MLP Classifier Assignment

Training Classifier using MLP (Neural Network)
Neural Network Assignment | Software Engineering Student

Aim and Task Overview

Aim: Learn the basic steps of creating and training neural networks

Task: Train a classifier for the selected dataset using a Neural Network approach.

Steps include:
- Dataset selection (Ecoli)
- Build a Multilayer Perceptron (MLP)
- Data preparation & normalization
- Training & evaluation
- Hyperparameter tuning
- Saving weights and reporting metrics

Dataset Overview

Dataset: Ecoli Dataset
- Total samples: 336
- Features: 7 (after dropping ID column)
- Classes: 8 (e.g., im, cp, pp, etc.)
- Source: UCI Machine Learning Repository
- Labels encoded using LabelEncoder from scikit-learn

MLP Model Architecture

- Input layer: 7 neurons (one for each feature)

• Hidden layer: 64 neurons with ReLU activation
• Output layer: 8 neurons (for 8 classes)
• Loss Function: CrossEntropyLoss
• Optimizer: Adam
• Device: CPU (or CUDA if available)

Training Hyperparameters
• Epochs: 500
• Batch size: 22
• Learning rate: 0.001
• Split: 80% training, 20% testing
• Optimizer: Adam
• Loss: Cross Entropy

Training Loss Over Epochs

Model Performance Metrics
• Accuracy: 88.24%
• Precision (per class): [0.96875, 0.8333, 0.5, 1.0, 0.9286]
• Recall (per class): [0.96875, 0.7143, 0.6667, 1.0, 0.9286]
• F1-score: [0.96875, 0.7692, 0.5714, 1.0, 0.9286]
• Confusion Matrix shown on next slide

Confusion Matrix

Python Code Summary

- Used libraries: NumPy, Pandas, Scikit-learn, PyTorch
- Data normalized using StandardScaler
- Labels encoded using LabelEncoder
- Model: 2-layer MLP with ReLU and Softmax (via CrossEntropyLoss)
- Loss and accuracy plotted
- Model weights saved using torch.save

Conclusion

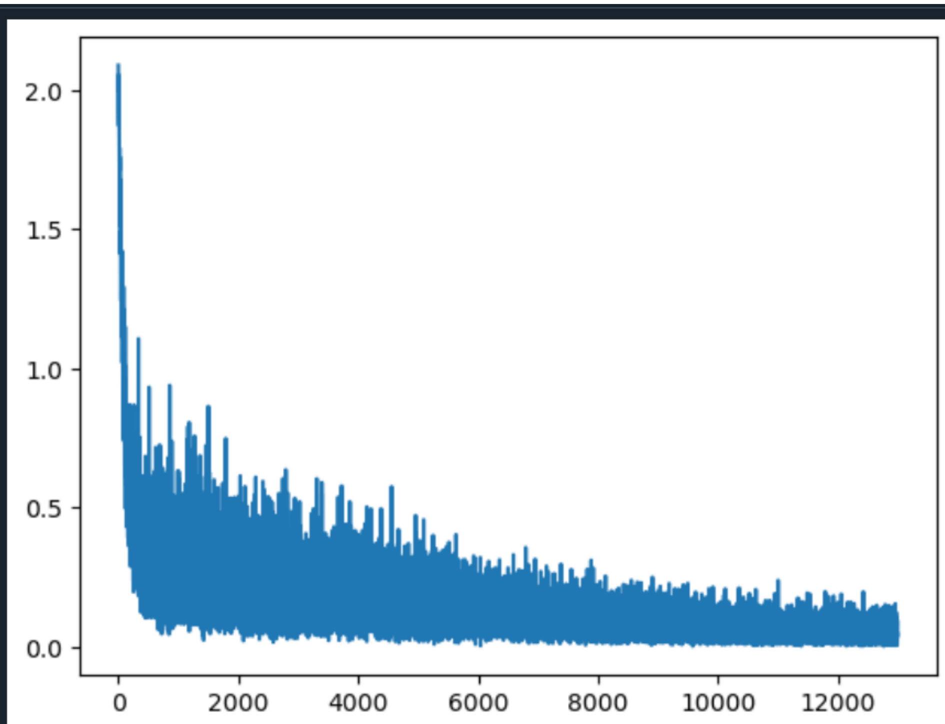
- Successfully implemented MLP for multi-class classification
- Achieved good accuracy with balanced precision/recall
- Training loss shows good convergence
- Model generalizes well with test data
- Next steps: experiment with deeper networks, regularization, and different learning rates

