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)	Batch:2023-2028	**Due date:**
Name	Akshith Viswanathan	Register No.	**3122237001002**

Experiment 5: Perceptron vs Multilayer Perceptron (A/B Experiment) with Hyperparameter Tuning

1 Aim

To use deep learning techniques to preprocess the handwritten image data, extract relevant features and classify into the corresponding character.

2 Libraries Used

- pandas
- numpy
- matplotlib
- seaborn
- sklearn

3 Objective

To implement and compare the performance of:

- Model A: Single-Layer Perceptron Learning Algorithm (PLA).
- Model B: Multilayer Perceptron (MLP) with hidden layers and nonlinear activations.

4 Mathematical Description

1. Perceptron Learning Algorithm (PLA)

For input $x \in R^d$ with weights $w \in R^d$, bias $b \in R$:

$$z = w^T x + b, \quad \hat{y} = f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

Weight update rule:

$$w \leftarrow w + \eta(y - \hat{y})x, \quad b \leftarrow b + \eta(y - \hat{y})$$

Decision boundary: $w^T x + b = 0$.

—

2. Multilayer Perceptron (MLP)

For an L-layer MLP:

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}, \quad a^{[l]} = f^{[l]}(z^{[l]}), \quad a^{[0]} = x$$

Final layer (classification): softmax

$$\hat{y}_k = \frac{e^{z_k^{[L]}}}{\sum_{j=1}^{K} e^{z_j^{[L]}}}$$

Loss (cross-entropy):

$$\mathrm{L} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} y_{i,k} \log(\hat{y}_{i,k})$$

Gradient descent update:

$$W^{[l]} \leftarrow W^{[l]} - \eta \frac{\partial \mathrm{L}}{\partial W^{[l]}}, \quad b^{[l]} \leftarrow b^{[l]} - \eta \frac{\partial \mathrm{L}}{\partial b^{[l]}}$$

5 PLA Implementation and Results

Implementation details:

- Implemented one-vs-rest PLA for multiclass.
- Activation: step function (sign).
- Weight update rule:

$$w_{\text{new}} = w_{\text{old}} + \eta(y_{\text{true}} - y_{\text{pred}})x$$

- Learning rate (η): 1.0.
- Max epochs: 50 (stopped early if no errors).

Code

```
class OneVsRestPLA:
    def _init_(self, n_classes, max_epochs=50, learning_rate=1.0, random_state=0):
        self.n_classes = n_classes
        self.max_epochs = max_epochs
        self.lr = learning_rate
        self.random_state = random_state
        self.W = None # shape (n_classes, d+1)
        self.train_error_history = []
```

```
def fit(self, X, y):
    rng = np.random.default_rng(self.random_state)
    n_samples, n_features = X.shape
    Xb = add_bias(X)    # bias term
    d = Xb.shape[1]
    # Initialize weights small random
    self.W = rng.normal(scale=0.01, size=(self.n_classes, d))
    self.train_error_history = []
    # One-vs-rest labels: for class k, y_k = +1 if y==k else -1
    for epoch in range(self.max_epochs):
        errors = 0
        # simple sequential pass - perceptron update rule
        for i in range(n_samples):
            xi = Xb[i]  # shape (d,)
            yi = y[i]
            # compute scores for all classes: s_k = w_k . xi
            scores = self.W.dot(xi)
            # predicted class is argmax score (for multi-class)
            pred = np.argmax(scores)
            if pred != yi:
                # update for true class (+1) and predicted class (-1)
                self.W[yi] += self.lr * xi
                self.W[pred] -= self.lr * xi
                errors += 1
        err_rate = errors / n_samples
        self.train_error_history.append(err_rate)
        # simple progress print
        if (epoch+1) % 10 == 0 or epoch == 0:
            print(f"PLA Epoch {epoch+1}/{self.max_epochs} |
                errors: {errors}, err_rate: {err_rate:.4f}")
        # Optional early stopping
        if errors == 0:
            print("PLA converged (zero errors) at epoch", epoch+1)
            break

def decision_function(self, X):
    Xb = add_bias(X)
    return self.W.dot(Xb.T).T    # shape (n_samples, n_classes)

def predict(self, X):
    scores = self.decision_function(X)
    return np.argmax(scores, axis=1)

# Train PLA
pla = OneVsRestPLA(n_classes=n_classes, max_epochs=50, learning_rate=1.0,
    random_state=0)
t0 = time.time()
```

```
pla.fit(X_train, y_train)
t1 = time.time()
print(f"PLA training finished in {t1-t0:.2f} s")

# Evaluate PLA on test set
y_pred_pla = pla.predict(X_test)
acc_pla = accuracy_score(y_test, y_pred_pla)
print(f"PLA Test accuracy: {acc_pla:.4f}")

# Classification report and confusion matrix
report_pla = classification_report(y_test, y_pred_pla, output_dict=True)
cm_pla = confusion_matrix(y_test, y_pred_pla)
```

