In [ ]:

Neural Networks for Recognition - Assignment 3

Instructor: Kris TAs: Arka, Jinkun, Rawal, Rohan, Sheng-Yu

Theory Questions (45 points)

Grading:

- The theory part consists of 7 questions.
- Please add your answers to the writeup (submitted as pdf to HW3:PDF). Insert images whenever necessary.
- Show all your work to obtain full credit.

Q1 (3 points)

The softmax function can be defined as, $softmax(x_i) = \frac{1}{S}s_i$ where $s_i = e^{x_i}$, $S = \sum_i s_i$. Using this definition, please answer Q1.1, Q1.2 and Q1.3 below.

Q1.1 (1 point)

Let $x \in \mathbb{R}^d$, what are the properties of $softmax(x)$, specifically, what is the range of each element in $softmax(x)$? What is the sum of all elements in $softmax(x)$?

Softmax is a function that turns a given vector into a vector of same dimensions such that the sum of the elements of the vector is 1. Each element in the input vector can vary from $[-\infty, +\infty]$. Each element in the output is in the range [0,1] and the sum of all the elements in output = 1.

Q1.2 (1 point)

"Softmax takes an arbitrary real valued vector x and turns it into a ___". **Please fill in the blank using an appropriate word/phrase**.

probability distribution of all the elements in x.

Q1.3 (1 point)

Let $x \in \mathbb{R}^d$, assume $v = softmax(softmax(...softmax(x)))$ where the softmax function is applied to x recursively N times. What is the value of v as a function of d $\forall x \in \mathbb{R}^d$, in the limit $N \to \infty$?

1/d
Each element in x will have equal probability = 1/d

Q1.1.1 (3 points, write-up)

Why is it not a good idea to initialize a network with all zeros? If you imagine that every layer has weights and biases, what can a zero-initialized network output after training?

Please include your answer to HW3:PDF

Initializing all the weights in the network with zero will not give us any gradient (gradient=0) therefore we wont be able to train anything since we havent learned any information. The output after training will be nothing but all zeroes.

Q1.1.3 (2 points, write-up)

Why is it a good practice to initialize the parameters using random numbers? Explain the intuition behind scaling the initializations depending on layer size (see near Fig 6 in Xavier initialization)?

Please include your answer to HW3:PDF

By initializing the parameters using random numbers, we make sure that the gradient is not approaching the same local minima and getting stuck there. With different initializations, the gradient will keep tracking different minimas.

We scale the initializations so as to prevent phenomenons like vanishing gradient or explosing gradient. By having a sufficiently large variance in the initialized weights the training becomes effective, as opposed to having gradients of very different magnitudes at different layers that yield to ill-conditioning and slower training.

Q2 (4 points)

Prove that softmax is invariant to translation, that is

$$softmax(x) = softmax(x + c) \qquad \forall c \in \mathbb{R}$$

Again, softmax is defined as, for each index i in a vector x.

$$softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

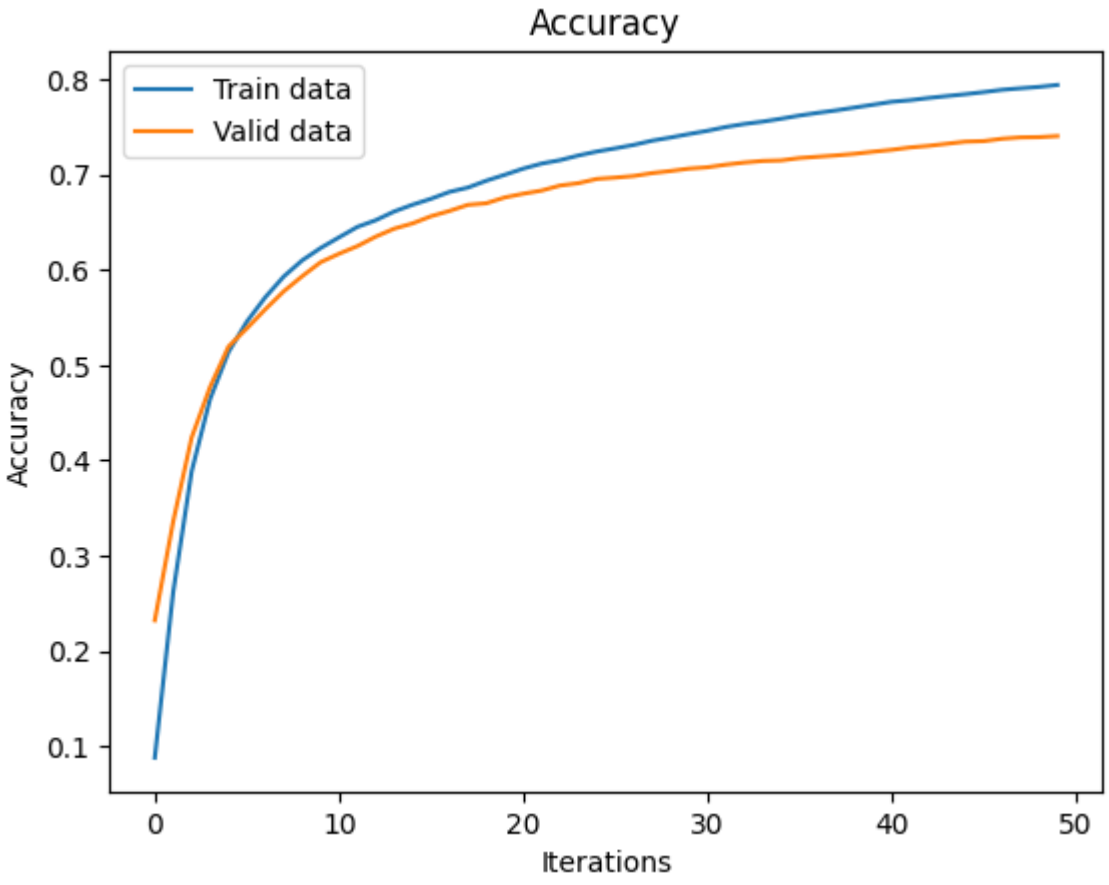
Often we use $c = -\max x_i$. Why is that a good idea? (Tip: consider the range of values that numerator will have with $c = 0$ and $c = -\max x_i$)

