Name : Sakshi Ravindra Aher

PRN : 202201040089

Batch : B(B-3)

Roll NO : 43

Implement the following :

1.Logistic Regression

2.Plotting Activation Functions (Sigmoid, ReLU, Tanh)

3.Log Loss

4.ANN Implementation using Skelearn

5.ANN Implementation using Keras

6.ANN from Scratch (Forward & Backpropagation)

Logistic Regression: Short Theory

1. Definition:
Logistic Regression is a **classification algorithm** used to predict **binary outcomes** (0 or 1). It calculates the probability of an event occurring using a **sigmoid function**.

2. Mathematical Model:
Logistic regression is based on a **linear equation**:
[z = W^T X + b]
Applying the **sigmoid function**:
[\sigma(z) = \frac{1}{1 + e^{-z}}]
This ensures the output is between **0 and 1** (probability).

3. Decision Rule:

- If (sigma(z) > 0.5), predict **1**

- If (sigma(z)<=leq 0.5), predict **0**

4. Cost Function (Log Loss):
Log Loss, also known as Binary Cross-Entropy Loss, is a performance metric used to evaluate the predictions of a binary classification model like Logistic Regression. It measures the difference between the predicted probabilities and the actual class labels. Log Loss quantifies how well the model's predicted probabilities match the true outcomes. The lower the log loss, the better the model is at predicting the correct class probabilities. The formula for **Log Loss** (also known as **Binary Cross-Entropy Loss**) is:

5. Training Process:

- **Initialize** weights and bias

- **Compute** predictions using sigmoid
- **Calculate** loss using log loss
- **Update** weights using gradient descent

 Used in **spam detection, medical diagnosis, and fraud detection**.

```
#defining the logistic regression model
class LogisticRegression:
    def _init_(self, learning_rate=0.01, epochs=1000):
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.weights = None
        self.bias = None

#learning_rate controls how fast the model updates the weights during
training.
#epochs is the number of iterations to train the model.
#weights and bias are parameters that the model will learn.
#sigmoid function

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    #Converts linear outputs (z) to probabilities between 0 and 1.

    def fit(self, X, y):
        num_samples, num_features = X.shape

        # Initialize weights and bias to zeros
        self.weights = np.zeros(num_features)
        self.bias = 0

        # Gradient descent for updating weights
        for _ in range(self.epochs):
            # Linear combination
            linear_model = np.dot(X, self.weights) + self.bias
            # Apply sigmoid to get predicted probabilities
            y_predicted = self.sigmoid(linear_model)

            # Compute gradients
            dw = (1 / num_samples) * np.dot(X.T, (y_predicted - y))
            db = (1 / num_samples) * np.sum(y_predicted - y)

            # Update weights and bias
            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db

    def predict(self, X):
```

```python
            # Compute linear combination and apply sigmoid
            linear_model = np.dot(X, self.weights) + self.bias
            y_predicted = self.sigmoid(linear_model)

            # Classify based on probability threshold of 0.5
            return [1 if i > 0.5 else 0 for i in y_predicted]

# Sample dataset (simple binary classification problem)
X = np.array([[1], [2], [3], [4], [5]])  # Input features (sweetness
levels)
y = np.array([0, 0, 1, 1, 1])  # Target labels (likes fruit or not)

model = LogisticRegression(learning_rate=0.1, epochs=1000)
model.fit(X, y)
predictions = model.predict(X)
print("Predictions:", predictions)
```

```
Predictions: [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0,
1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0,
1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0,
0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0,
1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0,
1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1,
0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1,
1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0,
1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1,
1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0,
1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1,
0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0,
0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1,
1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0,
1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0,
1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1,
0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0,
0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1,
1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0,
0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0,
1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1,
1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0]
```

```python
import numpy as np
from sklearn.linear_model import LogisticRegression

# Dataset (Sweetness levels and whether the fruit is liked)
X = np.array([[1], [2], [3], [4], [5]])  # Features
y = np.array([0, 0, 1, 1, 1])  # Target labels

# Create and train the Logistic Regression model
model = LogisticRegression()
model.fit(X, y)
```

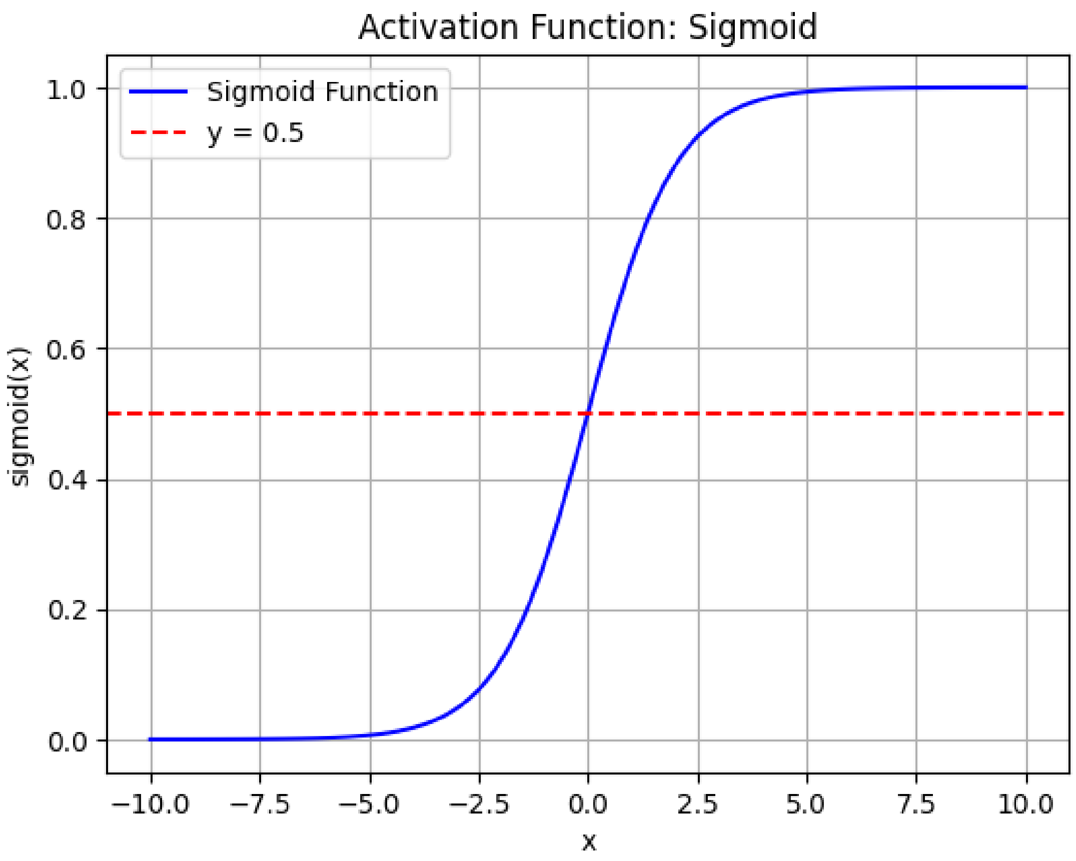
```
# Make predictions
predictions = model.predict(X)
print("Predictions:", predictions)

# Get probability estimates for each class
probabilities = model.predict_proba(X)
print("Probabilities:\n", probabilities)
```

```
Predictions: [0 0 1 1 1]
Probabilities:
 [[0.8157613  0.1842387 ]
 [0.60836998 0.39163002]
 [0.35275336 0.64724664]
 [0.16051755 0.83948245]
 [0.06286686 0.93713314]]
```

