**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:** |

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

# 1 Aim:

To implement and compare the performance of a Single-Layer Perceptron Learning Algorithm (PLA) and a Multilayer Perceptron (MLP) on the English Handwritten Characters dataset, and to evaluate them using accuracy, precision, recall, F1-score, confusion matrix, ROC curves, and convergence plots.

# 2 Libraries used:

- Numpy
- Pandas
- Matplotlib
- Scikit-learn
- Seaborn
- PIL
- PyTorch

# 3 Objective:

The objective of this assignment is to implement and compare the performance of a Single-Layer Perceptron Learning Algorithm (PLA) and a Multilayer Perceptron (MLP) on the English Handwritten Characters dataset. The task involves preprocessing the dataset, training both models, and tuning hyperparameters such as learning rate, batch size, activation functions, and optimizers. The models are evaluated using metrics like accuracy, precision, recall, F1-score, confusion matrix, ROC curves, and convergence plots. Finally, the results are analyzed to highlight the strengths, weaknesses, and practical differences between PLA and MLP.

# 4 Preprocessing Steps

## 4.1 Image Conversion and Resizing

All images were converted to grayscale to reduce complexity and ensure uniform pixel representation. They were then resized to $28 \times 28$ pixels to maintain consistency across the dataset.

## 4.2 Flattening and Normalization

Each resized image was flattened into a 1D feature vector to serve as input for the models. The pixel values were normalized to the range [0,1], which helps in stabilizing and accelerating the training process.

## 4.3 Label Encoding

The character labels, originally in alphanumeric form, were encoded into integer values using `LabelEncoder`. This step ensured compatibility with both PLA and MLP training.

## 4.4 Dataset Splitting

The dataset was stratified and split into training, validation, and test sets. Stratification preserved the class distribution across splits, ensuring fair evaluation of model performance.

# 5 PLA Implementation:

# pla

September 9, 2025

### 0.1 Deep Learning

[10]:
```python
import os
import numpy as np
import pandas as pd
from PIL import Image
from sklearn.preprocessing import LabelEncoder, LabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import (accuracy_score, precision_recall_fscore_support,
                             classification_report, confusion_matrix,␣
 ↪roc_curve, auc)
import matplotlib.pyplot as plt
import seaborn as sns

# reproducibility
RNG_SEED = 42
np.random.seed(RNG_SEED)
```

[2]:
```python
csv_path = "archive/english.csv"
df = pd.read_csv(csv_path)
print("Columns:", df.columns.tolist())
print(df.head())
```

```
Columns: ['image', 'label']
                image label
0  Img/img001-001.png     0
1  Img/img001-002.png     0
2  Img/img001-003.png     0
3  Img/img001-004.png     0
4  Img/img001-005.png     0
```

[3]:
```python
# params
ROOT = "archive"          # csv paths already contain "Img/..." so join ROOT +␣
 ↪df['image']
IMG_SIZE = (28, 28)       # width, height

images = []
labels = []
```

```python
missing = []

for idx, row in df.iterrows():
    img_rel = row["image"]       # e.g. "Img/img001-001.png"
    label_raw = row["label"]     # may be int-like or str
    img_path = os.path.join(ROOT, img_rel)
    if not os.path.exists(img_path):
        missing.append(img_path)
        continue
    im = Image.open(img_path).convert("L")        # grayscale
    im = im.resize(IMG_SIZE)                      # resize
    arr = np.asarray(im, dtype=np.float32).flatten() / 255.0
    images.append(arr)
    labels.append(label_raw)

print("Missing files (count):", len(missing))
X = np.vstack(images)   # shape (N, D)
y_raw = np.array(labels)  # raw labels (strings or numbers)
print("Loaded X:", X.shape, " y:", y_raw.shape, " unique labels:", np.
 ↪unique(y_raw).shape[0])
```

```
Missing files (count): 0
Loaded X: (3410, 784)  y: (3410,)  unique labels: 62
```

[4]:

```python
# encode to 0..C-1 for indexing in perceptron
le = LabelEncoder()
y = le.fit_transform(y_raw)   # y is integer array

# stratified splits
X_train_full, X_test, y_train_full, y_test = train_test_split(
    X, y, test_size=0.2, random_state=RNG_SEED, stratify=y)

# carve out validation from training
X_train, X_val, y_train, y_val = train_test_split(
    X_train_full, y_train_full, test_size=0.2, random_state=RNG_SEED,␣
 ↪stratify=y_train_full)

print("Shapes -> train:", X_train.shape, y_train.shape,
      "val:", X_val.shape, y_val.shape, "test:", X_test.shape, y_test.shape)
print("Classes (label -> idx) example:", list(zip(le.classes_[:8], range(8))))
```

```
Shapes -> train: (2182, 784) (2182,) val: (546, 784) (546,) test: (682, 784)
(682,)
Classes (label -> idx) example: [(np.str_('0'), 0), (np.str_('1'), 1),
(np.str_('2'), 2), (np.str_('3'), 3), (np.str_('4'), 4), (np.str_('5'), 5),
(np.str_('6'), 6), (np.str_('7'), 7)]
```

[13]:
```
class PLA:
    def __init__(self, input_dim, n_classes, lr=0.01, epochs=10):
        self.W = np.zeros((n_classes, input_dim))
        self.lr = lr
        self.epochs = epochs
        self.errors_ = []   # track misclassifications

    def fit(self, X, y):
        for _ in range(self.epochs):
            errors = 0
            for xi, target in zip(X, y):
                scores = self.W @ xi
                y_pred = np.argmax(scores)
                if y_pred != target:
                    self.W[target] += self.lr * xi
                    self.W[y_pred] -= self.lr * xi
                    errors += 1
            self.errors_.append(errors / len(y))  # error rate per epoch

    def predict(self, X):
        scores = self.W @ X.T
        return np.argmax(scores, axis=0)
```

