Mathematical Foundation of DNN : HW 1

Jeong Min Lee

March 28, 2024

Notations

1. $\otimes$: convolution
2. $v^{(i)}$: ith element of vector $v \in \mathbb{R}^n$
3. $A^{(i,j)}$: the element of matrix $A \in \mathbb{R}^{m \times n}$ in ith row and jth column
4. δ_{ij} : Kronecker delta symbol.

1

Let $\omega^{(1)}, \omega^{(2)} \in \mathbb{R}^{3\times3}$ denote the map from X to Y_1, Y_2, respectively. Then, $\omega \in \mathbb{R}^{(2\times3\times3)} = [\omega^{(1)}, \omega^{(2)}]$. Noting that the process of discrete convolution and being aware of where the filter maps each image pixel X_{ij}, $\omega^{(1)}, \omega^{(2)}$ can be represented as follow.

$$\omega^{(1)} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 1 & 0 \end{pmatrix} \tag{1}$$

$$\omega^{(2)} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & -1 & 1 \\ 0 & 0 & 0 \end{pmatrix} \tag{2}$$

For instance,

$$\begin{pmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 1 & 0 \end{pmatrix} = X_{32} - X_{22} = Y_{22}$$

2

We can formalized this problem as following example.

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \otimes \begin{pmatrix} & & \\ & ? & \\ & & \end{pmatrix} = \frac{1}{9}(a+b+c+d+e+f+g+h+i)$$

This is obvious that I_k/k^2 satisfy the condition above, where I_k is k dimensional identity matrix. Note that we have to divide all summation of subimage since there are k^2 pixels that overlap with the filter whose size is k.

3

This problem is obvious that $\omega = [0.299, 0.587, 0.114] \in \mathbb{R}^{3\times1\times1}$. As the given expression $Y_{ij} = 0.299X_{1,i,j} + 0.587X_{2,i,j} + 0.114X_{3,i,j}$ says, 1×1 convolution is a type of constant multiplicatioon for each channel. The reason we use 1×1 convolution can be elucidated by seeing the following example: Consider the 192 gray scale images $X \in \mathbb{R}^{192\times28\times28}$. Doing appropriate padding to conserve the output dimension, using a filter whose size is 5 and channel is 32 returns $32 \times 28 \times 28$ activation maps. The number of float multiplication calculation in this case is $28 \times 28 \times 32 \times 5 \times 192 = 120$M. However, inserting 1d convolution whose depth is 16 results in the activation map whose dimension is $16 \times 28 \times 28$.(Again, I assumed appropriate padding to preserve the dimension.) Then, applying $32 \times 5 \times 5$ makes the output activation map have $32 \times 28 \times 28$ dimension. (Note that this secondary filter is identical to the filter in the first case.) For second case, the number of computations is $28 \times 28 \times 1 \times 1 \times 192 + 28 \times 28 \times 32 \times 5 \times 5 \times 16 = 12.4$M, which is 10 times smaller than the first case. This example implies that inserting 1d convolution between the filter convolving the image reduces the time complexity of the model and can improve the model's performance.

4

CLAIM : σ can commute to max

proof : Consider $S = \{a_1, a_2, \cdots, a_n\} \in \mathbb{R}^n$. Withoug loss of generality, suppose $\max S = a_1$. Then, $\sigma(\max S) = \sigma(a_1)$. However, noting that σ is non-decreasing function, $\max\{\sigma(a_1), \cdots, \sigma(a_n)\} = \sigma(a_1)$. Otherwise, there is some $a_i \in S \setminus \{a_1\}$ s.t. $\sigma(a_i) > \sigma(a_1)$. However, since $a_i \le a_1$, it contradicts to the non-decreasing assumption of σ. Thus, σ and max commute each other.$\blacksquare$