Sigmoid Activation Function

Sigmoid function returns the value beteen 0 and 1. For activation function in deep learning network, Sigmoid function is considered not good since near the boundaries the network doesn't learn quickly. This is because gradient is almost zero near the boundaries.

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

def sigmoid(x):
    """Returns the sigmoid function value for a given input x."""
    return 1 / (1 + np.exp(-x))

# Generate x values from -10 to 10
x = np.linspace(-10, 10, 100)

# Compute sigmoid values
y = sigmoid(x)

# Plot the sigmoid function
plt.plot(x, y, label="Sigmoid Function", color="blue")
plt.axhline(y=0.5, color='r', linestyle='--', label="y = 0.5")
plt.xlabel("x")
plt.ylabel("sigmoid(x)")
plt.title("Activation Function: Sigmoid")
plt.legend()
plt.grid()
plt.show()
```



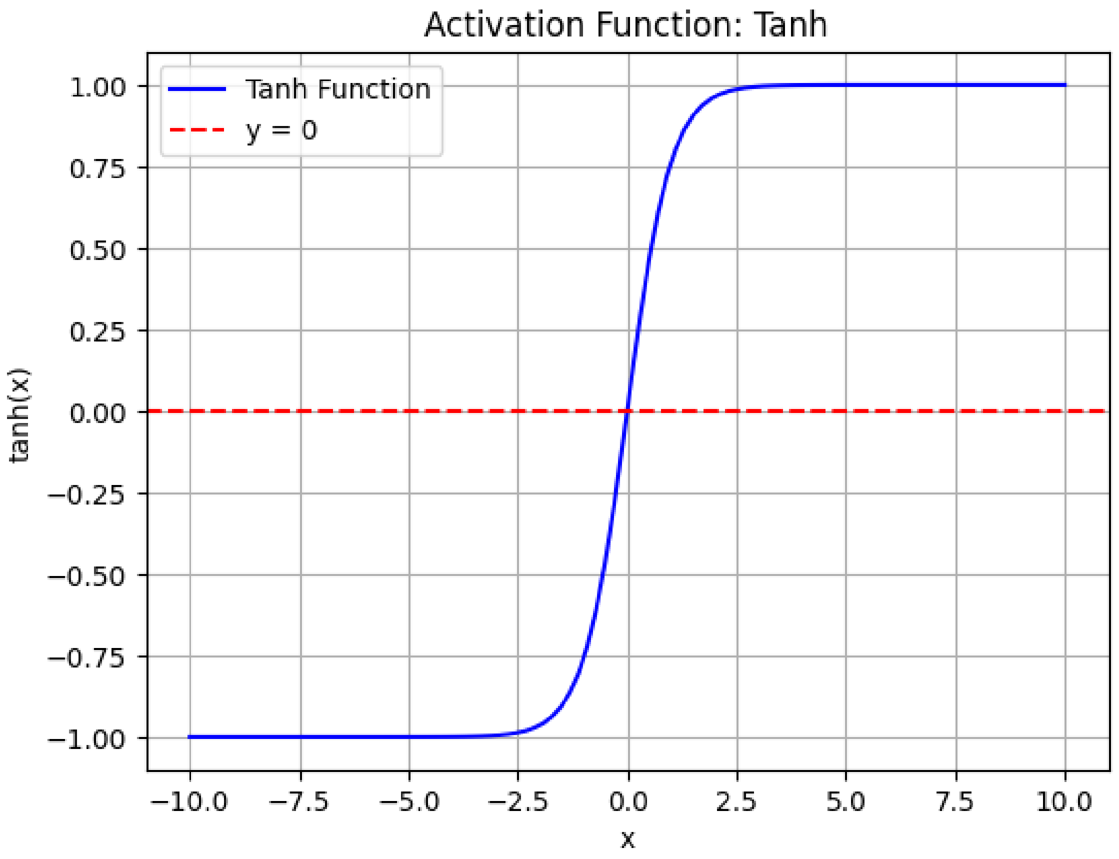
```
print(sigmoid(0))
print(sigmoid(2))
print(sigmoid(-2))

0.5
0.8807970779778823
0.11920292202211755
```

Tanh Activation Function

Tanh outputs between -1 and 1. Tanh also suffers from gradient problem near the boundaries just as Sigmoid activation function does.

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

def tanh(x):
    """Returns the tanh activation function value for a given input
x."""
    return np.tanh(x)

# Generate x values from -10 to 10
```

```
x = np.linspace(-10, 10, 100)

# Compute tanh values
y = tanh(x)

# Plot the tanh function
plt.plot(x, y, label="Tanh Function", color="blue")
plt.axhline(y=0, color='r', linestyle='--', label="y = 0")
plt.xlabel("x")
plt.ylabel("tanh(x)")
plt.title("Activation Function: Tanh")
plt.legend()
plt.grid()
plt.show()
```



```
print(tanh(1))
print(tanh(2))
print(tanh(-4))
```

```
0.7615941559557649
0.9640275800758169
-0.999329299739067
```