[14]:
```
# Hyperparameter search
lr_values = [0.1, 0.01, 0.001]
epoch_values = [10, 20, 50]

best_acc = 0
best_params = None
best_model = None

for lr in lr_values:
    for ep in epoch_values:
        pla = PLA(input_dim=X_train.shape[1], n_classes=len(le.classes_),␣
 ↪lr=lr, epochs=ep)
        pla.fit(X_train, y_train)
        val_preds = pla.predict(X_val)
        acc = accuracy_score(y_val, val_preds)
        print(f"lr={lr}, epochs={ep}, val_acc={acc:.4f}")
        if acc > best_acc:
            best_acc = acc
            best_params = (lr, ep)
            best_model = pla

print("\nBest Params:", best_params, "Validation Accuracy:", best_acc)
```

```
lr=0.1, epochs=10, val_acc=0.0916
lr=0.1, epochs=20, val_acc=0.1355
```

```
lr=0.1, epochs=50, val_acc=0.1667
lr=0.01, epochs=10, val_acc=0.0916
lr=0.01, epochs=20, val_acc=0.1355
lr=0.01, epochs=50, val_acc=0.1667
lr=0.001, epochs=10, val_acc=0.0916
lr=0.001, epochs=20, val_acc=0.1355
lr=0.001, epochs=50, val_acc=0.1667

Best Params: (0.1, 50) Validation Accuracy: 0.16666666666666666
```

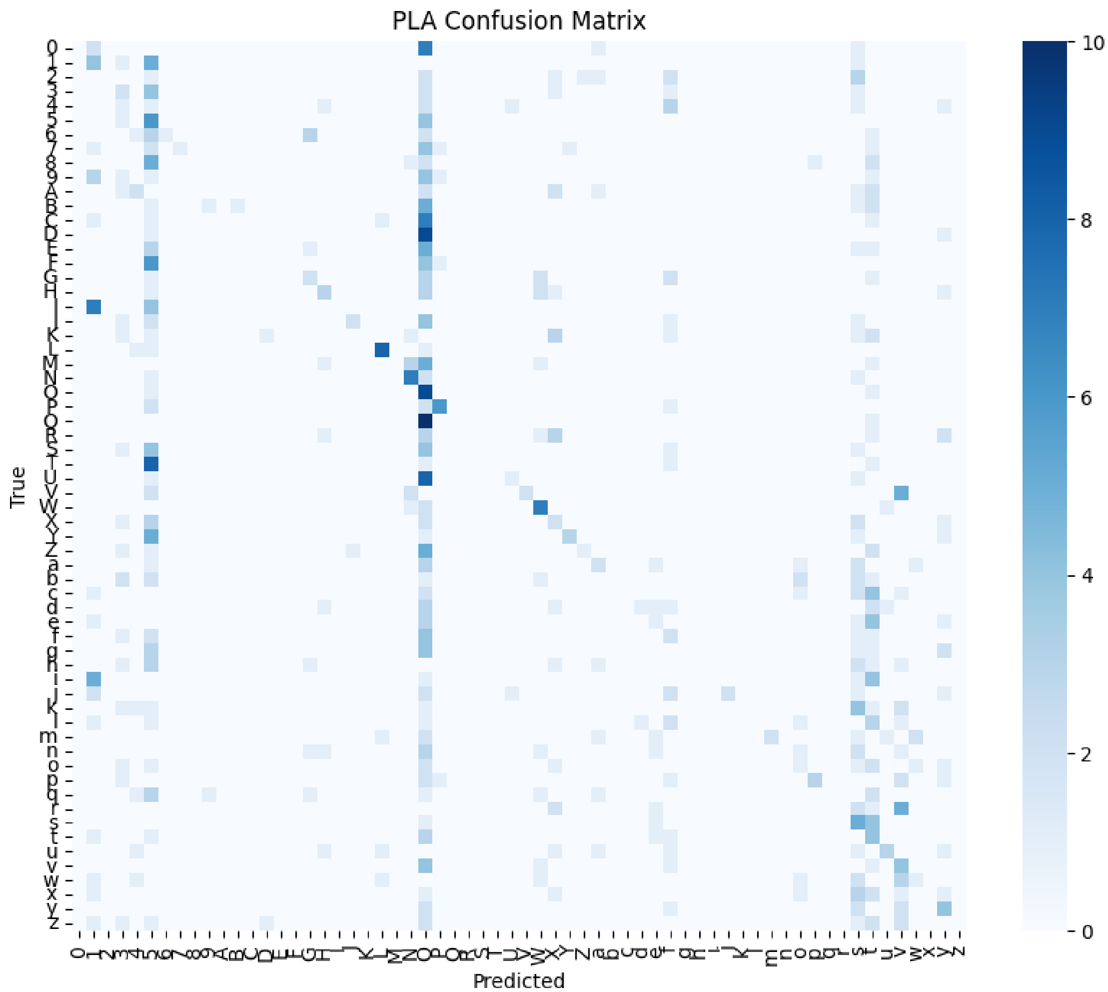
[15]:
```python
# Test evaluation using best PLA
pla_preds = best_model.predict(X_test)
```

[16]:
```python
# --- Basic Metrics ---
print("\nPLA Test Accuracy:", accuracy_score(y_test, pla_preds))
print("\nClassification Report:\n", classification_report(y_test, pla_preds,␣
  ↪target_names=le.classes_))

# --- Confusion Matrix ---
cm = confusion_matrix(y_test, pla_preds)
plt.figure(figsize=(10,8))
sns.heatmap(cm, annot=False, cmap="Blues", xticklabels=le.classes_,␣
  ↪yticklabels=le.classes_)
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("PLA Confusion Matrix")
plt.show()
```

```
PLA Test Accuracy: 0.15102639296187684

Classification Report:
               precision    recall  f1-score   support

           0       0.00      0.00      0.00        11
           1       0.12      0.36      0.19        11
           2       0.00      0.00      0.00        11
           3       0.10      0.18      0.13        11
           4       0.00      0.00      0.00        11
           5       0.06      0.55      0.11        11
           6       1.00      0.09      0.17        11
           7       1.00      0.09      0.17        11
           8       0.00      0.00      0.00        11
           9       0.00      0.00      0.00        11
           A       0.00      0.00      0.00        11
           B       1.00      0.09      0.17        11
           C       0.00      0.00      0.00        11
           D       0.00      0.00      0.00        11
```

