EXPERIMENT NUMBER : 01

AIM : Create a perceptron with appropriate no. of inputs and outputs. Train it using a fixed increment learning algorithm until no change in weights is required. Output the final weights.

THEORY : A perceptron is one of the simplest types of artificial neural networks. It was introduced by Frank Rosenblatt in 1958 and is based on the concept of a biological neuron.
The perceptron is a linear classifier that can classify input data points into two categories by learning from labeled training data. It works by applying a weighted sum to the input features and passing the result through an activation function, typically a step function.
The perceptron algorithm iteratively updates its weights based on the prediction errors it makes during the training process. The key steps involved are as follows:

1. **Initialization:** Randomly initialize the weights and bias for the perceptron.
2. **Weighted Sum:** Compute the weighted sum of inputs:

$$y_{\text{in}} = \sum_{i=1}^{n} w_i \cdot x_i + b$$

   where w_i are the weights, x_i are the input features, and b is the bias.

3. **Activation Function:** Apply a step activation function:

$$y = \begin{cases} 1 & \text{if } y_{\text{in}} > 0 \\ 0 & \text{otherwise} \end{cases}$$

4. **Weight Update Rule:** If the prediction is incorrect, update the weights using the following rule:

$$w_i(t+1) = w_i(t) + \Delta w_i$$

   where $$\Delta w_i = \eta \cdot (d - y) \cdot x_i$$

   Here, η is the learning rate, d is the desired output, and y is the predicted output.

5. **Training:**Continue updating the weights until the algorithm converges, i.e., when there are no changes in weights for all training samples.

<u>CODE</u> :

#ln[1]:

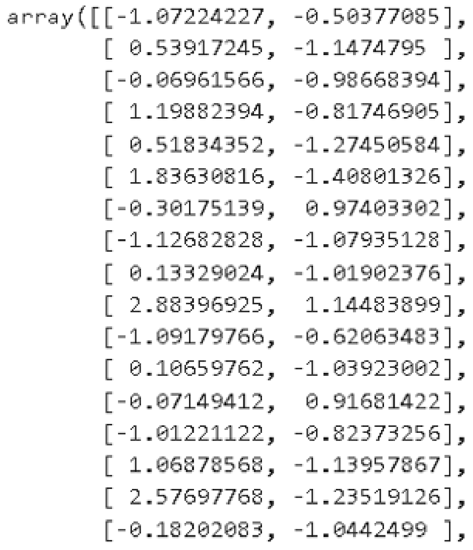
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Perceptron
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
# Generate linearly separable data
X, y = make_classification(n_samples=500, n_features=2, n_informative=1,
                           n_redundant=0, n_clusters_per_class=1, random_state=100)
```

#ln[2]:

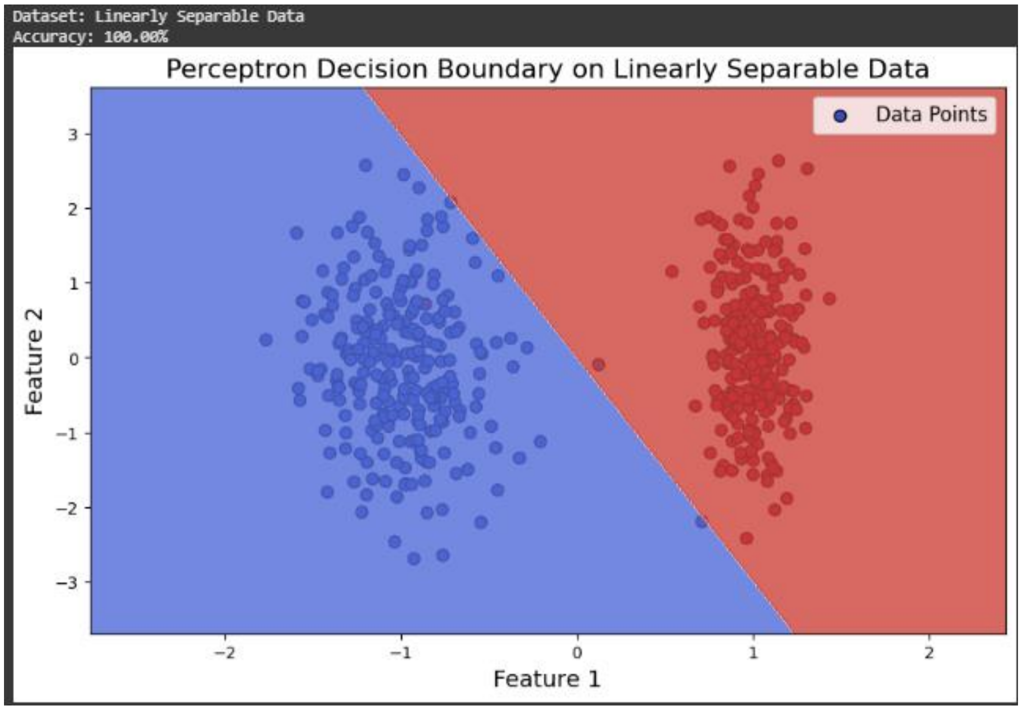
```
# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=100)
# Scale the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
# Train Perceptron
model = Perceptron(max_iter=2, tol=1e-2, random_state=100)
# model = Perceptron()
model.fit(X_train, y_train)
# Evaluate
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Dataset: Linearly Separable Data\nAccuracy: {accuracy * 100:.2f}%")
# Plotting
```