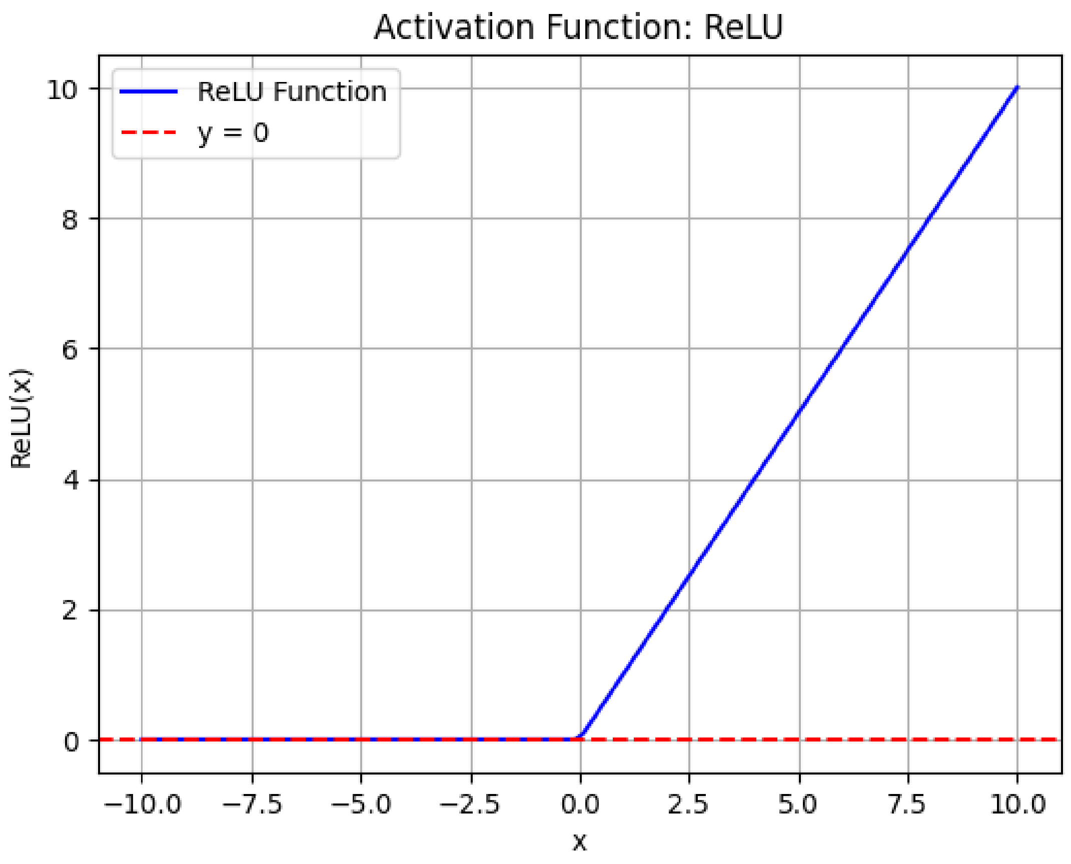
RELU Activation Function

RELU is more well known activation function which is used in the deep learning networks. RELU is less computational expensive than the other non linear activation functions.

1.RELU returns 0 if the x (input) is less than 0

2.RELU returns x if the x (input) is greater than 0

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

def relu(x):
    """Returns the ReLU activation function value for a given input
x."""
    return np.maximum(0, x)

# Generate x values from -10 to 10
x = np.linspace(-10, 10, 100)

# Compute ReLU values
y = relu(x)

# Plot the ReLU function
plt.plot(x, y, label="ReLU Function", color="blue")
plt.axhline(y=0, color='r', linestyle='--', label="y = 0")
plt.xlabel("x")
plt.ylabel("ReLU(x)")
plt.title("Activation Function: ReLU")
plt.legend()
plt.grid()
plt.show()
```



```
print(relu(6))
print(relu(7.9))
print(relu(-7.9))

6
7.9
0.0
```

Log Loss

Log Loss, also known as Binary Cross-Entropy Loss, is a metric used to evaluate the performance of binary classification models. It quantifies the difference between the predicted probabilities and the actual class labels (0 or 1). The formula for **Log Loss** is:

The formula for **Log Loss** is:

$$\text{Log Loss} = -\frac{1}{m}\sum_{i=1}^{m}\left[y_i \log\left(\hat{y}_i\right) + \left(1 - y_i\right)\log\left(1 - \hat{y}_i\right)\right]$$

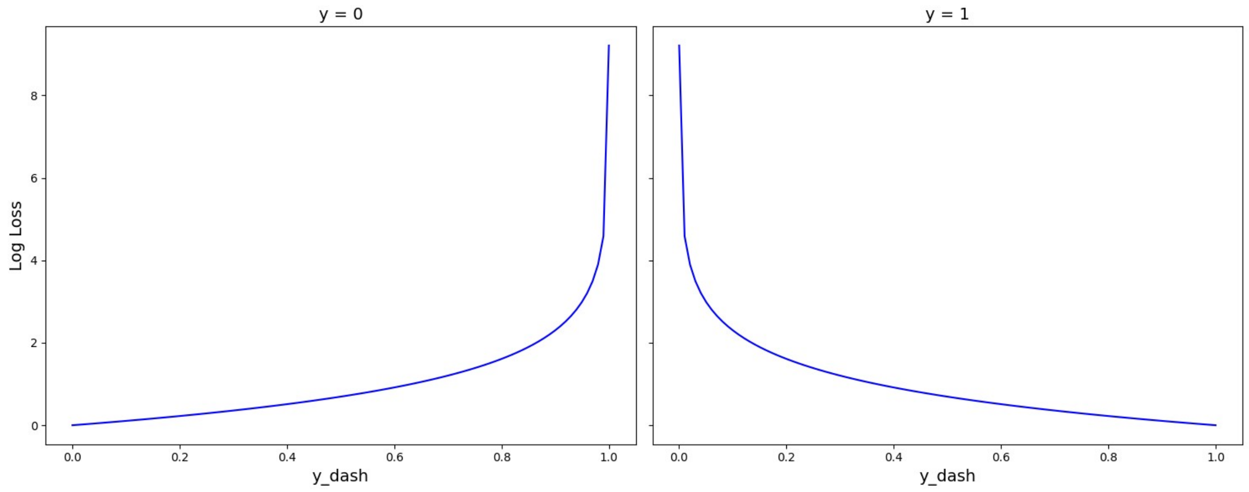
Where:

- (yi) is the true label (0 or 1) for the (i)-th sample.

- ({hat(yi)}) is the predicted probability that the (i)-th sample belongs to class 1.
- (m) is the total number of samples.

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

# Define log loss function
def log_loss(y, y_dash):
    """
    Computes log loss for a given true value (0 or 1) and predicted
probability.

    Args:
        y (int or array): True value(s), should be 0 or 1.
        y_dash (float or array): Predicted probability(ies) between 0
and 1.

    Returns:
        float or array: Log loss value.
    """
    return - (y * np.log(y_dash)) - ((1 - y) * np.log(1 - y_dash))

# Test cases
y, y_dash = 0, 0.6
print(f"log_loss({y}, {y_dash}) = {log_loss(y, y_dash)}")

y, y_dash = 1, 0.4
print(f"log_loss({y}, {y_dash}) = {log_loss(y, y_dash)}")

y, y_dash = 1, 0.8
print(f"log_loss({y}, {y_dash}) = {log_loss(y, y_dash)}")

y, y_dash = 0, 0.2
print(f"log_loss({y}, {y_dash}) = {log_loss(y, y_dash)}")

# Plot Log Loss for y = 0 and y = 1
fig, ax = plt.subplots(1, 2, figsize=(15, 6), sharex=True,
sharey=True)

y_dash = np.linspace(0.0001, 0.9999, 100)  # Avoid log(0) issue

ax[0].plot(y_dash, log_loss(0, y_dash), color='blue')
ax[0].set_title("y = 0", fontsize=14)
ax[0].set_xlabel("y_dash", fontsize=14)
ax[0].set_ylabel("Log Loss", fontsize=14)

ax[1].plot(y_dash, log_loss(1, y_dash), color='blue')
ax[1].set_title("y = 1", fontsize=14)
ax[1].set_xlabel("y_dash", fontsize=14)
```

```
plt.tight_layout()
plt.show()

log_loss(0, 0.6) = 0.916290731874155
log_loss(1, 0.4) = 0.916290731874155
log_loss(1, 0.8) = 0.2231435513142097
log_loss(0, 0.2) = 0.2231435513142097
```



