```
# getting data from google drive

from google.colab import drive
drive.mount('/content/drive')
```

```
# importing the libraries
import torch
import torch.nn as nn
import torchvision.datasets as datasets
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import torch.nn.functional as F
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np
```

```
# loading the dataset from google drive

trainset = datasets.MNIST(root='/content/drive/MyDrive/MLP_data', train=True, transform=transforms.ToTensor(), download=True)
testset = datasets.MNIST(root='/content/drive/MyDrive/MLP_data', train=False, transform=transforms.ToTensor(), download=True)

# Create the dataloaders
trainloader = DataLoader(trainset, batch_size=64, shuffle=True)
testloader = DataLoader(testset, batch_size=64, shuffle=False)
```

```
# Define the MLP class
class MLP(nn.Module):
    def __init__(self, num_classes=100):
        super(MLP, self).__init__()

        # Define the layers of the MLP
        self.fc1 = nn.Linear(32*32*3, 512) # input layer
        self.fc2 = nn.Linear(512, 256) # hidden layer
        self.fc3 = nn.Linear(256, 128) # hidden layer
        self.fc4 = nn.Linear(128, num_classes) # output layer

        # Add batch normalization layers
        self.bn1 = nn.BatchNorm1d(512)
        self.bn2 = nn.BatchNorm1d(256)

        # Add dropout layers
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        # Flatten the input images
        x = x.view(-1, 32*32*3)

        # Pass the input through the first fully connected layer and apply batch normalization and ReLU activation
        x = self.bn1(F.relu(self.fc1(x)))

        # Apply dropout to the output of the first fully connected layer
        x = self.dropout(x)

        # Pass the output through the second fully connected layer and apply batch normalization and ReLU activation
        x = self.bn2(F.relu(self.fc2(x)))

        # Apply dropout to the output of the second fully connected layer
        x = self.dropout(x)

        # Pass the output through the third fully connected layer and apply ReLU activation
        x = F.relu(self.fc3(x))

        # Apply dropout to the output of the third fully connected layer
        x = self.dropout(x)

        # Pass the output through the fourth fully connected layer and apply log softmax activation
        x = F.log_softmax(self.fc4(x), dim=1)

        return x
```

Batch normalization can help to stabilize the training process and reduce overfitting. We added batch normalization layers after the first two fully connected layers.

Dropout can help to regularize the model and reduce overfitting. We added dropout layers after the first two fully connected layers and after the third fully connected layer.