```
plt.figure(figsize=(10, 6))

# Scatter plot for data points

plt.scatter(X[:, 0], X[:, 1], c=y, cmap='coolwarm', s=50, edgecolor='k', label='Data Points')

# Plot decision boundary

x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1

y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),np.arange(y_min, y_max, 0.01))

Z = model.predict(scaler.transform(np.c_[xx.ravel(), yy.ravel()]))

Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.8, cmap='coolwarm')

# Labels and legend

plt.title("Perceptron Decision Boundary on Linearly Separable Data", fontsize=16)

plt.xlabel("Feature 1", fontsize=14)

plt.ylabel("Feature 2", fontsize=14)

plt.legend(fontsize=12)

plt.show()
```

<u>OUTPUT</u> :



```
array([[-1.07224227, -0.50377085],
       [ 0.53917245, -1.1474795 ],
       [-0.06961566, -0.98668394],
       [ 1.19882394, -0.81746905],
       [ 0.51834352, -1.27450584],
       [ 1.83630816, -1.40801326],
       [-0.30175139,  0.97403302],
       [-1.12682828, -1.07935128],
       [ 0.13329024, -1.01902376],
       [ 2.88396925,  1.14483899],
       [-1.09179766, -0.62063483],
       [ 0.10659762, -1.03923002],
       [-0.07149412,  0.91681422],
       [-1.01221122, -0.82373256],
       [ 1.06878568, -1.13957867],
       [ 2.57697768, -1.23519126],
       [-0.18202083, -1.0442499 ],
```



```
Dataset: Linearly Separable Data
Accuracy: 100.00%
```

<u>LEARNING OUTCOMES</u> :

EXPERIMENT NUMBER : 02

AIM : Create a simple ADALINE network with appropriate no. of input and output nodes. Train it using delta learning rules until no change in weights is required. Output the final weights.

THEORY :

Introduction to ADALINE :

ADALINE (Adaptive Linear Neuron) is a type of artificial neural network that consists of a single-layer perceptron using the Delta Learning Rule for weight adjustments. It was introduced by Bernard Widrow and Marcian Hoff in the 1960s. ADALINE is particularly useful for linear classification and function approximation problems.

Structure of ADALINE

- The ADALINE network consists of:
  - Input layer: Neurons representing features of the input data.
  - Weight vector (W): Weights associated with each input feature.
  - Summation unit: Computes the weighted sum of inputs.
  - Activation function: ADALINE uses an identity activation function (i.e., linear function), meaning the output is just the weighted sum.
  - Output: The final computed result.

Mathematically, the net input is calculated as:

$$Y = \sum wixi + b$$

where:

- xi are the input features,
- wi are the corresponding weights,
- b is the bias term,
- y is the output before applying any threshold.

For classification tasks, the final output is determined using a threshold function:

$$\hat{y} = \begin{cases} +1, & y \geq 0 \\ -1, & y < 0 \end{cases}$$

Delta Learning Rule :

The Delta Learning Rule (Widrow-Hoff rule) is used to train the ADALINE network by minimizing the mean squared error (MSE) between predicted and actual values. The weight update rule is:

$$\Delta\ wi = \eta(t - y)xi$$

$$wi^{\wedge\{new\}} = wi^{\wedge\{old\}} + \Delta\ wi$$

where:

- η is the learning rate (small positive value),

- t is the target output,
- y is the actual output (before thresholding),
- xi is the input feature.

The process is repeated iteratively until convergence (i.e., when there is no significant change in weights).

Steps to Implement ADALINE :

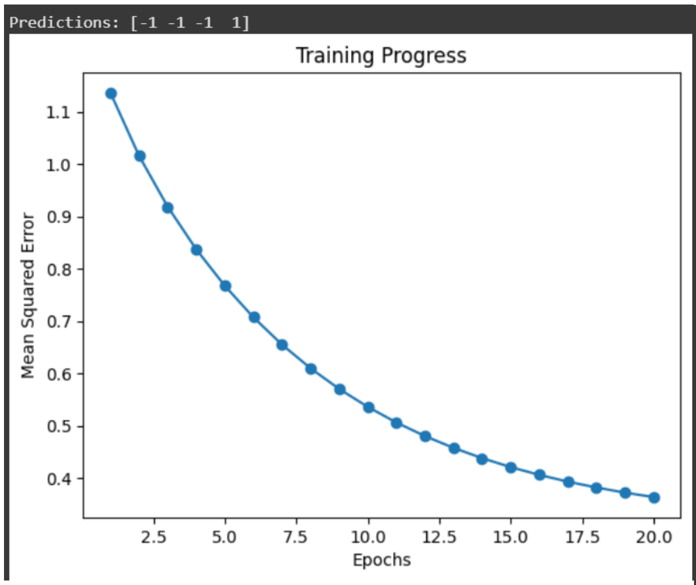
1. Initialize weights randomly or set them to small values.
2. Compute the net input using the weighted sum of inputs.
3. Calculate the error as the difference between the desired and actual output.
4. Update weights using the Delta Rule.
5. Repeat training until the change in weights becomes negligible.
6. Output the final weights after convergence.

CODE :

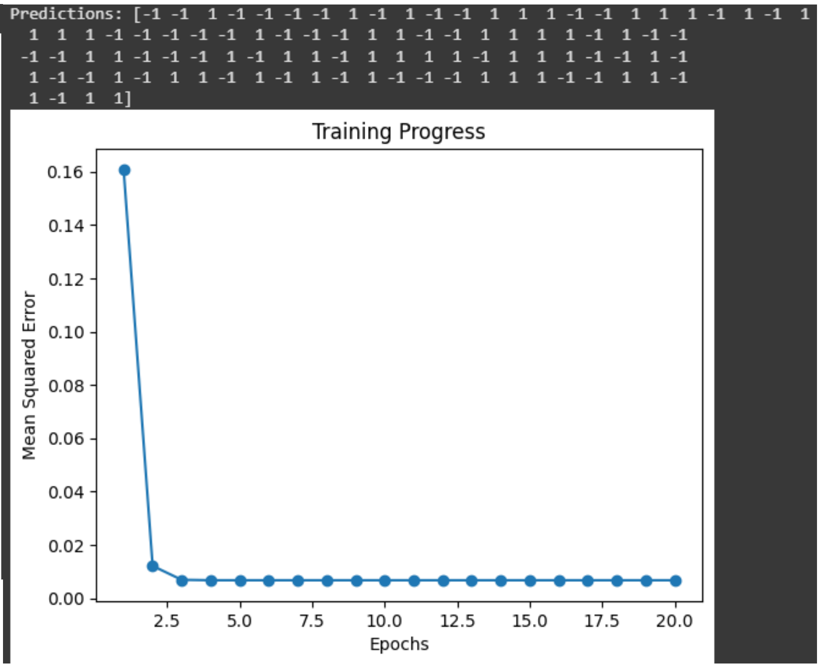
#ln[1]:

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

class Adaline:
    def init (self, learning_rate=0.01, epochs=100):
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.weights = None
        self.bias = None
        self.errors = []
    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0
        for _ in range(self.epochs):
            total_error = 0
            for xi, target in zip(X, y):
                output = self.predict_raw(xi)
                error = target - output
                self.weights += self.learning_rate * error * xi
                self.bias += self.learning_rate * error
                total_error += error**2    # Squared error for tracking performance
                self.errors.append(total_error / n_samples)
    def predict_raw(self, X):
        return np.dot(X, self.weights) + self.bias

    def predict(self, X):
        return np.where(self.predict_raw(X) >= 0.0, 1, -1)

    def plot_errors(self):
        plt.plot(range(1, len(self.errors) + 1),
        self.errors, marker='o')
        plt.xlabel('Epochs')
        plt.ylabel('Mean Squared Error')
        plt.title('Training Progress')
        plt.show()

    X, y = make_classification(
        n_samples=100, # Number of samples
        n_features=2, # Number of features
        n_informative=2,
        n_redundant=0,
        n_clusters_per_class=1,
        random_state=42 )

# Convert class labels from {0, 1} to {-1, 1}
y = np.where(y == 0, -1, 1)

# Training the ADALINE model
adaline = Adaline(learning_rate=0.1, epochs=20)
adaline.fit(X, y)

# Predictions
predictions = adaline.predict(X)
print("Predictions:", predictions)

# Plot the error over epochs
adaline.plot_errors()
```