Dataset : https://drive.google.com/file/d/1iIoWfurk58sTTuEm2tMhP7aWalUORI8s/view?usp=sharing

ANN Implementation Using sklearn

An **Artificial Neural Network (ANN)** in scikit-learn is implemented using the `MLPClassifier` for classification tasks. It consists of layers of neurons:

1. **Input Layer**: Receives input data.
2. **Hidden Layers**: Intermediate layers that process the input data.
3. **Output Layer**: Produces the final prediction.

Key Concepts:

- **Activation Function**: Functions like **ReLU** or **Sigmoid** are used to introduce non-linearity in neurons.
- **Backpropagation**: The method used to update weights in the network by minimizing the error.
- **Loss Function**: Measures the error, with **cross-entropy** commonly used for classification.

Workflow:

1. **Preprocess Data**: Handle missing values, encode categories, and standardize features.
2. **Define Model**: Specify the number of layers and neurons.
3. **Train Model**: Use `fit()` method with training data.

4. **Evaluate:** Use metrics like **accuracy** to evaluate the model.

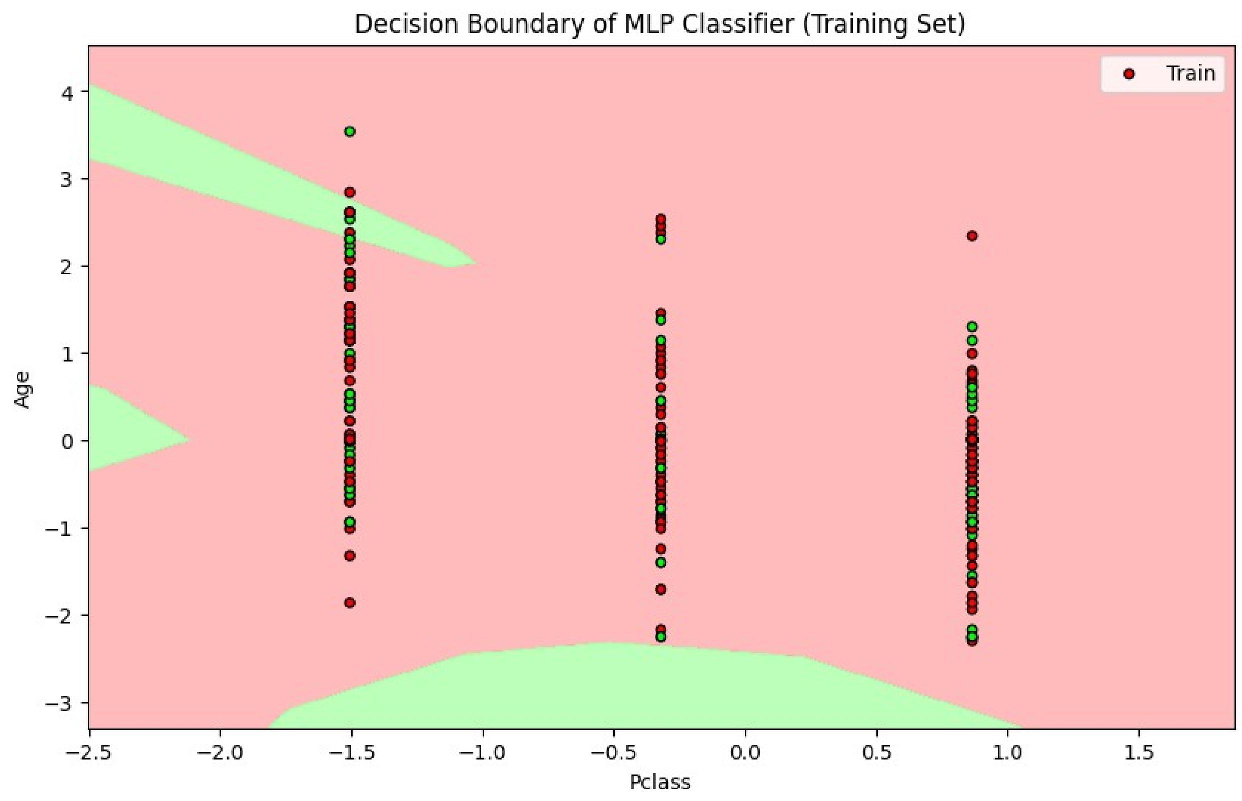
`MLPClassifier` automates much of the ANN's functionality, making it easy to implement neural networks for classification.

