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

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.A screen shot of a graph

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.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)