Console 1/A

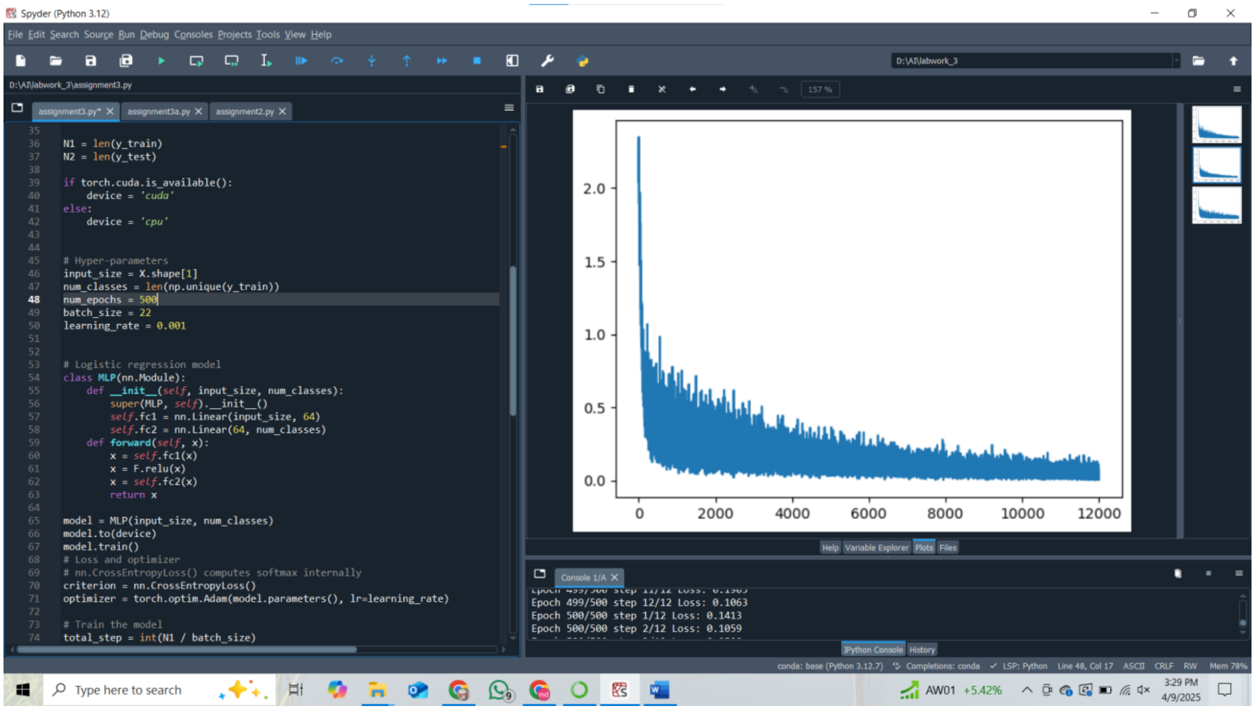
```
Epoch 500/500 step 3/12 Loss: 0.0500
Epoch 500/500 step 4/12 Loss: 0.1392
Epoch 500/500 step 5/12 Loss: 0.1605
Epoch 500/500 step 6/12 Loss: 0.2446
Epoch 500/500 step 7/12 Loss: 0.1275
Epoch 500/500 step 8/12 Loss: 0.1220
Epoch 500/500 step 9/12 Loss: 0.0780
Epoch 500/500 step 10/12 Loss: 0.0584
Epoch 500/500 step 11/12 Loss: 0.0921
Epoch 500/500 step 12/12 Loss: 0.0754
NN acc: 0.882353
NN recall: %f [0.96875    0.71428571 0.66666667 1.         0.92857143]
NN precision: [0.96875    0.83333333 0.5        1.         0.92857143]
NN f1: [0.96875    0.76923077 0.57142857 1.         0.92857143]
NN conf_mat: %f
[[31  0  0  0  1]
 [ 0 10  4  0  0]
 [ 0  2  4  0  0]
 [ 0  0  0  2  0]
 [ 1  0  0  0 13]]
```