<u>OUTPUT</u> :



```
Predictions: [-1 -1 -1  1]
```



```
Predictions: [-1 -1  1 -1 -1 -1 -1  1 -1  1 -1 -1  1  1  1 -1 -1  1  1  1 -1  1 -1  1
  1  1  1 -1 -1 -1 -1 -1  1 -1 -1 -1  1  1 -1 -1  1  1  1  1 -1  1 -1 -1
 -1 -1  1  1 -1 -1 -1  1 -1  1  1 -1  1  1  1  1 -1  1  1  1 -1 -1  1 -1
  1 -1 -1  1 -1  1  1 -1  1 -1  1 -1  1 -1 -1 -1  1  1  1 -1 -1  1  1 -1
  1 -1  1  1]
```

<u>LEARNING OUTCOMES</u> :

EXPERIMENT NUMBER : 03

AIM : Implement multilayer perceptron algorithm for MNIST Handwritten Digit Classification.

THEORY :
The Multilayer Perceptron (MLP) is a type of feedforward artificial neural network that consists of multiple layers of neurons. It is widely used for image classification tasks, such as the MNIST handwritten digit recognition, which consists of 28×28 grayscale images of digits (0-9).
Architecture of MLP :
A typical MLP for MNIST classification consists of:

1. Input Layer: 28×28 pixels = 784 neurons (one for each pixel)
2. Hidden Layers: 1 or more layers with a chosen number of neurons
3. Output Layer: 10 neurons (one for each digit from 0 to 9)

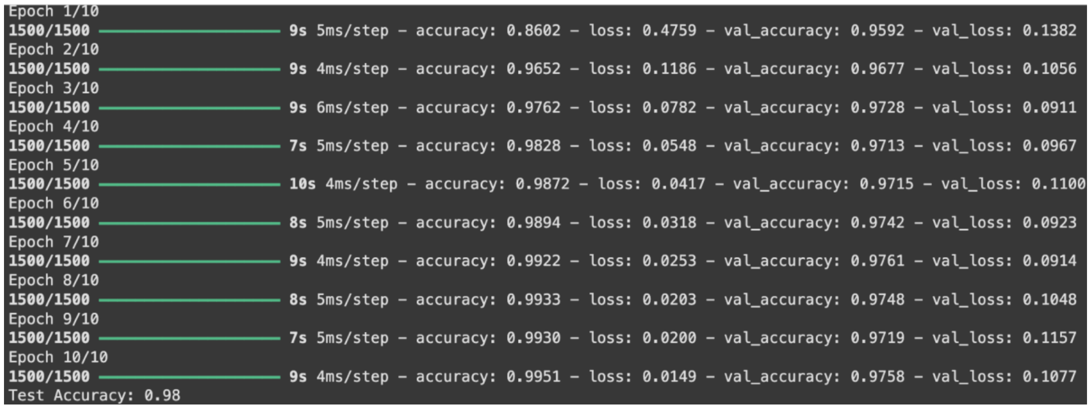
CODE :

