Assignment-1-C

April 20, 2025

1 Assignment 1 C

1.0.1 Import necessary modules

[1]:
```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
```

1.0.2 Layer class

[2]:
```
class Layer:
    def __init__(self, neurons_size: int, inputs_size: int, activation: str =␣
 ↪'sigmoid',
                 leaky_slope: float = 0.01, softmax_dim: int = 0):
        '''
        Initialize the layer with random weights and biases

        Parameters:
            neurons_size: int
                The number of neurons in the layer
            inputs_size: int
                The number of inputs to the layer
            activation: str
                The activation function to use
            leaky_slope: float
                The slope of the leaky relu activation function
            softmax_dim: int
                The dimension of the softmax activation function
        '''

        # initialize weights and biases with random values
        # use the He initialization method to initialize the weights if the␣
 ↪activation function is relu or leaky relu else use the Xavier initialization␣
 ↪method
        if activation == 'relu' or activation == 'leaky_relu':
```

```python
            self._weights: torch.Tensor = torch.randn(
                neurons_size, inputs_size) * torch.sqrt(2 / torch.
↪tensor(inputs_size, dtype=torch.float32))
        else:
            self._weights: torch.Tensor = torch.randn(
                neurons_size, inputs_size) * torch.sqrt(1 / torch.
↪tensor(inputs_size, dtype=torch.float32))
        self._biases: torch.Tensor = torch.zeros(neurons_size)
        self._activation: str = activation
        self._leaky_slope: float = leaky_slope
        self._softmax_dim: int = softmax_dim

        self._activation_function: dict[str, callable] = {
            'sigmoid': nn.Sigmoid(),
            'tanh': nn.Tanh(),
            'relu': nn.ReLU(),
            'leaky_relu': nn.LeakyReLU(negative_slope=self._leaky_slope),
            'softmax': nn.Softmax(dim=self._softmax_dim)
        }

    def set_weights(self, weights: torch.Tensor):
        '''
        Set the weights of the layer

        Parameters:
            weights: torch.Tensor
                The weights to set
        '''
        self._weights = weights

    def get_weights(self) -> torch.Tensor:
        '''
        Get the weights of the layer

        Returns:
            torch.Tensor
                The weights of the layer
        '''
        return self._weights

    def set_biases(self, biases: torch.Tensor):
        '''
        Set the biases of the layer

        Parameters:
            biases: torch.Tensor
                The biases to set
```

```
        '''
        self._biases = biases

    def get_biases(self) -> torch.Tensor:
        '''
        Get the biases of the layer

        Returns:
            torch.Tensor
                The biases of the layer
        '''
        return self._biases

    def forward(self, inputs: torch.Tensor) -> torch.Tensor:
        '''
        Forward pass

        Parameters:
            inputs: torch.Tensor
                The inputs to the layer
            activation: str
                The activation function to use
        Returns:
            torch.Tensor
                The outputs of the layer
        '''
        # calculate the sum of the inputs multiplied by the weights and add the␣
↪biases
        # sum: torch.Tensor = torch.matmul(self._weights, inputs) + self._biases
        sum: torch.Tensor = self._weights @ inputs + self._biases

        if self._activation in self._activation_function:
            return self._activation_function[self._activation](sum)
        else:
            raise ValueError(
                f"Activation function {self._activation} not found")
```

1.0.3 Custom Layer class

[3]:
```
# custom layer class to use the Layer class as a custom layer in PyTorch
class CustomLayer(nn.Module):
    def __init__(self, neurons_size: int, inputs_size: int, activation: str =␣
↪'sigmoid',
                 leaky_slope: float = 0.01, softmax_dim: int = 0):
        super(CustomLayer, self).__init__()

        '''
```

```
        Initialize the custom layer with the given neurons size, inputs size,
        activation, leaky slope, and softmax dimension

        '''

        # initialize the custom layer
        self.layer = Layer(neurons_size, inputs_size,
                           activation, leaky_slope, softmax_dim)

        # register the weights and biases as parameters to PyTorch
        self.weights = nn.Parameter(self.layer.get_weights())
        self.biases = nn.Parameter(self.layer.get_biases())

        # point the layer parameters to the PyTorch parameters
        self.layer.set_weights(self.weights)
        self.layer.set_biases(self.biases)

        # store the activation function parameters
        self.activation = activation
        self.leaky_slope = leaky_slope
        self.softmax_dim = softmax_dim

    def forward(self, inputs: torch.Tensor) -> torch.Tensor:

        # use custom layer forward method
        return self.layer.forward(inputs)
```