Let $\mathfrak{M}_{m,n}(F)$ be the set of $m \times n$ matrices over some field F. Then, in alegebra, it is well known fact that $\mathfrak{M}_{m,n}(F)$ is isomorphic to F^{mn}.[1] Letting $F = \mathbb{R}$, one can apply the **CLAIM** to the $X \in \mathfrak{M}_{m,n}(\mathbb{R})$. Since ρ maps from $\mathbb{R}^{m\times n}$ to $\mathbb{R}^{k\times l}$, the filter size that ρ uses is $(m/k, n/l)$. Considering the feature of max pooling, $\rho = \max$ for the input whose size is identical to the filter size. Let's denote

$$X_{i:i+m/k-1,j:j+n/l-1} = \begin{pmatrix} X_{ij} & \cdots & X_{i,j+n/l-1} \\ \vdots & \ddots & \vdots \\ X_{i+m/k-1,j} & \cdots & X_{i+m/k-1,j+n/l-1} \end{pmatrix}$$

and $y_{ij} = \max X_{i:i+m/k-1,j:j+n/l-1}$. Then, $\rho(X) = (y_{ij})$.

$$\begin{aligned}
\sigma(\rho(X_{i:i+m/k-1,j:j+n/l-1})) &= \sigma\left(\max\begin{pmatrix} X_{ij} & \cdots & X_{i,j+n/l-1} \\ \vdots & \ddots & \vdots \\ X_{i+m/k-1,j} & \cdots & X_{i+m/k-1,j+n/l-1} \end{pmatrix}\right) \\
&= \max\left(\begin{pmatrix} \sigma(X_{ij}) & \cdots & \sigma(X_{i,j+n/l-1}) \\ \vdots & \ddots & \vdots \\ \sigma(X_{i+m/k-1,j}) & \cdots & \sigma(X_{i+m/k-1,j+n/l-1}) \end{pmatrix}\right) \\
&= \rho(\sigma(X_{i:i+m/k-1,j:j+n/l-1})) \\
&= \sigma(y_{ij})
\end{aligned}$$

This implies that σ and ρ can commute.

5

To solve this problem, I refered the Chapter 2 code in the lecture. Please see appendix to get my implementations. Comparing the test result of two implementations, using KL-divergence and MSE loss, it was revealed that their performances are similar. To elucidate the reason using the KL-divergence rather than MSE in several vision tasks, I performed multiple times of training and testing. Furthermore, I recorded the time consumed by training and testing. See the table below.

	D_{KL}		MSE	
	Accuracy(%)	Time(s)	Accuracy(%)	Time(s)
1	92.16	2.8	95.58	6.4
2	96.63	2.8	95.38	6.4
3	95.98	3.0	95.28	6.3
4	96.08	2.8	94.98	6.5
5	96.84	2.8	88.15	6.4
average	95.538	2.84	93.874	6.4
std	1.71974882	0.08	2.86854388	0.06324555

Table 1: The performace test of MSE loss and KL-divergence loss(D_{KL})

As Table 1 says, the overal accuary of D_{KL} is slightly higher than MSE(95.538% > 93.874%). Furthermore, as the standard deviation of accuracies says, D_{KL} is more stable than MSE(1.720 < 2.869). Finally, the training and testing time of D_{KL} is much less than that of MSE. I thinks that this results from the computation of square in MSE. In general, the computation of power in floating numbers is slow in python. Considering that accuracy and computation time complexity are crucial aspects for the DL algorithms, implementation using D_{KL} loss is more choosable.

[1] This is corollary of the following theorem : $V \approx W \iff \dim V = \dim W$ for two vector spaces V, W.

6

(a)

Note that $y_L, b_L \in \mathbb{R}$ and $A_L \in \mathbb{R}^{1\times n_{L-1}}$

(i)

$$\frac{\partial y_L}{\partial b_L} = \frac{\partial b_L}{\partial b_L} = 1$$

(ii)

From $A_L y_{L-1} = A_L^{(1)} y_{L-1}^{(1)} + \cdots + A_L^{(n_{L-1})} y_{L-1}^{(n_{L-1})}$

$$\begin{aligned}
\frac{\partial y_L}{\partial y_{L-1}} &= \frac{\partial A_L y_{L-1}}{y_{L-1}} \\
&= \left(\frac{\partial A_L y_{L-1}}{\partial y_{L-1}^{(1)}}, \cdots, \frac{\partial A_L y_{L-1}}{\partial y_{L-1}^{(n_{L-1})}} \right) \\
&= \left(A_L^{(1)}, \cdots, A_L^{(n_{L-1})} \right) \\
&= A_L
\end{aligned}$$

