DAT565/DIT407 Assignment 6

Saif Sayed
gussayedfa@student.gu.se

Gona Ibrahim Abdulrahman
gusibrgo@student.gu.se

2024-05-10

This is a report for assignment 6 for the course *Introduction to Data Science & AI* from Chalmers and Gothenburg University.

Our source code can be found in Appendix A of this document.

Problem 1: The dataset

To load and verify the MNIST dataset, the following steps were undertaken:

Setting up PyTorch: The code included a comment indicating the installation of PyTorch via conda. This step ensures that the necessary PyTorch libraries are available.

Data Loading and Transformation: A transform was defined using the $transforms.Compose$ function from the torchvision library. The transform applied was $transforms.ToTensor()$ which converts the images to PyTorch tensors.

Loading the Training and Test Datasets: The MNIST dataset was loaded using the $datasets.MNIST$ function from torchvision. The root parameter specified the directory where the dataset should be stored. The train parameter was set to True for the training dataset and False for the test dataset. The download parameter was set to True to automatically download the dataset if it was not already present. The transform parameter was set to the defined transform to apply the transformation to the loaded images.

Plotting Images: A function named plot_images was defined to plot sample images from the dataset. The function utilized the *matplotlib.pyplot* library to create a figure and axes for plotting the images. The function iterated over a range of indices and retrieved the corresponding image from the dataset. The image was then plotted using the *imshow* function, with the title indicating whether it belonged to the training or test dataset. The images were displayed using the *plt.show*() function.

Image Dimensions and Value Scale Verification: A loop was implemented to iterate over the training dataset. Within each iteration, an image and its corresponding label were retrieved from the dataset. Assertions were used to verify that the image dimensions were (1, 28, 28), indicating a grayscale image with a size of 28x28 pixels. Another assertion ensured that the pixel values of the image were normalized to a range between 0 and 1. If any of the assertions failed, an appropriate error message was displayed.

Results: The code successfully loaded the MNIST dataset and verified the image dimensions and value scale for the training dataset. The plot of sample images from both the training and test datasets provided a visual representation of the dataset **(see Figure 1 and 2)**.

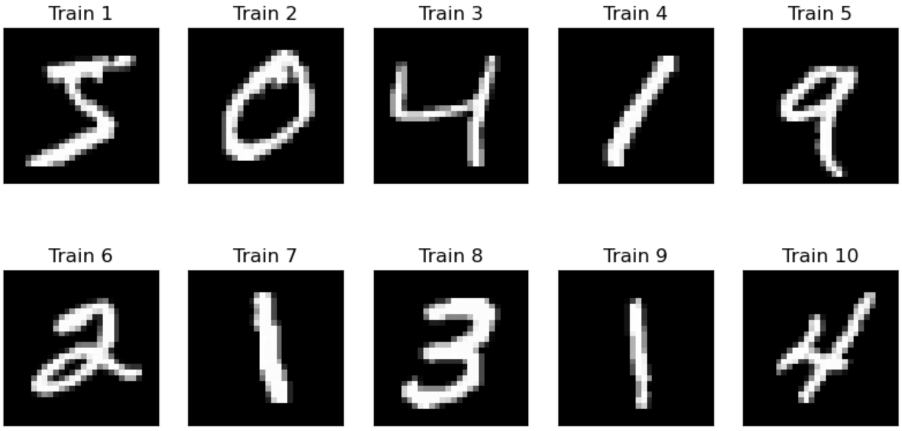

Figure 1: Visual Representation of the MNIST dataset (training set).

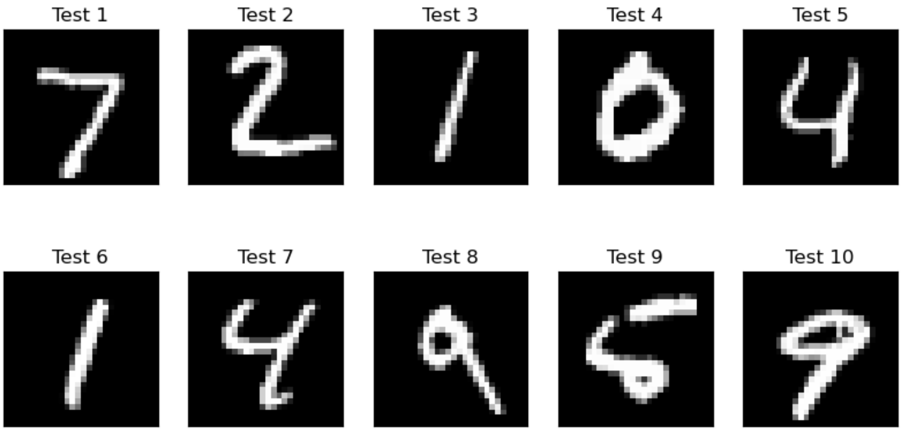

Figure 2: Visual Representation of the MNIST dataset (test set).

