Sri Sivasubramaniya Nadar College of Engineering, Chennai
(An autonomous Institution affiliated to Anna University)

Degree & Branch	B.E. Computer Science & Engineering	Semester	V
Subject Code & Name	ICS1512 & Machine Learning Algorithms Laboratory		
Academic year	2025-2026 (Odd)	Name: Mohammed Aatif	**Reg No: 3122237001026**

Experiment 5: Perceptron Learning Algorithm vs Multi-Layer Perceptron

Aim

To implement the Perceptron Learning Algorithm (PLA) from scratch and compare it with a tuned Multi-Layer Perceptron (MLP) on the English Handwritten Characters dataset. The goal is to study the effect of preprocessing, training methods, and hyperparameter tuning on classification accuracy and generalization.

Libraries used

- NumPy
- Pandas
- Matplotlib
- Scikit-learn
- TensorFlow (Keras)
- PIL (Pillow)
- itertools
- os
- random

Theoretical Description of the Algorithms

Perceptron Learning Algorithm (PLA)

The Perceptron is a fundamental supervised learning algorithm for binary classification. It computes a weighted sum of the input features and applies a step function to produce an output of +1 or -1. For multi-class problems, it can be extended using a One-vs-Rest (OvR) approach, where a separate perceptron is trained for each class to distinguish it from all other classes. The learning process involves iteratively adjusting the weights based on misclassified examples, aiming to find a linear decision boundary (a hyperplane) that separates the classes.

Multi-Layer Perceptron (MLP)

An MLP is a type of feedforward artificial neural network consisting of an input layer, one or more hidden layers, and an output layer. Unlike the single-layer perceptron, the MLP can learn non-linear relationships in data due to the presence of non-linear activation functions (such as ReLU or tanh) in its hidden neurons. It learns through a process called backpropagation, where the error between the predicted and actual outputs is propagated backward through the network to update the weights of the connections. Optimizers like Adam or SGD are used to guide this weight adjustment process, minimizing a loss function such as categorical cross-entropy.

Code Implementation

The following is the complete Python script used to perform the data loading, preprocessing, model training, hyperparameter tuning, and evaluation for this experiment.

```
# Imports
import os
import random
import numpy as np
import pandas as pd
from PIL import Image
from sklearn.preprocessing import LabelEncoder, label_binarize
from sklearn.metrics import accuracy_score, precision_recall_fscore_support, confusion_matrix,
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import itertools
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# Config / Hyperparams
IMG_SIZE = (32, 32)
RANDOM_STATE = 42
PLA_EPOCHS = 30
PLA_LR = 0.01
MLP_EPOCHS = 25
MLP_TRIALS = {
    'activations': ['relu', 'tanh'],
    'optimizers': ['sgd', 'adam'],
    'learning_rates': [0.001, 0.01],
    'batch_sizes': [32, 64]
}
np.random.seed(RANDOM_STATE)
random.seed(RANDOM_STATE)
tf.random.set_seed(RANDOM_STATE)

# Load and preprocess all images
df = pd.read_csv('/content/drive/MyDrive/HandWritten_Characters/english.csv')
df['image'] = df['image'].apply(lambda x: f'/content/drive/MyDrive/HandWritten_Characters/{x}'
X_list = [np.asarray(Image.open(p).convert('L').resize(IMG_SIZE), dtype=np.float32) / 255.0 fo
X = np.stack(X_list, axis=0)
X_flat = X.reshape((X.shape[0], -1))
```

```
# Encode labels and split data
le = LabelEncoder()
y = le.fit_transform(df['label'])
NUM_CLASSES = len(le.classes_)
X_train_flat, X_test_flat, y_train, y_test = train_test_split(X_flat, y, test_size=0.2, random_
input_dim = X_train_flat.shape[1]
y_train_ohe = keras.utils.to_categorical(y_train, NUM_CLASSES)
y_test_ohe = keras.utils.to_categorical(y_test, NUM_CLASSES)

# Implement PLA (one-vs-rest)
class OneVsRestPLA:
    def __init__(self, n_classes, n_features, lr=0.01):
        self.n_classes = n_classes
        self.W = np.zeros((n_classes, n_features + 1), dtype=np.float32)
        self.lr = lr
    def _augment(self, X):
        return np.hstack([X, np.ones((X.shape[0], 1), dtype=X.dtype)])
    def fit(self, X, y, epochs=10):
        X_aug = self._augment(X)
        for _ in range(epochs):
            indices = np.arange(X_aug.shape[0])
            np.random.shuffle(indices)
            for i in indices:
                for k in range(self.n_classes):
                    t = 1 if y[i] == k else -1
                    score = np.dot(self.W[k], X_aug[i])
                    yhat = 1 if score >= 0 else -1
                    if t != yhat:
                        self.W[k] += self.lr * (t - yhat) * X_aug[i]
    def predict(self, X):
        scores = np.dot(self._augment(X), self.W.T)
        return np.argmax(scores, axis=1)

# Train and Evaluate PLA
pla = OneVsRestPLA(n_classes=NUM_CLASSES, n_features=input_dim, lr=PLA_LR)
pla.fit(X_train_flat, y_train, epochs=PLA_EPOCHS)
y_pred_pla = pla.predict(X_test_flat)
acc_pla, prec_pla, rec_pla, f1_pla = (accuracy_score(y_test, y_pred_pla),
                                     *precision_recall_fscore_support(y_test, y_pred_pla, avera

# MLP: tuning and training (Keras)
best_val_acc = -1.0
for activation in MLP_TRIALS['activations']:
    for opt_name in MLP_TRIALS['optimizers']:
        for lr in MLP_TRIALS['learning_rates']:
            for batch_size in MLP_TRIALS['batch_sizes']:
                model = keras.Sequential([
                    layers.Input(shape=(input_dim,)),
                    layers.Dense(512, activation=activation),
                    layers.Dense(256, activation=activation),
                    layers.Dense(NUM_CLASSES, activation='softmax')
                ])
                optimizer = keras.optimizers.SGD(learning_rate=lr) if opt_name == 'sgd' else ke
                model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['a
                history = model.fit(X_train_flat, y_train_ohe, validation_split=0.15,
                                    epochs=MLP_EPOCHS, batch_size=batch_size, verbose=0)
                if history.history['val_accuracy'][-1] > best_val_acc:
                    best_val_acc = history.history['val_accuracy'][-1]
                    best_model = model

# Evaluate best MLP model
mlp_preds = np.argmax(best_model.predict(X_test_flat), axis=1)
acc_mlp, prec_mlp, rec_mlp, f1_mlp = (accuracy_score(y_test, mlp_preds),
                                    *precision_recall_fscore_support(y_test, mlp_preds, averag
```