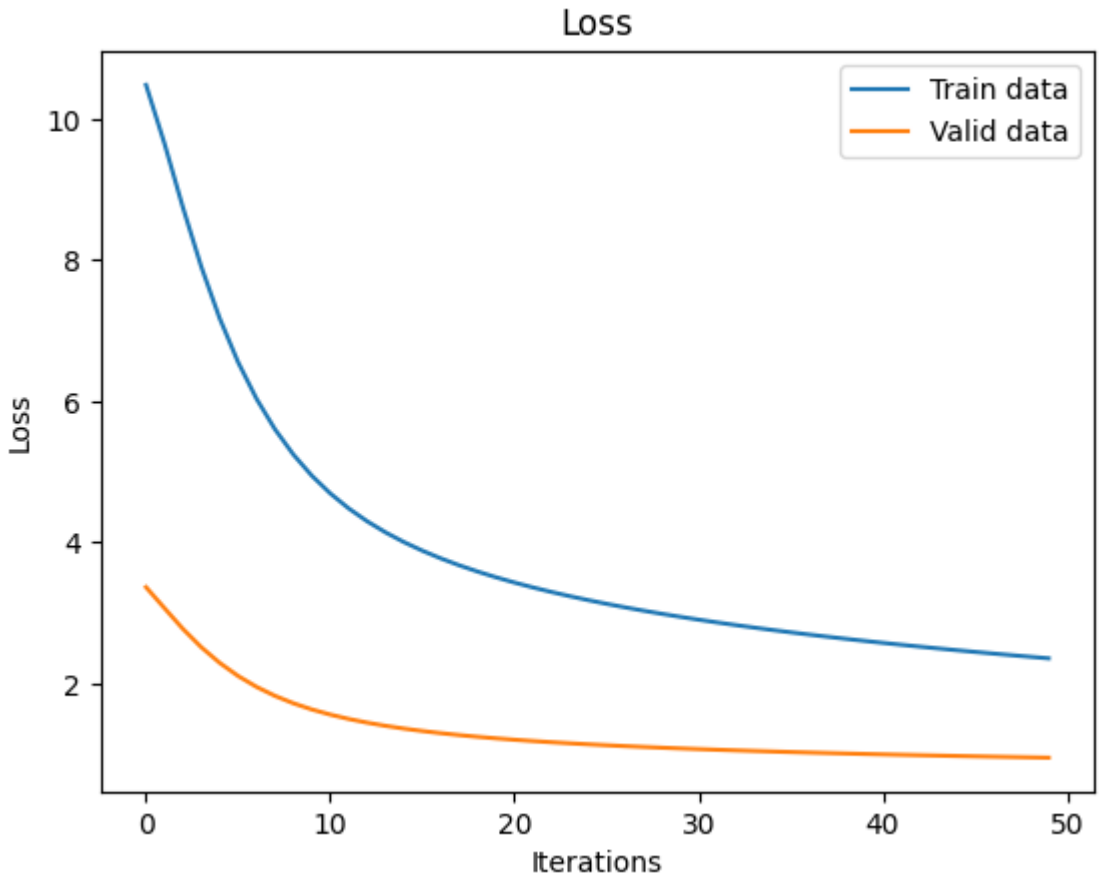
$$softmax(x + c) = \frac{e^{x_i+c}}{\sum_j e^{x_j+c}} = \frac{e^{x_i}e^c}{\sum_j e^{x_j}e^c} = \frac{e^{x_i}}{\sum_j e^{x_j}} = softmax(x)$$

Adding negative of the maximum value ensures that the exponential does not blow up. In case of a very large value the exponential will have a very high value. But the adding of -ve term, we bring down the exponential term and thus get resonable values. Had we not done this, i.e., for c = 0, the range of values that the numerator will have will vary from very small to way too high values!

Q2.1

Validation accuracy: 0.725 Accuracy | Loss :-------------------------:|:-------------------------:





In [ ]:
```
### Q2.1
import math
train_data = scipy.io.loadmat('data/nist36_train.mat')
valid_data = scipy.io.loadmat('data/nist36_valid.mat')

train_x, train_y = train_data['train_data'], train_data['train_labels']
valid_x, valid_y = valid_data['valid_data'], valid_data['valid_labels']

max_iters = 50
# pick a batch size, learning rate
batch_size = 3
learning_rate = 1e-3
# YOUR CODE HERE

# raise NotImplementedError()
hidden_size = 64

batches = get_random_batches(train_x,train_y,batch_size)
batch_num = len(batches)

params = {}

# initialize layers (named "layer1" and "output") here
# YOUR CODE HERE
initialize_weights(1024,hidden_size,params,'layer1')
initialize_weights(hidden_size,36,params,'output')
# raise NotImplementedError()
train_acc_arr = []
train_loss_arr = []
```

