Sri Sivasubramaniya Nadar College of Engineering, Chennai

(An autonomous Institution affiliated to Anna University)

Degree & Branch	M.Tech (Integrated) Computer Science & Engineering
Semester	V
Subject Code & Name	ICS1512 – Machine Learning Algorithms Laboratory
Academic Year	2025–2026 (Odd)
Batch	2023–2028

Name: I.S.Rajesh **Register No.: 3122237001042**

Experiment 5: Perceptron vs Multilayer Perceptron with Hyperparameter Tuning

Aim

To build and evaluate two neural network models, being perceptron and multilayer perceptron, and compare them along with hyperparameter tuning.

Libraries Used

`numpy, pandas, sklearn, matplotlib, seaborn, PIL, torch, tqdm, itertools`

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.

Also select and justify hyperparameters such as activation functions, cost functions, optimizers, learning rates, number of hidden layers, and batch sizes through systematic tuning.

Preprocessing Steps

```
# Config
DATA_DIR = "Img.zip"
IMG_SIZE = (28,28)
RANDOM_SEED = 42
NUM_CLASSES = 62
```

```
np.random.seed(RANDOM_SEED)
torch.manual_seed(RANDOM_SEED)
random.seed(RANDOM_SEED)

with zipfile.ZipFile("/content/Img.zip", 'r') as zf:
    zf.extractall("/content/images")

df = pd.read_csv("/content/english.csv")

def load_images_from_csv(csv_df, base_dir="/content/images",
   img_size=(28,28)):
    X, y = [], []
    for _, row in csv_df.iterrows():
        rel_path = row["image"]
        full_path = os.path.join(base_dir, rel_path)
        if os.path.exists(full_path):
            img = Image.open(full_path).convert("L").resize(
               img_size)
            arr = np.array(img, dtype=np.float32) / 255.0
            X.append(arr.flatten())
            y.append(row["label"])
        else:
            print("Missing file:", full_path)
    return np.array(X), np.array(y)

X, y_raw = load_images_from_csv(df, "/content/images", IMG_SIZE)
print("Dataset:", X.shape, "Unique labels:", len(set(y_raw)))

# Encode labels & split
le = LabelEncoder()
y = le.fit_transform(y_raw)

X_train, X_temp, y_train, y_temp = train_test_split(
    X, y, test_size=0.3, stratify=y, random_state=RANDOM_SEED
)
X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp, test_size=0.5, stratify=y_temp, random_state=
       RANDOM_SEED
)

# Normalize
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_val_s = scaler.transform(X_val)
X_test_s = scaler.transform(X_test)
```

PLA Implementation

```
# PLA (One-vs-Rest Perceptron from scratch)
```

```
class PerceptronOVR:
    def __init__(self, input_dim, num_classes, eta=0.01,
       max_epochs=10):
        self.W = np.zeros((num_classes, input_dim+1))
        self.eta = eta
        self.max_epochs = max_epochs
        self.num_classes = num_classes

    def _bias(self, X):
        return np.hstack([np.ones((X.shape[0],1)), X])

    def fit(self, X, y):
        Xb = self._bias(X)
        for epoch in range(self.max_epochs):
            errors = 0
            for i in range(Xb.shape[0]):
                xi, yi = Xb[i], y[i]
                pred = np.argmax(self.W @ xi)
                if pred != yi:
                    self.W[yi] += self.eta * xi
                    self.W[pred] -= self.eta * xi
                    errors += 1
            print(f"Epoch {epoch+1}, errors={errors}")

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

pla = PerceptronOVR(input_dim=X_train_s.shape[1], num_classes=
   NUM_CLASSES, eta=0.01, max_epochs=10)
pla.fit(X_train_s, y_train)
y_pred_pla = pla.predict(X_test_s)
print(classification_report(y_test, y_pred_pla))
```

1 PLA Output

```
Epoch 1, errors=2041
Epoch 2, errors=1717
Epoch 3, errors=1529
Epoch 4, errors=1399
Epoch 5, errors=1290
Epoch 6, errors=1231
Epoch 7, errors=1133
Epoch 8, errors=1110
Epoch 9, errors=1095
Epoch 10, errors=955
              precision    recall  f1-score   support

           0       0.00      0.00      0.00         8
           1       0.00      0.00      0.00         8
```