(iii)

Let $z_l = A_l y_{l-1} + b_l$.Then,

$$\begin{aligned}
\frac{\partial y_l}{\partial b_l} &= \frac{\partial \sigma(z_l)}{\partial b_l} \\
&= \begin{pmatrix} \frac{\partial \sigma(z_l^{(1)})}{\partial b_l^{(1)}} & \cdots & \frac{\partial \sigma(z_l^{(1)})}{\partial b_l^{(n_l)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \sigma(z_l^{(n_l)})}{\partial b_l^{(1)}} & \cdots & \frac{\partial \sigma(z_l^{(n_l)})}{\partial b_l^{(n_l)}} \end{pmatrix}
\end{aligned}$$

$$\begin{aligned}
\frac{\partial \sigma(z_l^{(i)})}{\partial b_l^{(j)}} &= \sigma'(z_l^{(i)}) \frac{\partial z_l^{(i)}}{\partial b_l^{(j)}} \\
&= \sigma'(z_l^{(i)}) \delta_{ij}
\end{aligned}$$

$$\therefore \frac{\partial y_l}{\partial b_l} = \begin{pmatrix} \sigma'(z_l^{(1)}) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma'(z_l^{(n_l)}) \end{pmatrix} = \mathrm{diag}(\sigma'(A_l y_l + b_l))$$

(iv)

$$\begin{aligned}
\frac{\partial y_l}{\partial y_{l-1}} &= \frac{\partial \sigma(z_l)}{\partial y_{l-1}} \\
&= \begin{pmatrix} \frac{\partial \sigma(z_l^{(1)})}{\partial y_{l-1}^{(1)}} & \cdots & \frac{\partial \sigma(z_l^{(1)})}{\partial y_{l-1}^{(n_{l-1})}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \sigma(z_l^{(n_l)})}{\partial y_{l-1}^{(1)}} & \cdots & \frac{\partial \sigma(z_l^{(n_l)})}{\partial y_{l-1}^{(n_{l-1})}} \end{pmatrix}
\end{aligned}$$

$$\frac{\partial \sigma(z_l^{(i)})}{\partial y_{l-1}^{(j)}} = \sigma'(z_l^{(i)}) \frac{\partial z_l^{(i)}}{\partial y_{l-1}^{(j)}} = \sigma'(z_l^{(i)}) A_l^{(i,j)} \quad \left(\because z_l^{(k)} = \sum_p A_l^{(k,p)} y_{l-1}^{(p)} + b_l^{(k)} \right)$$

$$\begin{aligned}
\therefore \frac{\partial y_l}{\partial y_{l-1}} &= \begin{pmatrix} \sigma'(z_l^{(1)}) A_l^{(1,1)} & \cdots & \sigma'(z_l^{(1)}) A_l^{(1,n_{l-1})} \\ \vdots & \ddots & \vdots \\ \sigma'(z_l^{(1)}) A_l^{(n_l,1)} & \cdots & \sigma'(z_l^{(1)}) A_l^{(n_l,n_{l-1})} \end{pmatrix} \\
&= \operatorname{diag}(\sigma'(A_l y_l + b_l)) A_l
\end{aligned}$$

(b)

(i)

$$\frac{\partial y_L}{\partial A_L} = \frac{\partial A_L y_{L-1}}{\partial A_L} = \left(\frac{\partial}{A_L^{(1)}} A_L y_{L-1}, \cdots, \frac{\partial}{A_L^{(n_{L-1})}} A_L y_{L-1} \right) = y_{L-1}^T$$

(ii)

To prove $\frac{\partial y_L}{\partial A_l} = \operatorname{diag}(\sigma'(A_l y_{l-1} + b_l)) \left(\frac{\partial y_L}{\partial y_l} \right)^T y_{l-1}^T$. it is enough to show the following, introducing $z_l = A_l y_{l-1} + b_l$.

$$\frac{\partial y_L}{\partial A_l^{(i,j)}} = \sigma'(z_l^{(i)}) \frac{\partial y_L}{\partial y_l^{(i)}} y_{l-1}^{(j)} \tag{3}$$

To show the above relation, consider the chain rule.

$$\frac{\partial y_L}{\partial A_l^{(i,j)}} = \frac{\partial y_L}{\partial y_l} \frac{\partial y_l}{\partial A_l^{(i,j)}}$$

Then, calculate both terms as follow.

$$\begin{aligned}
\frac{\partial y_l}{\partial A_l^{(i,j)}} &= \left(\frac{\partial \sigma(z_l^{(1)})}{\partial A_l^{(i,j)}}, \cdots, \frac{\partial \sigma(z_l^{(n_l)})}{\partial A_l^{(i,j)}} \right)^T \\
\frac{\partial \sigma(z_l^{(k)})}{\partial A_l^{(i,j)}} &= \sigma'(z_l^{(k)}) \frac{\partial z_l^{(k)}}{\partial A_l^{(i,j)}} = \sigma'(z_l^{(k)}) \delta_{ik} y_{l-1}^{(j)} \quad \left(\because z_l^{(k)} = \sum_p A_l^{(k,p)} y_{l-1}^{(p)} + b_l^{(k)} \right) \\
\therefore \frac{\partial y_l}{\partial A_l^{(i,j)}} &= \left(0, \cdots, \sigma'(z_l^{(i)}) y_{l-1}^{(j)}, \cdots, 0 \right)^T
\end{aligned}$$

Note that the nonzero element is located at the ith position. Then, since $\frac{\partial y_L}{\partial y_l} = \left(\frac{\partial y_L}{\partial y_l^{(1)}}, \cdots, \frac{\partial y_L}{\partial y_l^{(n_l)}} \right)$,

$$\frac{\partial y_L}{\partial A_l^{(i,j)}} = \left(\frac{\partial y_L}{\partial y_l^{(1)}}, \cdots, \frac{\partial y_L}{\partial y_l^{(n_l)}} \right) \cdot \left(0, \cdots, \sigma'(z_l^{(i)}) y_{l-1}^{(j)}, \cdots, 0 \right)^T = \sigma'(z_l^{(i)}) \frac{\partial y_L}{\partial y_l^{(i)}} y_{l-1}^{(j)}$$

7

The following is the code I implemented to solve this problem. Using the following code, the test accuracy for original LeNet5 was about 98.64%, while that for mordern LeNet5 was 98.71% The time consumed to train and test original LeNet5 was 18m 41sec, while mordern LeNet5 was 2m 42sec. Eventhough two algorithms have similar performance, the time complexity to train original LeNet5 takes approximately 10 time longer time. This is because C_3 layer in original LeNet5 is much more complex than that in mordern LeNet5.

Next, discussing the trainable parameters of both algorithms is meaningful. The only difference of original LeNet5 and mordern one is the C_3 layer. Thus, the difference of trainable parameters must be equal to that of C_3 layer. The number of trainable parameter of original LeNet5 was 60806, while that of mordern LeNet5 was 61706. According to the following calculation, the number of trainable parameters of original LeNet5 was 1516, while that of mordern LeNet5 was 2416.

$$6 \times (5 \times 5 \times 3 + 1) + 9 \times (5 \times 5 \times 4 + 1) + 1 \times (5 \times 5 \times 6 + 1)$$

The difference of the number of trainable parameters between original LeNet5 and mordern one was 900, which agrees to the difference between the total trainable parameters of original LeNet5 and mordern one. Thus, the handwritten computation matches to the result of the code.