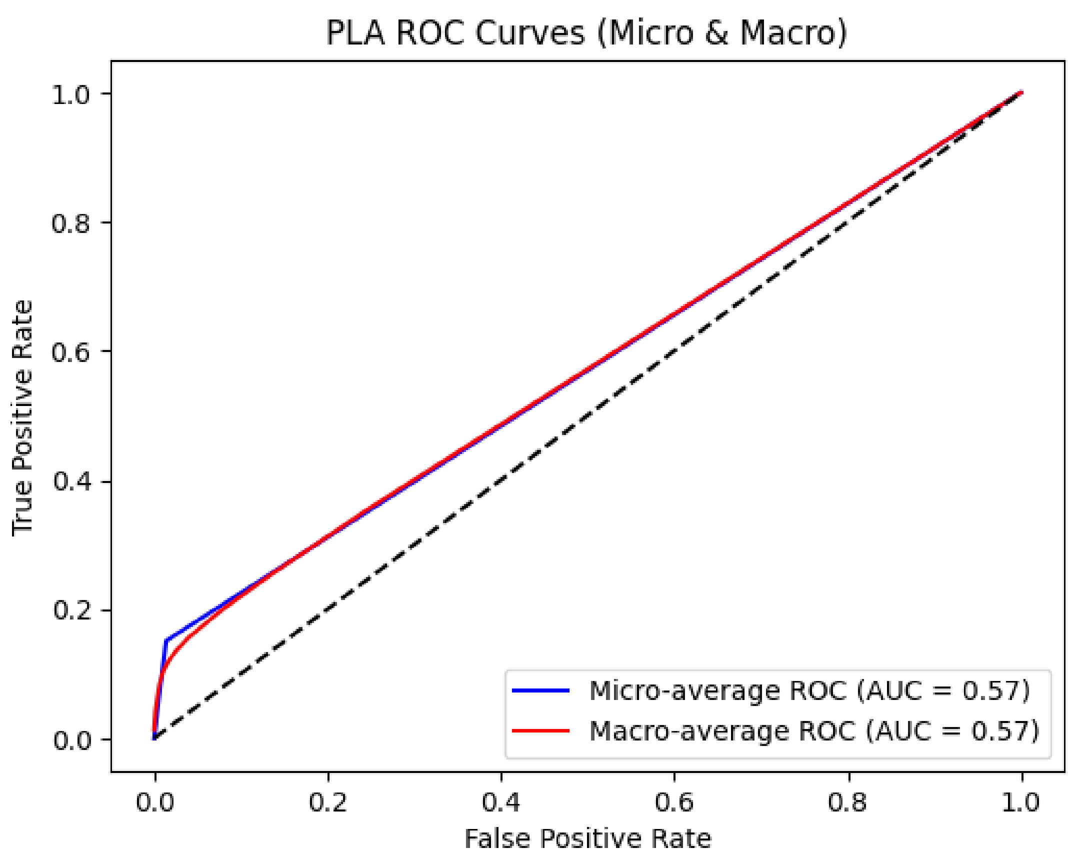
| | | | | |
|---|---|---|---|---|
| E | 0.00 | 0.00 | 0.00 | 11 |
| F | 0.00 | 0.00 | 0.00 | 11 |
| G | 0.22 | 0.18 | 0.20 | 11 |
| H | 0.33 | 0.27 | 0.30 | 11 |
| I | 0.00 | 0.00 | 0.00 | 11 |
| J | 0.67 | 0.18 | 0.29 | 11 |
| K | 0.00 | 0.00 | 0.00 | 11 |
| L | 0.67 | 0.73 | 0.70 | 11 |
| M | 0.00 | 0.00 | 0.00 | 11 |
| N | 0.47 | 0.64 | 0.54 | 11 |
| O | 0.05 | 0.82 | 0.10 | 11 |
| P | 0.60 | 0.55 | 0.57 | 11 |
| Q | 0.00 | 0.00 | 0.00 | 11 |
| R | 0.00 | 0.00 | 0.00 | 11 |
| S | 0.00 | 0.00 | 0.00 | 11 |
| T | 0.00 | 0.00 | 0.00 | 11 |
| U | 0.33 | 0.09 | 0.14 | 11 |
| V | 1.00 | 0.18 | 0.31 | 11 |
| W | 0.39 | 0.64 | 0.48 | 11 |
| X | 0.10 | 0.18 | 0.13 | 11 |
| Y | 0.75 | 0.27 | 0.40 | 11 |
| Z | 0.50 | 0.09 | 0.15 | 11 |
| a | 0.22 | 0.18 | 0.20 | 11 |
| b | 0.00 | 0.00 | 0.00 | 11 |
| c | 0.00 | 0.00 | 0.00 | 11 |
| d | 0.50 | 0.09 | 0.15 | 11 |
| e | 0.12 | 0.09 | 0.11 | 11 |
| f | 0.08 | 0.18 | 0.11 | 11 |
| g | 0.00 | 0.00 | 0.00 | 11 |
| h | 0.00 | 0.00 | 0.00 | 11 |
| i | 0.00 | 0.00 | 0.00 | 11 |
| j | 1.00 | 0.18 | 0.31 | 11 |
| k | 0.00 | 0.00 | 0.00 | 11 |
| l | 0.00 | 0.00 | 0.00 | 11 |
| m | 1.00 | 0.18 | 0.31 | 11 |
| n | 0.00 | 0.00 | 0.00 | 11 |
| o | 0.11 | 0.09 | 0.10 | 11 |
| p | 0.75 | 0.27 | 0.40 | 11 |
| q | 0.00 | 0.00 | 0.00 | 11 |
| r | 0.00 | 0.00 | 0.00 | 11 |
| s | 0.09 | 0.45 | 0.15 | 11 |
| t | 0.07 | 0.36 | 0.11 | 11 |
| u | 0.50 | 0.27 | 0.35 | 11 |
| v | 0.13 | 0.36 | 0.20 | 11 |
| w | 0.20 | 0.09 | 0.12 | 11 |
| x | 0.00 | 0.00 | 0.00 | 11 |
| y | 0.21 | 0.36 | 0.27 | 11 |
| z | 0.00 | 0.00 | 0.00 | 11 |

```
    accuracy                           0.15       682
   macro avg       0.23      0.15      0.13       682
weighted avg       0.23      0.15      0.13       682

/home/pranesh/Downloads/ML Lab/.venv/lib/python3.12/site-
packages/sklearn/metrics/_classification.py:1731: 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", result.shape[0])
/home/pranesh/Downloads/ML Lab/.venv/lib/python3.12/site-
packages/sklearn/metrics/_classification.py:1731: 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", result.shape[0])
/home/pranesh/Downloads/ML Lab/.venv/lib/python3.12/site-
packages/sklearn/metrics/_classification.py:1731: 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", result.shape[0])
```