2	0.27	0.38	0.32	8
3	0.33	0.25	0.29	8
4	0.20	0.12	0.15	8
5	0.17	0.12	0.14	8
6	0.22	0.25	0.24	8
7	0.50	0.33	0.40	9
8	0.00	0.00	0.00	8
9	0.27	0.38	0.32	8
10	0.45	0.62	0.53	8
11	0.17	0.22	0.19	9
12	0.44	0.50	0.47	8
13	0.10	0.12	0.11	8
14	0.15	0.25	0.19	8
15	0.20	0.12	0.15	8
16	0.19	0.38	0.25	8
17	1.00	0.38	0.55	8
18	0.00	0.00	0.00	9
19	0.33	0.38	0.35	8
20	1.00	0.12	0.22	8
21	1.00	0.11	0.20	9
22	0.25	0.22	0.24	9
23	0.22	0.25	0.24	8
24	0.23	0.33	0.27	9
25	0.67	0.50	0.57	8
26	0.00	0.00	0.00	8
27	0.23	0.38	0.29	8
28	0.12	0.11	0.12	9
29	0.50	0.56	0.53	9
30	0.38	0.75	0.50	8
31	0.00	0.00	0.00	8
32	0.43	0.38	0.40	8
33	0.50	0.38	0.43	8
34	0.00	0.00	0.00	8
35	0.29	0.22	0.25	9
36	0.00	0.00	0.00	8
37	0.38	0.33	0.35	9
38	0.13	0.22	0.17	9
39	0.14	0.12	0.13	8
40	0.06	0.12	0.08	8
41	0.18	0.25	0.21	8
42	0.00	0.00	0.00	9
43	0.17	0.25	0.20	8
44	0.00	0.00	0.00	9
45	0.22	0.25	0.24	8
46	0.00	0.00	0.00	8
47	0.16	0.38	0.22	8
48	0.17	0.25	0.20	8
49	0.00	0.00	0.00	9
50	0.17	0.25	0.20	8
51	0.50	0.11	0.18	9
52	0.00	0.00	0.00	8

```
          53       0.00      0.00      0.00         8
          54       0.33      0.25      0.29         8
          55       0.17      0.25      0.20         8
          56       0.00      0.00      0.00         8
          57       0.27      0.50      0.35         8
          58       0.33      0.38      0.35         8
          59       0.00      0.00      0.00         8
          60       0.67      0.44      0.53         9
          61       0.07      0.12      0.09         8

    accuracy                           0.22       512
   macro avg       0.24      0.22      0.21       512
weighted avg       0.24      0.22      0.21       512
```