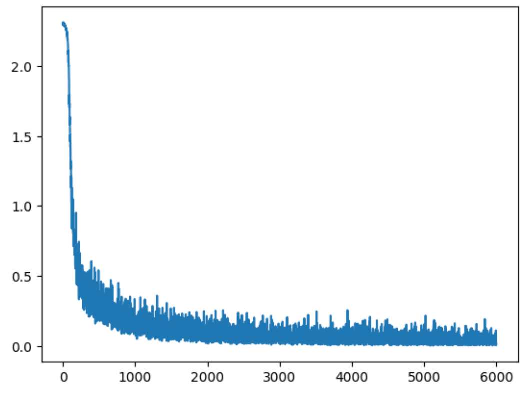

Figure 1: The training loss of problem 7 for each iteration.

```python
import torch
import torch.nn as nn
from torch.optim import Optimizer
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import transforms
import matplotlib.pyplot as plt

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

'''
Step 1:
'''

# MNIST dataset
train_dataset = datasets.MNIST(root='./mnist_data/',
                               train=True,
                               transform=transforms.ToTensor(),
                               download=True)

test_dataset = datasets.MNIST(root='./mnist_data/',
                              train=False,
                              transform=transforms.ToTensor())


'''
Step 2: LeNet5
'''

# Modern LeNet uses this layer for C3
class C3_layer_full(nn.Module):
    def __init__(self):
        super(C3_layer_full, self).__init__()
        self.conv_layer = nn.Conv2d(6, 16, kernel_size=5)

    def forward(self, x):
        return self.conv_layer(x)

# Original LeNet uses this layer for C3
class C3_layer(nn.Module):
    def __init__(self):
        super(C3_layer, self).__init__()
        self.ch_in_3 = [[0, 1, 2],
                        [1, 2, 3],
                        [2, 3, 4],
                        [3, 4, 5],
                        [0, 4, 5],
                        [0, 1, 5]] # filter with 3 subset of input channels
        self.ch_in_4 = [[0, 1, 2, 3],
                        [1, 2, 3, 4],
                        [2, 3, 4, 5],
                        [0, 3, 4, 5],
                        [0, 1, 4, 5],
                        [0, 1, 2, 5],
                        [0, 1, 3, 4],
                        [1, 2, 4, 5],
                        [0, 2, 3, 5]] # filter with 4 subset of input channels
        # put implementation here
        self.linear = nn.ModuleList()
        for _ in range(6):
            self.linear.append(nn.Conv2d(in_channels=3, out_channels=1,kernel_size=(5,5)))
        for _ in range(9):
            self.linear.append(nn.Conv2d(in_channels=4, out_channels=1,kernel_size=(5,5)))
        self.linear.append(nn.Conv2d(in_channels=6, out_channels=1, kernel_size=(5,5)))
    def forward(self, x):
        output = []
        for i in range(16):
            if i < 6:
                inputs = x[:,self.ch_in_3[i],:,:].to(device)
            elif i>=6 and i<15:
                inputs = x[:,self.ch_in_4[i-6],:,:].to(device)
            else:
                inputs = x
            output.append(self.linear[i](inputs))
        output = torch.cat(output, dim=1)
        return output

class LeNet(nn.Module) :
    def __init__(self) :
        super(LeNet, self).__init__()
        #padding=2 makes 28x28 image into 32x32
        self.C1_layer = nn.Sequential(
                nn.Conv2d(1, 6, kernel_size=5, padding=2),
                nn.Tanh()
                )
        self.P2_layer = nn.Sequential(
                nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Tanh()
                )
        self.C3_layer = nn.Sequential(
                # C3_layer_full(),
                C3_layer(),
                nn.Tanh()
                )
        self.P4_layer = nn.Sequential(
                nn.AvgPool2d(kernel_size=2, stride=2),
                nn.Tanh()
                )
        self.C5_layer = nn.Sequential(
                nn.Linear(5*5*16, 120),
                nn.Tanh()
                )
        self.F6_layer = nn.Sequential(
                nn.Linear(120, 84),
                nn.Tanh()
                )
        self.F7_layer = nn.Linear(84, 10)
        self.tanh = nn.Tanh()
    def forward(self, x) :
        output = self.C1_layer(x)
        output = self.P2_layer(output)
        output = self.C3_layer(output)
        output = self.P4_layer(output)
        output = output.view(-1,5*5*16)
        output = self.C5_layer(output)
        output = self.F6_layer(output)
        output = self.F7_layer(output)
        return output


'''
Step 3
'''
model = LeNet().to(device)
loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-1)

# print total number of trainable parameters
param_ct = sum([p.numel() for p in model.parameters()])
print(f"Total number of trainable parameters: {param_ct}")

'''
Step 4
'''
train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=100, shuffle=True
                                            )
losses = []
import time
start = time.time()
for epoch in range(10) :
    print("{}th epoch starting.".format(epoch))
    for images, labels in train_loader :
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        train_loss = loss_function(model(images), labels)
        losses.append(train_loss.item())
        train_loss.backward()

        optimizer.step()
end = time.time()
print("Time ellapsed in training is: {}".format(end - start))

plt.plot(range(6000),losses)


'''
Step 5
'''
test_loss, correct, total = 0, 0, 0

test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=100, shuffle=False)

for images, labels in test_loader :
    images, labels = images.to(device), labels.to(device)

    output = model(images)
    test_loss += loss_function(output, labels).item()

    pred = output.max(1, keepdim=True)[1]
    correct += pred.eq(labels.view_as(pred)).sum().item()

    total += labels.size(0)

print('[Test set] Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)\n'.format(
        test_loss /total, correct, total,
        100. * correct / total))
```

