Problem 2: Multinomial Logistic Regression from Pre-trained Feature Extractor

Author: Yankun (Alex) Meng

In [1]:
```
import ssl
import torch
import torchvision
import matplotlib.pyplot as plt
from torchvision import transforms
from pretrained_model.Encoder import extractor
```

Step 1

In this problem, you will implement a logistic regression model from scratch to classify MNIST given a pre-trained feature extractor. Specifically, you will walk through the following steps:

Step 1: Load the pre-trained weights to your feature extractor. The code defining the architecture of your feature extractor is included in the **attachment**.

In [2]:
```
# Instantiate the model
device = "cpu" # i will just use cpu
feature_extractor = extractor().to(device)
```

In [3]:
```
# load the pretrained weights into the extractor
state_dict = torch.load("./pretrained_model/feature_extractor_weights.pth", map_loc
feature_extractor.load_state_dict(state_dict)
```

Out[3]:
```
<All keys matched successfully>
```

In [4]:
```
feature_extractor.eval()

for p in feature_extractor.parameters():
    p.requires_grad = False
```

Step 2

Step 2: Extract the latent representation (denoted as $h \in \mathbb{R}^k$) from each sample in MNIST (denoted as $x \in \mathbb{R}^{784}$) using the pre-trained feature extractor.

In [5]:
```
ssl._create_default_https_context = ssl._create_unverified_context
```

In [6]:
```
# Load MNIST from torchvision
train = torchvision.datasets.MNIST(
    root='./data',
    train=True,
    download=True,
    transform=transforms.ToTensor()
)

test = torchvision.datasets.MNIST(
    root='./data',
    train=False,
    download=True,
    transform=transforms.ToTensor()
)
```

In [7]:
```
x, y = train[0] # (image, label)
print(x.shape)
print(y)
```

```
torch.Size([1, 28, 28])
5
```

In [8]:
```
# Create Data Loader wrappers
trainloader = torch.utils.data.DataLoader(
    train, batch_size=64, shuffle=True
)

testloader = torch.utils.data.DataLoader(
    test, batch_size=64, shuffle=False
)
```

Now we are ready to extract the features using our pretrained feature extractor.

In [9]:
```
# (H, Y)
H_train, Y_train = [], []
H_test,  Y_test  = [], []
```

In [10]:
```
# Loop Assignment
with torch.no_grad():
    # train set
    for x, y in trainloader:
        h = feature_extractor(x)
        H_train.append(h)
        Y_train.append(y)

    # test set
    for x, y in testloader:
        h = feature_extractor(x)
        H_test.append(h)
        Y_test.append(y)
```

In [11]:
```
# Concatenate all batches
H_train = torch.cat(H_train, dim=0)
Y_train = torch.cat(Y_train, dim=0)
H_test = torch.cat(H_test, dim=0)
Y_test = torch.cat(Y_test, dim=0)
print(H_train.shape)
print(Y_train.shape)
print(H_test.shape)
print(Y_test.shape)
```

```
torch.Size([60000, 256])
torch.Size([60000])
torch.Size([10000, 256])
torch.Size([10000])
```

Step 3: Derive the gradient of W and b with respect to the *cross-entropy* loss between the label and prediction $\hat{y}$ of the model:

$$\hat{y} = \sigma(W^T h + b), \tag{1}$$

where $\sigma(\cdot)$ denotes the softmax function.

$$\hat{\mathbf{y}} = \sigma(\mathbf{z}) = \sigma(\mathbf{W}^\top \mathbf{h} + \mathbf{b})$$

For a single sample with one-hot label $\mathbf{y} \in \mathbb{R}^C$, the cross-entropy loss is

$$L = -\sum_{c=1}^{C} y_c \log \hat{y}_c$$

For a batch of N samples, the average loss is

$$L = -\frac{1}{N} \sum_{i=1}^{N} \sum_{c=1}^{C} y_{i,c} \log \hat{y}_{i,c}$$

We first compute the derivative of the loss with respect to the input to softmax $\mathbf{z}$

$$\begin{aligned}
\hat{y}_c &= \frac{e^{z_c}}{\sum_{j=1}^{C} e^{z_j}} \\
L &= -\sum_{k=1}^{C} y_k \log \hat{y}_k \\
\frac{\partial L}{\partial z_c} &= \sum_{k=1}^{C} \frac{\partial L}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial z_c} \\
\frac{\partial L}{\partial \hat{y}_k} &= -\frac{y_k}{\hat{y}_k} \\
\frac{\partial \hat{y}_k}{\partial z_c} &= \begin{cases} \hat{y}_c(1-\hat{y}_c), & k = c \\ -\hat{y}_k \hat{y}_c, & k \neq c \end{cases} \\
\frac{\partial L}{\partial z_c} &= -\frac{y_c}{\hat{y}_c} \cdot \hat{y}_c(1-\hat{y}_c) + \sum_{k \neq c} -\frac{y_k}{\hat{y}_k}(-\hat{y}_k \hat{y}_c) \\
&= \hat{y}_c - y_c
\end{aligned}$$

Vectorized for the batch

$$\frac{\partial L}{\partial \mathbf{Z}} = \hat{\mathbf{Y}} - \mathbf{Y},$$

where $\hat{\mathbf{Y}}, \mathbf{Y} \in \mathbb{R}^{N \times C}$.