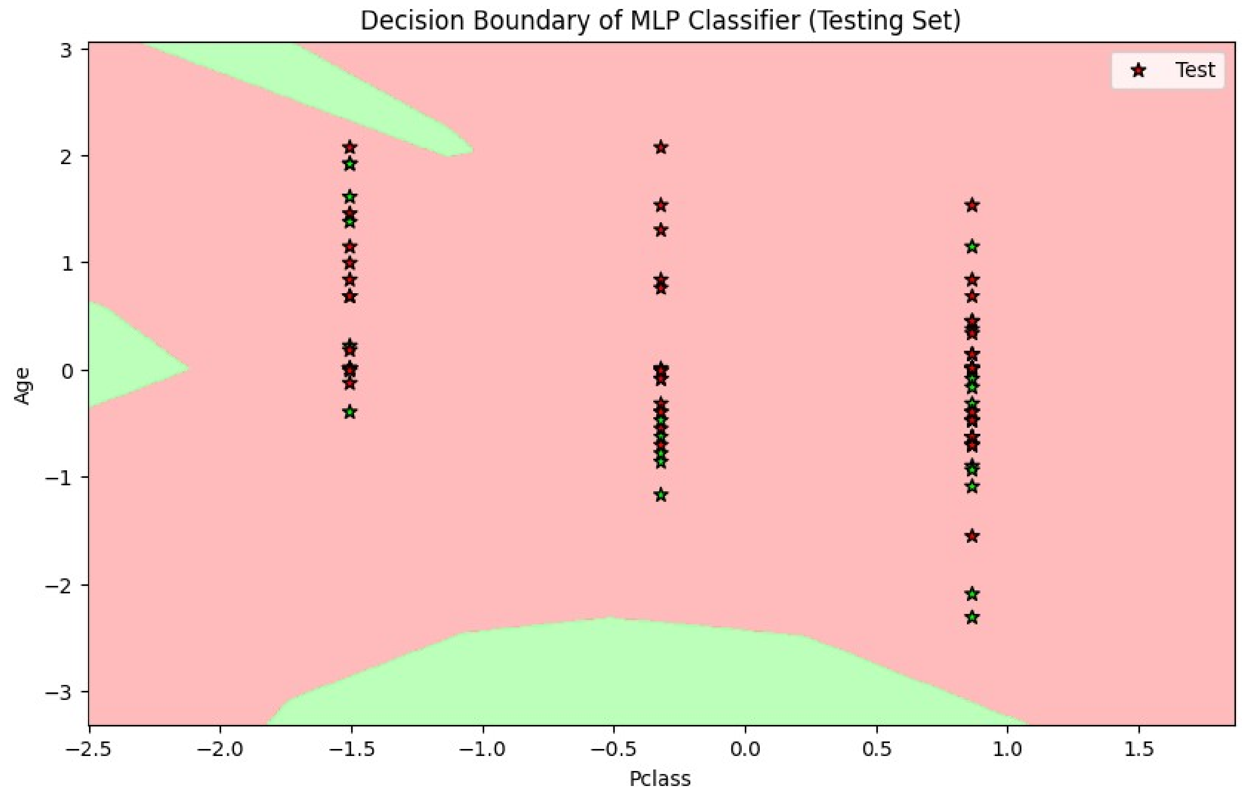
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, classification_report

# Load Titanic dataset
url = '/content/Titanic Dataset.csv'
data = pd.read_csv(url)

# Preprocess the data
data = data.drop(['Name', 'Ticket', 'Cabin'], axis=1)  # Drop
irrelevant columns
data = pd.get_dummies(data, drop_first=True)  # One-hot encode
categorical variables
data['Age'].fillna(data['Age'].mean(), inplace=True)  # Fill missing
Age values

# Select only 'Pclass' and 'Age' as features for simplicity
X = data[['Pclass', 'Age']].values
y = data['Survived'].values

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Define the MLP classifier
mlp = MLPClassifier(hidden_layer_sizes=(10, 10), max_iter=1000,
random_state=42)

# Train the MLP classifier
mlp.fit(X_train, y_train)

# Make predictions
y_pred = mlp.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
```

```python
print("Accuracy:", accuracy)
print("\nClassification Report:\n", classification_report(y_test,
y_pred))

# Plot decision boundaries for training set
x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                     np.arange(y_min, y_max, 0.01))

Z_train = mlp.predict(np.c_[xx.ravel(), yy.ravel()])
Z_train = Z_train.reshape(xx.shape)

# Define color maps
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00'])

plt.figure(figsize=(10, 6))
plt.contourf(xx, yy, Z_train, alpha=0.8, cmap=cmap_light)
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cmap_bold,
edgecolor='k', s=20, label='Train')
plt.title("Decision Boundary of MLP Classifier (Training Set)")
plt.xlabel("Pclass")
plt.ylabel("Age")
plt.legend()
plt.show()

# Plot decision boundaries for testing set
x_min, x_max = X_test[:, 0].min() - 1, X_test[:, 0].max() + 1
y_min, y_max = X_test[:, 1].min() - 1, X_test[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                     np.arange(y_min, y_max, 0.01))

Z_test = mlp.predict(np.c_[xx.ravel(), yy.ravel()])
Z_test = Z_test.reshape(xx.shape)

plt.figure(figsize=(10, 6))
plt.contourf(xx, yy, Z_test, alpha=0.8, cmap=cmap_light)
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cmap_bold,
edgecolor='k', s=50, label='Test', marker='*')
plt.title("Decision Boundary of MLP Classifier (Testing Set)")
plt.xlabel("Pclass")
plt.ylabel("Age")
plt.legend()
plt.show()
```

```
<ipython-input-28-7bb405ac7406>:17: FutureWarning: A value is trying
to be set on a copy of a DataFrame or Series through chained
assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
```

```
work because the intermediate object on which we are setting values
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try
using 'df.method({col: value}, inplace=True)' or df[col] =
df[col].method(value) instead, to perform the operation inplace on the
original object.


  data['Age'].fillna(data['Age'].mean(), inplace=True)  # Fill missing
Age values
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classificatio
n.py:1565: UndefinedMetricWarning: Precision is ill-defined and being
set to 0.0 in labels with no predicted samples. Use `zero_division`
parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is",
len(result))
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classificatio
n.py:1565: UndefinedMetricWarning: Precision is ill-defined and being
set to 0.0 in labels with no predicted samples. Use `zero_division`
parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is",
len(result))
/usr/local/lib/python3.11/dist-packages/sklearn/metrics/_classificatio
n.py:1565: UndefinedMetricWarning: Precision is ill-defined and being
set to 0.0 in labels with no predicted samples. Use `zero_division`
parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is",
len(result))