MLP Implementation

```python
# MLP Model (PyTorch)
class MLP(nn.Module):
    def __init__(self, input_dim, hidden_layers, num_classes,
       activation='relu', dropout=0.0):
        super().__init__()
        layers, prev = [], input_dim
        act_map = {'relu':nn.ReLU, 'tanh':nn.Tanh, 'sigmoid':nn.
           Sigmoid}
        for h in hidden_layers:
            layers += [nn.Linear(prev, h), act_map[activation]()]
            if dropout>0: layers.append(nn.Dropout(dropout))
            prev = h
        layers.append(nn.Linear(prev, num_classes))
        self.net = nn.Sequential(*layers)
    def forward(self, x): return self.net(x)

# Training Utilities
def make_loader(X, y, batch=64, shuffle=True):
    ds = TensorDataset(torch.tensor(X, dtype=torch.float32),
       torch.tensor(y, dtype=torch.long))
    return DataLoader(ds, batch_size=batch, shuffle=shuffle)

def train_epoch(model, loader, opt, device):
    model.train(); total=0
    for xb,yb in loader:
        xb,yb = xb.to(device), yb.to(device)
        out = model(xb)
        loss = F.cross_entropy(out, yb)
        opt.zero_grad(); loss.backward(); opt.step()
        total += loss.item()*xb.size(0)
    return total/len(loader.dataset)

def eval_model(model, loader, device):
    model.eval(); preds, labs = [], []
    total=0
    with torch.no_grad():
        for xb,yb in loader:
            xb,yb = xb.to(device), yb.to(device)
            out = model(xb)
            loss = F.cross_entropy(out, yb)
            total += loss.item()*xb.size(0)
            preds.append(out.argmax(1).cpu().numpy())
            labs.append(yb.cpu().numpy())
    return total/len(loader.dataset), np.concatenate(preds), np.
       concatenate(labs)

# Hyperparameter grid
hyperparams = {
    'hidden_layers': [[128], [256], [256,128], [512,256]],
    'activation': ['relu', 'tanh', 'sigmoid'],
    'dropout': [0.0, 0.2, 0.5],
    'lr': [1e-2, 1e-3, 1e-4],
    'batch_size': [64, 128, 256],
    'optimizer': ['adam', 'sgd', 'rmsprop']
}

# Train MLP
device = "cuda" if torch.cuda.is_available() else "cpu"

best_val_acc = 0
best_config = None
best_model_state = None

# Iterate over all hyperparameter combinations
for hl, act, do, lr, bs, opt_name in product(
    hyperparams['hidden_layers'],
    hyperparams['activation'],
    hyperparams['dropout'],
    hyperparams['lr'],
    hyperparams['batch_size'],
    hyperparams['optimizer']
):
    # Prepare loaders with current batch size
    train_loader = make_loader(X_train_s, y_train, batch=bs)
    val_loader = make_loader(X_val_s, y_val, batch=bs, shuffle=
       False)

    # Initialize model and optimizer
    model = MLP(X_train_s.shape[1], hl, NUM_CLASSES, activation=
       act, dropout=do).to(device)
    if opt_name == 'adam':
        opt = optim.Adam(model.parameters(), lr=lr)
    elif opt_name == 'sgd':
        opt = optim.SGD(model.parameters(), lr=lr, momentum=0.9)
            # momentum is standard for SGD
    elif opt_name == 'rmsprop':
        opt = optim.RMSprop(model.parameters(), lr=lr)

    # Train for a few epochs (e.g., 5 for tuning speed)
    for epoch in range(5):
        train_epoch(model, train_loader, opt, device)

    # Evaluate on validation set
    _, y_val_pred, y_val_true = eval_model(model, val_loader,
       device)
    val_acc = (y_val_pred == y_val_true).mean()

    # Save best
    if val_acc > best_val_acc:
        best_val_acc = val_acc
        best_config = {'hidden_layers': hl, 'activation': act, '
           dropout': do, 'lr': lr, 'batch_size': bs, 'optimizer':
            opt_name}
        best_model_state = model.state_dict()

print("Best hyperparameters:", best_config, "Validation Accuracy:
   ", best_val_acc)

train_loader = make_loader(X_train_s, y_train, batch=best_config[
   'batch_size'])
val_loader = make_loader(X_val_s, y_val, batch=best_config['
   batch_size'], shuffle=False)
test_loader = make_loader(X_test_s, y_test, batch=best_config['
   batch_size'], shuffle=False)

model = MLP(X_train_s.shape[1], best_config['hidden_layers'],
   NUM_CLASSES,
            activation=best_config['activation'], dropout=
               best_config['dropout']).to(device)
model.load_state_dict(best_model_state)  # Optionally start from
   tuned weights

if best_config['optimizer'] == 'adam':
    opt = optim.Adam(model.parameters(), lr=best_config['lr'])
elif best_config['optimizer'] == 'sgd':
    opt = optim.SGD(model.parameters(), lr=best_config['lr'],
       momentum=0.9)
elif best_config['optimizer'] == 'rmsprop':
    opt = optim.RMSprop(model.parameters(), lr=best_config['lr'])

history = {'train_loss':[], 'val_loss':[], 'val_acc':[]}
for epoch in range(20):
    tl = train_epoch(model, train_loader, opt, device)
    vl, vp, vlabs = eval_model(model, val_loader, device)
    acc = (vp==vlabs).mean()
    history['train_loss'].append(tl); history['val_loss'].append(
       vl); history['val_acc'].append(acc)
    print(f"Epoch {epoch+1}: train={tl:.3f}, val={vl:.3f}, acc={
       acc:.3f}")
```