Using the chain rule

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \mathbf{Z}} \cdot \frac{\partial \mathbf{Z}}{\partial \mathbf{W}}$$

Since $\mathbf{Z} = \mathbf{H}\mathbf{W} + \mathbf{1}_N \mathbf{b}^\top$

$$\frac{\partial \mathbf{Z}}{\partial \mathbf{W}} = \mathbf{H}^\top$$

Thus, the gradient w.r.t. weights for the batch is

$$\boxed{\frac{\partial L}{\partial \mathbf{W}} = \frac{1}{N}\mathbf{H}^\top(\hat{\mathbf{Y}} - \mathbf{Y})}$$

Similarly, the gradient w.r.t. bias is

$$\frac{\partial L}{\partial \mathbf{b}} = \frac{\partial L}{\partial \mathbf{Z}} \cdot \frac{\partial \mathbf{Z}}{\partial \mathbf{b}}$$

Since $\frac{\partial \mathbf{Z}}{\partial \mathbf{b}} = \mathbf{1}_N$ (broadcasted sum over batch)

$$\boxed{\frac{\partial L}{\partial \mathbf{b}} = \frac{1}{N}\sum_{i=1}^{N}(\hat{\mathbf{y}}_i - \mathbf{y}_i)}$$

with resulting shape $\mathbb{R}^C$.

Step 4: Manually implement Stochastic Gradient Descent (SGD) from *scratch (using matrix operations only)*. Use the gradient derived in **Step 3** to run training and update the parameters W and b. Finally, report the classification accuracy on the test set.

In [12]:
```python
# Parameters
num_classes = 10
num_features = H_train.shape[1]  # 256 from extractor
lr = 0.1
num_epochs = 20
batch_size = 256
```

In [13]:
```python
# Initialize weights and bias
W = torch.zeros((num_features, num_classes), dtype=torch.float32)
b = torch.zeros(num_classes, dtype=torch.float32)
```

In [14]:
```python
# Convert labels to one-hot
def to_one_hot(y, C):
    return torch.eye(C)[y]  # shape: [N, C]
```

```
Y_train_onehot = to_one_hot(Y_train, num_classes)
Y_test_onehot  = to_one_hot(Y_test, num_classes)
```

In [15]:
```
# Softmax function
def softmax(z):
    # subtract max for numerical stability
    exp_z = torch.exp(z - z.max(dim=1, keepdim=True)[0])
    return exp_z / exp_z.sum(dim=1, keepdim=True)
```

In [16]:
```
# Cross-entropy loss
def cross_entropy(pred, target):
    return - (target * torch.log(pred + 1e-8)).sum(dim=1).mean()
```

In [17]:
```
# Number of training samples
num_train = H_train.shape[0]

for epoch in range(num_epochs):
    # Shuffle dataset
    perm = torch.randperm(num_train)
    H_shuffled = H_train[perm]
    Y_shuffled = Y_train_onehot[perm]

    # Mini-batch SGD
    for i in range(0, num_train, batch_size):
        H_batch = H_shuffled[i:i+batch_size]  # [batch, features]
        Y_batch = Y_shuffled[i:i+batch_size]  # [batch, C]

        # Forward pass: linear + softmax
        z = H_batch @ W + b            # [batch, C]
        y_hat = softmax(z)             # [batch, C]

        # Compute gradients manually (from Step 3)
        dW = H_batch.T @ (y_hat - Y_batch) / H_batch.shape[0]  # [features, C]
        db = (y_hat - Y_batch).mean(dim=0)                      # [C]

        # Update parameters
        W -= lr * dW
        b -= lr * db

    # Compute training loss per epoch
    z_train = H_train @ W + b
    y_hat_train = softmax(z_train)
    loss = cross_entropy(y_hat_train, Y_train_onehot)
    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {loss.item():.4f}")
```

```
Epoch 1/20, Loss: 0.0281
Epoch 2/20, Loss: 0.0181
Epoch 3/20, Loss: 0.0139
Epoch 4/20, Loss: 0.0115
Epoch 5/20, Loss: 0.0099
Epoch 6/20, Loss: 0.0087
Epoch 7/20, Loss: 0.0079
Epoch 8/20, Loss: 0.0072
Epoch 9/20, Loss: 0.0066
Epoch 10/20, Loss: 0.0061
Epoch 11/20, Loss: 0.0057
Epoch 12/20, Loss: 0.0054
Epoch 13/20, Loss: 0.0051
Epoch 14/20, Loss: 0.0048
Epoch 15/20, Loss: 0.0046
Epoch 16/20, Loss: 0.0044
Epoch 17/20, Loss: 0.0042
Epoch 18/20, Loss: 0.0040
Epoch 19/20, Loss: 0.0038
Epoch 20/20, Loss: 0.0037
```

In [18]:
```python
# ---- Evaluate on test set ----
z_test = H_test @ W + b
y_hat_test = softmax(z_test)
y_pred = torch.argmax(y_hat_test, dim=1)

accuracy = (y_pred == Y_test).float().mean()
print(f"Test set classification accuracy: {accuracy.item() * 100:.2f}%")
```

```
Test set classification accuracy: 99.08%
```