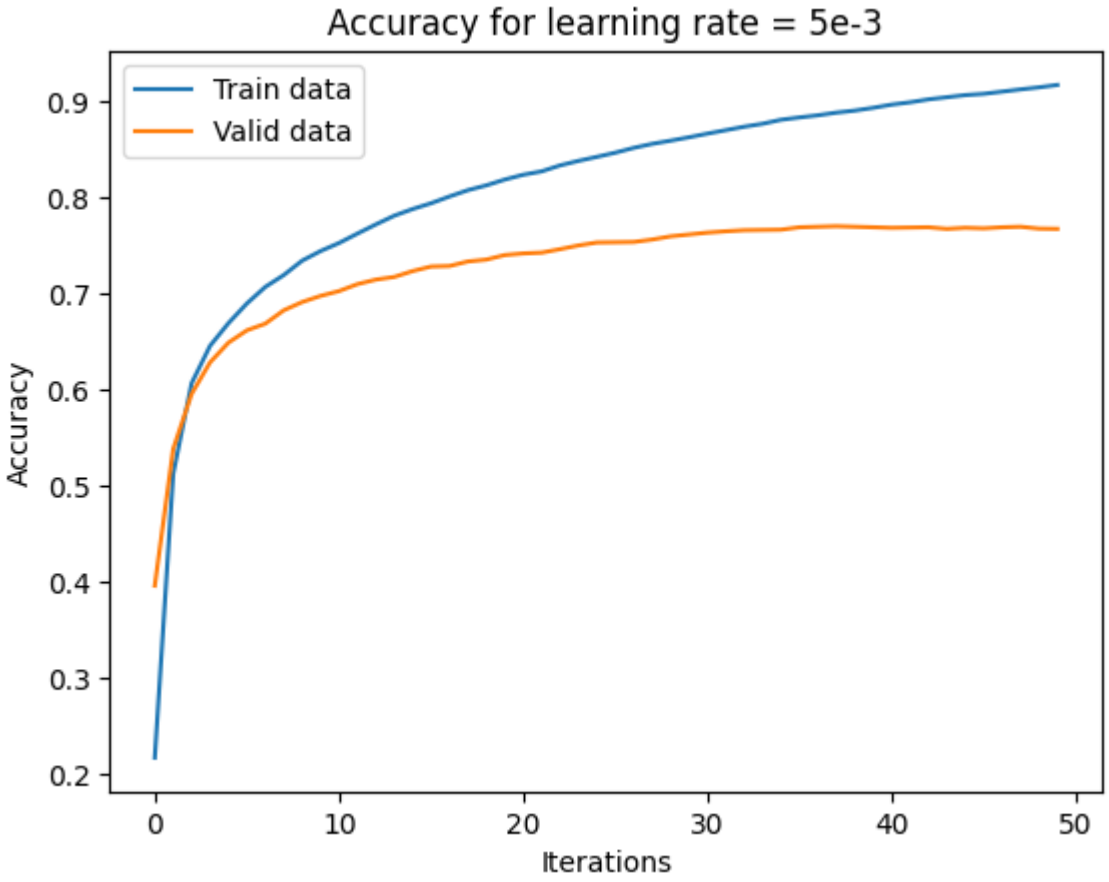
```
valid_acc_arr = []
valid_loss_arr = []
itr_arr = []
# with default settings, you should get loss < 150 and accuracy > 80%
for itr in range(max_iters):
    itr_arr.append(itr)
    total_loss = 0
    total_acc = 0
    valid_loss = 0
    valid_acc = 0
    for xb,yb in batches:
        # print("xb shape = ",xb.shape)
        # training loop can be exactly the same as q2!
        # YOUR CODE HERE
        post_act = forward(xb,params,'layer1',sigmoid)
        pred_output = forward(post_act,params,'output',softmax)
        # raise NotImplementedError()

        # loss
        # be sure to add loss and accuracy to epoch totals
        # YOUR CODE HERE
        loss, acc = compute_loss_and_acc(yb, pred_output)
        total_loss += loss/len(batches)
        total_acc += acc/len(batches)

        # raise NotImplementedError()

        # backward
        # YOUR CODE HERE
        last_layer_backprop = backwards(pred_output - yb, params, 'output', li
        hidden_layer_backprop  = backwards(last_layer_backprop, params, 'layer
        # raise NotImplementedError()

        # apply gradient
        # YOUR CODE HERE
        params['Woutput'] = params['Woutput'] - learning_rate*params['grad_Wou
        params['boutput'] = params['boutput'] - learning_rate*params['grad_bou
        params['Wlayer1'] = params['Wlayer1'] - learning_rate*params['grad_Wla
        params['blayer1'] = params['blayer1'] - learning_rate*params['grad_bla

        # raise NotImplementedError()
    print("acc = ", total_acc)
    train_acc_arr.append(total_acc)
    train_loss_arr.append(total_loss)
    if itr % 2 == 0:
        print("itr: {:02d} \t loss: {:.2f} \t acc : {:.2f}".format(itr,total_l

# run on validation set and report accuracy! should be above 70%

    post_act = forward(valid_x,params,'layer1',sigmoid)
    pred_output = forward(post_act,params,'output',softmax)
    # raise NotImplementedError()

    # loss
    # be sure to add loss and accuracy to epoch totals
    # YOUR CODE HERE
    loss, acc = compute_loss_and_acc(valid_y, pred_output)
```