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.datasets import mnist
import matplotlib.pyplot as plt

# Load the MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Normalize the data
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0

# One-hot encode the labels
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Build the ANN model
model = Sequential([
        Flatten(input_shape=(28, 28)), # Flatten the 28x28 images into a 1D vector
```

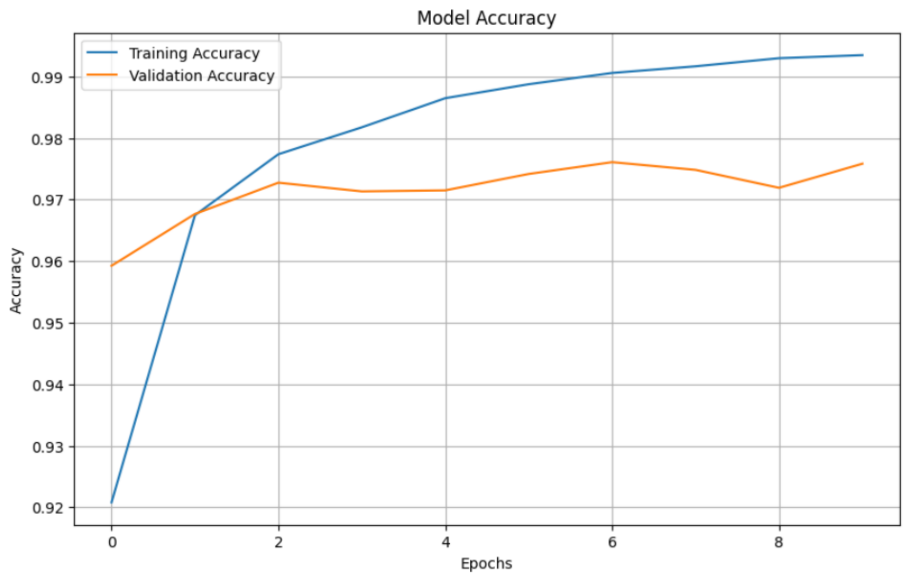
```
    Dense(128, activation='relu'), # Hidden layer with 128 neurons
    Dense(64, activation='relu'), # Hidden layer with 64 neurons
    Dense(10, activation='softmax') # Output layer for 10 classes
    ])

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

# Train the model
history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2, verbose=1)

# Evaluate the model on the test data
test_loss, test_accuracy = model.evaluate(X_test, y_test, verbose=0)
print(f"Test Accuracy: {test_accuracy:.2f}")

# Plot training and validation accuracy
plt.figure(figsize=(10, 6))
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.grid()
plt.show()

# Plot training and validation loss
plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid()
plt.show()
```

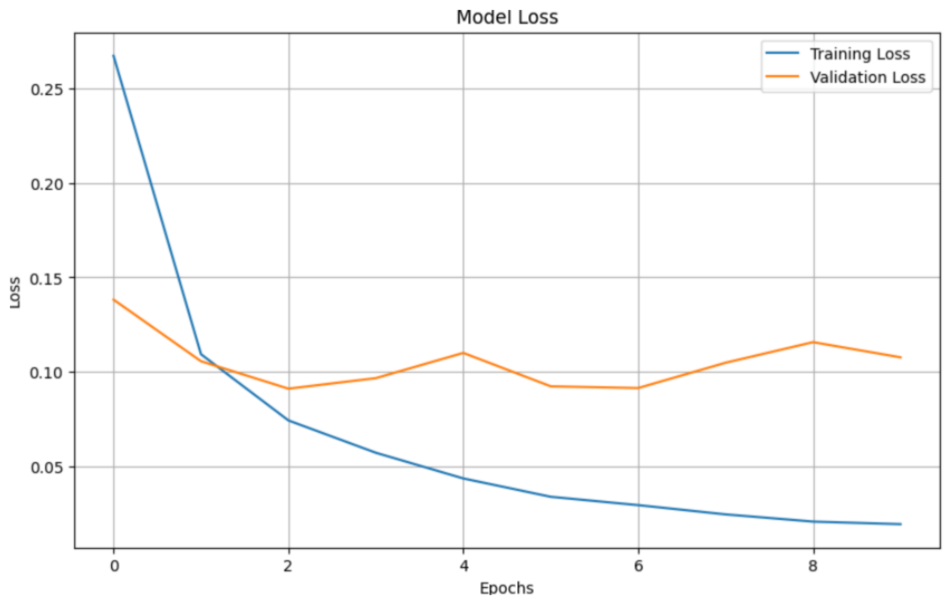
```
# Predict on test data and visualize some predictions
import numpy as np
predictions = np.argmax(model.predict(X_test), axis=1)
actual = np.argmax(y_test, axis=1)

