[CLS] simple_mlp_pytorch

October 3, 2025

[1]:
```
import numpy as np
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import␣
 ↪mean_squared_error,accuracy_score,classification_report,confusion_matrix,ConfusionMatrixDis
import matplotlib.pyplot as plt
import torch.nn.functional as F
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
import pandas
```

[2]:
```
# CHOOSE DATASET

# Binary classification dataset
data = datasets.fetch_openml(name="diabetes",version=1, as_frame=True)

X = data.data.values
y = data.target.values
X.shape

# Keep as DataFrame for named-column ops
df = data.data.copy()
y = np.where(data.target.values == "tested_positive", 1, 0).astype(np.float32)

# indices of features with invalid zeros
invalid_idx = [1, 2, 3, 4, 5, 7]

# count zeros per feature
zero_counts = (X[:, invalid_idx] == 0).sum(axis=0)
rows_with_zero = (X[:, invalid_idx] == 0).any(axis=1).sum()

print("Zeros per feature:\n", zero_counts)
print(f"Rows with  1 zero: {rows_with_zero} / {len(df)}")
```

```
(len(X), zero_counts, rows_with_zero)

# Drop columns 3 and 4 (0-based indexing)
X = np.delete(X, [3, 4], axis=1)

# Keep only rows where Glucose, BloodPressure, BMI are non-zero
mask = (X[:, [1, 2, 3, 4]] != 0).all(axis=1)
X = X[mask]
y = y[mask]
```

```
Zeros per feature:
 [  5  35 227 374  11   0]
Rows with  1 zero: 376 / 768
```

[3]:
```
#train test spliting
test_size=0.2
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=test_size,␣
 ↪random_state=42)
```

[4]:
```
# Standardize features
scaler=StandardScaler()
Xtr= scaler.fit_transform(Xtr)
Xte= scaler.transform(Xte)
```

[5]:
```
class MLP(nn.Module):
    def __init__(self, input_size, output_size=1, dropout_prob=0.5):
        super(MLP, self).__init__()

        self.fc1 = nn.Linear(input_size, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, 64)
        self.fc4 = nn.Linear(64, 64)
        self.out = nn.Linear(64, output_size)

        self.dropout = nn.Dropout(p=dropout_prob)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.dropout(x)

        x = F.relu(self.fc2(x))
        x = self.dropout(x)

        x = F.relu(self.fc3(x))
        x = self.dropout(x)

        x = F.relu(self.fc4(x))
        x = self.dropout(x)
```

```
        x = self.out(x)
        return x
```

[6]:
```
num_epochs=100
lr=0.0003
dropout=0.2
batch_size=64
```

[7]:
```
Xtr = torch.tensor(Xtr, dtype=torch.float32)
ytr = torch.tensor(ytr, dtype=torch.float32)
Xte = torch.tensor(Xte, dtype=torch.float32)
yte = torch.tensor(yte, dtype=torch.float32)

# Wrap Xtr and ytr into a dataset
train_dataset = TensorDataset(Xtr, ytr)

# Create DataLoader
train_dataloader = DataLoader(train_dataset, batch_size=batch_size,␣
 ↪shuffle=True)
```

[8]:
```
# Model, Loss, Optimizer
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = MLP(input_size=Xtr.shape[1], dropout_prob=dropout).to(device)
criterion = nn.BCEWithLogitsLoss()  # for binary classification
optimizer = optim.Adam(model.parameters(), lr=lr)
```