1.0.4 Perceptron class

[4]:
```
class Perceptron(nn.Module):
    def __init__(self, input_size: int, hidden_size: int, output_size: int,␣
 ↪hidden_activation: str = 'relu',
                 output_activation: str = 'softmax', leaky_slope: float = 0.01,␣
 ↪softmax_dim: int = 0):
        super(Perceptron, self).__init__()

        '''
        Initialize the perceptron with the given input size, hidden size,␣
 ↪output size,
        hidden activation, output activation, leaky slope, and softmax dimension

        Parameters:
            input_size: int
                The size of the input layer
            hidden_size: int
                The size of the hidden layer
            output_size: int
```

```
                The size of the output layer
            hidden_activation: str
                The activation function to use for the hidden layer
            output_activation: str
                The activation function to use for the output layer
            leaky_slope: float
                The slope of the leaky relu activation function
            softmax_dim: int
                The dimension of the softmax activation function
        '''

        # create a list of layer sizes including the input and output sizes
        layer_sizes: list[int] = [input_size] + hidden_size + [output_size]
        layers: list[CustomLayer] = []

        # create hidden layers
        for i in range(len(layer_sizes) - 2):
            layers.append(CustomLayer(
                layer_sizes[i+1], layer_sizes[i], hidden_activation, 
↪leaky_slope, softmax_dim))

        # create output layer
        layers.append(CustomLayer(
            layer_sizes[-1], layer_sizes[-2], output_activation, leaky_slope, 
↪softmax_dim))

        # store the layers
        self.layers = nn.ModuleList(layers)

    def forward(self, inputs: torch.Tensor) -> torch.Tensor:

        # get the dimension of the input
        input_dim: int = inputs.dim()

        # if the input is a single sample, unsqueeze it
        if input_dim == 1:
            inputs = inputs.unsqueeze(0)

        # if the input is a batch of samples, process each sample individually
        batch_output: list[torch.Tensor] = []

        # process each sample in the batch
        for sample in inputs:

            # process the sample through the layers
            for layer in self.layers:
                sample = layer(sample)
```

```
            # store the output of the sample
            batch_output.append(sample)

        # stack the outputs of the samples into a single tensor
        return torch.stack(batch_output, dim=0).squeeze(0)
```