# Visualize first 10 test samples
plt.figure(figsize=(12, 6))
for i in range(10):
        plt.subplot(2, 5, i+1)
        plt.imshow(X_test[i], cmap='gray')
        plt.title(f"Pred: {predictions[i]} | Actual: {actual[i]}")
        plt.axis('off')
plt.tight_layout()
plt.show()
```

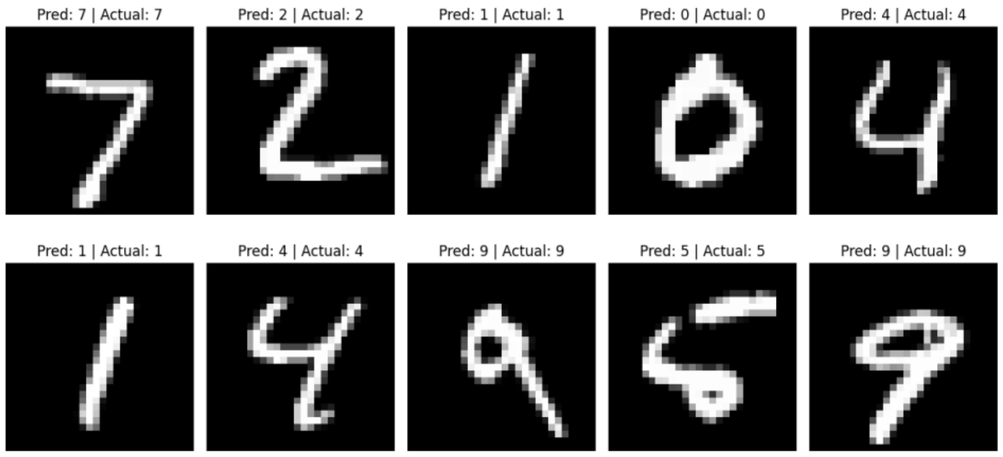
<u>OUTPUT</u> :



```
Epoch 1/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - accuracy: 0.8602 - loss: 0.4759 - val_accuracy: 0.9592 - val_loss: 0.1382
Epoch 2/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 4ms/step - accuracy: 0.9652 - loss: 0.1186 - val_accuracy: 0.9677 - val_loss: 0.1056
Epoch 3/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.9762 - loss: 0.0782 - val_accuracy: 0.9728 - val_loss: 0.0911
Epoch 4/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.9828 - loss: 0.0548 - val_accuracy: 0.9713 - val_loss: 0.0967
Epoch 5/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy: 0.9872 - loss: 0.0417 - val_accuracy: 0.9715 - val_loss: 0.1100
Epoch 6/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.9894 - loss: 0.0318 - val_accuracy: 0.9742 - val_loss: 0.0923
Epoch 7/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 4ms/step - accuracy: 0.9922 - loss: 0.0253 - val_accuracy: 0.9761 - val_loss: 0.0914
Epoch 8/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.9933 - loss: 0.0203 - val_accuracy: 0.9748 - val_loss: 0.1048
Epoch 9/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.9930 - loss: 0.0200 - val_accuracy: 0.9719 - val_loss: 0.1157
Epoch 10/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 4ms/step - accuracy: 0.9951 - loss: 0.0149 - val_accuracy: 0.9758 - val_loss: 0.1077
Test Accuracy: 0.98
```







<u>LEARNING OUTCOMES</u> :