A Problem 5 : Implementation

A.1 KL-Divergence

```
import torch
import torch.nn as nn
from torch.optim import Optimizer
from torch.utils.data import DataLoader

# torchvision: popular datasets, model architectures, and common image transformations for
                                                    computer vision.
from torchvision import datasets
from torchvision.transforms import transforms

from random import randint
from random import shuffle
import numpy as np
import matplotlib.pyplot as plt


'''
Step 1: Prepare dataset
'''
# Use data with only 4 and 9 as labels: which is hardest to classify
label_1, label_2 = 4, 9

# MNIST training data
train_set = datasets.MNIST(root='./mnist_data/', train=True, transform=transforms.ToTensor(),
                                                    download=True)

# Use data with two labels
idx = (train_set.targets == label_1) + (train_set.targets == label_2)
train_set.data = train_set.data[idx]
train_set.targets = train_set.targets[idx]
train_set.targets[train_set.targets == label_1] = -1
train_set.targets[train_set.targets == label_2] = 1

# MNIST testing data
test_set = datasets.MNIST(root='./mnist_data/', train=False, transform=transforms.ToTensor())

# Use data with two labels
idx = (test_set.targets == label_1) + (test_set.targets == label_2)
test_set.data = test_set.data[idx]
test_set.targets = test_set.targets[idx]
test_set.targets[test_set.targets == label_1] = -1
test_set.targets[test_set.targets == label_2] = 1

'''
Step 2: Define the neural network class
'''
class LR(nn.Module) :
    '''
    Initialize model
        input_dim : dimension of given input data
    '''
    # MNIST data is 28x28 images
    def __init__(self, input_dim=28*28) :
        super().__init__()
        self.linear = nn.Linear(input_dim, 1, bias=True)

    ''' forward given input x '''
    def forward(self, x) :
        return self.linear(x.float().view(-1, 28*28))

'''
Step 3: Create the model, specify loss function and optimizer.
'''
model = LR()                                        # Define a Neural Network Model

def logistic_loss(output, target):
    return -torch.nn.functional.logsigmoid(target*output)

loss_function = logistic_loss                                                       # Specify loss
                                                    function
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)    # specify SGD with learning rate

'''
Step 4: Train model with SGD
'''
losses = []
```

```python
for _ in range(50000) :
    # Sample a random data for training
    ind = randint(0, len(train_set.data)-1)
    image, label = train_set.data[ind], train_set.targets[ind]

    # Clear previously computed gradient
    optimizer.zero_grad()

    # then compute gradient with forward and backward passes
    train_loss = loss_function(model(image), label.float())
    train_loss.backward()
    losses.append(train_loss.item())
    #(This syntax will make more sense once we learn about minibatches)

    # perform SGD step (parameter update)
    optimizer.step()
'''
Step 5: Test model (Evaluate the accuracy)
'''
test_loss, correct = 0, 0
misclassified_ind = []
correct_ind = []

# Evaluate accuracy using test data
for ind in range(len(test_set.data)) :

    image, label = test_set.data[ind], test_set.targets[ind]

    # evaluate model
    output = model(image)

    # Calculate cumulative loss
    test_loss += loss_function(output, label.float()).item()

    # Make a prediction
    if output.item() * label.item() >= 0 :
        correct += 1
        correct_ind += [ind]
    else:
        misclassified_ind += [ind]

# Print out the results
print('[Test set] Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)\n'.format(
        test_loss /len(test_set.data), correct, len(test_set.data),
        100. * correct / len(test_set.data)))

import torch
import torch.nn as nn
from torch.optim import Optimizer