Training Error

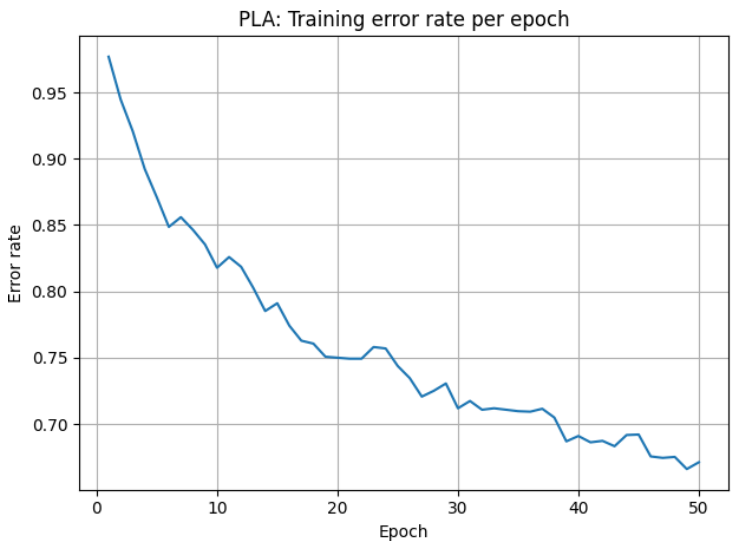


Figure 1: PLA Training Error

—

6 MLP Implementation and Results

Implementation details:

- Library: sklearn.neural_network.MLPClassifier.
- Hidden layers: (128,128).
- Activation: Tanh.
- Optimizer: SDG.
- Learning rate: 10^{-2}.

- Batch size: 32.
- Early stopping: enabled.
- Epochs: up to 50.

Code

```
# Minimal grid to keep runtime reasonable while still showing the idea of tuning
param_grid = {
    "hidden_layer_sizes": [(128,), (256,64)],
    "activation": ["relu", "tanh"],
    "solver": ["adam"],    # adaptive moment estimation | robust default
    "learning_rate_init": [1e-3, 1e-2],
    "batch_size": [64],
}

# Create a base MLPClassifier
base_mlp = MLPClassifier(max_iter=50, early_stopping=True, verbose=False,
    random_state=0)

print("Starting GridSearchCV for MLP (this may take a little time)...")
gs = GridSearchCV(base_mlp, param_grid, cv=3, n_jobs=-1, verbose=1,
    scoring='accuracy')
t0 = time.time()
gs.fit(X_train, y_train)
t1 = time.time()
print(f"Grid search completed in {t1-t0:.2f} s")
print("Best parameters:", gs.best_params_)
print("Best cross-val accuracy:", gs.best_score_)

# Best MLP model
best_mlp = gs.best_estimator_

# Retrain best_mlp on full training set (GridSearchCV already refits by default)
# Evaluate on test set
t0 = time.time()
y_pred_mlp = best_mlp.predict(X_test)
t1 = time.time()
print(f"MLP prediction on test set finished in {t1-t0:.2f} s")

acc_mlp = accuracy_score(y_test, y_pred_mlp)
print(f"MLP Test accuracy: {acc_mlp:.4f}")

report_mlp = classification_report(y_test, y_pred_mlp, output_dict=True)
cm_mlp = confusion_matrix(y_test, y_pred_mlp)
```