Instead of using ReLU activation function for all the layers, we added batch normalization layers and used ReLU activation function for the first two fully connected layers, and used only ReLU activation function for the third fully connected layer.

```
# Initialize the model and optimizer
# Define the model
model = MLP()

# Define the optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Define the loss function
criterion = nn.CrossEntropyLoss()

num_epochs = 10
```

```
# Train model
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(trainloader):
        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i+1) % 100 == 0:
            print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'.format(epoch+1, num_epochs, i+1, len(trainloader), loss.item())
```

```
Epoch [2/10], Step [600/782], Loss: 3.8801
Epoch [2/10], Step [700/782], Loss: 3.6288
Epoch [3/10], Step [100/782], Loss: 3.9580
Epoch [3/10], Step [200/782], Loss: 3.9558
Epoch [3/10], Step [300/782], Loss: 3.8788
Epoch [3/10], Step [400/782], Loss: 4.0242
Epoch [3/10], Step [500/782], Loss: 3.8433
Epoch [3/10], Step [600/782], Loss: 3.6007
Epoch [3/10], Step [700/782], Loss: 3.8738
Epoch [4/10], Step [100/782], Loss: 3.9789
Epoch [4/10], Step [200/782], Loss: 3.7609
Epoch [4/10], Step [300/782], Loss: 3.9748
Epoch [4/10], Step [400/782], Loss: 3.8763
Epoch [4/10], Step [500/782], Loss: 4.0305
Epoch [4/10], Step [600/782], Loss: 4.1171
Epoch [4/10], Step [700/782], Loss: 4.0862
Epoch [5/10], Step [100/782], Loss: 3.8942
Epoch [5/10], Step [200/782], Loss: 4.0446
Epoch [5/10], Step [300/782], Loss: 3.9479
Epoch [5/10], Step [400/782], Loss: 3.7219
Epoch [5/10], Step [500/782], Loss: 3.6615
Epoch [5/10], Step [600/782], Loss: 3.9151
Epoch [5/10], Step [700/782], Loss: 3.6865
Epoch [6/10], Step [100/782], Loss: 3.6484
Epoch [6/10], Step [200/782], Loss: 3.8249
Epoch [6/10], Step [300/782], Loss: 3.9109
Epoch [6/10], Step [400/782], Loss: 3.8526
Epoch [6/10], Step [500/782], Loss: 3.8373
Epoch [6/10], Step [600/782], Loss: 4.1370
Epoch [6/10], Step [700/782], Loss: 4.1223
Epoch [7/10], Step [100/782], Loss: 3.7403
Epoch [7/10], Step [200/782], Loss: 3.6059
Epoch [7/10], Step [300/782], Loss: 3.8795
Epoch [7/10], Step [400/782], Loss: 3.9663
Epoch [7/10], Step [500/782], Loss: 3.5932
Epoch [7/10], Step [600/782], Loss: 3.8928
Epoch [7/10], Step [700/782], Loss: 3.9830
Epoch [8/10], Step [100/782], Loss: 3.7288
Epoch [8/10], Step [200/782], Loss: 3.8721
Epoch [8/10], Step [300/782], Loss: 3.6976
Epoch [8/10], Step [400/782], Loss: 3.4182
Epoch [8/10], Step [500/782], Loss: 3.8970
```

```
Epoch [10/10], Step [400/782], Loss: 4.0215
Epoch [10/10], Step [500/782], Loss: 3.6394
Epoch [10/10], Step [600/782], Loss: 3.7810
Epoch [10/10], Step [700/782], Loss: 3.5625
```

```
# Evaluate model
from sklearn.metrics import f1_score

# Initialize lists
y_pred_list = []
y_true_list = []

# Set model to evaluation mode
model.eval()

# Iterate over test data
for images, labels in trainloader:
    # Forward pass
    outputs = model(images)
    # Get predictions from the maximum value
    _, predicted = torch.max(outputs.data, 1)
    # Append predictions
    y_pred_list.append(predicted)
    # Append ground truths
    y_true_list.append(labels)

# Convert lists to tensors
y_pred_list = torch.cat(y_pred_list).cpu().numpy()
y_true_list = torch.cat(y_true_list).cpu().numpy()

# Calculate F1 score
print("The F1_score for the training set:" ,f1_score(y_true_list, y_pred_list, average='macro'))
```

```
The F1_score for the training set: 0.15116660595711442
```

```
# Test model
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in testloader:
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Test Accuracy of the model on the 10000 test images: {} %'.format(100 * correct / total))
```

```
Test Accuracy of the model on the 10000 test images: 17.7 %
```

```
# Initialize lists
y_pred_list = []
y_true_list = []

# Set model to evaluation mode
model.eval()

# Iterate over test data
for images, labels in testloader:
    # Forward pass
    outputs = model(images)
    # Get predictions from the maximum value
    _, predicted = torch.max(outputs.data, 1)
    # Append predictions
    y_pred_list.append(predicted)
    # Append ground truths
    y_true_list.append(labels)

# Convert lists to tensors
y_pred_list = torch.cat(y_pred_list).cpu().numpy()
y_true_list = torch.cat(y_true_list).cpu().numpy()

# Calculate F1 score
print("The F1_score for the testing set:" ,f1_score(y_true_list, y_pred_list, average='macro'))
```

```
The F1_score for the testing set: 0.14622218194988856
```

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

# Load model
#model.load_state_dict(torch.load('model.ckpt'))
```