Accuracy: 0.5952380952380952

Classification Report:
               precision    recall  f1-score   support

           0       0.60      1.00      0.75        50
           1       0.00      0.00      0.00        34

    accuracy                           0.60        84
   macro avg       0.30      0.50      0.37        84
weighted avg       0.35      0.60      0.44        84
```


Decision Boundary of MLP Classifier (Training Set)
Train
Age
Pclass
4
3
2
1
0
−1
−2
−3
−2.5
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5


Decision Boundary of MLP Classifier (Testing Set)
Test
Age
Pclass
3
2
1
0
−1
−2
−3
−2.5
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

Artificial Neural Network Using Keras

Introduction

An **Artificial Neural Network (ANN)** mimics the human brain's structure to make predictions. In this example, we use **Keras** to create an ANN that predicts whether a passenger survived the Titanic disaster, based on features like age, sex, and fare.

Model Architecture

- **Input Layer**: Takes in features such as age, fare, and sex.
- **Hidden Layers**:
  - **Layer 1**: 16 neurons, with **ReLU** (Rectified Linear Unit) activation function.
  - **Layer 2**: 8 neurons, with **ReLU** activation.
- **Output Layer**: 1 neuron, with **Sigmoid** activation, providing a probability for survival (1 for survived, 0 for not).

Training Process

- **Loss Function**: `binary_crossentropy` is used, as it's suitable for binary classification (survived or not).
- **Optimizer**: `adam`, an adaptive learning rate optimizer that helps the network converge faster.
- **Evaluation**: We use **accuracy**, loss curves, and a **confusion matrix** to evaluate the model's performance during training.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import classification_report,
ConfusionMatrixDisplay

# Load the Titanic dataset
df = pd.read_csv("Titanic Dataset.csv")

# Data Preprocessing
df.drop(["PassengerId", "Name", "Ticket", "Cabin"], axis=1,
inplace=True)
df["Age"].fillna(df["Age"].median(), inplace=True)
df["Fare"].fillna(df["Fare"].median(), inplace=True)
df["Embarked"].fillna(df["Embarked"].mode()[0], inplace=True)

# Encode categorical variables
label_encoder = LabelEncoder()
df["Sex"] = label_encoder.fit_transform(df["Sex"])
df["Embarked"] = label_encoder.fit_transform(df["Embarked"])
```

```python
# Define features and target
X = df.drop("Survived", axis=1)
y = df["Survived"]

# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Build the ANN model
model = Sequential()
model.add(Dense(16, activation='relu',
input_shape=(X_train.shape[1],)))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

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

# Train the model
history = model.fit(X_train, y_train, epochs=50, batch_size=8,
validation_data=(X_test, y_test), verbose=1)

# Evaluate the model
y_pred = (model.predict(X_test) > 0.5).astype("int32")
print(classification_report(y_test, y_pred))

# Display confusion matrix
ConfusionMatrixDisplay.from_predictions(y_test, y_pred)
plt.show()

# Plot accuracy history
plt.figure(figsize=(12, 4))
plt.subplot(1, 3, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Model Accuracy')

# Plot loss history
plt.subplot(1, 3, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Model Loss')

# Display confusion matrix as heatmap
plt.subplot(1, 3, 3)
ConfusionMatrixDisplay.from_predictions(y_test, y_pred, ax=plt.gca())
plt.title('Confusion Matrix')

plt.tight_layout()
plt.show()
```

```
Epoch 1/50
```

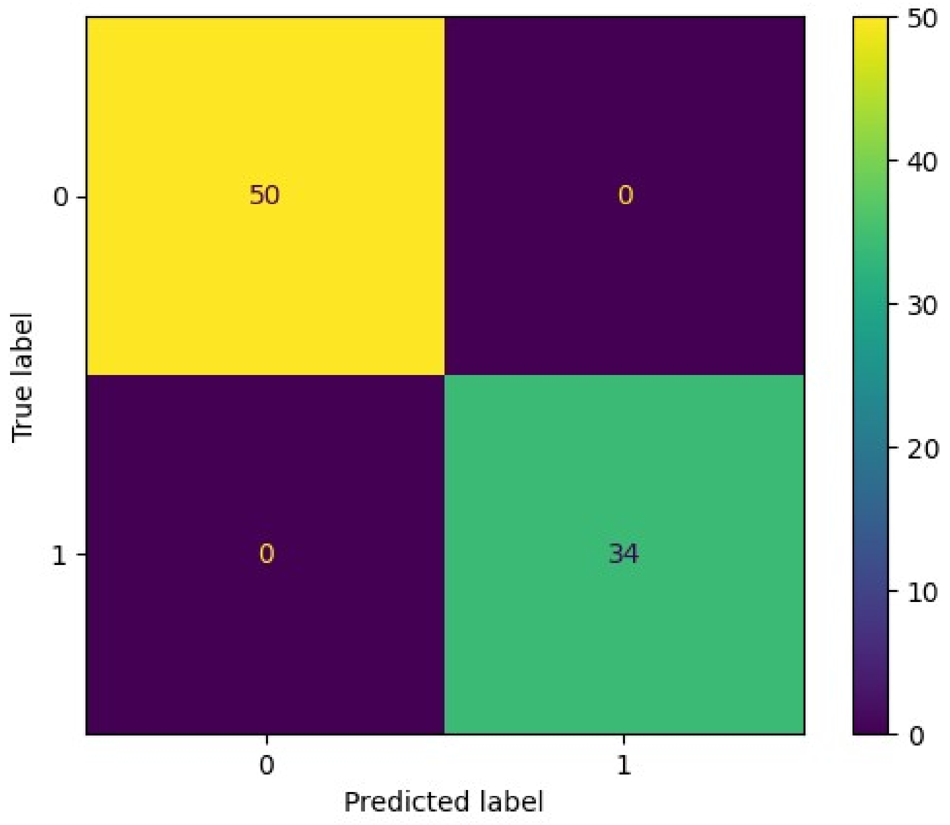
```
<ipython-input-30-e3552360cb59>:15: FutureWarning: A value is trying
to be set on a copy of a DataFrame or Series through chained
assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try
using 'df.method({col: value}, inplace=True)' or df[col] =
df[col].method(value) instead, to perform the operation inplace on the
original object.


  df["Age"].fillna(df["Age"].median(), inplace=True)
<ipython-input-30-e3552360cb59>:16: FutureWarning: A value is trying
to be set on a copy of a DataFrame or Series through chained
assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try
using 'df.method({col: value}, inplace=True)' or df[col] =
df[col].method(value) instead, to perform the operation inplace on the
original object.


  df["Fare"].fillna(df["Fare"].median(), inplace=True)
<ipython-input-30-e3552360cb59>:17: FutureWarning: A value is trying
to be set on a copy of a DataFrame or Series through chained
assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try
```