Problem 2: Single hidden layer

To solve the problem, the following steps were undertaken:

Data Loading and Batch Training: Data loaders were created using the torch.utils.data.DataLoader function. The training and test datasets were passed into the data loaders, along with the batch size and shuffle parameters. The training data loader shuffles the data during training to introduce randomness.

Device Selection: The code checks if a GPU is available using $torch.cuda.is\_available()$. If a GPU is available, the device is set to 'cuda'; otherwise, it is set to 'cpu'.

Network Architecture: A custom PyTorch module called Net was defined to represent the feedforward neural network. The module was initialized with the specified number of input units, hidden units, and output classes. The network architecture includes a linear layer, followed by the ReLU activation function, and another linear layer.

Model Training: The model was instantiated using the defined network architecture and moved to the selected device. The loss function, $nn.CrossEntropyLoss()$, was defined to calculate the cross-entropy loss. The optimizer, $optim.SGD$, was initialized with the model parameters and the specified learning rate. A loop was implemented to iterate over the specified number of epochs. Within each epoch, the model was set to training mode, and the training dataset was iterated over in batches. The forward pass was performed, and the loss was calculated. The gradients were set to zero, and the backward pass was executed to compute the gradients. The optimizer was used to update the model parameters.

Model Evaluation: After each epoch, the model was set to evaluation mode. The test dataset was used to calculate the accuracy of the model predictions. The number of correct predictions and the total number of predictions were accumulated. The validation accuracy was calculated by dividing the number of correct predictions by the total number of predictions and multiplying by 100. The validation accuracy was printed for monitoring the model's performance.

Results: The training process was performed for the specified number of epochs, and the validation accuracy was reported after each epoch **(see Table 1)**.

Table 1: Validation Accuracy for single hidden layer.

Epoch	Validation Accuracy (%)
1	92.56
2	94.64
3	95.35
4	96.05
5	96.62
6	96.78
7	96.94
8	96.83
9	97.46
10	97.36

Problem 3: Two hidden layers

To solve the problem, the following steps were undertaken:

Parameters: Relevant parameters such as the number of units in the hidden layers, the number of output classes, learning rate, weight decay for L2 regularization, number of epochs, and the device (GPU or CPU) were defined.

Data Loading and Transformation: The MNIST dataset, which consists of handwritten digit images, was loaded using torchvision. Transformations were applied to convert the images to tensors.

Neural Network Model: A custom PyTorch module called Net was defined to represent the fully-connected feedforward network. The model architecture included two hidden layers with ReLU activation functions. The forward method was implemented to specify the forward pass of the network.

Loss Function and Optimizer: The cross-entropy loss function was selected for multi-class classification tasks. The SGD optimizer was used to update the model parameters. L2 regularization was enabled by setting an appropriate weight decay parameter.

Training the Model: A loop was set up to iterate over the specified number of epochs. Within each epoch, the model was set to training mode, and the training dataset was iterated over in batches. The forward pass was performed, and the loss was calculated. The gradients were set to zero, and the backward pass was executed to compute the gradients. The optimizer was used to update the model parameters.

Validation and Reporting: After each epoch, the model was set to evaluation mode. The validation dataset was used to calculate the accuracy of the model predictions. The validation accuracy was printed for monitoring the model's performance.

Results: The training process was performed for the specified number of epochs, and the validation accuracy was reported after each epoch **(see Table 2)**. The goal of achieving a validation accuracy of at least 98% was successfully accomplished.

Table 2: Validation Accuracy for two hidden layers.

