# A1_3

March 27, 2024

## 0.1 MNIST Data Download

[1]:
```python
# Load and preprocess MNIST dataset using NumPy
from keras.datasets import mnist
```

## 0.2 Part I

[5]:
```python
import numpy as np

(X_tr, y_tr), (X_t, y_t) = mnist.load_data()
X_tr = X_tr.reshape(-1, 28*28) / 255.
X_t = X_t.reshape(-1, 28*28) / 255.

# Define neural network architecture
class NeuralNetwork:
    def __init__(self, in_s, hid_s, out_s):
        self.W1 = np.random.randn(in_s, hid_s)
        self.b1 = np.zeros(hid_s)
        self.W2 = np.random.randn(hid_s, out_s)
        self.b2 = np.zeros(out_s)

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def forward(self, x):
        self.z1 = np.dot(x, self.W1) + self.b1
        self.a1 = self.sigmoid(self.z1)
        self.z2 = np.dot(self.a1, self.W2) + self.b2
        self.a2 = self.sigmoid(self.z2)
        return self.a2

# Convert data to NumPy arrays
in_s = X_tr.shape[1]
hid_s = 64
out_s = 10
X_tr, y_tr = np.array(X_tr), np.array(y_tr)
X_t, y_t = np.array(X_t), np.array(y_t)
```

```python
# Initialize the neural network
model = NeuralNetwork(in_s, hid_s, out_s)

# Define loss function
def mse_loss(predictions, targets):
    return np.mean((predictions - targets)**2)

# Define sigmoid derivative
def sigmoid_derivative(x):
    return x * (1 - x)

# Define learning rate
learning_rate = 0.1

# Train the model
num_epochs = 10
batch_size = 64
num_batches = X_tr.shape[0]

# Train the model with SGD
for epoch in range(num_epochs):
    epoch_l = 0.0
    for i in range(X_tr.shape[0]):
        # Select a random sample
        index = np.random.randint(X_tr.shape[0])
        input_sample = X_tr[index:index+1]
        label_sample = y_tr[index:index+1]

        # Forward pass
        output = model.forward(input_sample)

        # One-hot encode labels
        label_onehot = np.eye(out_s)[label_sample.astype(int)]

        # Compute loss
        loss = mse_loss(output, label_onehot)
        epoch_l += loss

        # Backpropagation
        d_loss = 2 * (output - label_onehot)
        d_z2 = d_loss * sigmoid_derivative(model.a2)
        d_a1 = np.dot(d_z2, model.W2.T)
        d_z1 = d_a1 * sigmoid_derivative(model.a1)

        # Update parameters
        model.W2 -= learning_rate * np.dot(model.a1.T, d_z2)
        model.b2 -= learning_rate * np.sum(d_z2, axis=0)
```

```
        model.W1 -= learning_rate * np.dot(input_sample.T, d_z1)
        model.b1 -= learning_rate * np.sum(d_z1, axis=0)

    epoch_l /= X_tr.shape[0]
    print(f"Ep [{epoch+1}/{num_epochs}], L: {epoch_l:.4f}")


# Evaluate the model
correct = 0
total = 0

for i in range(0, len(X_t), batch_size):
    inputs = X_t[i:i+batch_size]
    labels = y_t[i:i+batch_size]
    outputs = model.forward(inputs)
    predicted = np.argmax(outputs, axis=1)
    total += batch_size
    correct += np.sum(predicted == labels)

accuracy = correct / total * 100
print(f"Accuracy on the test set: {accuracy:.2f}%")
```

```
Ep [1/10], L: 0.0468
Ep [2/10], L: 0.0197
Ep [3/10], L: 0.0111
Ep [4/10], L: 0.0091
Ep [5/10], L: 0.0082
Ep [6/10], L: 0.0074
Ep [7/10], L: 0.0072
Ep [8/10], L: 0.0066
Ep [9/10], L: 0.0059
Ep [10/10], L: 0.0058
Accuracy on the test set: 94.69%
```