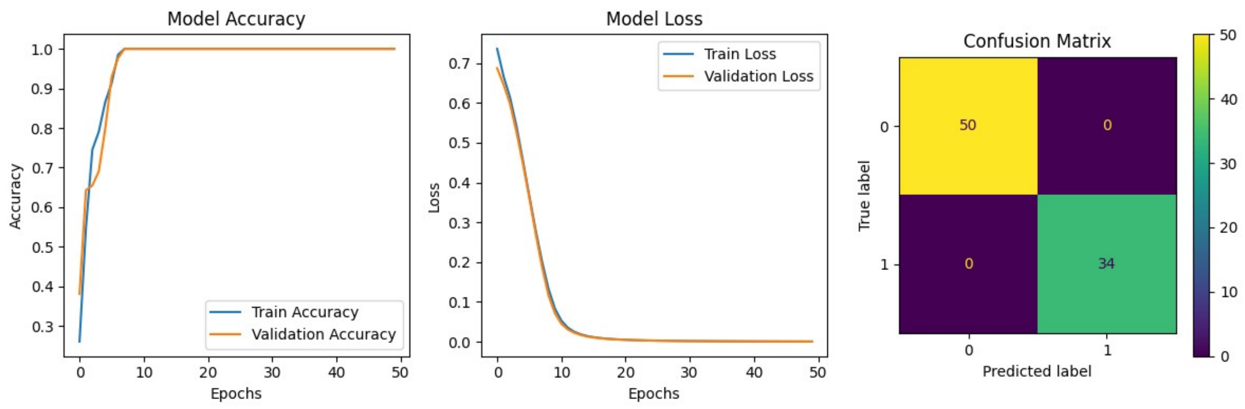
```
using 'df.method({col: value}, inplace=True)' or df[col] =
df[col].method(value) instead, to perform the operation inplace on the
original object.


  df["Embarked"].fillna(df["Embarked"].mode()[0], inplace=True)
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py
:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to
a layer. When using Sequential models, prefer using an `Input(shape)`
object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)

42/42 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.2496 - loss:
0.7486 - val_accuracy: 0.3810 - val_loss: 0.6873
Epoch 2/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.4080 - loss:
0.6830 - val_accuracy: 0.6429 - val_loss: 0.6461
Epoch 3/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.7544 - loss:
0.6248 - val_accuracy: 0.6548 - val_loss: 0.5980
Epoch 4/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - accuracy: 0.7846 - loss:
0.5549 - val_accuracy: 0.6905 - val_loss: 0.5277
Epoch 5/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.8681 - loss:
0.4695 - val_accuracy: 0.7976 - val_loss: 0.4484
Epoch 6/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.8781 - loss:
0.4057 - val_accuracy: 0.9286 - val_loss: 0.3624
Epoch 7/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9685 - loss:
0.2994 - val_accuracy: 0.9762 - val_loss: 0.2716
Epoch 8/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss:
0.2234 - val_accuracy: 1.0000 - val_loss: 0.1873
Epoch 9/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.1430 - val_accuracy: 1.0000 - val_loss: 0.1161
Epoch 10/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.0947 - val_accuracy: 1.0000 - val_loss: 0.0712
Epoch 11/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss:
0.0628 - val_accuracy: 1.0000 - val_loss: 0.0450
Epoch 12/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.0387 - val_accuracy: 1.0000 - val_loss: 0.0307
Epoch 13/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
```

```
0.0250 - val_accuracy: 1.0000 - val_loss: 0.0224
Epoch 14/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.0187 - val_accuracy: 1.0000 - val_loss: 0.0168
Epoch 15/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss:
0.0153 - val_accuracy: 1.0000 - val_loss: 0.0129
Epoch 16/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.0112 - val_accuracy: 1.0000 - val_loss: 0.0104
Epoch 17/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.0098 - val_accuracy: 1.0000 - val_loss: 0.0086
Epoch 18/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.0083 - val_accuracy: 1.0000 - val_loss: 0.0072
Epoch 19/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 1.0000 - loss:
0.0066 - val_accuracy: 1.0000 - val_loss: 0.0061
Epoch 20/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss:
0.0056 - val_accuracy: 1.0000 - val_loss: 0.0052
Epoch 21/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.0049 - val_accuracy: 1.0000 - val_loss: 0.0045
Epoch 22/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.0043 - val_accuracy: 1.0000 - val_loss: 0.0040
Epoch 23/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.0037 - val_accuracy: 1.0000 - val_loss: 0.0035
Epoch 24/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.0033 - val_accuracy: 1.0000 - val_loss: 0.0032
Epoch 25/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.0029 - val_accuracy: 1.0000 - val_loss: 0.0028
Epoch 26/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 1.0000 - loss:
0.0026 - val_accuracy: 1.0000 - val_loss: 0.0026
Epoch 27/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 1.0000 - loss:
0.0023 - val_accuracy: 1.0000 - val_loss: 0.0023
Epoch 28/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 1.0000 - loss:
0.0021 - val_accuracy: 1.0000 - val_loss: 0.0021
Epoch 29/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.0021 - val_accuracy: 1.0000 - val_loss: 0.0019
```

```
Epoch 30/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.0018 - val_accuracy: 1.0000 - val_loss: 0.0018
Epoch 31/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.0017 - val_accuracy: 1.0000 - val_loss: 0.0016
Epoch 32/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.0017 - val_accuracy: 1.0000 - val_loss: 0.0015
Epoch 33/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 1.0000 - loss:
0.0014 - val_accuracy: 1.0000 - val_loss: 0.0014
Epoch 34/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 1.0000 - loss:
0.0013 - val_accuracy: 1.0000 - val_loss: 0.0013
Epoch 35/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.0012 - val_accuracy: 1.0000 - val_loss: 0.0012
Epoch 36/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.0012 - val_accuracy: 1.0000 - val_loss: 0.0011
Epoch 37/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 1.0000 - loss:
0.0011 - val_accuracy: 1.0000 - val_loss: 0.0011
Epoch 38/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
0.0010 - val_accuracy: 1.0000 - val_loss: 9.8929e-04
Epoch 39/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss:
9.0442e-04 - val_accuracy: 1.0000 - val_loss: 9.2908e-04
Epoch 40/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 1.0000 - loss:
8.8975e-04 - val_accuracy: 1.0000 - val_loss: 8.7118e-04
Epoch 41/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 1.0000 - loss:
8.2843e-04 - val_accuracy: 1.0000 - val_loss: 8.2067e-04
Epoch 42/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 1.0000 - loss:
7.2730e-04 - val_accuracy: 1.0000 - val_loss: 7.7322e-04
Epoch 43/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 1.0000 - loss:
7.0946e-04 - val_accuracy: 1.0000 - val_loss: 7.2941e-04
Epoch 44/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 1.0000 - loss:
6.4979e-04 - val_accuracy: 1.0000 - val_loss: 6.9121e-04
Epoch 45/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - accuracy: 1.0000 - loss:
6.3900e-04 - val_accuracy: 1.0000 - val_loss: 6.5423e-04
Epoch 46/50
```

```
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 1.0000 - loss:
5.9197e-04 - val_accuracy: 1.0000 - val_loss: 6.2019e-04
Epoch 47/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 1.0000 - loss:
5.4877e-04 - val_accuracy: 1.0000 - val_loss: 5.8760e-04
Epoch 48/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 1.0000 - loss:
5.3348e-04 - val_accuracy: 1.0000 - val_loss: 5.5958e-04
Epoch 49/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 1.0000 - loss:
5.0756e-04 - val_accuracy: 1.0000 - val_loss: 5.3193e-04
Epoch 50/50
42/42 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 1.0000 - loss:
4.9841e-04 - val_accuracy: 1.0000 - val_loss: 5.0691e-04
3/3 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        50
           1       1.00      1.00      1.00        34

    accuracy                           1.00        84
   macro avg       1.00      1.00      1.00        84
weighted avg       1.00      1.00      1.00        84
```