1.0.5 Test Training with graphs

[5]:
```
# set the random seed for reproducibility
torch.manual_seed(42)

# use cpu for training
device = torch.device('cpu')

# define the transformations to apply to the dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# load the MNIST dataset
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
)

# create the data loaders
train_loader = DataLoader(
    train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(
    test_dataset, batch_size=64, shuffle=False)

# create the model
input_size = 28 * 28  # MNIST images are 28x28 pixels
hidden_size = [128]  # one hidden layer
output_size = 10  # 10 classes (digits 0-9)
model = Perceptron(
```

```python
    input_size=input_size,
    hidden_size=hidden_size,
    output_size=output_size,
    hidden_activation='relu',
    output_activation='softmax',
    softmax_dim=0
).to(device)

# define the optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# lists to store metrics for visualization
train_losses = []
test_losses = []
test_accuracies = []

# training loop
num_epochs = 10
print("Starting training...")
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0

    for i, (images, labels) in enumerate(train_loader):
        # flatten the images
        images = images.view(-1, input_size).to(device)
        labels = labels.to(device)

        # forward propagation
        outputs = model.forward(images)
        loss = criterion(outputs, labels)

        # back propagation and optimization step
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # calculate the running loss for the training set
        running_loss += loss.detach().item()

        # print the loss for the training set every 100 steps
        if (i+1) % 100 == 0:
            print(
                f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/
↪{len(train_loader)}], Loss: {loss.detach().item():.4f}')
```

```
    # calculate the average loss for the training set
    epoch_loss = running_loss / len(train_loader)
    train_losses.append(epoch_loss)
    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {epoch_loss:.4f}\n')

    # evaluate on test set after each epoch (without printing during training)
    model.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        test_loss = 0.0

        for images, labels in test_loader:
            images = images.view(-1, input_size).to(device)
            labels = labels.to(device)
            outputs = model.forward(images)

            loss = criterion(outputs, labels)
            test_loss += loss.item()

            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        avg_test_loss = test_loss / len(test_loader)
        test_accuracy = 100 * correct / total

        # Store metrics for visualization
        test_losses.append(avg_test_loss)
        test_accuracies.append(test_accuracy)

print("Training complete!")

# evaluate the model
model.eval()
with torch.no_grad():
    correct = 0  # correct predictions
    total = 0  # total samples
    test_loss = 0.0  # running loss

    # process each sample in the test set
    for images, labels in test_loader:
        images = images.view(-1, input_size).to(device)
        labels = labels.to(device)
        outputs = model.forward(images)

        # calculate the loss for the test set
```

```
        loss = criterion(outputs, labels)
        test_loss += loss.item()

        # calculate the accuracy for the test set
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

# calculate the average loss for the test set
final_test_loss = test_loss / len(test_loader)
print(f'Test Loss: {final_test_loss:.4f}')

# calculate the accuracy for the test set
final_accuracy = 100 * correct / total
print(f'Test Accuracy: {final_accuracy:.2f}%')
print('')

# now create the visualization graphs
plt.figure(figsize=(12, 10))

# plot training loss
plt.subplot(2, 1, 1)
plt.plot(range(1, num_epochs + 1), train_losses, label='Training Loss',␣
 ↪marker='o', color='blue')
plt.plot(range(1, num_epochs + 1), test_losses, label='Test Loss', marker='s',␣
 ↪color='red')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Test Loss')
plt.legend()
plt.grid(True)

# plot test accuracy
plt.subplot(2, 1, 2)
plt.plot(range(1, num_epochs + 1), test_accuracies, label='Test Accuracy',␣
 ↪marker='o', color='green')
plt.xlabel('Epochs')
plt.ylabel('Accuracy (%)')
plt.title('Test Accuracy')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```

```
Starting training…
Epoch [1/10], Step [100/938], Loss: 1.6681
```

```
Epoch [1/10], Step [200/938], Loss: 1.6458
Epoch [1/10], Step [300/938], Loss: 1.6796
Epoch [1/10], Step [400/938], Loss: 1.5789
Epoch [1/10], Step [500/938], Loss: 1.6485
Epoch [1/10], Step [600/938], Loss: 1.5806
Epoch [1/10], Step [700/938], Loss: 1.5931
Epoch [1/10], Step [800/938], Loss: 1.5502
Epoch [1/10], Step [900/938], Loss: 1.5129
Epoch [1/10], Loss: 1.6436

Epoch [2/10], Step [100/938], Loss: 1.6527
Epoch [2/10], Step [200/938], Loss: 1.5527
Epoch [2/10], Step [300/938], Loss: 1.5786
Epoch [2/10], Step [400/938], Loss: 1.5990
Epoch [2/10], Step [500/938], Loss: 1.5768
Epoch [2/10], Step [600/938], Loss: 1.5422
Epoch [2/10], Step [700/938], Loss: 1.5262
Epoch [2/10], Step [800/938], Loss: 1.5072
Epoch [2/10], Step [900/938], Loss: 1.5341
Epoch [2/10], Loss: 1.5429

Epoch [3/10], Step [100/938], Loss: 1.5515
Epoch [3/10], Step [200/938], Loss: 1.5376
Epoch [3/10], Step [300/938], Loss: 1.5240
Epoch [3/10], Step [400/938], Loss: 1.5739
Epoch [3/10], Step [500/938], Loss: 1.5059
Epoch [3/10], Step [600/938], Loss: 1.5236
Epoch [3/10], Step [700/938], Loss: 1.5505
Epoch [3/10], Step [800/938], Loss: 1.5458
Epoch [3/10], Step [900/938], Loss: 1.4739
Epoch [3/10], Loss: 1.5257