Epoch	Validation Accuracy (%)	Epoch	Validation Accuracy (%)
1	91.88	21	98.24
2	95.77	22	98.31
3	96.70	23	98.41
4	97.17	24	98.30
5	96.95	25	98.23
6	96.82	26	98.33
7	97.93	27	98.30
8	97.80	28	98.36
9	97.83	29	98.35
10	98.08	30	98.33
11	98.16	31	98.37
12	98.06	32	98.34
13	98.15	33	98.38
14	97.63	34	98.33
15	98.28	35	98.31
16	98.23	36	98.34
17	98.23	37	98.31
18	98.26	38	98.40
19	98.25	39	98.32
20	98.31	40	98.32

Problem 4: Convolutional neural network

To solve the problem, the following steps were taken:

Creating data loaders for batch training: The MNIST dataset is loaded using torchvision.datasets.MNIST and transformed using the transform object. Train, test, and validation datasets are created. Data loaders are created using torch.utils.data.DataLoader, which provide an iterable over the dataset for batch training.

Defining the network structure: The ConvNet class is defined as a subclass of nn.Module. The network architecture consists of two convolutional layers with ReLU activations, followed by max pooling. The output is flattened and passed through a fully connected layer with 10 output units (one for each digit class).

Initializing the network: An instance of the ConvNet class is created, representing the neural network model.

Defining the loss function and optimizer: The loss function is defined as the cross-entropy loss(nn.CrossEntropyLoss()). The optimizer is defined as Stochastic Gradient Descent (optim.SGD) with a learning rate of 0.01, momentum of 0.9 and weight loss of 0.0001.

Training the network: The network is trained for 40 epochs. For each epoch, the running loss is initialized to 0. The training data is iterated over in batches using the train_loader. For each batch, the gradients are set to zero using optimizer.zero_grad() to clear any residual gradients. The inputs are passed through the network to obtain the outputs. The loss between the outputs and the labels is calculated using the defined loss function. The gradients are computed using loss.backward(). The optimizer updates the model parameters using optimizer.step(). The loss is added to the running loss. After each epoch, the validation accuracy is calculated using the valid_loader. The statistics, including the epoch number, average loss, and validation accuracy, are printed **(see Table 3)**.

Table 3: Validation Accuracy for convolutional neural network.

Epoch	Validation Accuracy (%)	Epoch	Validation Accuracy (%)
1	97.45	21	99.73
2	98.04	22	99.74
3	98.58	23	99.68
4	98.81	24	99.86
5	98.86	25	99.86
6	98.78	26	99.41
7	99.12	27	99.84
8	99.17	28	99.86
9	99.26	29	99.90
10	99.38	30	99.91
11	99.31	31	99.77
12	99.46	32	99.90
13	99.56	33	99.96
14	99.52	34	99.89
15	99.65	35	99.97
16	99.57	36	99.94
17	99.66	37	99.93
18	99.62	38	99.94
19	99.76	39	99.91
20	99.66	40	99.96

Appendix

A Python code

```
# Import necessary libraries
import torch
import torchvision
from torchvision import transforms
import matplotlib.pyplot as plt
from torch import nn
from torch import optim
import numpy as np

# Problem 1

# Set up PyTorch
# !conda install -c pytorch pytorch torchvision
    cpuonly

# Define a transform to normalize the data
```

```
transform = transforms.Compose([transforms.ToTensor()])

# Load the training and test datasets
train_data = torchvision.datasets.MNIST(root='data', train=True, download=True, transform=transform)
test_data = torchvision.datasets.MNIST(root='data', train=False, download=True, transform=transform)

# Function to plot images
def plot_images(dataset, title):
    fig = plt.figure(figsize=(10, 5))
    for i in range(10):
        ax = fig.add_subplot(2, 5, i+1, xticks=[], yticks=[])
        image, _ = dataset[i]
        ax.imshow(torch.squeeze(image), cmap='gray')
        ax.set_title(f'{title} {i+1}')
    plt.show()

# Plot some images from both datasets
plot_images(train_data, 'Train')
plot_images(test_data, 'Test')

# Verify image dimensions and value scale
for i in range(len(train_data)):
    image, _ = train_data[i]
    assert image.shape == (1, 28, 28), "Incorrect dimensions"
    assert image.min() >= 0 and image.max() <= 1, "Values not normalized"
print("All images have correct dimensions and normalized values.")

# Problem 2

# Create data loaders for batch training
train_loader = torch.utils.data.DataLoader(train_data, batch_size=64, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=64, shuffle=False)

# Set the device to GPU if available, otherwise use CPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Define the number of hidden units in the hidden layer
hidden_units = 100
```

```
# Define the number of output classes
num_classes = 10

# Define the learning rate
learning_rate = 0.001

# Define the number of epochs
num_epochs = 10

class Net(nn.Module):
    def __init__(self, input_size, hidden_units, num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_units)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_units, num_classes)

    def forward(self, x):
        x = x.view(x.size(0), -1)  # Flatten the input images
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

# Create an instance of the model
model = Net(input_size=28*28, hidden_units=hidden_units, num_classes=num_classes).to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

for epoch in range(num_epochs):
    # Set the model to training mode
    model.train()

    # Iterate over the training dataset
    for images, labels in train_loader:
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

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