[17]:
```
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize

# Binarize labels for ROC
y_test_bin = label_binarize(y_test, classes=np.arange(len(le.classes_)))
pla_preds_bin = label_binarize(pla_preds, classes=np.arange(len(le.classes_)))

# Micro-average ROC
fpr_micro, tpr_micro, _ = roc_curve(y_test_bin.ravel(), pla_preds_bin.ravel())
roc_auc_micro = auc(fpr_micro, tpr_micro)

# Macro-average ROC
fpr_dict, tpr_dict, roc_auc_dict = {}, {}, {}
for i in range(len(le.classes_)):
    fpr_dict[i], tpr_dict[i], _ = roc_curve(y_test_bin[:, i], pla_preds_bin[:,
 ↪i])
```

```
    roc_auc_dict[i] = auc(fpr_dict[i], tpr_dict[i])

all_fpr = np.unique(np.concatenate([fpr_dict[i] for i in range(len(le.
 ↪classes_))]))
mean_tpr = np.zeros_like(all_fpr)
for i in range(len(le.classes_)):
    mean_tpr += np.interp(all_fpr, fpr_dict[i], tpr_dict[i])
mean_tpr /= len(le.classes_)
roc_auc_macro = auc(all_fpr, mean_tpr)

# Plot ROC
plt.figure()
plt.plot(fpr_micro, tpr_micro, label=f"Micro-average ROC (AUC = {roc_auc_micro:.
 ↪2f})", color="blue")
plt.plot(all_fpr, mean_tpr, label=f"Macro-average ROC (AUC = {roc_auc_macro:.
 ↪2f})", color="red")
plt.plot([0, 1], [0, 1], "k--")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("PLA ROC Curves (Micro & Macro)")
plt.legend(loc="lower right")
plt.show()
```



[18]:
```
plt.figure()
plt.plot(range(1, len(best_model.errors_)+1), best_model.errors_, marker="o")
plt.xlabel("Epochs")
plt.ylabel("Training Error Rate")
plt.title("PLA Training Error vs Epochs")
plt.show()
```

# 6 MLP Implementation:

# mlp

September 9, 2025

[12]:
```python
import os
import numpy as np
import pandas as pd
from PIL import Image
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, label_binarize
from sklearn.metrics import (
    classification_report, confusion_matrix,
    accuracy_score, roc_curve, auc
)

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

import matplotlib.pyplot as plt
import seaborn as sns
```

[13]:
```python
csv_path = "archive/english.csv"
df = pd.read_csv(csv_path)

ROOT = "archive"
IMG_SIZE = (28, 28)  # width, height

images, labels, missing = [], [], []
for idx, row in df.iterrows():
    img_rel = row["image"]
    label_raw = row["label"]
    img_path = os.path.join(ROOT, img_rel)

    if not os.path.exists(img_path):
        missing.append(img_path)
        continue

    im = Image.open(img_path).convert("L")
    im = im.resize(IMG_SIZE)
    arr = np.asarray(im, dtype=np.float32).flatten() / 255.0  # normalize [0,1]
```

```
    images.append(arr)
    labels.append(label_raw)

print("Missing files:", len(missing))

X = np.vstack(images)  # (N, D)
y_raw = np.array(labels)
print("Loaded X:", X.shape, " y:", y_raw.shape)
```

```
Missing files: 0
Loaded X: (3410, 784)  y: (3410,)
```

[14]:
```
# Encode to integers 0..C-1
le = LabelEncoder()
y = le.fit_transform(y_raw)

# Stratified split -> train, validation, test
RNG_SEED = 42
X_train_full, X_test, y_train_full, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=RNG_SEED
)

X_train, X_val, y_train, y_val = train_test_split(
    X_train_full, y_train_full, test_size=0.2,
    stratify=y_train_full, random_state=RNG_SEED
)

print("Shapes -> train:", X_train.shape, "val:", X_val.shape, "test:", X_test.
 ↪shape)
print("Num classes:", len(le.classes_))
```

```
Shapes -> train: (2182, 784) val: (546, 784) test: (682, 784)
Num classes: 62
```

[15]:
```
X_train_t = torch.tensor(X_train, dtype=torch.float32)
y_train_t = torch.tensor(y_train, dtype=torch.long)
X_val_t   = torch.tensor(X_val, dtype=torch.float32)
y_val_t   = torch.tensor(y_val, dtype=torch.long)
X_test_t  = torch.tensor(X_test, dtype=torch.float32)
y_test_t  = torch.tensor(y_test, dtype=torch.long)

def get_loader(X, y, batch_size):
    ds = TensorDataset(X, y)
    return DataLoader(ds, batch_size=batch_size, shuffle=True)
```

[16]:
```
class MLP(nn.Module):
```

```
    def __init__(self, input_dim, hidden_dim, output_dim, activation="relu",␣
↪num_hidden=1):
        super().__init__()
        act_fn = {"relu": nn.ReLU(), "tanh": nn.Tanh(), "sigmoid": nn.
↪Sigmoid()}[activation]

        layers = []
        layers.append(nn.Linear(input_dim, hidden_dim))
        layers.append(act_fn)

        if num_hidden == 2:
            layers.append(nn.Linear(hidden_dim, hidden_dim))
            layers.append(act_fn)

        layers.append(nn.Linear(hidden_dim, output_dim))
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)
```

[17]:
```
def train_model(params):
    batch_size, lr, hidden_dim, activation, optimizer_name, num_hidden = params

    train_loader = get_loader(X_train_t, y_train_t, batch_size)
    val_loader   = get_loader(X_val_t, y_val_t, batch_size)

    model = MLP(X_train.shape[1], hidden_dim, len(le.classes_), activation,␣
↪num_hidden)
    criterion = nn.CrossEntropyLoss()

    if optimizer_name == "sgd":
        optimizer = optim.SGD(model.parameters(), lr=lr)
    else:
        optimizer = optim.Adam(model.parameters(), lr=lr)

    history = {"train_loss": [], "val_loss": [], "val_acc": []}
    EPOCHS = 20

    for epoch in range(EPOCHS):
        model.train()
        train_loss = 0
        for xb, yb in train_loader:
            optimizer.zero_grad()
            out = model(xb)
            loss = criterion(out, yb)
            loss.backward()
            optimizer.step()
```

```python
            train_loss += loss.item()

        # validation
        model.eval()
        val_loss, correct = 0, 0
        with torch.no_grad():
            for xb, yb in val_loader:
                out = model(xb)
                loss = criterion(out, yb)
                val_loss += loss.item()
                preds = out.argmax(dim=1)
                correct += (preds == yb).sum().item()
        acc = correct / len(val_loader.dataset)

        history["train_loss"].append(train_loss/len(train_loader))
        history["val_loss"].append(val_loss/len(val_loader))
        history["val_acc"].append(acc)

    return model, history
```

[11]:
```python
search_space = [
    (bs, lr, hd, act, opt, nh)
    for bs in [32, 64, 128]            # batch sizes
    for lr in [0.1, 0.01, 0.001]       # learning rates
    for hd in [128, 256]               # hidden layer size
    for act in ["relu", "tanh", "sigmoid"]  # activations
    for opt in ["sgd", "adam"]         # optimizers
    for nh in [1, 2]                   # number of hidden layers
]

best_acc, best_params, best_model, best_history = 0, None, None, None

for params in search_space:
    model, hist = train_model(params)
    final_acc = hist["val_acc"][-1]

    if final_acc > best_acc:
        best_acc = final_acc
        best_params = params
        best_model = model
        best_history = hist

print("\nBest Params:", best_params)
print("Best Val Accuracy:", best_acc)
```

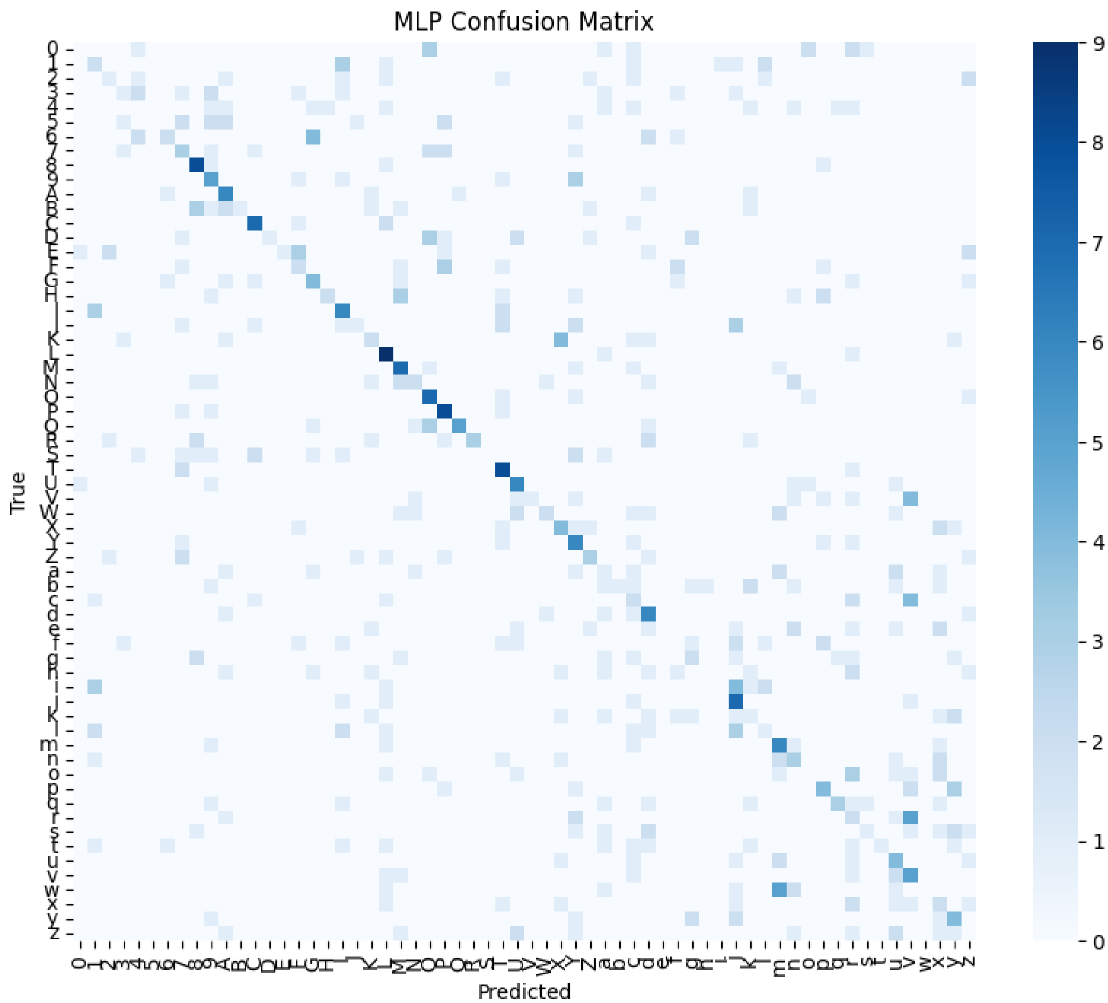
```
Best Params: (32, 0.001, 256, 'sigmoid', 'adam', 1)
Best Val Accuracy: 0.30036630036630035
```

[18]:
```python
best_model.eval()
with torch.no_grad():
    preds = best_model(X_test_t).argmax(dim=1).numpy()

print("\nMLP Test Accuracy:", accuracy_score(y_test, preds))
print("\nClassification Report:\n", classification_report(y_test, preds,␣
↪target_names=le.classes_))

# Confusion Matrix
cm = confusion_matrix(y_test, preds)
plt.figure(figsize=(10,8))
sns.heatmap(cm, cmap="Blues", xticklabels=le.classes_, yticklabels=le.classes_,␣
↪annot=False)
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("MLP Confusion Matrix")
plt.show()
```

```
MLP Test Accuracy: 0.2668621700879765

Classification Report:
               precision    recall  f1-score   support

           0       0.00      0.00      0.00        11
           1       0.15      0.18      0.17        11
           2       0.20      0.09      0.12        11
           3       0.20      0.09      0.12        11
           4       0.00      0.00      0.00        11
           5       0.00      0.00      0.00        11
           6       0.40      0.18      0.25        11
           7       0.19      0.27      0.22        11
           8       0.44      0.73      0.55        11
           9       0.23      0.45      0.30        11
           A       0.32      0.55      0.40        11
           B       1.00      0.09      0.17        11
           C       0.54      0.64      0.58        11
           D       1.00      0.09      0.17        11
           E       1.00      0.09      0.17        11
           F       0.20      0.18      0.19        11
           G       0.31      0.36      0.33        11
           H       0.67      0.18      0.29        11
           I       0.30      0.55      0.39        11
           J       0.33      0.09      0.14        11
           K       0.22      0.18      0.20        11
           L       0.35      0.82      0.49        11
           M       0.37      0.64      0.47        11
           N       0.33      0.18      0.24        11
```

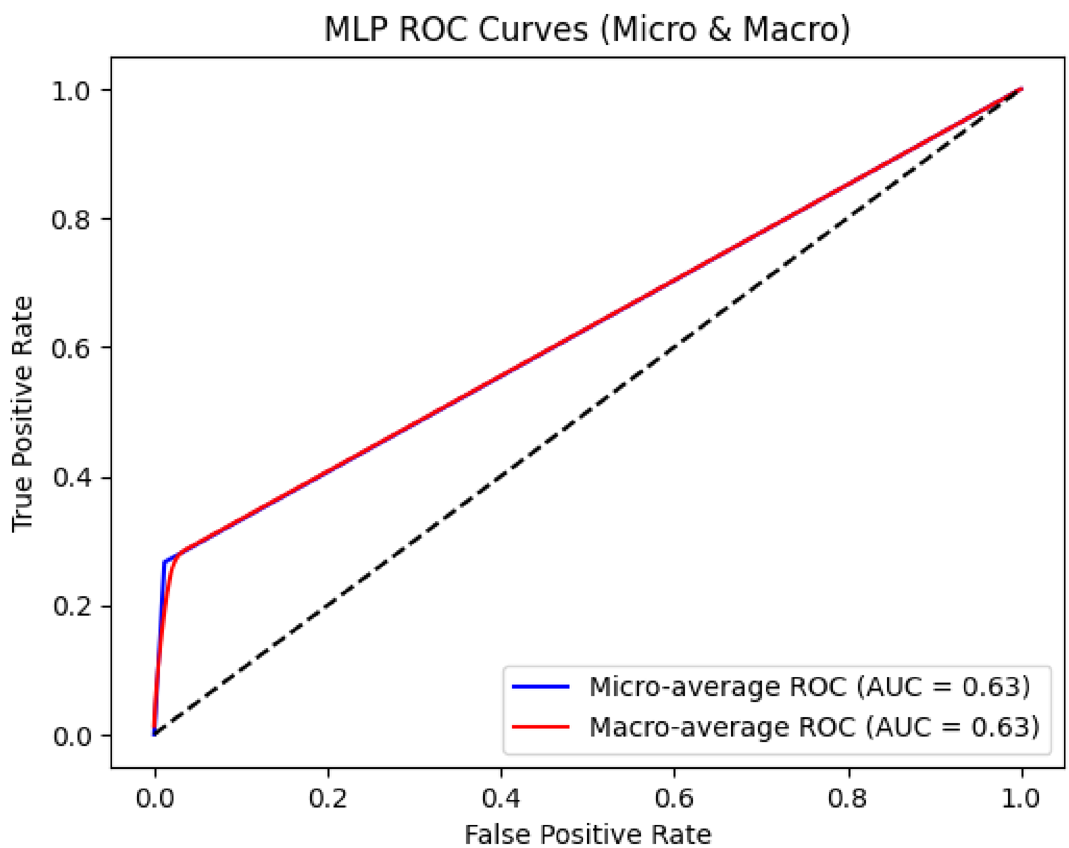
```
           O       0.35      0.64      0.45        11
           P       0.40      0.73      0.52        11
           Q       0.83      0.45      0.59        11
           R       1.00      0.27      0.43        11
           S       0.00      0.00      0.00        11
           T       0.35      0.73      0.47        11
           U       0.38      0.55      0.44        11
           V       1.00      0.09      0.17        11
           W       0.50      0.18      0.27        11
           X       0.31      0.36      0.33        11
           Y       0.22      0.55      0.32        11
           Z       0.38      0.27      0.32        11
           a       0.07      0.09      0.08        11
           b       1.00      0.09      0.17        11
           c       0.10      0.18      0.12        11
           d       0.24      0.55      0.33        11
           e       0.00      0.00      0.00        11
           f       0.00      0.00      0.00        11
           g       0.22      0.18      0.20        11
           h       0.00      0.00      0.00        11
           i       0.00      0.00      0.00        11
           j       0.24      0.64      0.35        11
           k       0.09      0.09      0.09        11
           l       0.14      0.09      0.11        11
           m       0.29      0.55      0.38        11
           n       0.19      0.27      0.22        11
           o       0.00      0.00      0.00        11
           p       0.33      0.36      0.35        11
           q       0.60      0.27      0.38        11
           r       0.08      0.18      0.11        11
           s       0.33      0.09      0.14        11
           t       1.00      0.09      0.17        11
           u       0.21      0.36      0.27        11
           v       0.21      0.45      0.29        11
           w       0.00      0.00      0.00        11
           x       0.11      0.18      0.14        11
           y       0.25      0.36      0.30        11
           z       0.00      0.00      0.00        11

    accuracy                           0.27       682
   macro avg       0.33      0.27      0.23       682
weighted avg       0.33      0.27      0.23       682

/home/pranesh/Downloads/ML Lab/.venv/lib/python3.12/site-
packages/sklearn/metrics/_classification.py:1731: 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", result.shape[0])
/home/pranesh/Downloads/ML Lab/.venv/lib/python3.12/site-
packages/sklearn/metrics/_classification.py:1731: 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", result.shape[0])
/home/pranesh/Downloads/ML Lab/.venv/lib/python3.12/site-
packages/sklearn/metrics/_classification.py:1731: 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", result.shape[0])
```

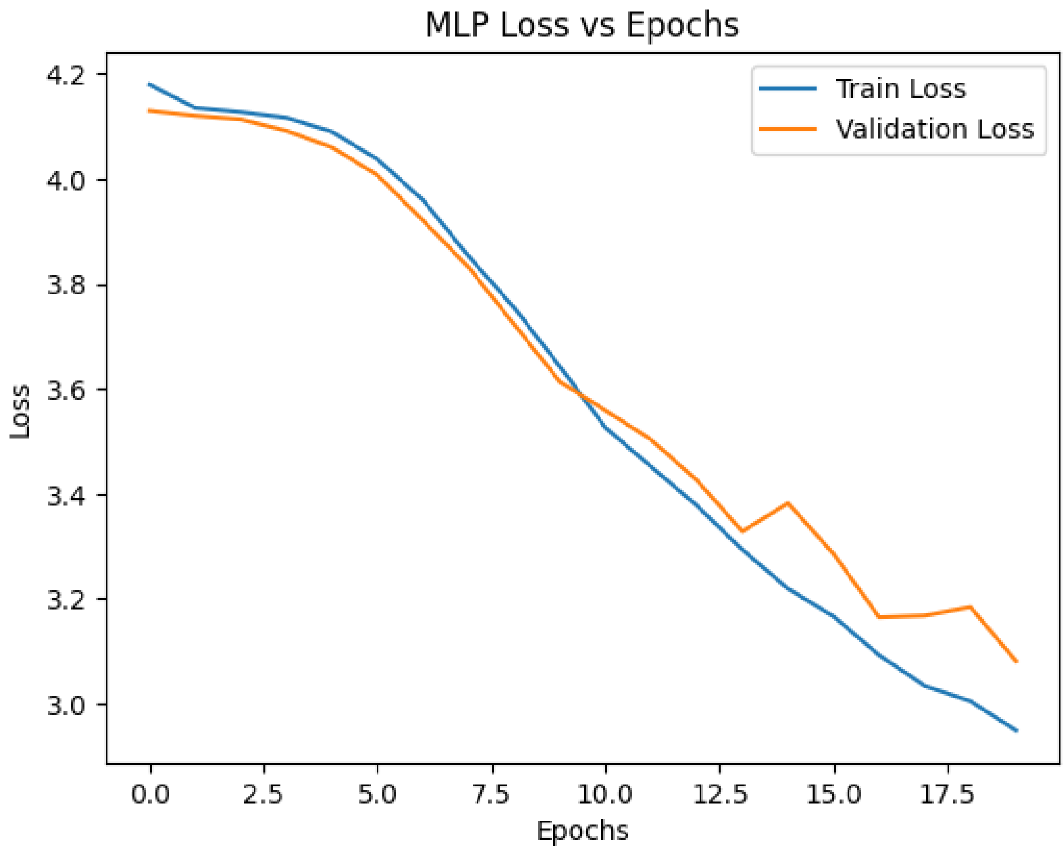


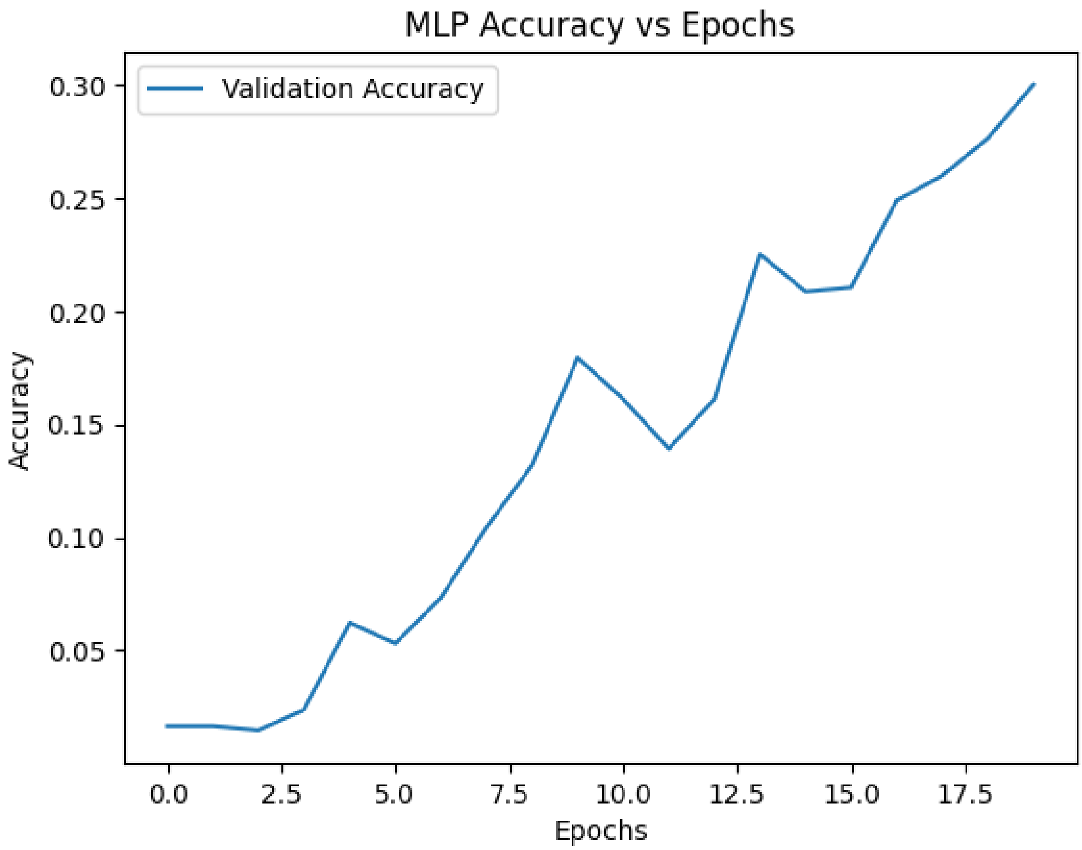

[19]:
```python
y_test_bin = label_binarize(y_test, classes=np.arange(len(le.classes_)))
preds_bin = label_binarize(preds, classes=np.arange(len(le.classes_)))

fpr_micro, tpr_micro, _ = roc_curve(y_test_bin.ravel(), preds_bin.ravel())
```

```
roc_auc_micro = auc(fpr_micro, tpr_micro)

fpr_dict, tpr_dict, roc_auc_dict = {}, {}, {}
for i in range(len(le.classes_)):
    fpr_dict[i], tpr_dict[i], _ = roc_curve(y_test_bin[:, i], preds_bin[:, i])
    roc_auc_dict[i] = auc(fpr_dict[i], tpr_dict[i])

all_fpr = np.unique(np.concatenate([fpr_dict[i] for i in range(len(le.
 ↪classes_))]))
mean_tpr = np.zeros_like(all_fpr)
for i in range(len(le.classes_)):
    mean_tpr += np.interp(all_fpr, fpr_dict[i], tpr_dict[i])
mean_tpr /= len(le.classes_)
roc_auc_macro = auc(all_fpr, mean_tpr)

plt.figure()
plt.plot(fpr_micro, tpr_micro, label=f"Micro-average ROC (AUC = {roc_auc_micro:.
 ↪2f})", color="blue")
plt.plot(all_fpr, mean_tpr, label=f"Macro-average ROC (AUC = {roc_auc_macro:.
 ↪2f})", color="red")
plt.plot([0, 1], [0, 1], "k--")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("MLP ROC Curves (Micro & Macro)")
plt.legend(loc="lower right")
plt.show()
```