Epoch [4/10], Step [100/938], Loss: 1.5638
Epoch [4/10], Step [200/938], Loss: 1.5299
Epoch [4/10], Step [300/938], Loss: 1.5604
Epoch [4/10], Step [400/938], Loss: 1.4964
Epoch [4/10], Step [500/938], Loss: 1.5160
Epoch [4/10], Step [600/938], Loss: 1.5395
Epoch [4/10], Step [700/938], Loss: 1.4756
Epoch [4/10], Step [800/938], Loss: 1.5394
Epoch [4/10], Step [900/938], Loss: 1.4798
Epoch [4/10], Loss: 1.5162

Epoch [5/10], Step [100/938], Loss: 1.4801
Epoch [5/10], Step [200/938], Loss: 1.5192
Epoch [5/10], Step [300/938], Loss: 1.4835
Epoch [5/10], Step [400/938], Loss: 1.5304
Epoch [5/10], Step [500/938], Loss: 1.4768
```

```
Epoch [5/10], Step [600/938], Loss: 1.5039
Epoch [5/10], Step [700/938], Loss: 1.4835
Epoch [5/10], Step [800/938], Loss: 1.4859
Epoch [5/10], Step [900/938], Loss: 1.5083
Epoch [5/10], Loss: 1.5087

Epoch [6/10], Step [100/938], Loss: 1.4709
Epoch [6/10], Step [200/938], Loss: 1.4802
Epoch [6/10], Step [300/938], Loss: 1.4773
Epoch [6/10], Step [400/938], Loss: 1.5314
Epoch [6/10], Step [500/938], Loss: 1.4881
Epoch [6/10], Step [600/938], Loss: 1.4932
Epoch [6/10], Step [700/938], Loss: 1.4990
Epoch [6/10], Step [800/938], Loss: 1.5216
Epoch [6/10], Step [900/938], Loss: 1.4829
Epoch [6/10], Loss: 1.5031

Epoch [7/10], Step [100/938], Loss: 1.5222
Epoch [7/10], Step [200/938], Loss: 1.5041
Epoch [7/10], Step [300/938], Loss: 1.5004
Epoch [7/10], Step [400/938], Loss: 1.5026
Epoch [7/10], Step [500/938], Loss: 1.5287
Epoch [7/10], Step [600/938], Loss: 1.4896
Epoch [7/10], Step [700/938], Loss: 1.5044
Epoch [7/10], Step [800/938], Loss: 1.4919
Epoch [7/10], Step [900/938], Loss: 1.4779
Epoch [7/10], Loss: 1.4986

Epoch [8/10], Step [100/938], Loss: 1.5097
Epoch [8/10], Step [200/938], Loss: 1.4773
Epoch [8/10], Step [300/938], Loss: 1.4935
Epoch [8/10], Step [400/938], Loss: 1.4673
Epoch [8/10], Step [500/938], Loss: 1.5089
Epoch [8/10], Step [600/938], Loss: 1.5228
Epoch [8/10], Step [700/938], Loss: 1.4956
Epoch [8/10], Step [800/938], Loss: 1.4797
Epoch [8/10], Step [900/938], Loss: 1.5282
Epoch [8/10], Loss: 1.4962

Epoch [9/10], Step [100/938], Loss: 1.5104
Epoch [9/10], Step [200/938], Loss: 1.5063
Epoch [9/10], Step [300/938], Loss: 1.4763
Epoch [9/10], Step [400/938], Loss: 1.4828
Epoch [9/10], Step [500/938], Loss: 1.4811
Epoch [9/10], Step [600/938], Loss: 1.4766
Epoch [9/10], Step [700/938], Loss: 1.5493
Epoch [9/10], Step [800/938], Loss: 1.4619
Epoch [9/10], Step [900/938], Loss: 1.5280
```

```
Epoch [9/10], Loss: 1.4942

Epoch [10/10], Step [100/938], Loss: 1.4769
Epoch [10/10], Step [200/938], Loss: 1.4857
Epoch [10/10], Step [300/938], Loss: 1.4649
Epoch [10/10], Step [400/938], Loss: 1.5376
Epoch [10/10], Step [500/938], Loss: 1.4945
Epoch [10/10], Step [600/938], Loss: 1.4634
Epoch [10/10], Step [700/938], Loss: 1.4923
Epoch [10/10], Step [800/938], Loss: 1.4620
Epoch [10/10], Step [900/938], Loss: 1.4647
Epoch [10/10], Loss: 1.4916

Training complete!
Test Loss: 1.4933
Test Accuracy: 96.94%
```