[9]:
```
# Training loop
for epoch in range(num_epochs):
    model.train()
    epoch_loss = 0.0

    for batch_x, batch_y in train_dataloader:
        batch_x = batch_x.to(device)
        batch_y = batch_y.to(device)

        logits = model(batch_x)
        loss = criterion(logits, batch_y.view(-1, 1))

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()

    avg_loss = epoch_loss / len(train_dataloader)
```

```
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {avg_loss:.4f}")
```

```
Epoch [1/100], Loss: 0.6730
Epoch [2/100], Loss: 0.6669
Epoch [3/100], Loss: 0.6624
Epoch [4/100], Loss: 0.6630
Epoch [5/100], Loss: 0.6381
Epoch [6/100], Loss: 0.6343
Epoch [7/100], Loss: 0.6055
Epoch [8/100], Loss: 0.6017
Epoch [9/100], Loss: 0.5675
Epoch [10/100], Loss: 0.5774
Epoch [11/100], Loss: 0.5246
Epoch [12/100], Loss: 0.5239
Epoch [13/100], Loss: 0.5293
Epoch [14/100], Loss: 0.4761
Epoch [15/100], Loss: 0.4889
Epoch [16/100], Loss: 0.5030
Epoch [17/100], Loss: 0.4604
Epoch [18/100], Loss: 0.4668
Epoch [19/100], Loss: 0.4959
Epoch [20/100], Loss: 0.4499
Epoch [21/100], Loss: 0.4668
Epoch [22/100], Loss: 0.4624
Epoch [23/100], Loss: 0.4320
Epoch [24/100], Loss: 0.4519
Epoch [25/100], Loss: 0.4800
Epoch [26/100], Loss: 0.4419
Epoch [27/100], Loss: 0.5479
Epoch [28/100], Loss: 0.5238
Epoch [29/100], Loss: 0.4859
Epoch [30/100], Loss: 0.4435
Epoch [31/100], Loss: 0.4712
Epoch [32/100], Loss: 0.4592
Epoch [33/100], Loss: 0.4637
Epoch [34/100], Loss: 0.5141
Epoch [35/100], Loss: 0.4478
Epoch [36/100], Loss: 0.5066
Epoch [37/100], Loss: 0.4727
Epoch [38/100], Loss: 0.4530
Epoch [39/100], Loss: 0.4711
Epoch [40/100], Loss: 0.4277
Epoch [41/100], Loss: 0.4638
Epoch [42/100], Loss: 0.4734
Epoch [43/100], Loss: 0.4410
Epoch [44/100], Loss: 0.4750
Epoch [45/100], Loss: 0.4241
Epoch [46/100], Loss: 0.4414
```

```
Epoch [47/100], Loss: 0.5735
Epoch [48/100], Loss: 0.4204
Epoch [49/100], Loss: 0.4649
Epoch [50/100], Loss: 0.4405
Epoch [51/100], Loss: 0.4783
Epoch [52/100], Loss: 0.4769
Epoch [53/100], Loss: 0.4284
Epoch [54/100], Loss: 0.4111
Epoch [55/100], Loss: 0.4272
Epoch [56/100], Loss: 0.4863
Epoch [57/100], Loss: 0.4392
Epoch [58/100], Loss: 0.4137
Epoch [59/100], Loss: 0.4195
Epoch [60/100], Loss: 0.4857
Epoch [61/100], Loss: 0.4733
Epoch [62/100], Loss: 0.4258
Epoch [63/100], Loss: 0.4185
Epoch [64/100], Loss: 0.4048
Epoch [65/100], Loss: 0.4563
Epoch [66/100], Loss: 0.4183
Epoch [67/100], Loss: 0.4831
Epoch [68/100], Loss: 0.5334
Epoch [69/100], Loss: 0.4311
Epoch [70/100], Loss: 0.4908
Epoch [71/100], Loss: 0.4396
Epoch [72/100], Loss: 0.4244
Epoch [73/100], Loss: 0.4514
Epoch [74/100], Loss: 0.4238
Epoch [75/100], Loss: 0.4764
Epoch [76/100], Loss: 0.4454
Epoch [77/100], Loss: 0.4340
Epoch [78/100], Loss: 0.4671
Epoch [79/100], Loss: 0.4570
Epoch [80/100], Loss: 0.4382
Epoch [81/100], Loss: 0.4522
Epoch [82/100], Loss: 0.4574
Epoch [83/100], Loss: 0.4529
Epoch [84/100], Loss: 0.4441
Epoch [85/100], Loss: 0.4592
Epoch [86/100], Loss: 0.4888
Epoch [87/100], Loss: 0.4332
Epoch [88/100], Loss: 0.4039
Epoch [89/100], Loss: 0.4684
Epoch [90/100], Loss: 0.4263
Epoch [91/100], Loss: 0.4360
Epoch [92/100], Loss: 0.4245
Epoch [93/100], Loss: 0.4292
Epoch [94/100], Loss: 0.4371
```