2 MLP Output

```
Best hyperparameters: {'hidden_layers': [512, 256], 'activation':
    'relu', 'dropout': 0.0, 'lr': 0.001, 'batch_size': 128, '
   optimizer': 'rmsprop'} Validation Accuracy: 0.4755381604696673
Epoch 1: train=1.675, val=2.323, acc=0.477
Epoch 2: train=0.807, val=2.383, acc=0.513
Epoch 3: train=0.581, val=2.522, acc=0.513
Epoch 4: train=0.478, val=2.504, acc=0.534
Epoch 5: train=0.403, val=2.591, acc=0.534
Epoch 6: train=0.310, val=2.661, acc=0.536
Epoch 7: train=0.268, val=2.739, acc=0.517
Epoch 8: train=0.226, val=2.870, acc=0.528
Epoch 9: train=0.191, val=2.886, acc=0.536
Epoch 10: train=0.161, val=2.967, acc=0.552
Epoch 11: train=0.136, val=3.028, acc=0.534
Epoch 12: train=0.123, val=3.021, acc=0.566
Epoch 13: train=0.099, val=3.018, acc=0.560
Epoch 14: train=0.087, val=3.119, acc=0.558
Epoch 15: train=0.074, val=3.193, acc=0.540
Epoch 16: train=0.067, val=3.222, acc=0.558
Epoch 17: train=0.058, val=3.253, acc=0.542
Epoch 18: train=0.048, val=3.323, acc=0.542
Epoch 19: train=0.047, val=3.378, acc=0.562
Epoch 20: train=0.046, val=3.438, acc=0.560
```

3 MLP vs PLA Comparison

```
# Evaluation (MLP vs PLA)
# Test set
_, y_pred_mlp, _ = eval_model(model, test_loader, device)
print("MLP Classification Report:\n", classification_report(
   y_test, y_pred_mlp))
print("PLA Classification Report:\n", classification_report(
   y_test, y_pred_pla))

# Confusion Matrix Example (MLP)
cm = confusion_matrix(y_test, y_pred_mlp)
plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, fmt='d', cmap="Blues")
plt.title("MLP Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()

# Confusion Matrix Example (PLA)
cm = confusion_matrix(y_test, y_pred_pla)
plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, fmt='d', cmap="Blues")
plt.title("PLA Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()

# ROC (micro-average)
def predict_proba(model, loader):
    model.eval(); out=[]
    with torch.no_grad():
        for xb,_ in loader:
            logits = model(xb.to(device))
            out.append(F.softmax(logits,1).cpu().numpy())
    return np.vstack(out)

probs = predict_proba(model, test_loader)
y_onehot = np.eye(NUM_CLASSES)[y_test]
fpr,tpr,_ = roc_curve(y_onehot.ravel(), probs.ravel())
plt.plot(fpr,tpr); plt.plot([0,1],[0,1],'k--')
plt.title("Micro-average ROC"); plt.show()

# Plot Training Curves
plt.plot(history['train_loss'], label="train_loss")
plt.plot(history['val_loss'], label="val_loss")
plt.plot(history['val_acc'], label="val_acc")
plt.legend(); plt.title("Training Curves"); plt.show()
```

```
MLP Classification Report:
              precision    recall  f1-score   support

           0       0.25      0.12      0.17         8
           1       0.56      0.62      0.59         8
           2       0.71      0.62      0.67         8
           3       0.70      0.88      0.78         8
           4       0.30      0.38      0.33         8
           5       0.40      0.25      0.31         8
           6       0.50      0.62      0.56         8
           7       0.55      0.67      0.60         9
           8       0.67      1.00      0.80         8
           9       0.60      0.75      0.67         8
          10       0.88      0.88      0.88         8
          11       0.42      0.56      0.48         9
          12       0.86      0.75      0.80         8
```