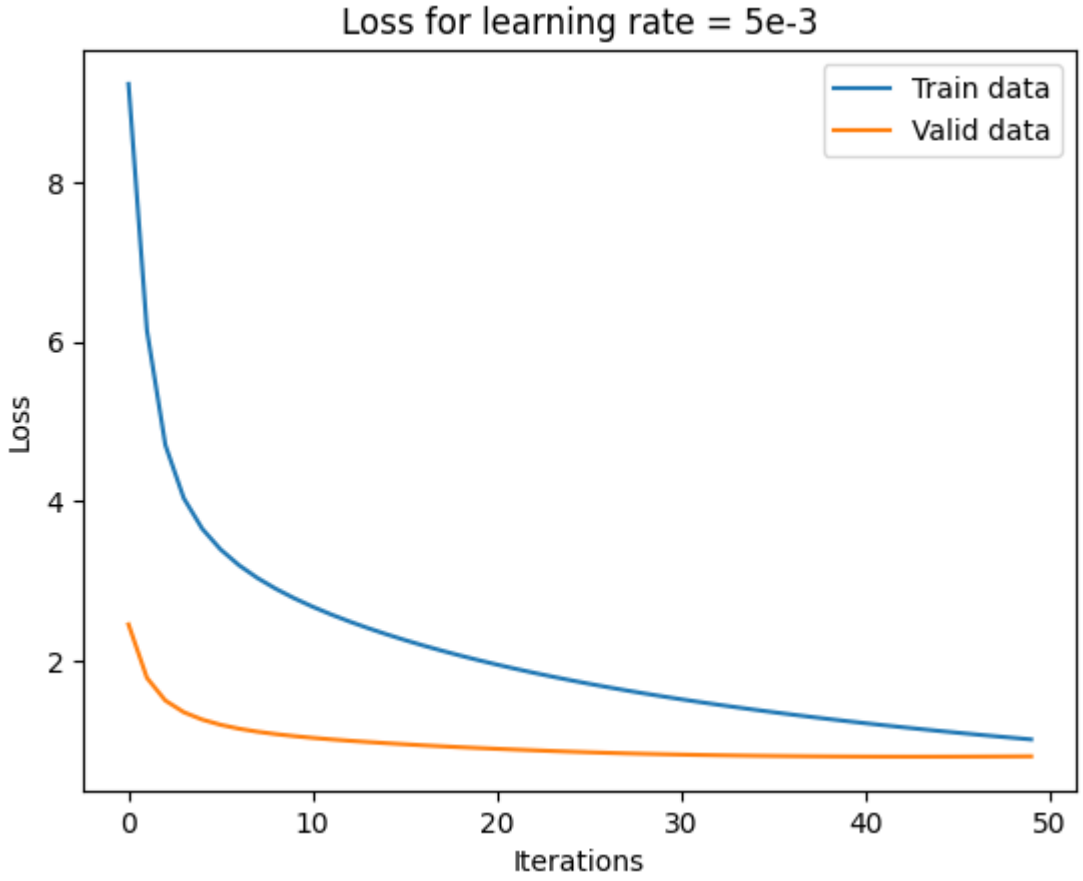
```
    valid_loss += loss/len(batches)
    valid_acc += acc
    valid_acc_arr.append(valid_acc)
    valid_loss_arr.append(valid_loss)
    # raise NotImplementedError()
    print('Validation accuracy: ',valid_acc)


import matplotlib.pyplot as plt
plt.plot(itr_arr,train_acc_arr, label = "Train data")
plt.plot(itr_arr,valid_acc_arr, label = "Valid data")
plt.xlabel('Iterations')
plt.ylabel('Accuracy')
plt.legend()
plt.title("Accuracy")
plt.show()

plt.plot(itr_arr,train_loss_arr, label = "Train data")
plt.plot(itr_arr,valid_loss_arr, label = "Valid data")
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
plt.title("Loss")
plt.show()
```