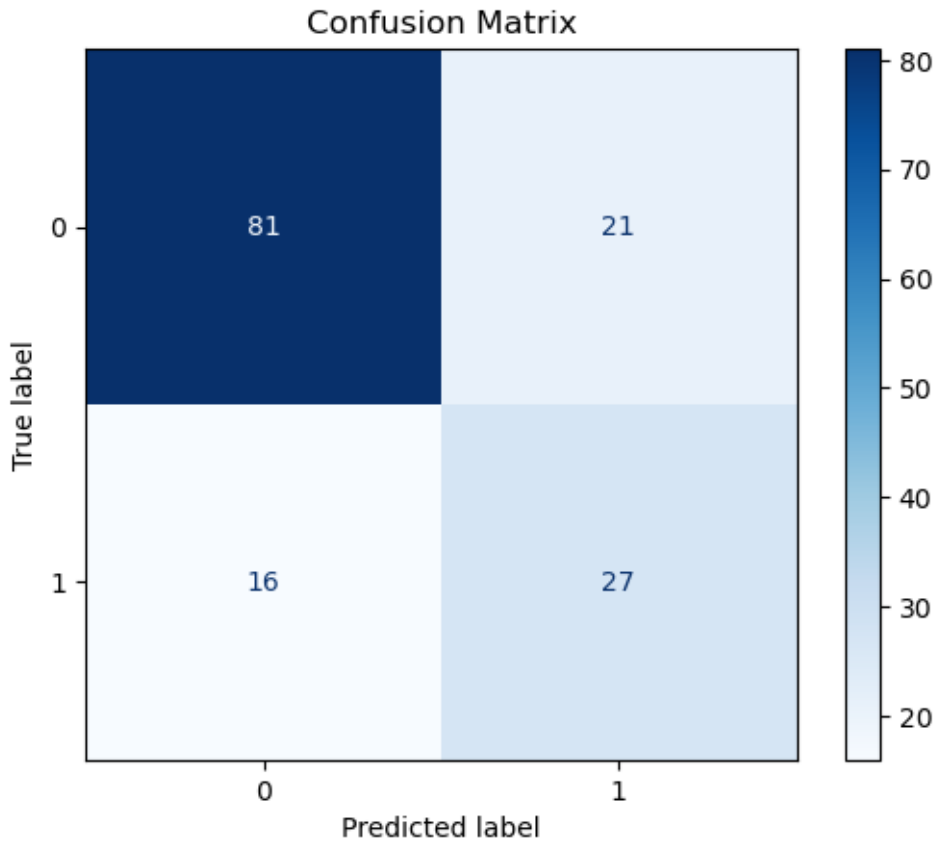
```
Epoch [95/100], Loss: 0.4524
Epoch [96/100], Loss: 0.4075
Epoch [97/100], Loss: 0.4413
Epoch [98/100], Loss: 0.4369
Epoch [99/100], Loss: 0.4529
Epoch [100/100], Loss: 0.4235
```

[10]:
```
y_pred=model(Xte)
print(f'ACC:{accuracy_score(yte.detach().numpy(),y_pred.detach().numpy()>0.
↪5)}') #classification
```

```
ACC:0.7931034482758621
```