from torch.utils.data import DataLoader

# torchvision: popular datasets, model architectures, and common image transformations for
                                               computer vision.
from torchvision import datasets
from torchvision.transforms import transforms

from random import randint
from random import shuffle
import numpy as np
import matplotlib.pyplot as plt


'''
Step 1: Prepare dataset
'''
# Use data with only 4 and 9 as labels: which is hardest to classify
label_1, label_2 = 4, 9

# MNIST training data
train_set = datasets.MNIST(root='./mnist_data/', train=True, transform=transforms.ToTensor(),
                                               download=True)

# Use data with two labels
idx = (train_set.targets == label_1) + (train_set.targets == label_2)
train_set.data = train_set.data[idx]
train_set.targets = train_set.targets[idx]
train_set.targets[train_set.targets == label_1] = -1
train_set.targets[train_set.targets == label_2] = 1

# MNIST testing data
test_set = datasets.MNIST(root='./mnist_data/', train=False, transform=transforms.ToTensor())

# Use data with two labels
idx = (test_set.targets == label_1) + (test_set.targets == label_2)
test_set.data = test_set.data[idx]
test_set.targets = test_set.targets[idx]
test_set.targets[test_set.targets == label_1] = -1
test_set.targets[test_set.targets == label_2] = 1

'''
Step 2: Define the neural network class
'''
class LR(nn.Module) :
    '''
    Initialize model
        input_dim : dimension of given input data
    '''
    # MNIST data is 28x28 images
    def __init__(self, input_dim=28*28) :
        super().__init__()
        self.linear = nn.Linear(input_dim, 1, bias=True)

    ''' forward given input x '''
    def forward(self, x) :
        return self.linear(x.float().view(-1, 28*28))

'''
Step 3: Create the model, specify loss function and optimizer.
'''
model = LR()                                    # Define a Neural Network Model

def loss_func(target,output):
    return 0.5*(1-output)*((1-torch.sigmoid(-target))**2 + torch.sigmoid(target)**2)+0.5*(1+
                                                         output)*((torch.sigmoid(-target))**2+(1-
                                                         torch.sigmoid(target))**2)

loss_function = logistic_loss                                                         # Specify loss
                                                      function
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)   # specify SGD with learning rate

'''
Step 4: Train model with SGD
'''
losses = []
for _ in range(1000) :
    # Sample a random data for training
    ind = randint(0, len(train_set.data)-1)
    image, label = train_set.data[ind], train_set.targets[ind]

    # Clear previously computed gradient
    optimizer.zero_grad()

    # then compute gradient with forward and backward passes
    train_loss = loss_function(model(image), label.float())
    train_loss.backward()
    losses.append(train_loss.item())
    #(This syntax will make more sense once we learn about minibatches)

    # perform SGD step (parameter update)
    optimizer.step()
'''
Step 5: Test model (Evaluate the accuracy)
'''
test_loss, correct = 0, 0
misclassified_ind = []
correct_ind = []

# Evaluate accuracy using test data
for ind in range(len(test_set.data)) :

    image, label = test_set.data[ind], test_set.targets[ind]
    # evaluate model
    output = model(image)

    # Calculate cumulative loss
    test_loss += loss_function(output, label.float()).item()

    # Make a prediction
    if output.item() * label.item() >= 0 :
        correct += 1
        correct_ind += [ind]
    else:
        misclassified_ind += [ind]

# Print out the results
print('[Test set] Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)\n'.format(
        test_loss /len(test_set.data), correct, len(test_set.data),
        100. * correct / len(test_set.data)))
```