*Using the above trained model to predict this image:*

[7]:
```
from PIL import Image

img = Image.open('img_1.jpg')

# Assuming X_custom contains your own data
X_custom = np.array(img)
X_custom = X_custom.reshape(-1, 28*28)

# Normalize your custom data (if necessary)
```

```
X_custom_normalized = X_custom / 255.0

# Create an empty array to store predictions
predictions = []

# Loop over each sample in the custom data
for sample in X_custom_normalized:
    # Reshape the sample to match the input size of the model
    sample = sample.reshape(1, -1)

    # Forward pass through the model
    output = model.forward(sample)

    # Get the predicted label (index of the maximum value in the output)
    predicted_label = np.argmax(output)

    # Append the predicted label to the list of predictions
    predictions.append(predicted_label)

# Convert the list of predictions to a NumPy array
predictions = np.array(predictions)

# Print the predictions
print("Predictions:", predictions)
```

```
Predictions: [2]
```

## 0.3 Pytorch Dependencies

[21]:
```
# Setting up Pytorch imports and Data

import torch
import torch.nn as nn
import torch.optim as opt
import torchvision.datasets as datset
import torchvision.transforms as trans
from torch.utils.data import DataLoader

# Set device
dev = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Load MNIST dataset and preprocess
trans = trans.Compose([
    trans.ToTensor(),
    trans.Normalize((0.5,), (0.5,))
])
```

```
tr_dataset = datset.MNIST(root='./data', train=True, transform=trans,␣
↪download=True)
t_dataset = datset.MNIST(root='./data', train=False, transform=trans,␣
↪download=True)

tr_loader = DataLoader(tr_dataset, batch_size=batch_size, shuffle=True)
t_loader = DataLoader(t_dataset, batch_size=batch_size, shuffle=False)
```

### 0.4 Part II

[25]:
```
# Define the autoencoder architecture
class Autoencoder(nn.Module):
    def __init__(self, in_s, hid_s):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(in_s, hid_s),
            nn.Tanh()
        )
        self.decoder = nn.Linear(hid_s, in_s)

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x

# Initialize the autoencoder
in_s = 28 * 28  # MNIST image size
hid_s = 64
autoenc = Autoencoder(in_s, hid_s).to(dev)

# Loss and Optimizer
c = nn.MSELoss()
op = opt.Adam(autoenc.parameters(), lr=0.001)

# Train the autoencoder
num_epochs = 10
for ep in range(num_epochs):
    r_loss = 0.0
    for img, _ in tr_loader:
        img = img.view(img.size(0), -1).to(dev)

        out = autoenc(img)

        loss = c(out, img)

        op.zero_grad()
        loss.backward()
```

```
        op.step()

        r_loss += loss.item()

    ep_loss = r_loss / len(tr_loader)
    print(f"Ep [{ep+1}/{num_epochs}], L: {ep_loss:.4f}")
```

```
Ep [1/10], L: 0.1265
Ep [2/10], L: 0.0585
Ep [3/10], L: 0.0505
Ep [4/10], L: 0.0470
Ep [5/10], L: 0.0455
Ep [6/10], L: 0.0442
Ep [7/10], L: 0.0433
Ep [8/10], L: 0.0428
Ep [9/10], L: 0.0423
Ep [10/10], L: 0.0420
```