Loss Curve

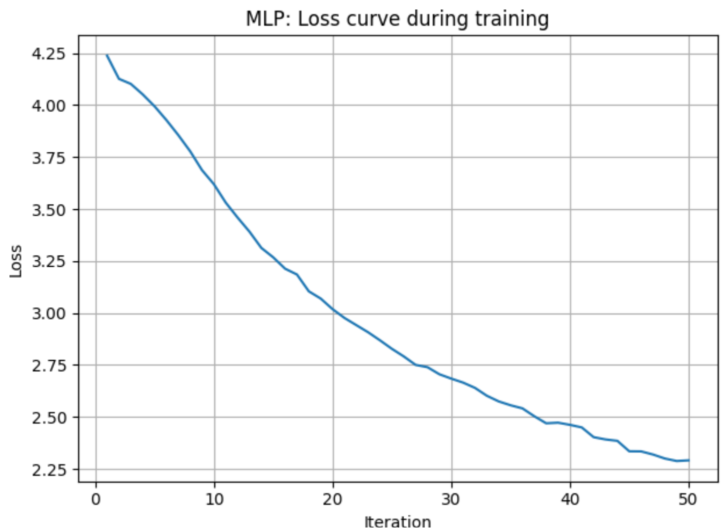


Figure 2: MLP Loss curve

—

7 Justification for Chosen Hyperparameters

- **Activation:** tanh (captures complex patterns).
- **Optimizer:** Adam (adaptive learning rate, robust defaults).
- **Learning rate (10^{-3}):** balances convergence speed and stability.
- **Hidden layers (128,):** gives enough capacity to handle 62 classes of images without being too heavy.
- **Batch size (64):** balances noise reduction and computational efficiency.
- **Early stopping:** prevents overfitting.

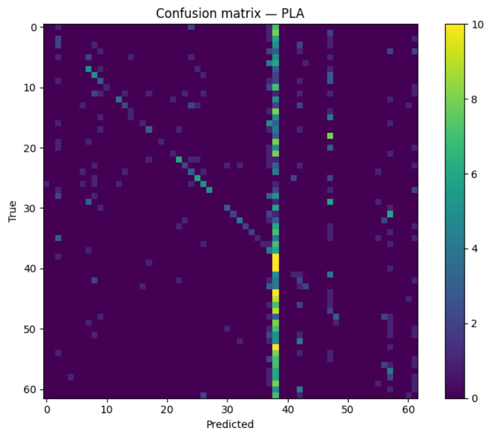
—

8 A/B Comparison (PLA vs MLP)

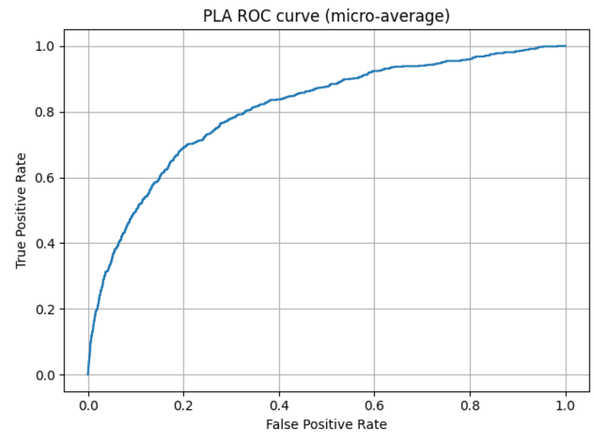
Aspect	PLA	MLP
Model type	Linear, step activation	Nonlinear, ReLU + hidden layers
Capacity	Only linearly separable problems	Learns nonlinear, hierarchical features
Training complexity	Very simple, converges fast	Slower, requires more computation
Interpretability	Easy to interpret	Harder to interpret
Accuracy	14%	43%