[20]:
```
plt.figure()
plt.plot(best_history["train_loss"], label="Train Loss")
plt.plot(best_history["val_loss"], label="Validation Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("MLP Loss vs Epochs")
plt.legend()
plt.show()

plt.figure()
plt.plot(best_history["val_acc"], label="Validation Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.title("MLP Accuracy vs Epochs")
plt.legend()
plt.show()
```


MLP Loss vs Epochs
Train Loss
Validation Loss
Loss
Epochs
4.2
4.0
3.8
3.6
3.4
3.2
3.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


MLP Accuracy vs Epochs
Validation Accuracy
Accuracy
0.30
0.25
0.20
0.15
0.10
0.05
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs

# 7 Justification for Chosen Hyperparameters

The hyperparameters were tuned systematically over different values of batch size, learning rate, hidden layer size, activation function, optimizer, and number of hidden layers. The best performing configuration was found to be:

- Batch size: 32
- Learning rate: 0.001
- Hidden layer size: 256 neurons
- Activation function: Sigmoid
- Optimizer: Adam
- Number of hidden layers: 1

This configuration achieved a validation accuracy of approximately 30.03%. The smaller batch size of 32 allowed more frequent weight updates, leading to better generalization. A low learning rate of 0.001 provided stable convergence and avoided overshooting during optimization. The Sigmoid activation function was found to be effective in this case, providing nonlinear decision boundaries suitable for the dataset. Adam optimizer outperformed SGD due to its adaptive learning rate adjustment, leading to faster and more stable convergence. A single hidden layer with 256 neurons struck a balance between model complexity and overfitting, resulting in the best validation performance.

# 8 A/B Comparison (PLA vs MLP)

## 8.1 Strengths and Weaknesses of PLA vs MLP

The Perceptron Learning Algorithm (PLA) is simple, interpretable, and computationally efficient. However, it is limited to linearly separable data, which makes it unsuitable for complex multi-class problems such as handwritten character recognition. On the other hand, the Multilayer Perceptron (MLP) is capable of learning nonlinear decision boundaries through hidden layers and nonlinear activations. This allowed the MLP to achieve significantly higher accuracy compared to PLA, albeit at the cost of higher computational requirements and a greater risk of overfitting.

## 8.2 Impact of Hyperparameter Tuning on Convergence and Accuracy

Hyperparameter tuning had a significant impact on the performance of the MLP. Batch size and learning rate directly affected the stability and speed of convergence, with smaller batches and a lower learning rate yielding smoother training dynamics. The choice of optimizer also influenced performance: Adam provided faster convergence and better generalization compared to SGD. Similarly, the activation function determined the quality of nonlinear transformations, with Sigmoid providing the best accuracy in this experiment. Overall, systematic hyperparameter tuning was crucial in improving both the convergence rate and the final accuracy of the MLP, whereas PLA showed limited improvement regardless of parameter adjustments.

Table 1: Comparison of PLA and MLP

| **Aspect** | **PLA (Perceptron Learning Algorithm)** | **MLP (Multilayer Perceptron)** |
|---|---|---|
| Capability | Can only handle linearly separable data | Learns nonlinear decision boundaries with hidden layers |
| Complexity | Simple, fast, easy to implement | Computationally intensive, requires backpropagation |
| Accuracy | Low on multi-class handwritten recognition tasks | Significantly higher accuracy after tuning |
| Flexibility | Limited to binary or simple multi-class extensions | Flexible architecture with tunable layers, activations, and optimizers |
| Overfitting | Less prone due to simplicity | Can overfit without regularization |
| Scalability | Not suitable for large/complex datasets | Scales well with larger datasets and deeper networks |

# 9 Observations:

## 9.1 Why does PLA underperform compared to MLP?

PLA is a linear model and cannot capture complex non-linear relationships in image data. It relies only on a single separating hyperplane, which is insufficient for multi-class classification. MLP, with hidden layers and non-linear activations, learns richer feature representations.

## 9.2 Which hyperparameters had the most impact on MLP performance?

Learning rate strongly influenced convergence speed and stability. Activation functions like ReLU and sigmoid shaped how non-linear features were extracted. Optimizer choice (Adam vs SGD) and hidden layer size also had significant effects on accuracy.

## 9.3 Did optimizer choice (SGD vs Adam) affect convergence?

Yes, Adam generally converged faster and more stably than SGD in our experiments. SGD often required smaller learning rates and more epochs to achieve comparable accuracy. Adam's adaptive learning rate adjustment helped avoid poor local minima.

## 9.4 Did adding more hidden layers always improve results? Why or why not?

Adding extra hidden layers did not always improve accuracy. Shallow networks with sufficient neurons often learned adequately for this dataset. Too many layers sometimes led to overfitting or slower training without better generalization.

### 9.5 Did MLP show overfitting? How could it be mitigated?

Although training accuracy was not recorded, validation and test accuracies can indicate overfitting. If validation accuracy is noticeably lower than test accuracy, it suggests the model may not generalize well. To mitigate potential overfitting, techniques such as regularization, dropout, and early stopping could be applied.

### 9.6 Did MLP show overfitting? How could it be mitigated?

### 9.7 Did MLP show overfitting? How could it be mitigated?

Validation accuracy was 0.30 while test accuracy was 0.266, indicating minor overfitting as the model performs slightly worse on unseen data. To mitigate this, techniques such as regularization, dropout, or early stopping could be applied to improve generalization. Additionally, increasing the size of training data or performing data augmentation could help reduce overfitting.

## 10 Conclusion

- The Multilayer Perceptron (MLP) outperformed the Single-Layer Perceptron (PLA) in terms of validation and test accuracy, demonstrating the benefit of non-linear transformations and multiple layers.
- Hyperparameter tuning, especially choice of activation function, learning rate, and optimizer, significantly impacted MLP performance and convergence speed.
- Overfitting was minimal based on validation and test accuracies, but could be further reduced using techniques like regularization, dropout, or early stopping.