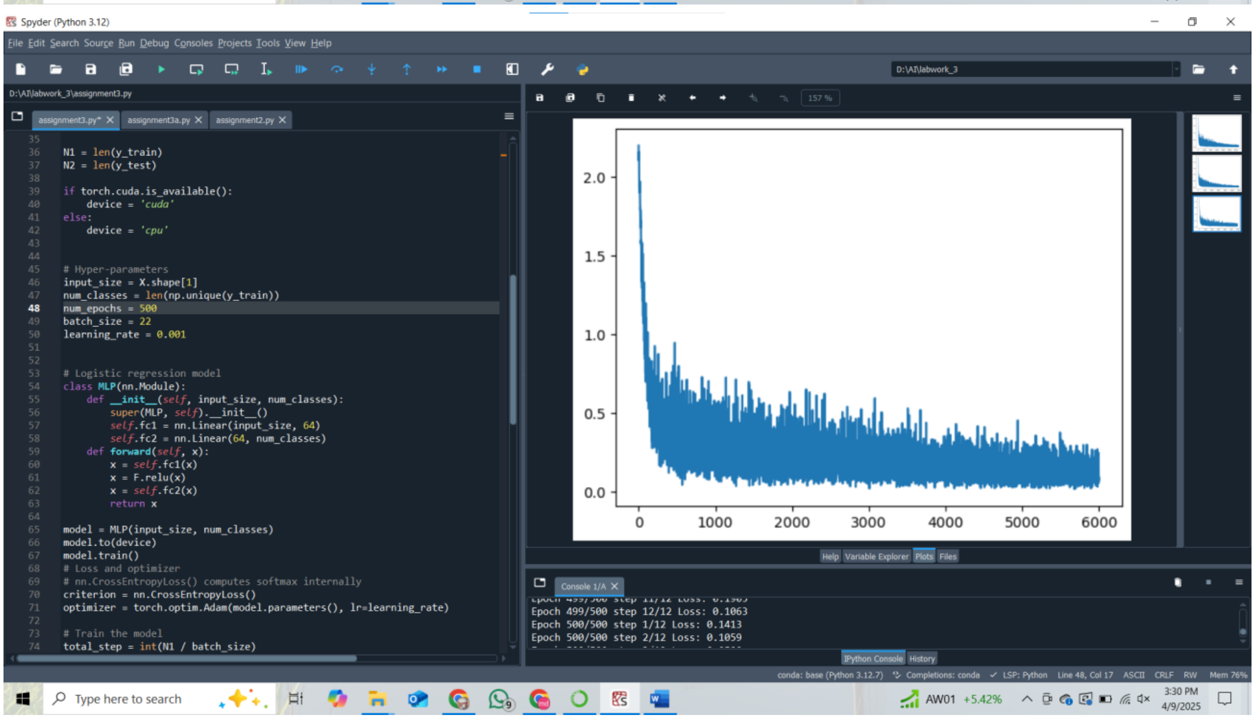
Name	Type	Size	Value
optimizer	optim.adam.Adam	1	Adam object of torch.optim.adam module
outputs	Tensor	(68, 8)	Tensor object of torch module
prec	Array of float64	(5,)	[0.96875 0.83333333 0.5 1. 0.92857143]
predicted	Tensor	(68,)	Tensor object of torch module
recall	Array of float64	(5,)	[0.96875 0.71428571 0.66666667 1. 0.92857143]
scaler	preprocessing._data.StandardScaler	1	StandardScaler object of sklearn.preprocessing._data module
total_step	int	1	12
X	Array of float64	(336, 7)	[[-0.0517614 -1.41953086 -0.17514236 ... 0.49078096 -1.20771743 -0 ...
x	Array of float32	(268, 7)	[[-1.2866843 -0.8787572 -0.17514236 ... -0.73678035 -0.78994834 -0 ...
X_test	Array of float64	(68, 7)	[[-0.82358817 -1.48712757 -0.17514236 ... -0.73678034 -0.69711074 -0 ...
X_train	Array of float64	(268, 7)	[[5.65700021e-01 -2.02790123e-01 -1.75142361e-01 ... -4.09211456e+00 ...
xb	Tensor	(68, 7)	Tensor object of torch module
y	Array of int64	(268,)	[0 0 0 ... 1 0 1]
y_pred	Array of int64	(68,)	[0 0 1 ... 4 0 0]
y_test	Array of int64	(68,)	[0 0 1 ... 1 0 0]
y_train	Array of int64	(268,)	[1 7 7 ... 0 4 0]
yb	Array of int64	(22,)	[0 0 0 ... 1 1 0]