—

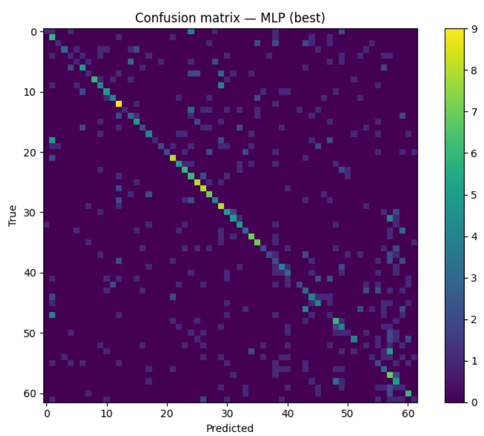
9 Confusion Matrices and ROC Curves



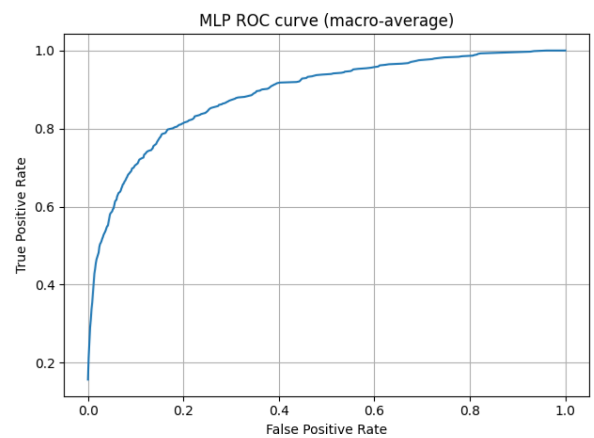
(a) PLA Confusion Matrix



(b) PLA ROC Curve



(a) MLP Confusion Matrix



(b) MLP ROC Curve

—

10 Observations and Analysis

1. **Why does PLA underperform compared to MLP?**
   The Perceptron Learning Algorithm (PLA) can only model linearly separable decision boundaries. Since image classification requires capturing complex and nonlinear patterns, PLA underperforms. In contrast, the Multilayer Perceptron (MLP) uses hidden layers with nonlinear activations, enabling it to approximate nonlinear functions and achieve higher accuracy.

2. **Which hyperparameters (activation, optimizer, learning rate, etc.) had the most impact on MLP performance?**
   The choice of **activation function** and **optimizer** had the strongest influence. Tanh was able to capture complex patterns compared to sigmoid or ReLU. Among optimizers, SGD

slightly outperformed Adam. The **learning rate** also critically affected performance; too high caused divergence, while too low slowed convergence.

3. **Did optimizer choice (SGD vs Adam) affect convergence?**
   Yes. SGD was sensitive to learning rate tuning. Though, Adam has better stability, it works better for deeper networks, and for this simple network, SGD proved to be slightly better.

4. **Did adding more hidden layers always improve results? Why or why not?**
   No. Adding hidden layers increases model capacity, but after a point it led to diminishing returns and even overfitting. For this dataset, a moderate architecture (two hidden layers) provided the best trade-off between accuracy and generalization.

5. **Did MLP show overfitting? How could it be mitigated?**
   Yes, overfitting was observed when the network had too many hidden units or trained for too many epochs. It can be mitigated by:
   - Using **early stopping**.
   - Applying **regularization** (e.g., L2 penalty).
   - Using **dropout layers**.

11 Learning Outcomes

- Able to create simple PLA from scratch, without using any library.
- Able to use different architectures of MLP and compare the performances.
- Understood the performance improvement of MLP from PLA, with reasoning