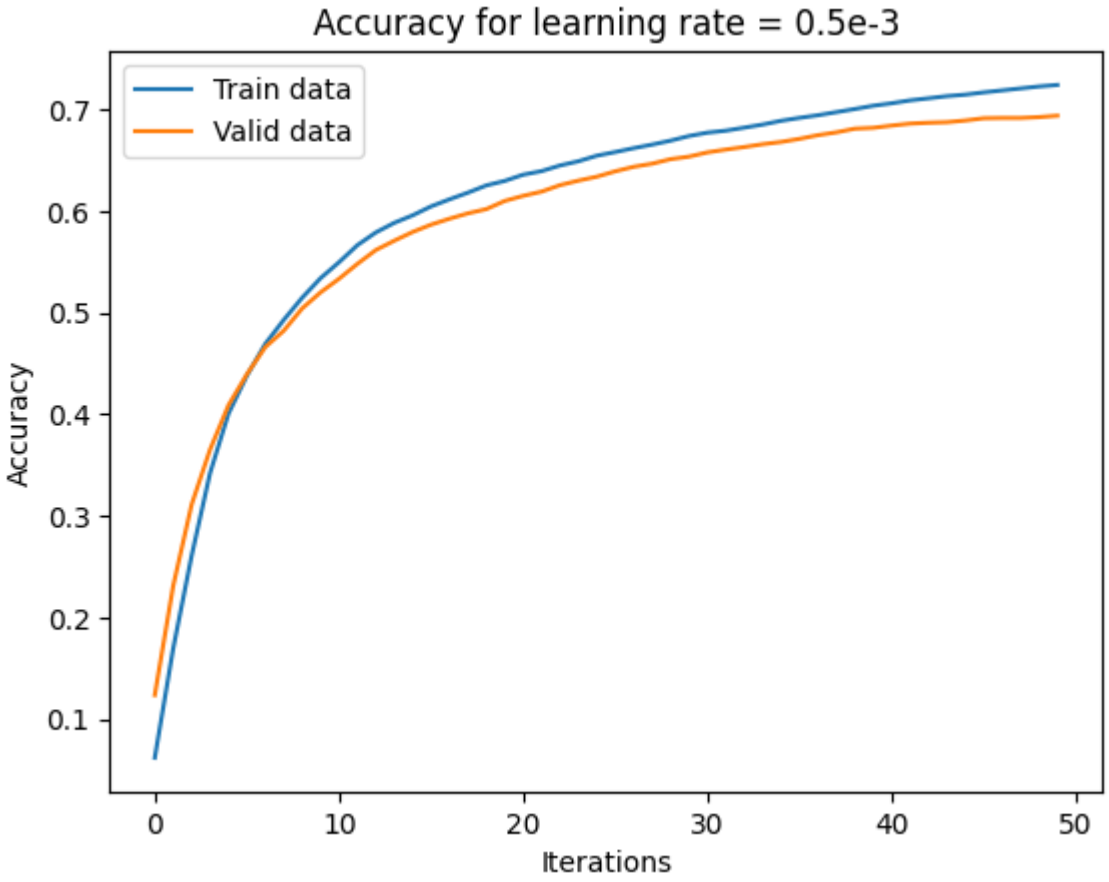
Q2.2

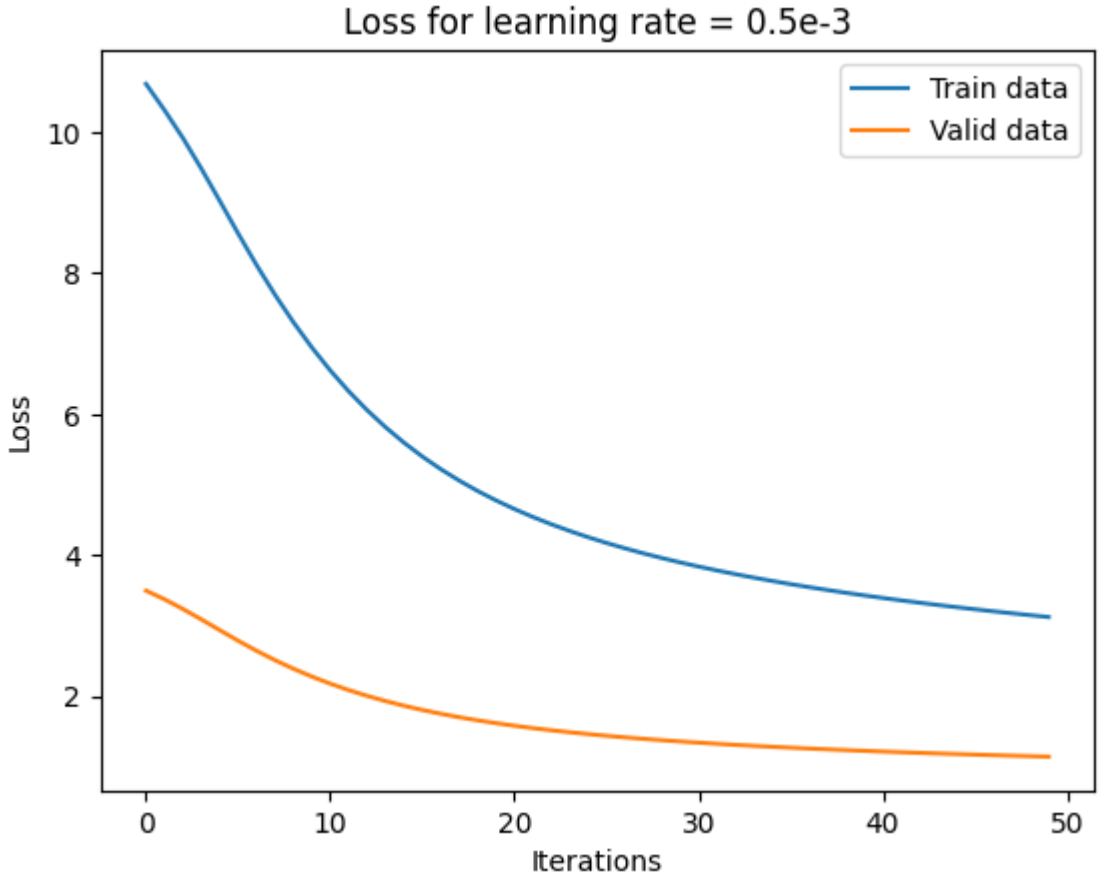
Test accuracy: 0.7633333333333333 for learning rate 5e^-3 on test data.

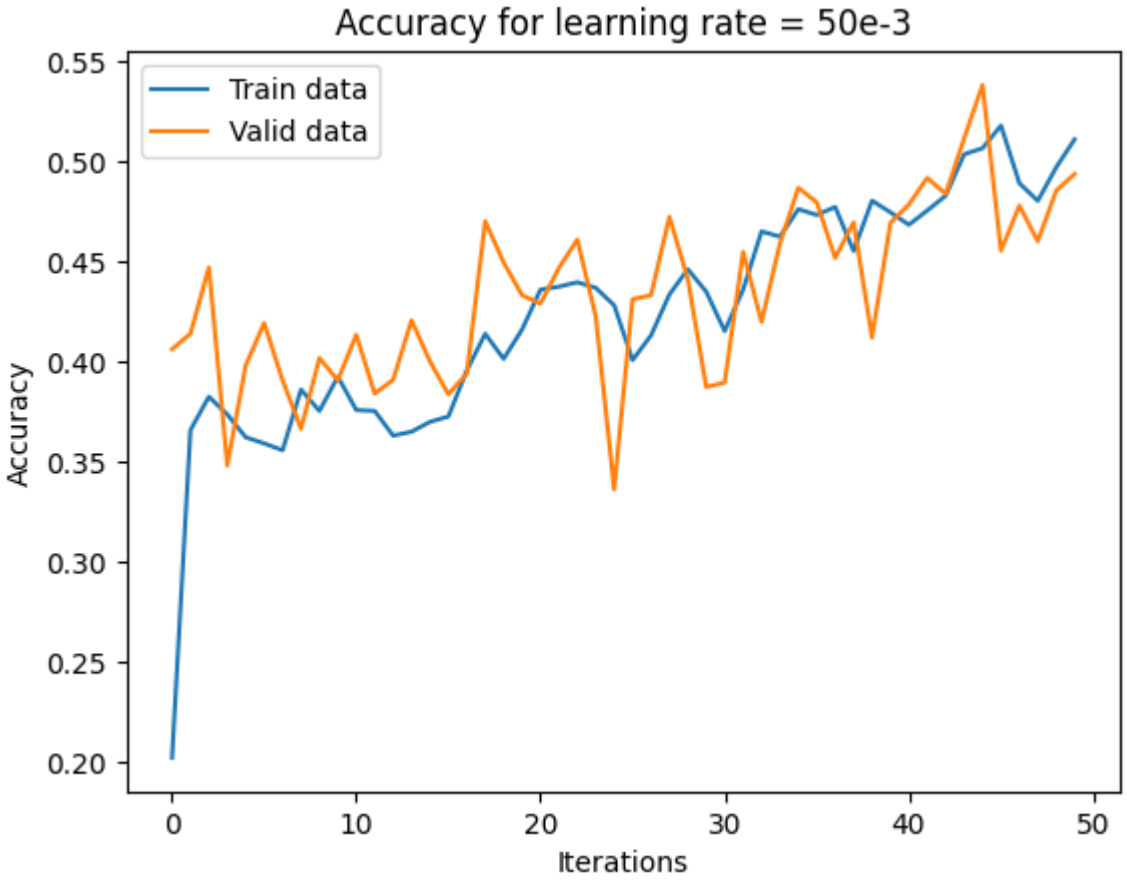
With increase in learning rate, the changes in the weights are big, i.e., the accuracy and loss go haywire. The model approaches the minima of the loss function in less time but takes big steps in random directions. Whereas with a slow learning rate, the model track the minima of the loss function very slowly and steadily with small steps that seem very smooth. Accuracy | Loss :---------:|:----------:


Accuracy for learning rate = 5e-3
Train data
Valid data
Accuracy
Iterations
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0
10
20
30
40
50


Loss for learning rate = 5e-3
Train data
Valid data
Loss
Iterations
2
4
6
8
0
10
20
30
40
50


Accuracy for learning rate = 0.5e-3
Train data
Valid data
Accuracy
Iterations
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0
10
20
30
40
50

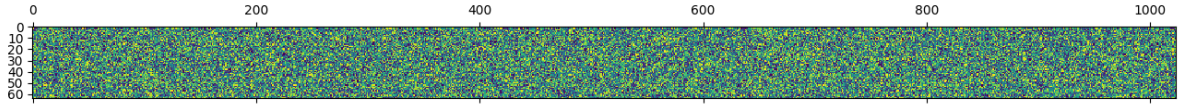

Loss for learning rate = 0.5e-3
Train data
Valid data
Loss
Iterations
2
4
6
8
10
0
10
20
30
40
50


Accuracy for learning rate = 50e-3
Train data
Valid data
Accuracy
Iterations
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0
10
20
30
40
50

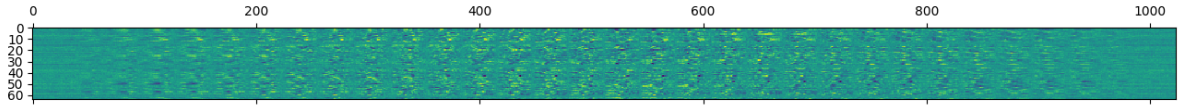

Loss for learning rate = 50e-3
Train data
Valid data
Loss
Iterations
8
7
6
5
4
3
2
0
10
20
30
40
50

Q2.3

Initialized Weights:



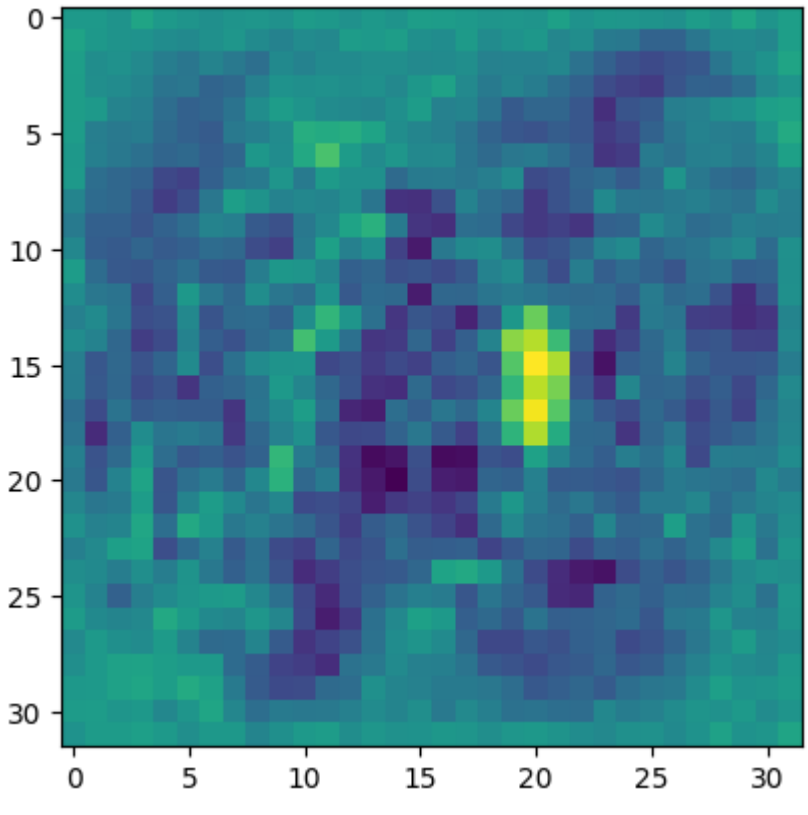
Trained Weights:

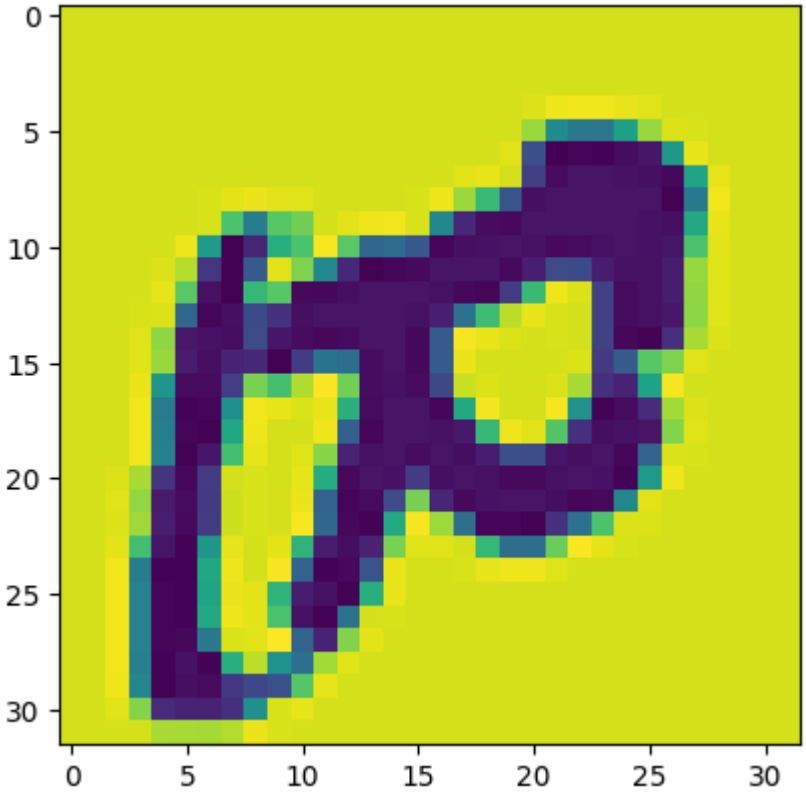


After the first glance one can comment that the initialized weights look all random (and they should!) and the trained weights have some form of pattern (the least one can say is the they dont look random).
Now after a deeper look at the learned weights, we can see that there seems to be a pattern that sort of keep repeating after regular intervals. There is no way that we can back track as to what the weights truly represent, but we can be sure that they have gained values that correspond to a particular type of input. We can also see that at the two extremeties, there isnt much change or visible pattern. We can relate this to the type of input images that we gave for training. The main content (text in out case) lies in the central part of the image and the boundary regions are plane. And we can relate this to the trained weights picture above: way=vy pattern is absent in the extreme regions.

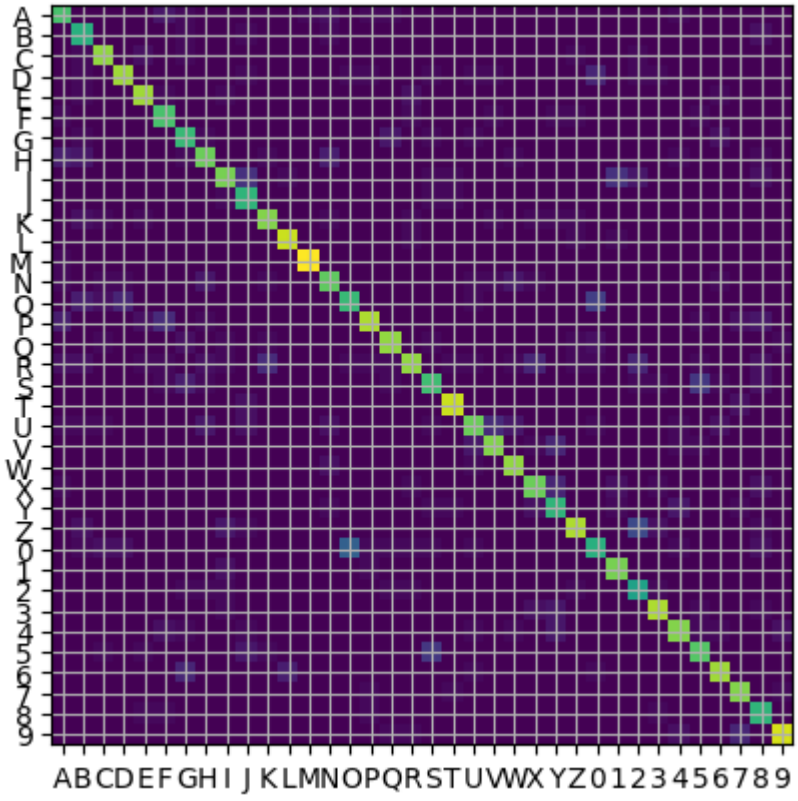
Q2.4

Weights | Image :----------:|:----------:





Q2.5



We can see that there are some blocks that correspond to zero(0) and "O"; "V" and "U"; one(1) and "I"; five(5) and "S". Which means that the network is not able to distinguish these similar characters, which makes sense as well. They all have similar shapes and structure. So the

weights corresponding them will have almost similar values and so will the gradient track the similar minima. Hence, we get these confused outputs.

Q3 (3 points)

Show that the functions represented by a multi-layer fully-connected neural networks without a non-linear activation function are linear functions of the input.

Let input be "X". And the function that maps from one layer (i) to the next (j) be

$$F_{i,j} = W_{i,j} * X + b_{i,j}$$

For the first layer X will be the input, and for the subsequent layers X will be the output of the subsequent layers.

$$F_{i,j} = W_{i,j} * F_{i-1,j-1} + b_{i,j}$$

Here we can say that each next layer is a linear function of its previous layer.

$$F_{i,j} = W_{i,j} * F(X) + b_{i,j}$$

If we keep tracking back to the first layer, $F_{i-1,j-1}$ will be a function of X. Therefore, every layer is a linear function of X.