Help Variable Explorer Plots Files



Help Variable Explorer Plots Files






```
import numpy as np


from sklearn.datasets import load_iris

from sklearn import metrics

from sklearn import model_selection
```

```
from sklearn.utils import shuffle
import sklearn.preprocessing
import torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.preprocessing import LabelEncoder


# Load dataset
df = pd.read_csv('D:/AI/labwork_3/39_Ecoli/ecoli.data', sep=r'\s+', header=None)

X = df.iloc[:, 1:-1]  # Skips the first column (ID) and selects features
y = df.iloc[:, -1]  # Last column as target

scaler = sklearn.preprocessing.StandardScaler()
X = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y,
                    test_size=0.2, random_state=1)

# Convert string labels to integers
label_encoder = LabelEncoder()
y_train = label_encoder.fit_transform(y_train)
y_test = label_encoder.transform(y_test)
```

```
y_train = np.array(y_train, dtype=np.int64)
y_test = np.array(y_test, dtype=np.int64)

N1 = len(y_train)
N2 = len(y_test)

if torch.cuda.is_available():
   device = 'cuda'
else:
   device = 'cpu'


# Hyper-parameters
input_size = X.shape[1]
num_classes = len(np.unique(y_train))
num_epochs = 500
batch_size = 22
learning_rate = 0.001


# Logistic regression model
class MLP(nn.Module):
   def __init__(self, input_size, num_classes):
      super(MLP, self).__init__()
      self.fc1 = nn.Linear(input_size, 64)
```

```
        self.fc2 = nn.Linear(64, num_classes)
    def forward(self, x):
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        return x

model = MLP(input_size, num_classes)
model.to(device)
model.train()
# Loss and optimizer
# nn.CrossEntropyLoss() computes softmax internally
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# Train the model
total_step = int(N1 / batch_size)
losses = []
for epoch in range(num_epochs):
    x, y = shuffle(X_train, y_train)
    x = np.array(x, dtype=np.float32)

    for i in range(total_step):
        xb = x[i*batch_size : (i+1)*batch_size]
        xb = torch.from_numpy(xb).to(device)
        yb = y[i*batch_size : (i+1)*batch_size]
```

```
        labels = torch.from_numpy(yb).to(device)
        # Forward pass
        outputs = model(xb)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        losses.append(loss.item())
        print ('Epoch %d/%d step %d/%d Loss: %.4f' %(epoch+1, num_epochs, i+1, total_step,
loss.item()) )

# Test the model
# In test phase, we don't need to compute gradients (for memory efficiency)

plt.figure()
plt.plot(losses)
plt.show()

model.eval()
with torch.no_grad():
  xb = np.array(X_test, dtype=np.float32)
  xb = torch.from_numpy(xb).to(device)
  outputs = model(xb)
  _, predicted = torch.max(outputs.data, 1)
```

```
    y_pred = predicted.cpu().numpy()

# Save the model checkpoint
torch.save(model.state_dict(), 'model.ckpt')

acc = metrics.accuracy_score(y_test, y_pred)
print("NN acc: %f" %acc )

recall = metrics.recall_score(y_test, y_pred, average=None)
print("NN recall: %f",  recall )

prec = metrics.precision_score(y_test, y_pred, average=None)
print("NN precision:", prec )

f1 = metrics.f1_score(y_test, y_pred, average=None)
print("NN f1:", f1 )

conf_mat = metrics.confusion_matrix(y_test, y_pred)
print("NN conf_mat: %f")
print(conf_mat)
```