A.2 MSE

```
    import torch
import torch.nn as nn
from torch.optim import Optimizer
from torch.utils.data import DataLoader

# torchvision: popular datasets, model architectures, and common image transformations for
                                                   computer vision.
from torchvision import datasets
from torchvision.transforms import transforms

from random import randint
from random import shuffle
import numpy as np
import matplotlib.pyplot as plt


'''
Step 1: Prepare dataset
'''
# Use data with only 4 and 9 as labels: which is hardest to classify
label_1, label_2 = 4, 9

# MNIST training data
train_set = datasets.MNIST(root='./mnist_data/', train=True, transform=transforms.ToTensor(),
                                                   download=True)

# Use data with two labels
idx = (train_set.targets == label_1) + (train_set.targets == label_2)
train_set.data = train_set.data[idx]
train_set.targets = train_set.targets[idx]
train_set.targets[train_set.targets == label_1] = -1
train_set.targets[train_set.targets == label_2] = 1

# MNIST testing data
test_set = datasets.MNIST(root='./mnist_data/', train=False, transform=transforms.ToTensor())

# Use data with two labels
idx = (test_set.targets == label_1) + (test_set.targets == label_2)
test_set.data = test_set.data[idx]
test_set.targets = test_set.targets[idx]
test_set.targets[test_set.targets == label_1] = -1
test_set.targets[test_set.targets == label_2] = 1

'''
Step 2: Define the neural network class
'''
class LR(nn.Module) :
    '''
    Initialize model
        input_dim : dimension of given input data
    '''
    # MNIST data is 28x28 images
    def __init__(self, input_dim=28*28) :
        super().__init__()
        self.linear = nn.Linear(input_dim, 1, bias=True)

    ''' forward given input x '''
    def forward(self, x) :
        return self.linear(x.float().view(-1, 28*28))

'''
Step 3: Create the model, specify loss function and optimizer.
'''
model = LR()                                    # Define a Neural Network Model

def loss(target,output):
    return 0.5*(1-output)*((1-torch.sigmoid(-target))**2 + torch.sigmoid(target))+0.5*(1+
                                                    output)*((torch.sigmoid(-target))**2+(1-
                                                    torch.sigmoid(target))**2)

loss_function = loss                                            # Specify loss function
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)   # specify SGD with learning rate

'''
Step 4: Train model with SGD
'''
losses = []
for _ in range(50000) :
    # Sample a random data for training
    ind = randint(0, len(train_set.data)-1)
    image, label = train_set.data[ind], train_set.targets[ind]

    # Clear previously computed gradient
    optimizer.zero_grad()

    # then compute gradient with forward and backward passes
    train_loss = loss_function(model(image), label.float())
    train_loss.backward()
    losses.append(train_loss.item())
    #(This syntax will make more sense once we learn about minibatches)

    # perform SGD step (parameter update)
    optimizer.step()
'''
Step 5: Test model (Evaluate the accuracy)
'''
test_loss, correct = 0, 0
misclassified_ind = []
correct_ind = []

# Evaluate accuracy using test data
for ind in range(len(test_set.data)) :

    image, label = test_set.data[ind], test_set.targets[ind]

    # evaluate model
    output = model(image)

    # Calculate cumulative loss
    test_loss += loss_function(output, label.float()).item()

    # Make a prediction
    if output.item() * label.item() >= 0 :
        correct += 1
        correct_ind += [ind]
    else:
        misclassified_ind += [ind]

# Print out the results
print('[Test set] Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)\n'.format(
        test_loss /len(test_set.data), correct, len(test_set.data),
        100. * correct / len(test_set.data)))
```