### 0.5 Part III

[23]:

```
# Define the neural network architecture
class MLP(nn.Module):
    def __init__(self, in_s, hid_s1, hid_s2, out_s):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(in_s, hid_s1)
        self.fc2 = nn.Linear(hid_s1, hid_s2)
        self.fc3 = nn.Linear(hid_s2, out_s)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.sigmoid(self.fc1(x))
        x = self.sigmoid(self.fc2(x))
        x = self.fc3(x)
        return x

# Hyperparameters
in_s = 784  # 28x28 for MNIST images
hid_s1 = 256
hid_s2 = 128
out_s = 10  # 10 classes for MNIST digits
lr = 0.2
batch_size = 64
num_epochs = 10




# Initialize the model
```

```python
model = MLP(in_s, hid_s1, hid_s2, out_s)

# Training loop
tot_stp = len(tr_loader)
for ep in range(num_epochs):
    for i, (img, lbl) in enumerate(tr_loader):
        # Flatten
        img = img.view(-1, 28*28)

        # Forward pass
        out = model(img)

        # Compute MSE loss
        c = nn.MSELoss()
        loss = c(out, torch.nn.functional.one_hot(lbl, num_classes=out_s).
 ↪float())

        # Zero gradients, backward pass, and optimize
        model.zero_grad()
        loss.backward()

        # SGD update
        with torch.no_grad():
            for par in model.parameters():
                par -= lr * par.grad

        if (i+1) % 100 == 0:
            print ('Ep [{}/{}], S [{}/{}], L: {:.4f}'
                   .format(ep+1, num_epochs, i+1, tot_stp, loss.item()))

# Test the model
with torch.no_grad():
    corr = 0
    total = 0
    for img, lbl in t_loader:
        img = img.view(-1, 28*28)
        out = model(img)
        _, predicted = torch.max(out.data, 1)
        total += lbl.size(0)
        corr += (predicted == lbl).sum().item()

    print('Accuracy of the network on the 10000 test images: {:.2f} %'.
 ↪format(100 * corr / total))
```

```
Ep [1/10], S [100/938], L: 0.0918
Ep [1/10], S [200/938], L: 0.0869
Ep [1/10], S [300/938], L: 0.0881
```

```
Ep [1/10], S [400/938], L: 0.0838
Ep [1/10], S [500/938], L: 0.0789
Ep [1/10], S [600/938], L: 0.0718
Ep [1/10], S [700/938], L: 0.0695
Ep [1/10], S [800/938], L: 0.0708
Ep [1/10], S [900/938], L: 0.0665
Ep [2/10], S [100/938], L: 0.0626
Ep [2/10], S [200/938], L: 0.0567
Ep [2/10], S [300/938], L: 0.0578
Ep [2/10], S [400/938], L: 0.0549
Ep [2/10], S [500/938], L: 0.0550
Ep [2/10], S [600/938], L: 0.0588
Ep [2/10], S [700/938], L: 0.0544
Ep [2/10], S [800/938], L: 0.0523
Ep [2/10], S [900/938], L: 0.0510
Ep [3/10], S [100/938], L: 0.0415
Ep [3/10], S [200/938], L: 0.0541
Ep [3/10], S [300/938], L: 0.0515
Ep [3/10], S [400/938], L: 0.0463
Ep [3/10], S [500/938], L: 0.0426
Ep [3/10], S [600/938], L: 0.0403
Ep [3/10], S [700/938], L: 0.0463
Ep [3/10], S [800/938], L: 0.0475
Ep [3/10], S [900/938], L: 0.0493
Ep [4/10], S [100/938], L: 0.0466
Ep [4/10], S [200/938], L: 0.0467
Ep [4/10], S [300/938], L: 0.0554
Ep [4/10], S [400/938], L: 0.0423
Ep [4/10], S [500/938], L: 0.0446
Ep [4/10], S [600/938], L: 0.0422
Ep [4/10], S [700/938], L: 0.0452
Ep [4/10], S [800/938], L: 0.0361
Ep [4/10], S [900/938], L: 0.0393
Ep [5/10], S [100/938], L: 0.0418
Ep [5/10], S [200/938], L: 0.0411
Ep [5/10], S [300/938], L: 0.0366
Ep [5/10], S [400/938], L: 0.0466
Ep [5/10], S [500/938], L: 0.0390
Ep [5/10], S [600/938], L: 0.0396
Ep [5/10], S [700/938], L: 0.0446
Ep [5/10], S [800/938], L: 0.0406
Ep [5/10], S [900/938], L: 0.0313
Ep [6/10], S [100/938], L: 0.0471
Ep [6/10], S [200/938], L: 0.0364
Ep [6/10], S [300/938], L: 0.0424
Ep [6/10], S [400/938], L: 0.0460
Ep [6/10], S [500/938], L: 0.0351
Ep [6/10], S [600/938], L: 0.0388
```

```
Ep [6/10], S [700/938], L: 0.0328
Ep [6/10], S [800/938], L: 0.0389
Ep [7/10], S [100/938], L: 0.0388
Ep [7/10], S [200/938], L: 0.0404
Ep [7/10], S [300/938], L: 0.0331
Ep [7/10], S [400/938], L: 0.0354
Ep [7/10], S [500/938], L: 0.0389
Ep [7/10], S [600/938], L: 0.0317
Ep [7/10], S [700/938], L: 0.0350
Ep [7/10], S [800/938], L: 0.0295
Ep [7/10], S [900/938], L: 0.0309
Ep [8/10], S [100/938], L: 0.0371
Ep [8/10], S [200/938], L: 0.0272
Ep [8/10], S [300/938], L: 0.0298
Ep [8/10], S [400/938], L: 0.0288
Ep [8/10], S [500/938], L: 0.0370
Ep [8/10], S [600/938], L: 0.0299
Ep [8/10], S [700/938], L: 0.0267
Ep [8/10], S [800/938], L: 0.0335
Ep [8/10], S [900/938], L: 0.0319
Ep [9/10], S [100/938], L: 0.0267
Ep [9/10], S [200/938], L: 0.0324
Ep [9/10], S [300/938], L: 0.0333
Ep [9/10], S [400/938], L: 0.0289
Ep [9/10], S [500/938], L: 0.0312
Ep [9/10], S [600/938], L: 0.0288
Ep [9/10], S [700/938], L: 0.0332
Ep [9/10], S [800/938], L: 0.0259
Ep [9/10], S [900/938], L: 0.0273
Ep [10/10], S [100/938], L: 0.0272
Ep [10/10], S [200/938], L: 0.0320
Ep [10/10], S [300/938], L: 0.0292
Ep [10/10], S [400/938], L: 0.0205
Ep [10/10], S [500/938], L: 0.0256
Ep [10/10], S [600/938], L: 0.0247
Ep [10/10], S [700/938], L: 0.0238
Ep [10/10], S [800/938], L: 0.0298
Ep [10/10], S [900/938], L: 0.0392
Accuracy of the network on the 10000 test images: 89.18 %
```