Artificial Neural Network (ANN) from Scratch using Backpropagation

An **Artificial Neural Network (ANN)** is a computational model inspired by the human brain. It consists of layers of interconnected neurons that process and learn from data. **Backpropagation** is the key algorithm used to train ANNs by adjusting weights based on errors.

Key Concepts:

1. **Neural Network Structure:**
   - **Input Layer**: Takes feature data (e.g., Titanic passenger attributes).
   - **Hidden Layer(s)**: Applies transformations using activation functions (e.g., ReLU, Sigmoid).
   - **Output Layer**: Produces final predictions (e.g., survival probability).
2. **Forward Propagation:**
   - Computes neuron activations using weighted sums and an activation function.
   - Moves data from input → hidden layers → output layer.
3. **Loss Function:**
   - Measures the difference between predicted and actual values (e.g., Binary Cross-Entropy for classification).
4. **Backpropagation:**
   - Computes **gradients** of the loss function with respect to weights using the **chain rule**.
   - Adjusts weights to minimize error using **Gradient Descent**.
5. **Training Process:**
   - **Initialize** random weights.
   - **Iterate** through data, performing forward propagation and backpropagation.
   - **Update** weights using learning rate until loss reduces.

6. **Evaluation:**
   - Once trained, the model predicts unseen data and is assessed using accuracy, confusion matrix, etc.

Why Backpropagation?

Backpropagation enables efficient learning by updating weights iteratively, ensuring the network **learns meaningful patterns** from data.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder

# Load the Titanic dataset
df = pd.read_csv("Titanic Dataset.csv")

# Data Preprocessing
df.drop(["PassengerId", "Name", "Ticket", "Cabin"], axis=1,
inplace=True)
df["Age"].fillna(df["Age"].median(), inplace=True)
df["Fare"].fillna(df["Fare"].median(), inplace=True)
df["Embarked"].fillna(df["Embarked"].mode()[0], inplace=True)

# Encode categorical variables
label_encoder = LabelEncoder()
df["Sex"] = label_encoder.fit_transform(df["Sex"])
df["Embarked"] = label_encoder.fit_transform(df["Embarked"])

# Define features and target
X = df.drop("Survived", axis=1).values
y = df["Survived"].values.reshape(-1, 1)

# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Initialize network parameters
input_size = X_train.shape[1]
hidden_size = 8
output_size = 1
learning_rate = 0.01
epochs = 500
```

```python
# Weights and biases
W1 = np.random.randn(input_size, hidden_size)
b1 = np.zeros((1, hidden_size))
W2 = np.random.randn(hidden_size, output_size)
b2 = np.zeros((1, output_size))

# Activation function and its derivative
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def sigmoid_derivative(a):
    return a * (1 - a)

# Training loop
for epoch in range(epochs):
    # Forward propagation
    Z1 = np.dot(X_train, W1) + b1
    A1 = sigmoid(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = sigmoid(Z2)

    # Compute loss (binary cross-entropy)
    loss = -np.mean(y_train * np.log(A2) + (1 - y_train) * np.log(1 -
A2))

    # Backpropagation
    dZ2 = A2 - y_train
    dW2 = np.dot(A1.T, dZ2) / X_train.shape[0]
    db2 = np.sum(dZ2, axis=0, keepdims=True) / X_train.shape[0]

    dZ1 = np.dot(dZ2, W2.T) * sigmoid_derivative(A1)
    dW1 = np.dot(X_train.T, dZ1) / X_train.shape[0]
    db1 = np.sum(dZ1, axis=0, keepdims=True) / X_train.shape[0]

    # Update weights
    W1 -= learning_rate * dW1
    b1 -= learning_rate * db1
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2

    # Print loss every 50 epochs
    if epoch % 50 == 0:
        print(f"Epoch {epoch}, Loss: {loss:.4f}")

# Predictions on test set
Z1_test = np.dot(X_test, W1) + b1
A1_test = sigmoid(Z1_test)
Z2_test = np.dot(A1_test, W2) + b2
A2_test = sigmoid(Z2_test)
y_pred = (A2_test > 0.5).astype(int)
```

```python
# Calculate accuracy
accuracy = np.mean(y_pred == y_test)
print(f"Test Accuracy: {accuracy:.4f}")
```

```
Epoch 0, Loss: 1.0041
Epoch 50, Loss: 0.8913
Epoch 100, Loss: 0.8040
Epoch 150, Loss: 0.7379
Epoch 200, Loss: 0.6880
Epoch 250, Loss: 0.6496
Epoch 300, Loss: 0.6190
Epoch 350, Loss: 0.5935
Epoch 400, Loss: 0.5713
Epoch 450, Loss: 0.5514
Test Accuracy: 0.6548
```

```
<ipython-input-35-7e6d2ceef23b>:11: FutureWarning: A value is trying
to be set on a copy of a DataFrame or Series through chained
assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try
using 'df.method({col: value}, inplace=True)' or df[col] =
df[col].method(value) instead, to perform the operation inplace on the
original object.


  df["Age"].fillna(df["Age"].median(), inplace=True)
<ipython-input-35-7e6d2ceef23b>:12: FutureWarning: A value is trying
to be set on a copy of a DataFrame or Series through chained
assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try
using 'df.method({col: value}, inplace=True)' or df[col] =
df[col].method(value) instead, to perform the operation inplace on the
original object.


  df["Fare"].fillna(df["Fare"].median(), inplace=True)
<ipython-input-35-7e6d2ceef23b>:13: FutureWarning: A value is trying
to be set on a copy of a DataFrame or Series through chained
assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
```

```
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try
using 'df.method({col: value}, inplace=True)' or df[col] =
df[col].method(value) instead, to perform the operation inplace on the
original object.


  df["Embarked"].fillna(df["Embarked"].mode()[0], inplace=True)
```