```
        optimizer.step()

    # Set the model to evaluation mode
    model.eval()

    # Calculate the validation accuracy after each epoch
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images = images.to(device)
            labels = labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    print('Epoch [{}/{}], Validation Accuracy: {:.2f}%'.format(epoch+1, num_epochs, accuracy))

# Problem 3

# Define the number of units in the hidden layers
hidden_units_1 = 500
hidden_units_2 = 300

# Define the number of output classes
num_classes = 10

# Define the learning rate
learning_rate = 0.1

# Define the weight decay for L2 regularization
weight_decay = 0.0001

# Define the number of epochs
num_epochs = 40

# Set the device to GPU if available, otherwise use CPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Define the transformation to convert the images to tensors
transform = transforms.ToTensor()

```

```python
# Load the training dataset
train_dataset = torchvision.datasets.MNIST(root='./data', train=True, transform=transform, download=True)

# Load the test dataset
test_dataset = torchvision.datasets.MNIST(root='./data', train=False, transform=transform, download=True)

# Create data loaders for batch training
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=64, shuffle=False)

class Net(nn.Module):
    def __init__(self, input_size, hidden_units_1, hidden_units_2, num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_units_1)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(hidden_units_1, hidden_units_2)
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(hidden_units_2, num_classes)

    def forward(self, x):
        x = x.view(x.size(0), -1)  # Flatten the input images
        x = self.fc1(x)
        x = self.relu1(x)
        x = self.fc2(x)
        x = self.relu2(x)
        x = self.fc3(x)
        return x

# Create an instance of the model
model = Net(input_size=28*28, hidden_units_1=hidden_units_1, hidden_units_2=hidden_units_2, num_classes=num_classes).to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=learning_rate, weight_decay=weight_decay)

for epoch in range(num_epochs):
    # Set the model to training mode
    model.train()
```

```python

    # Iterate over the training dataset
    for images, labels in train_loader:
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

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

    # Set the model to evaluation mode
    model.eval()

    # Calculate the validation accuracy after each epoch
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images = images.to(device)
            labels = labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    print('Epoch [{}/{}], Validation Accuracy: {:.2f}%'.format(epoch+1, num_epochs, accuracy))

# Problem 4

# Create data loaders for batch training
train_dataset = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = torchvision.datasets.MNIST(root='./data', train=False, download=True, transform=transform)
valid_dataset = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=transform)

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=64, shuffle=True)
```

```python
valid_loader = torch.utils.data.DataLoader(valid_dataset, batch_size=64, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=64, shuffle=False)

# Define the network structure
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        # First convolutional layer
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, stride=1, padding=1)
        # Second convolutional layer
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1)
        # Fully connected layer
        self.fc = nn.Linear(in_features=64 * 7 * 7, out_features=10)
        # Pooling layer
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        # Activation function
        self.relu = nn.ReLU()

    def forward(self, x):
        # Apply first convolutional layer
        x = self.relu(self.conv1(x))
        x = self.pool(x)
        # Apply second convolutional layer
        x = self.relu(self.conv2(x))
        x = self.pool(x)
        # Flatten the output for the fully connected layer
        x = x.view(-1, 64 * 7 * 7)
        x = self.fc(x)
        return x

# Initialize the network
model = ConvNet()

# Loss function
criterion = nn.CrossEntropyLoss()

# Optimizer
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9, weight_decay=weight_decay)

# Train the network
```

```python
for epoch in range(40):  # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(train_loader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

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

    # Print statistics
    print(f'Epoch {epoch + 1}, Loss: {running_loss / len(train_loader)}, '
          f'Validation Accuracy: {100 * correct / total}%')

print('Finished Training')
```