## 0.6 Part IV

[27]:
```python
# Define Autoencoder architecture
class Autoencoder(nn.Module):
    def __init__(self, in_s, hid_s):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Linear(in_s, hid_s)
```

```python
        self.decoder = nn.Linear(hid_s, in_s)

    def forward(self, x):
        x = torch.sigmoid(self.encoder(x))
        x = torch.sigmoid(self.decoder(x))
        return x

# Hyperparameters
in_s = 784  # 28x28 for MNIST images
hid_s = 256
batch_size = 64
num_epochs = 10

# Initialize the autoencoder
autoenc = Autoencoder(in_s, hid_s)

# Define the loss function and optimizer
c = nn.MSELoss()
op = opt.SGD(autoenc.parameters(), lr=0.1)

# Training loop for the autoencoder
tot_stp = len(tr_loader)
for ep in range(num_epochs):
    for i, (img, _) in enumerate(tr_loader):
        img = img.view(-1, in_s)

        # Forward pass
        out = autoenc(img)
        loss = c(out, img)

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

        if (i+1) % 100 == 0:
            print ('Autoencoder: Ep [{}/{}], S [{}/{}], L: {:.4f}'
                   .format(ep+1, num_epochs, i+1, tot_stp, loss.item()))

# Save the pre-trained weights of the first layer
pretrained_weights = autoenc.encoder.weight.detach().clone()

# Define the MLP architecture with pre-trained weights
class MLP(nn.Module):
    def __init__(self, in_s, hid_s1, hid_s2, out_s):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(in_s, hid_s1)
```

```
        self.fc2 = nn.Linear(hid_s1, hid_s2)
        self.fc3 = nn.Linear(hid_s2, out_s)
        self.sigmoid = nn.Sigmoid()

        # Initialize first two layers with pre-trained weights
        with torch.no_grad():
            self.fc1.weight = nn.Parameter(pretrained_weights)
            self.fc1.requires_grad = False  # Freeze the parameters

    def forward(self, x):
        x = self.sigmoid(self.fc1(x))
        x = self.sigmoid(self.fc2(x))
        x = self.sigmoid(self.fc3(x))
        return x

# Re-initialize the autoencoder for the fine-tuning phase
autoenc = Autoencoder(in_s, hid_s)

# Load MNIST dataset again for fine-tuning
tr_loader = DataLoader(tr_dataset, batch_size=batch_size, shuffle=True)

# Initialize the MLP model with pre-trained weights
model = MLP(in_s, hid_s1=hid_s, hid_s2=128, out_s=10)

# Define the loss function and optimizer
c = nn.MSELoss()
op = opt.SGD(model.parameters(), lr=0.1)

# Training loop for fine-tuning the entire MLP
for ep in range(num_epochs):
    for i, (img, lbl) in enumerate(tr_loader):
        img = img.view(-1, in_s)

        # Forward pass
        out = model(img)
        lbl = torch.nn.functional.one_hot(lbl, num_classes=10).float()
        loss = c(out, lbl)

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

        if (i+1) % 100 == 0:
            print ('Fine-tuning: Ep [{}/{}], S [{}/{}], L: {:.4f}'
                   .format(ep+1, num_epochs, i+1, tot_stp, loss.item()))
```

```
Autoencoder: Ep [1/10], S [100/938], L: 1.4770
Autoencoder: Ep [1/10], S [200/938], L: 1.1810
Autoencoder: Ep [1/10], S [300/938], L: 1.0684
Autoencoder: Ep [1/10], S [400/938], L: 1.0222
Autoencoder: Ep [1/10], S [500/938], L: 1.0032
Autoencoder: Ep [1/10], S [600/938], L: 0.9807
Autoencoder: Ep [1/10], S [700/938], L: 0.9776
Autoencoder: Ep [1/10], S [800/938], L: 0.9693
Autoencoder: Ep [1/10], S [900/938], L: 0.9638
Autoencoder: Ep [2/10], S [100/938], L: 0.9585
Autoencoder: Ep [2/10], S [200/938], L: 0.9520
Autoencoder: Ep [2/10], S [300/938], L: 0.9516
Autoencoder: Ep [2/10], S [400/938], L: 0.9499
Autoencoder: Ep [2/10], S [500/938], L: 0.9467
Autoencoder: Ep [2/10], S [600/938], L: 0.9468
Autoencoder: Ep [2/10], S [700/938], L: 0.9458
Autoencoder: Ep [2/10], S [800/938], L: 0.9425
Autoencoder: Ep [2/10], S [900/938], L: 0.9428
Autoencoder: Ep [3/10], S [100/938], L: 0.9417
Autoencoder: Ep [3/10], S [200/938], L: 0.9372
Autoencoder: Ep [3/10], S [300/938], L: 0.9365
Autoencoder: Ep [3/10], S [400/938], L: 0.9417
Autoencoder: Ep [3/10], S [500/938], L: 0.9407
Autoencoder: Ep [3/10], S [600/938], L: 0.9411
Autoencoder: Ep [3/10], S [700/938], L: 0.9350
Autoencoder: Ep [3/10], S [800/938], L: 0.9370
Autoencoder: Ep [3/10], S [900/938], L: 0.9336
Autoencoder: Ep [4/10], S [100/938], L: 0.9369
Autoencoder: Ep [4/10], S [200/938], L: 0.9366
Autoencoder: Ep [4/10], S [300/938], L: 0.9398
Autoencoder: Ep [4/10], S [400/938], L: 0.9308
Autoencoder: Ep [4/10], S [500/938], L: 0.9385
Autoencoder: Ep [4/10], S [600/938], L: 0.9308
Autoencoder: Ep [4/10], S [700/938], L: 0.9340
Autoencoder: Ep [4/10], S [800/938], L: 0.9346
Autoencoder: Ep [4/10], S [900/938], L: 0.9329
Autoencoder: Ep [5/10], S [100/938], L: 0.9324
Autoencoder: Ep [5/10], S [200/938], L: 0.9353
Autoencoder: Ep [5/10], S [300/938], L: 0.9342
Autoencoder: Ep [5/10], S [400/938], L: 0.9311
Autoencoder: Ep [5/10], S [500/938], L: 0.9294
Autoencoder: Ep [5/10], S [600/938], L: 0.9298
Autoencoder: Ep [5/10], S [700/938], L: 0.9317
Autoencoder: Ep [5/10], S [800/938], L: 0.9290
Autoencoder: Ep [5/10], S [900/938], L: 0.9290
Autoencoder: Ep [6/10], S [100/938], L: 0.9319
Autoencoder: Ep [6/10], S [200/938], L: 0.9315
Autoencoder: Ep [6/10], S [300/938], L: 0.9310
```

```
Autoencoder: Ep [6/10], S [400/938], L: 0.9297
Autoencoder: Ep [6/10], S [500/938], L: 0.9310
Autoencoder: Ep [6/10], S [600/938], L: 0.9328
Autoencoder: Ep [6/10], S [700/938], L: 0.9308
Autoencoder: Ep [6/10], S [800/938], L: 0.9326
Autoencoder: Ep [6/10], S [900/938], L: 0.9309
Autoencoder: Ep [7/10], S [100/938], L: 0.9334
Autoencoder: Ep [7/10], S [200/938], L: 0.9297
Autoencoder: Ep [7/10], S [300/938], L: 0.9309
Autoencoder: Ep [7/10], S [400/938], L: 0.9309
Autoencoder: Ep [7/10], S [500/938], L: 0.9315
Autoencoder: Ep [7/10], S [600/938], L: 0.9298
Autoencoder: Ep [7/10], S [700/938], L: 0.9312
Autoencoder: Ep [7/10], S [800/938], L: 0.9284
Autoencoder: Ep [7/10], S [900/938], L: 0.9305
Autoencoder: Ep [8/10], S [100/938], L: 0.9277
Autoencoder: Ep [8/10], S [200/938], L: 0.9302
Autoencoder: Ep [8/10], S [300/938], L: 0.9297
Autoencoder: Ep [8/10], S [400/938], L: 0.9286
Autoencoder: Ep [8/10], S [500/938], L: 0.9317
Autoencoder: Ep [8/10], S [600/938], L: 0.9272
Autoencoder: Ep [8/10], S [700/938], L: 0.9315
Autoencoder: Ep [8/10], S [800/938], L: 0.9279
Autoencoder: Ep [8/10], S [900/938], L: 0.9291
Autoencoder: Ep [9/10], S [100/938], L: 0.9291
Autoencoder: Ep [9/10], S [200/938], L: 0.9296
Autoencoder: Ep [9/10], S [300/938], L: 0.9294
Autoencoder: Ep [9/10], S [400/938], L: 0.9290
Autoencoder: Ep [9/10], S [500/938], L: 0.9243
Autoencoder: Ep [9/10], S [600/938], L: 0.9298
Autoencoder: Ep [9/10], S [700/938], L: 0.9292
Autoencoder: Ep [9/10], S [800/938], L: 0.9306
Autoencoder: Ep [9/10], S [900/938], L: 0.9288
Autoencoder: Ep [10/10], S [100/938], L: 0.9299
Autoencoder: Ep [10/10], S [200/938], L: 0.9291
Autoencoder: Ep [10/10], S [300/938], L: 0.9302
Autoencoder: Ep [10/10], S [400/938], L: 0.9309
Autoencoder: Ep [10/10], S [500/938], L: 0.9292
Autoencoder: Ep [10/10], S [600/938], L: 0.9291
Autoencoder: Ep [10/10], S [700/938], L: 0.9292
Autoencoder: Ep [10/10], S [800/938], L: 0.9304
Autoencoder: Ep [10/10], S [900/938], L: 0.9285
Fine-tuning: Ep [1/10], S [100/938], L: 0.0910
Fine-tuning: Ep [1/10], S [200/938], L: 0.0900
Fine-tuning: Ep [1/10], S [300/938], L: 0.0898
Fine-tuning: Ep [1/10], S [400/938], L: 0.0899
Fine-tuning: Ep [1/10], S [500/938], L: 0.0900
Fine-tuning: Ep [1/10], S [600/938], L: 0.0901
```

```
Fine-tuning: Ep [1/10], S [700/938], L: 0.0897
Fine-tuning: Ep [1/10], S [800/938], L: 0.0901
Fine-tuning: Ep [1/10], S [900/938], L: 0.0900
Fine-tuning: Ep [2/10], S [100/938], L: 0.0902
Fine-tuning: Ep [2/10], S [200/938], L: 0.0900
Fine-tuning: Ep [2/10], S [300/938], L: 0.0896
Fine-tuning: Ep [2/10], S [400/938], L: 0.0899
Fine-tuning: Ep [2/10], S [500/938], L: 0.0900
Fine-tuning: Ep [2/10], S [600/938], L: 0.0898
Fine-tuning: Ep [2/10], S [700/938], L: 0.0900
Fine-tuning: Ep [2/10], S [800/938], L: 0.0895
Fine-tuning: Ep [2/10], S [900/938], L: 0.0902
Fine-tuning: Ep [3/10], S [100/938], L: 0.0900
Fine-tuning: Ep [3/10], S [200/938], L: 0.0899
Fine-tuning: Ep [3/10], S [300/938], L: 0.0899
Fine-tuning: Ep [3/10], S [400/938], L: 0.0898
Fine-tuning: Ep [3/10], S [500/938], L: 0.0902
Fine-tuning: Ep [3/10], S [600/938], L: 0.0900
Fine-tuning: Ep [3/10], S [700/938], L: 0.0899
Fine-tuning: Ep [3/10], S [800/938], L: 0.0902
Fine-tuning: Ep [3/10], S [900/938], L: 0.0902
Fine-tuning: Ep [4/10], S [100/938], L: 0.0899
Fine-tuning: Ep [4/10], S [200/938], L: 0.0898
Fine-tuning: Ep [4/10], S [300/938], L: 0.0900
Fine-tuning: Ep [4/10], S [400/938], L: 0.0899
Fine-tuning: Ep [4/10], S [500/938], L: 0.0900
Fine-tuning: Ep [4/10], S [600/938], L: 0.0898
Fine-tuning: Ep [4/10], S [700/938], L: 0.0899
Fine-tuning: Ep [4/10], S [800/938], L: 0.0899
Fine-tuning: Ep [4/10], S [900/938], L: 0.0899
Fine-tuning: Ep [5/10], S [100/938], L: 0.0901
Fine-tuning: Ep [5/10], S [200/938], L: 0.0901
Fine-tuning: Ep [5/10], S [300/938], L: 0.0899
Fine-tuning: Ep [5/10], S [400/938], L: 0.0902
Fine-tuning: Ep [5/10], S [500/938], L: 0.0899
Fine-tuning: Ep [5/10], S [600/938], L: 0.0898
Fine-tuning: Ep [5/10], S [700/938], L: 0.0899
Fine-tuning: Ep [5/10], S [800/938], L: 0.0900
Fine-tuning: Ep [5/10], S [900/938], L: 0.0902
Fine-tuning: Ep [6/10], S [100/938], L: 0.0901
Fine-tuning: Ep [6/10], S [200/938], L: 0.0900
Fine-tuning: Ep [6/10], S [300/938], L: 0.0904
Fine-tuning: Ep [6/10], S [400/938], L: 0.0901
Fine-tuning: Ep [6/10], S [500/938], L: 0.0898
Fine-tuning: Ep [6/10], S [600/938], L: 0.0900
Fine-tuning: Ep [6/10], S [700/938], L: 0.0901
Fine-tuning: Ep [6/10], S [800/938], L: 0.0898
Fine-tuning: Ep [6/10], S [900/938], L: 0.0899
```

```
Fine-tuning: Ep [7/10], S [100/938], L: 0.0901
Fine-tuning: Ep [7/10], S [200/938], L: 0.0901
Fine-tuning: Ep [7/10], S [300/938], L: 0.0900
Fine-tuning: Ep [7/10], S [400/938], L: 0.0901
Fine-tuning: Ep [7/10], S [500/938], L: 0.0897
Fine-tuning: Ep [7/10], S [600/938], L: 0.0900
Fine-tuning: Ep [7/10], S [700/938], L: 0.0903
Fine-tuning: Ep [7/10], S [800/938], L: 0.0900
Fine-tuning: Ep [7/10], S [900/938], L: 0.0901
Fine-tuning: Ep [8/10], S [100/938], L: 0.0900
Fine-tuning: Ep [8/10], S [200/938], L: 0.0900
Fine-tuning: Ep [8/10], S [300/938], L: 0.0902
Fine-tuning: Ep [8/10], S [400/938], L: 0.0902
Fine-tuning: Ep [8/10], S [500/938], L: 0.0898
Fine-tuning: Ep [8/10], S [600/938], L: 0.0898
Fine-tuning: Ep [8/10], S [700/938], L: 0.0901
Fine-tuning: Ep [8/10], S [800/938], L: 0.0901
Fine-tuning: Ep [8/10], S [900/938], L: 0.0902
Fine-tuning: Ep [9/10], S [100/938], L: 0.0902
Fine-tuning: Ep [9/10], S [200/938], L: 0.0900
Fine-tuning: Ep [9/10], S [300/938], L: 0.0899
Fine-tuning: Ep [9/10], S [400/938], L: 0.0899
Fine-tuning: Ep [9/10], S [500/938], L: 0.0901
Fine-tuning: Ep [9/10], S [600/938], L: 0.0901
Fine-tuning: Ep [9/10], S [700/938], L: 0.0899
Fine-tuning: Ep [9/10], S [800/938], L: 0.0898
Fine-tuning: Ep [9/10], S [900/938], L: 0.0899
Fine-tuning: Ep [10/10], S [100/938], L: 0.0899
Fine-tuning: Ep [10/10], S [200/938], L: 0.0901
Fine-tuning: Ep [10/10], S [300/938], L: 0.0900
Fine-tuning: Ep [10/10], S [400/938], L: 0.0901
Fine-tuning: Ep [10/10], S [500/938], L: 0.0902
Fine-tuning: Ep [10/10], S [600/938], L: 0.0899
Fine-tuning: Ep [10/10], S [700/938], L: 0.0901
Fine-tuning: Ep [10/10], S [800/938], L: 0.0899
Fine-tuning: Ep [10/10], S [900/938], L: 0.0897
```

## 0.7 Part V

[ ]:
```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms

# Define LeNet model
class LeNet(nn.Module):
```

```python
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.fc1 = nn.Linear(16 * 4 * 4, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = torch.nn.functional.relu(self.conv1(x))
        x = torch.nn.functional.max_pool2d(x, kernel_size=2, stride=2)
        x = torch.nn.functional.relu(self.conv2(x))
        x = torch.nn.functional.max_pool2d(x, kernel_size=2, stride=2)
        x = torch.flatten(x, 1)
        x = torch.nn.functional.relu(self.fc1(x))
        x = torch.nn.functional.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# Load MNIST dataset
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=32, shuffle=True)

# Initialize LeNet model
net = LeNet()

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

# Training loop
for epoch in range(5):  # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if i % 1000 == 999:    # print every 1000 mini-batches
```

```
            print('[%d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_loss /
 ↪1000))
            running_loss = 0.0

print('Finished Training')
```

```
[1,  1000] loss: 1.675
[2,  1000] loss: 0.157
[3,  1000] loss: 0.100
[4,  1000] loss: 0.079
[5,  1000] loss: 0.064
Finished Training
```

[ ]:
```
import torch
import torchvision
import torchvision.transforms as transforms

# Load the test dataset
testset = torchvision.datasets.MNIST(root='./data', train=False, download=True,
 ↪transform=transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
]))

testloader = torch.utils.data.DataLoader(testset, batch_size=32, shuffle=False)

# Function to calculate accuracy
def calculate_accuracy(net, dataloader):
    correct = 0
    total = 0
    with torch.no_grad():
        for data in dataloader:
            images, labels = data
            outputs = net(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    return correct / total

# Calculate accuracy on test dataset
accuracy = calculate_accuracy(net, testloader)
print('Accuracy of the network on the test images: {:.2f}%'.format(accuracy *
 ↪100))
```

```
Accuracy of the network on the test images: 98.13%
```