13	0.80	0.50	0.62	8
14	0.44	0.50	0.47	8
15	0.67	0.50	0.57	8
16	0.86	0.75	0.80	8
17	0.67	0.50	0.57	8
18	0.45	0.56	0.50	9
19	0.50	0.75	0.60	8
20	0.38	0.38	0.38	8
21	0.60	0.33	0.43	9
22	0.75	0.67	0.71	9
23	0.57	0.50	0.53	8
24	0.47	0.78	0.58	9
25	0.71	0.62	0.67	8
26	1.00	0.25	0.40	8
27	0.62	0.62	0.62	8
28	0.38	0.33	0.35	9
29	0.75	0.67	0.71	9
30	0.57	1.00	0.73	8
31	0.50	0.38	0.43	8
32	0.71	0.62	0.67	8
33	0.80	0.50	0.62	8
34	0.86	0.75	0.80	8
35	0.78	0.78	0.78	9
36	0.36	0.62	0.45	8
37	0.67	0.44	0.53	9
38	0.70	0.78	0.74	9
39	0.57	0.50	0.53	8
40	0.38	0.38	0.38	8
41	0.50	0.50	0.50	8
42	0.40	0.22	0.29	9
43	0.27	0.38	0.32	8
44	0.38	0.56	0.45	9
45	0.62	0.62	0.62	8
46	0.25	0.25	0.25	8
47	0.33	0.12	0.18	8
48	0.56	0.62	0.59	8
49	0.25	0.22	0.24	9
50	0.25	0.25	0.25	8
51	0.50	0.78	0.61	9
52	0.43	0.38	0.40	8
53	0.67	0.25	0.36	8
54	0.67	0.50	0.57	8
55	0.62	0.62	0.62	8
56	0.33	0.25	0.29	8
57	0.38	0.62	0.48	8
58	0.67	1.00	0.80	8
59	0.50	0.12	0.20	8
60	0.56	0.56	0.56	9

```
          61       0.27      0.38      0.32         8

    accuracy                           0.54       512
   macro avg       0.55      0.54      0.53       512
weighted avg       0.55      0.54      0.53       512

PLA Classification Report:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00         8
           1       0.00      0.00      0.00         8
           2       0.27      0.38      0.32         8
           3       0.33      0.25      0.29         8
           4       0.20      0.12      0.15         8
           5       0.17      0.12      0.14         8
           6       0.22      0.25      0.24         8
           7       0.50      0.33      0.40         9
           8       0.00      0.00      0.00         8
           9       0.27      0.38      0.32         8
          10       0.45      0.62      0.53         8
          11       0.17      0.22      0.19         9
          12       0.44      0.50      0.47         8
          13       0.10      0.12      0.11         8
          14       0.15      0.25      0.19         8
          15       0.20      0.12      0.15         8
          16       0.19      0.38      0.25         8
          17       1.00      0.38      0.55         8
          18       0.00      0.00      0.00         9
          19       0.33      0.38      0.35         8
          20       1.00      0.12      0.22         8
          21       1.00      0.11      0.20         9
          22       0.25      0.22      0.24         9
          23       0.22      0.25      0.24         8
          24       0.23      0.33      0.27         9
          25       0.67      0.50      0.57         8
          26       0.00      0.00      0.00         8
          27       0.23      0.38      0.29         8
          28       0.12      0.11      0.12         9
          29       0.50      0.56      0.53         9
          30       0.38      0.75      0.50         8
          31       0.00      0.00      0.00         8
          32       0.43      0.38      0.40         8
          33       0.50      0.38      0.43         8
          34       0.00      0.00      0.00         8
          35       0.29      0.22      0.25         9
          36       0.00      0.00      0.00         8
          37       0.38      0.33      0.35         9
          38       0.13      0.22      0.17         9
          39       0.14      0.12      0.13         8
          40       0.06      0.12      0.08         8
          41       0.18      0.25      0.21         8
          42       0.00      0.00      0.00         9
          43       0.17      0.25      0.20         8
          44       0.00      0.00      0.00         9
          45       0.22      0.25      0.24         8
          46       0.00      0.00      0.00         8
          47       0.16      0.38      0.22         8
          48       0.17      0.25      0.20         8
          49       0.00      0.00      0.00         9
          50       0.17      0.25      0.20         8
          51       0.50      0.11      0.18         9
          52       0.00      0.00      0.00         8
          53       0.00      0.00      0.00         8
          54       0.33      0.25      0.29         8
          55       0.17      0.25      0.20         8
          56       0.00      0.00      0.00         8
          57       0.27      0.50      0.35         8
          58       0.33      0.38      0.35         8
          59       0.00      0.00      0.00         8
          60       0.67      0.44      0.53         9
          61       0.07      0.12      0.09         8

    accuracy                           0.22       512
   macro avg       0.24      0.22      0.21       512
weighted avg       0.24      0.22      0.21       512
```