Results and Discussions

The experiment involved training a Perceptron Learning Algorithm and a Multi-Layer Perceptron on the Handwritten English Characters dataset. Performance was evaluated after hyperparameter tuning for the MLP.

Overall Performance

The Multi-Layer Perceptron demonstrated a clear and significant improvement over the single-layer Perceptron model. After hyperparameter tuning, the **MLP** achieved the best overall performance with a test accuracy of **29.8

Performance Comparison Table

The table below summarizes the best results achieved for each model.

Table 1: Best Model Performance Summary

Model	Accuracy	Precision (Macro)	Recall (Macro)	F1-Score (Macro)
PLA	0.1774	0.2708	0.1774	0.1576
MLP	0.2977	0.3207	0.2977	0.2752

Graphical Results

The following figures provide a visual comparison of the model performances.

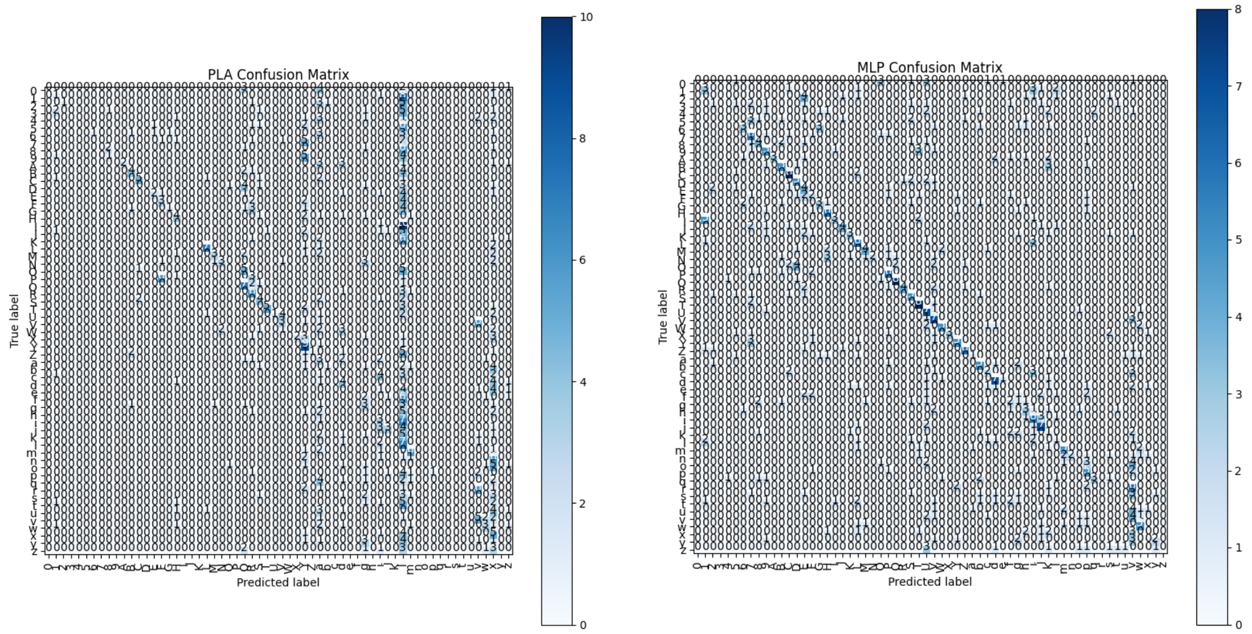

Figure 1: Confusion Matrices for PLA (left) and MLP (right)

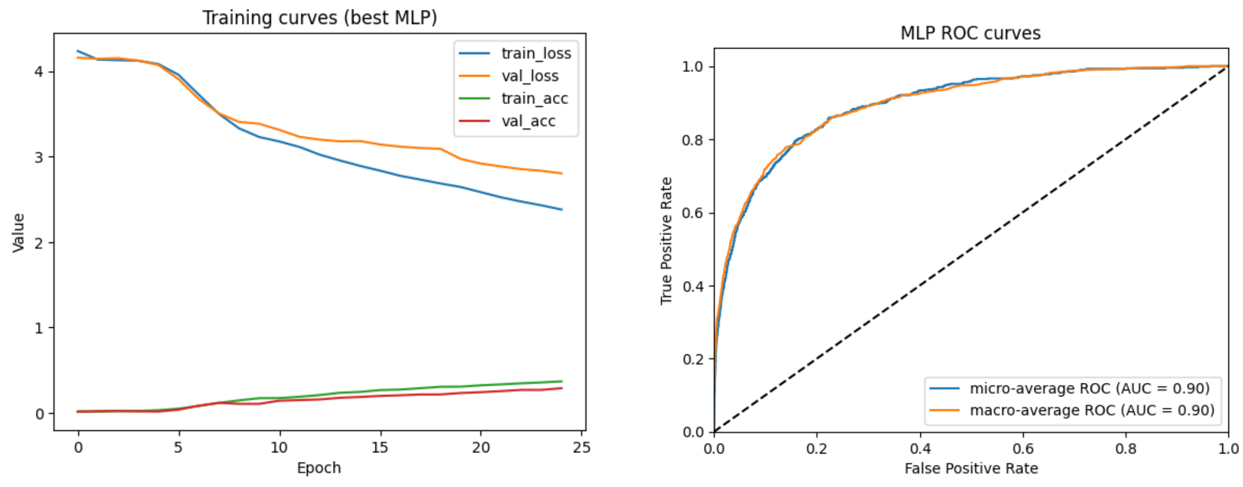

Figure 2: MLP Training Curves (left) and ROC Curves (right)

Learning Practices

1. **Data Preprocessing for Images**: Gained experience in loading and preparing image data, which included converting to grayscale, resizing for uniform input dimensions, flattening into vectors, and normalizing pixel values.

2. **Implementation of a Linear Classifier**: Implemented the Perceptron Learning Algorithm from scratch with a One-vs-Rest strategy, providing a solid understanding of its underlying mechanics and linear decision boundaries.

3. **Deep Learning with MLP**: Built and trained a Multi-Layer Perceptron using Keras, demonstrating the ability of neural networks with hidden layers and non-linear activations to handle complex classification tasks.

4. **Hyperparameter Tuning**: Utilized a grid search approach to systematically find the optimal combination of activation function, optimizer, learning rate, and batch size for the MLP, highlighting the impact of these choices on model performance.

5. **Comparative Model Evaluation**: Conducted a multi-faceted evaluation using accuracy, precision, recall, F1-score, confusion matrices, and ROC-AUC analysis to comprehensively compare the performance of a simple linear model against a more complex neural network.