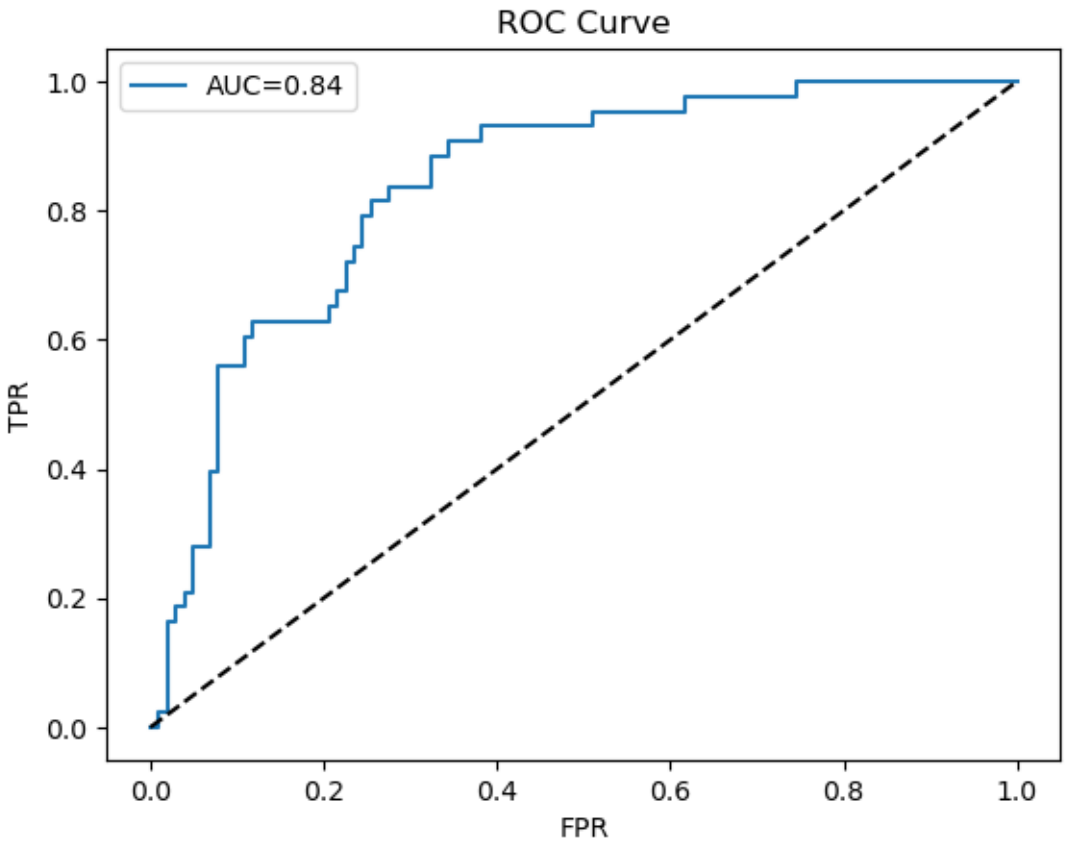
[11]:
```
model.eval()
device = next(model.parameters()).device

Xte_t = torch.as_tensor(Xte, dtype=torch.float32, device=device)
with torch.no_grad():
    logits = model(Xte_t)
    # make it 1D
    if logits.ndim > 1: logits = logits.squeeze(-1)
    y_proba = torch.sigmoid(logits).cpu().numpy()

print('proba min/mean/max:', y_proba.min(), y_proba.mean(), y_proba.max())
y_pred = (y_proba > 0.5).astype(int)

# ---- confusion matrix ----
cm = confusion_matrix(yte, y_pred)
ConfusionMatrixDisplay(cm).plot(cmap="Blues")
plt.title("Confusion Matrix")
plt.show()
plt.savefig("../Plots/ConfusionMatrixMLP.pdf", format="pdf",␣
↪bbox_inches="tight")
```

```
proba min/mean/max: 0.0020300266 0.36886007 0.93190086
```



<Figure size 640x480 with 0 Axes>

[12]:
```
fpr, tpr, _ = roc_curve(yte, y_proba)
roc_auc = auc(fpr, tpr)
plt.plot(fpr, tpr, label=f"AUC={roc_auc:.2f}")
plt.plot([0,1],[0,1],'k--')
plt.xlabel("FPR"); plt.ylabel("TPR"); plt.title("ROC Curve"); plt.legend(); plt.
↪show()
plt.savefig("../Plots/RoCMLP.pdf", format="pdf", bbox_inches="tight")
```




```
<Figure size 640x480 with 0 Axes>
```

[ ]: