.183580	-0.394229	...	0.886073	0.531452	0.311377	-0.021919	-0.71368
3	0.490473	-1.914407	-3.616364	-4.318823	-4.268016	-3.881110	-2.993280	-1.671131	-1.333884	-0.965629	...	0.350816	0.499111	0.600345	0.842069	0.95207
4	0.800232	-0.874252	-2.384761	-3.973292	-4.338224	-3.802422	-2.534510	-1.783423	-1.594450	-0.753199	...	1.148884	0.958434	1.059025	1.371682	1.27739

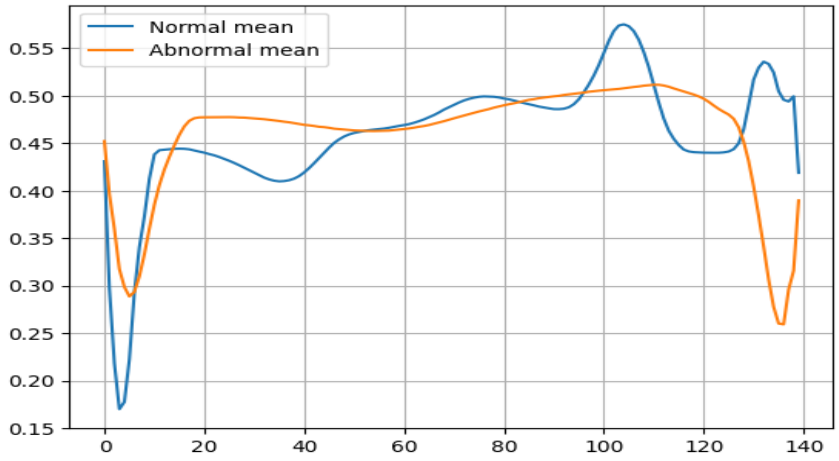
5 rows × 141 columns

```
from sklearn.model_selection import train_test_split
from tensorflow.keras import Model, layers
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import accuracy_score, precision_score, recall_score

labels = raw_data[:, -1]

# The other data points are the electrocadriogram data
data = raw_data[:, 0:-1]

train_data, test_data, train_labels, test_labels = train_test_split(
  data, labels, test_size=0.2, random_state=21
)
min_val = tf.reduce_min(train_data)
max_val = tf.reduce_max(train_data)

train_data = (train_data - min_val) / (max_val - min_val)
test_data = (test_data - min_val) / (max_val - min_val)

train_data = tf.cast(train_data, tf.float32)
test_data = tf.cast(test_data, tf.float32)
train_labels = train_labels.astype(bool)
test_labels = test_labels.astype(bool)

normal_train_data = train_data[train_labels]
normal_test_data = test_data[test_labels]

anomalous_train_data = train_data[~train_labels]
anomalous_test_data = test_data[~test_labels]
```

```
import matplotlib.pyplot as plt
plt.plot(normal_train_data[0], label="Normal")
plt.plot(anomalous_train_data[0], label="Abnormal")
plt.grid(True)

plt.legend()
plt.show()
```




```
plt.plot(normal_train_data.numpy().mean(axis = 0), label="Normal mean")
plt.plot(anomalous_train_data.numpy().mean(axis = 0), label="Abnormal mean")
plt.grid(True)
plt.legend()
plt.show()
```

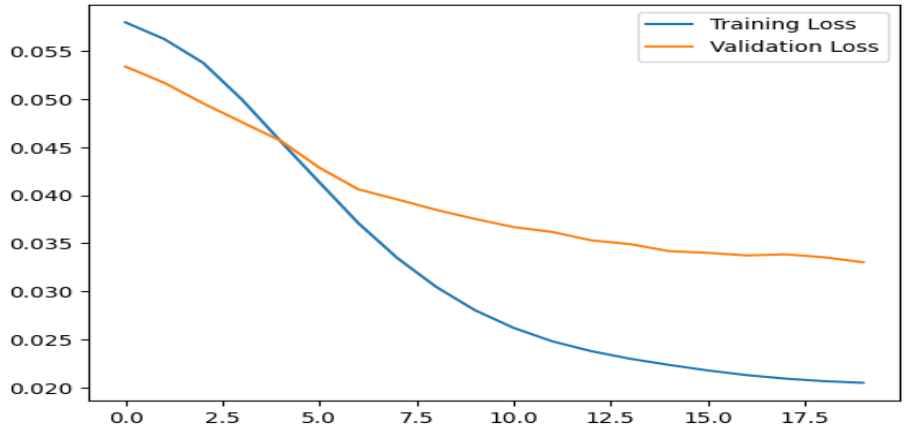



```
class AnomalyDetector(Model):
 def __init__(self):
  super(AnomalyDetector, self).__init__()
  self.encoder = tf.keras.Sequential([
   layers.Dense(32, activation="relu"),
   layers.Dense(16, activation="relu"),
   layers.Dense(8, activation="relu")])
```

```
    self.decoder = tf.keras.Sequential([
      layers.Dense(16, activation="relu"),
      layers.Dense(32, activation="relu"),
      layers.Dense(140, activation="sigmoid")])

  def call(self, x):
    encoded = self.encoder(x)
    decoded = self.decoder(encoded)
    return decoded

autoencoder = AnomalyDetector()
autoencoder.compile(optimizer='adam', loss='mae')
history = autoencoder.fit(normal_train_data, normal_train_data,
          epochs=20,
          batch_size=512,
          validation_data=(test_data, test_data),
          shuffle=True)
```

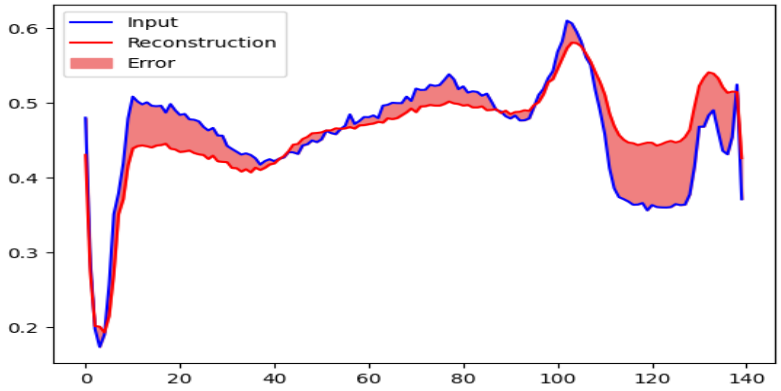
```
Epoch 17/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - loss: 0.0211 - val_loss: 0.0337
Epoch 18/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: 0.0210 - val_loss: 0.0338
Epoch 19/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - loss: 0.0207 - val_loss: 0.0335
Epoch 20/20
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - loss: 0.0205 - val_loss: 0.0330
```

```
plt.plot(history.history["loss"], label="Training Loss")
plt.plot(history.history["val_loss"], label="Validation Loss")
plt.legend()
```

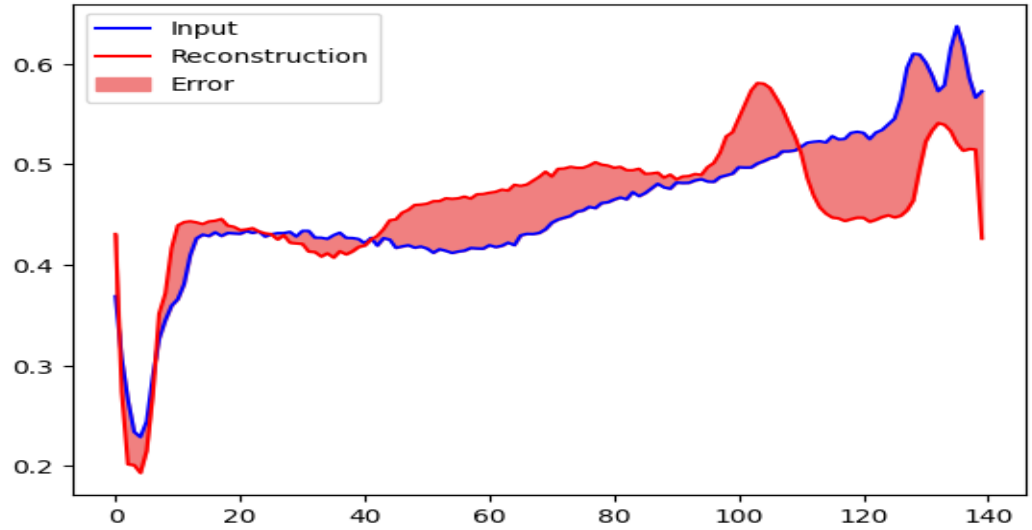



```
y = autoencoder.predict(normal_test_data[0:1])
y = y.flatten()
plt.plot(normal_test_data[0], 'b')
plt.plot(y, 'r')
```

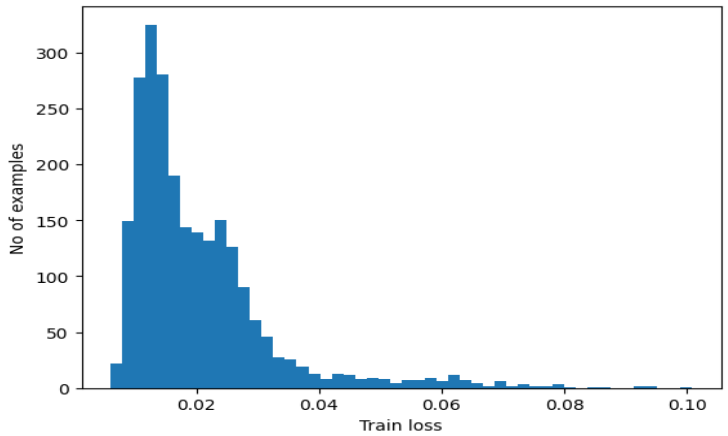
```
plt.fill_between(np.arange(140), y, normal_test_data[0], color='lightcoral')
plt.legend(labels=["Input", "Reconstruction", "Error"])
plt.show()
```




```
y1 = autoencoder.predict(anomalous_test_data[0:1])
y1 = y1.flatten()
plt.plot(anomalous_test_data[0], 'b')
plt.plot(y, 'r')
plt.fill_between(np.arange(140), y, anomalous_test_data[0], color='lightcoral')
plt.legend(labels=["Input", "Reconstruction", "Error"])
plt.show()
```



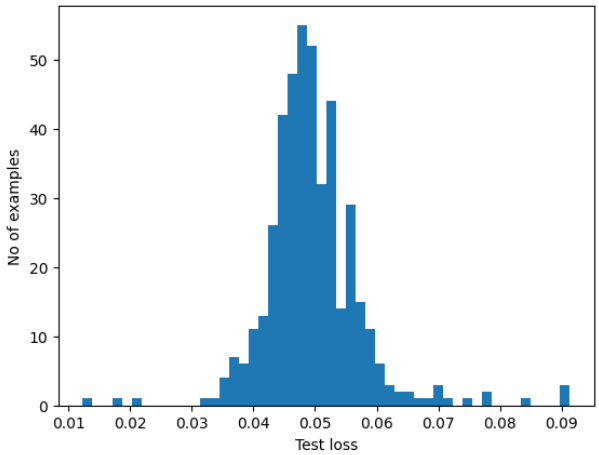

```
reconstructions = autoencoder.predict(normal_train_data)
train_loss = tf.keras.losses.mae(reconstructions, normal_train_data)

plt.hist(train_loss[None,:], bins=50)
plt.xlabel("Train loss")
plt.ylabel("No of examples")
plt.show()
```




```
threshold = np.mean(train_loss) + np.std(train_loss)
print("Threshold: ", threshold)
```

```
Threshold:  0.032382492
```

```
reconstructions = autoencoder.predict(anomalous_test_data)
test_loss = tf.keras.losses.mae(reconstructions, anomalous_test_data)

plt.hist(test_loss[None, :], bins=50)
plt.xlabel("Test loss")
plt.ylabel("No of examples")
plt.show()
```




```
def predict(model, data, threshold):
  reconstructions = model(data)
  loss = tf.keras.losses.mae(reconstructions, data)
  return tf.math.less(loss, threshold)

def print_stats(predictions, labels):
  print("Accuracy = {}".format(accuracy_score(labels, predictions)))
  print("Precision = {}".format(precision_score(labels, predictions)))
  print("Recall = {}".format(recall_score(labels, predictions)))
```

```
preds = predict(autoencoder, test_data, threshold)
print_stats(preds, test_labels)
```

```
Accuracy = 0.945
Precision = 0.9922027290448343
Recall = 0.9089285714285714
```

REINFORCEMENT LEARNING

Reinforcement learning (RL) is an exciting area of machine learning that focuses on how intelligent agents can learn to take actions in dynamic environments to maximize cumulative rewards. Let's dive into the details:

1. **Basic Idea**:
    - In RL, an **agent** interacts with an **environment** by taking actions.
    - The environment responds with **rewards** or **penalties** based on the agent's actions.
    - The agent's goal is to learn a policy (a strategy) that maximizes the long-term reward.
2. **Key Concepts**:
    - **Markov Decision Process (MDP)**:
        - The environment is typically modelled as an MDP.
        - It consists of states, actions, transition probabilities, and rewards.
    - **Policy**:
        - A mapping from states to actions.
        - Determines the agent's behaviour.
    - **Value Function**:
        - Estimates the expected cumulative reward from a given state.
        - Helps the agent evaluate different policies.
    - **Q-Function (Action-Value Function)**:
        - Estimates the expected cumulative reward for taking a specific action in each state.
3. **Exploration vs. Exploitation**:
    - RL faces a trade-off:
        - **Exploration**: Trying new actions to discover better strategies.
        - **Exploitation**: Leveraging current knowledge to maximize immediate rewards.
    - Balancing exploration and exploitation are crucial.
4. **Applications**:
    - **Game Playing**: RL agents excel in games like chess, Go, and video games.
    - **Robotics**: Teaching robots to perform tasks in real-world environments.
    - **Recommendation Systems**: Personalizing content for users.
    - **Finance**: Portfolio optimization, trading, and risk management.

CODE

```
import gym
env = gym.make('CartPole-v0')

import gym
env = gym.make("LunarLander-v2", render_mode="human")
env.action_space.seed(42)

observation, info = env.reset(seed=42)

for _ in range(1000):
  observation, reward, terminated, truncated, info =
env.step(env.action_space.sample())

  if terminated or truncated:
    observation, info = env.reset()

env.close()
```

BATCH NORMALIZATION IMPLEMENTATION ON FPGA BOARD

Implementing batch normalization on an FPGA board involves several steps, as outlined in the document. Here's a summary of the key points:

1. Understanding Batch Normalization: Batch normalization is a technique used to stabilize and accelerate the training of deep neural networks by normalizing the inputs to each layer. It involves calculating the mean and variance of the inputs and then scaling and shifting them using learned parameters.

2. Integration with Convolution Layers: The document discusses merging the batch normalization layer with the convolution layer to optimize resource usage on the FPGA. This is done by combining the parameters of both layers, which reduces the number of operations and memory required during inference.

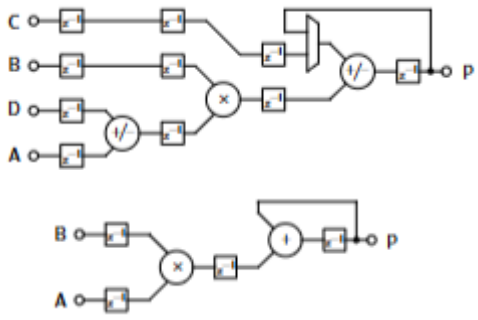
3. Fixed-Point Representation: Since FPGAs typically operate on fixed-point arithmetic, the floating-point calculations involved in batch normalization need to be adapted. This involves quantizing the mean, variance, scale, and offset parameters to fixed-point representations, which can be efficiently processed by the FPGA.

4. Resource Utilization: The implementation should be designed to fit within the constraints of the FPGA's resources. The document mentions that by optimizing the

operations (e.g., merging multiplication and addition), the batch normalization can be efficiently executed within a single DSP slice on the FPGA.

5. Synthesis and Testing: After designing the architecture, the next step is to synthesize the design using hardware description languages (HDLs) like VHDL or Verilog. The synthesized design should then be tested on the FPGA board to ensure it meets the desired performance metrics, such as processing speed and accuracy.

6. Performance Optimization: The document highlights that reducing the bit precision (e.g., from 32 bits to 16 bits) can lead to higher clock frequencies and better resource utilization, albeit with a trade-off in accuracy. This is crucial for real-time applications where speed is essential.



DISPLAYING ACCURACY & RECALL SCORE FOR AN AUTOENCODER

```
from sklearn.metrics import *
def print_stats(predictions, labels):
  print("Accuracy = {}".format(accuracy_score(labels, predictions)))
  print("Precision = {}".format(precision_score(labels, predictions)))
  print("Recall = {}".format(recall_score(labels, predictions)))
```