Aspect	**PLA (Perceptron Learning Algorithm)**	**MLP (Multilayer Perceptron)**
Final Chosen Parameters	— Hidden layers: N/A — Learning rate: implicit updates — Epochs: 10 — Errors reduced from 2041 $\rightarrow$ 955	— Hidden layers: [512, 256] — Activation: ReLU — Dropout: 0.0 — Learning rate: 0.001 — Batch size: 128 — Optimizer: RMSprop
Validation Accuracy	0.22	0.54
Training Behavior	— Errors decrease steadily over epochs (2041 $\rightarrow$ 955) — Shows convergence but limited by linear separability	— Training loss decreases smoothly — Validation loss diverges (overfitting beyond $\sim$5 epochs)
Strengths	— Extremely simple and fast — Easy to implement — Good for linearly separable data	— Learns complex non-linear decision boundaries — Supports hyperparameter tuning to a better extent (layers, activation, optimizers, dropout) — Achieves much higher accuracy
Weaknesses	— Cannot model non-linear relationships — Accuracy too low for practical use — Very sensitive to class imbalance	— Computationally heavier — Risk of overfitting (seen in validation loss curve) — Needs careful tuning of hyperparameters

Confusion Matrices and ROC Curves

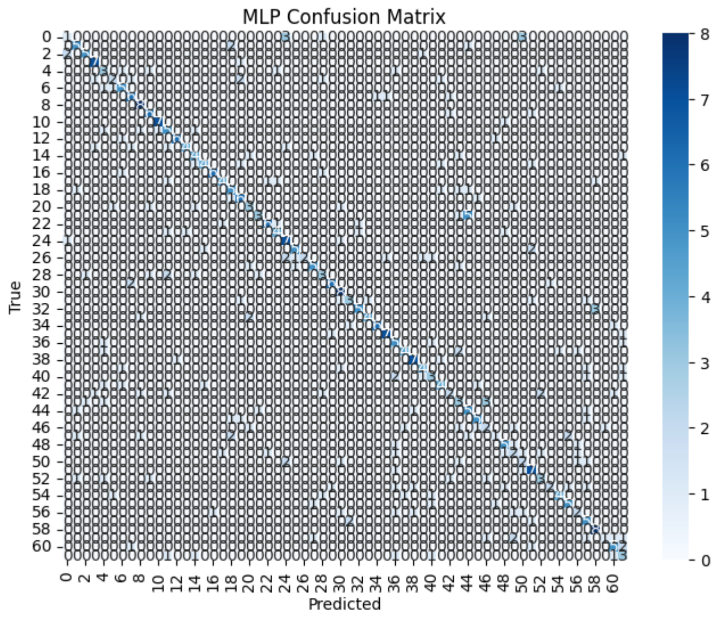

Figure 1: Confusion Matrix for MLP Model

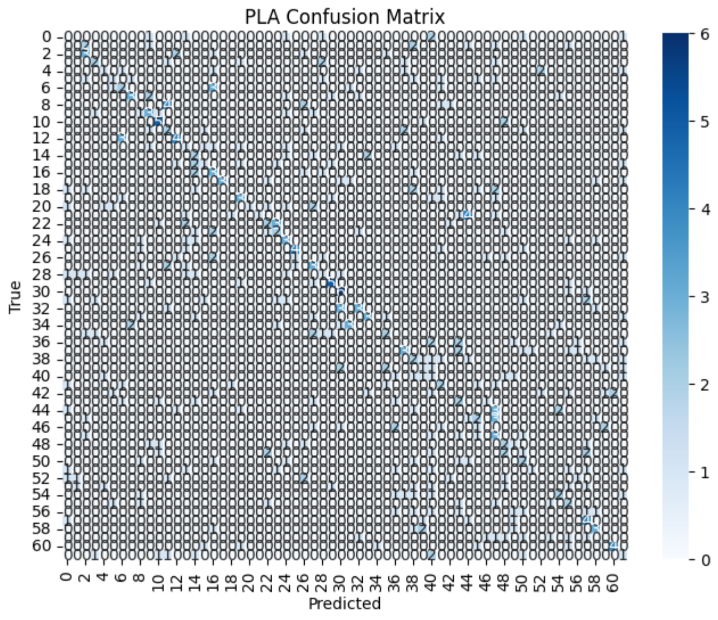

Figure 2: Confusion Matrix or PLA Model

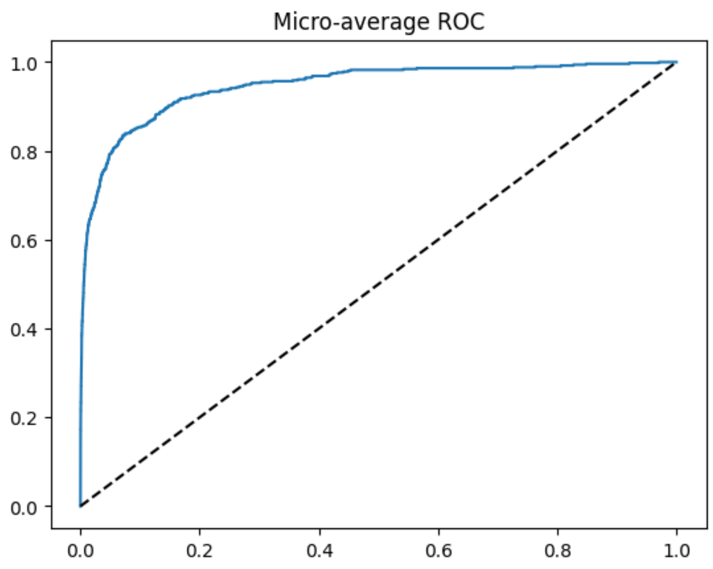

Figure 3: ROC Curve for MLP

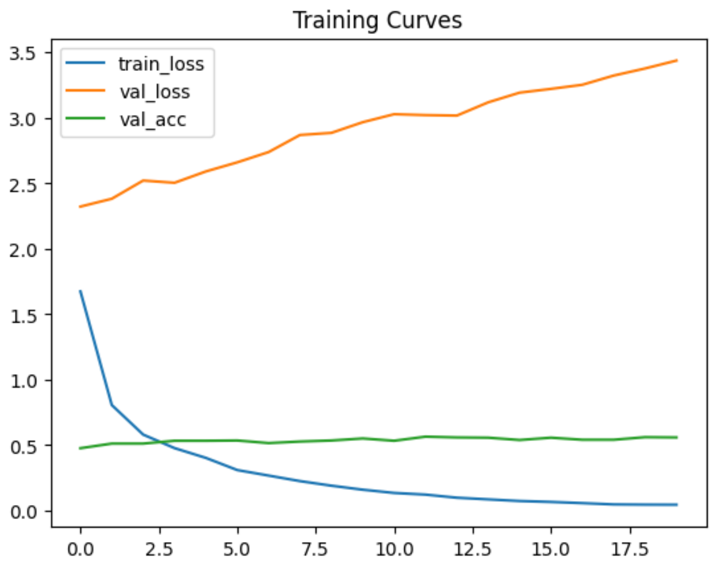

Figure 4: Training Curves

Observations and Analysis

- As we go through the epochs for the MLP model, the training loss keeps decreasing but validation loss increases steadily after a few epochs. Also the testing accuracy is around 50, indicating a likely overfit model.
- However, on the basis of the area under the ROC curve, it is visible that it is able to distinguish properly between the classes.
- Linear models like PLA are insufficient for practical real-world datasets like Handwritten Digit Classification, whereas deep models like MLP can work with some improvements.

4 Learning Outcomes and Best Practices

- Learned to implement neural network models like PLA and MLP on real-world datasets.
- Learned to view training and validation curves to reveal overfitting in the MLP, emphasizing the importance of balancing model complexity with regularization techniques.
- Evaluated result using respective classification metrics like accuracy, and compared the confusion matrices for better understanding.
- Preprocessed images in dataset with transformation methods like resizing, flattening and normalization.
- Systematically tuned learning rate, batch size, and hidden layer sizes to yield significant performance gains.