Q3.1 (3 points, write-up)

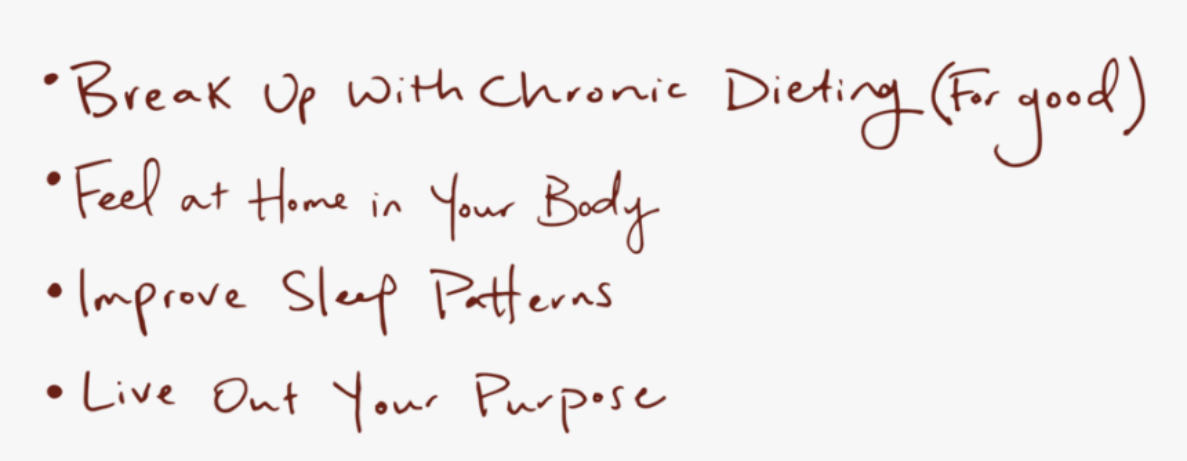
The method outlined above is pretty simplistic, and makes several assumptions. What are two big assumptions that the sample method makes. In your writeup, include two example images where you expect the character detection to fail (either miss valid letters, or respond to non-letters).

Please include your answer to HW3:PDF

First assumption: This algo assumes that the characters arent overwritten and have sufficient spaces between them. This will fail for cursive handwriting or cases where the characters are very close to each other.

Second assumption: We also assume that each single character is fully connected and has no dis-joints, or is not made up of mixture of multiple small connections.

This algorithm can classify bullet points as zeroes or Os, i.e., a non-letter. Or classify 5 as "S", 1 as "I" as miss valid letters.



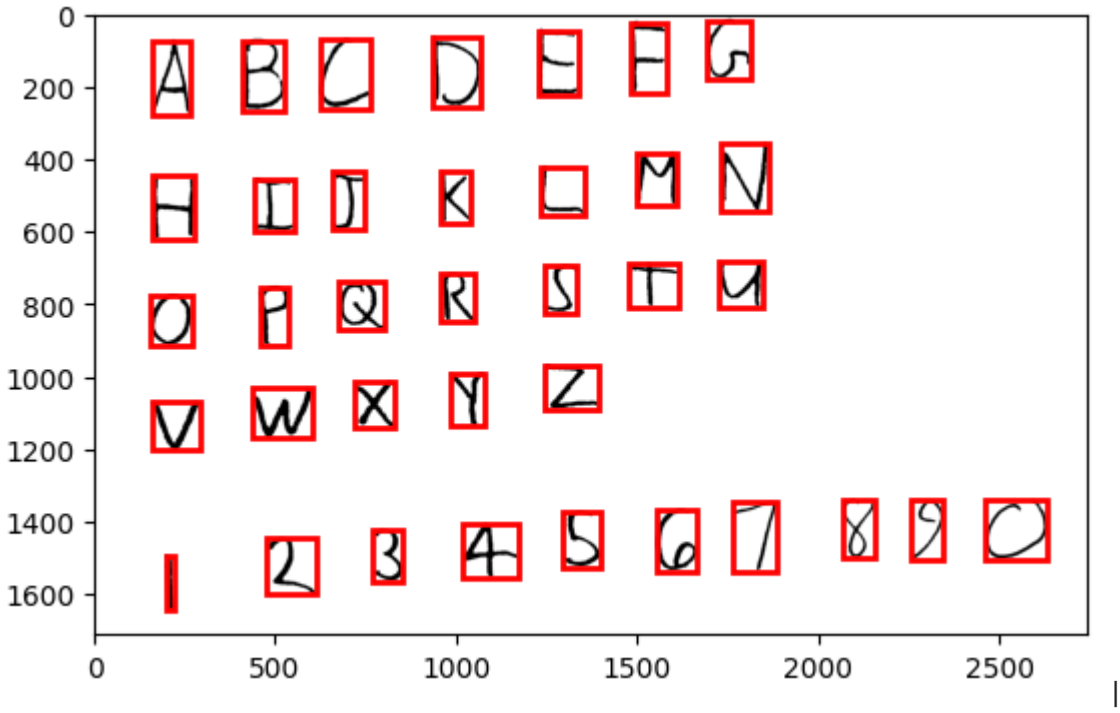
In [ ]:
```
### Q3.2
import numpy as np

import skimage
import skimage.measure
import skimage.color
import skimage.restoration
import skimage.io
import skimage.filters
import skimage.morphology
import skimage.segmentation
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

from skimage import data, img_as_float
from skimage.filters import threshold_otsu
from skimage.segmentation import clear_border
from skimage.measure import label, regionprops
from skimage.morphology import closing, square
from skimage.color import label2rgb
from skimage.color import rgb2gray
from skimage.restoration import denoise_tv_chambolle, estimate_sigma, denoise_
from skimage.morphology import square

# takes a color image
# returns a list of bounding boxes and black_and_white image
def findLetters(image):
    image = rgb2gray(image)
    image = img_as_float(image)
    est_noise = estimate_sigma(image, average_sigmas=True)
    # print("estimated noise = ", est_noise)
    denoised_img = denoise_tv_chambolle(image, weight=0.1)
```

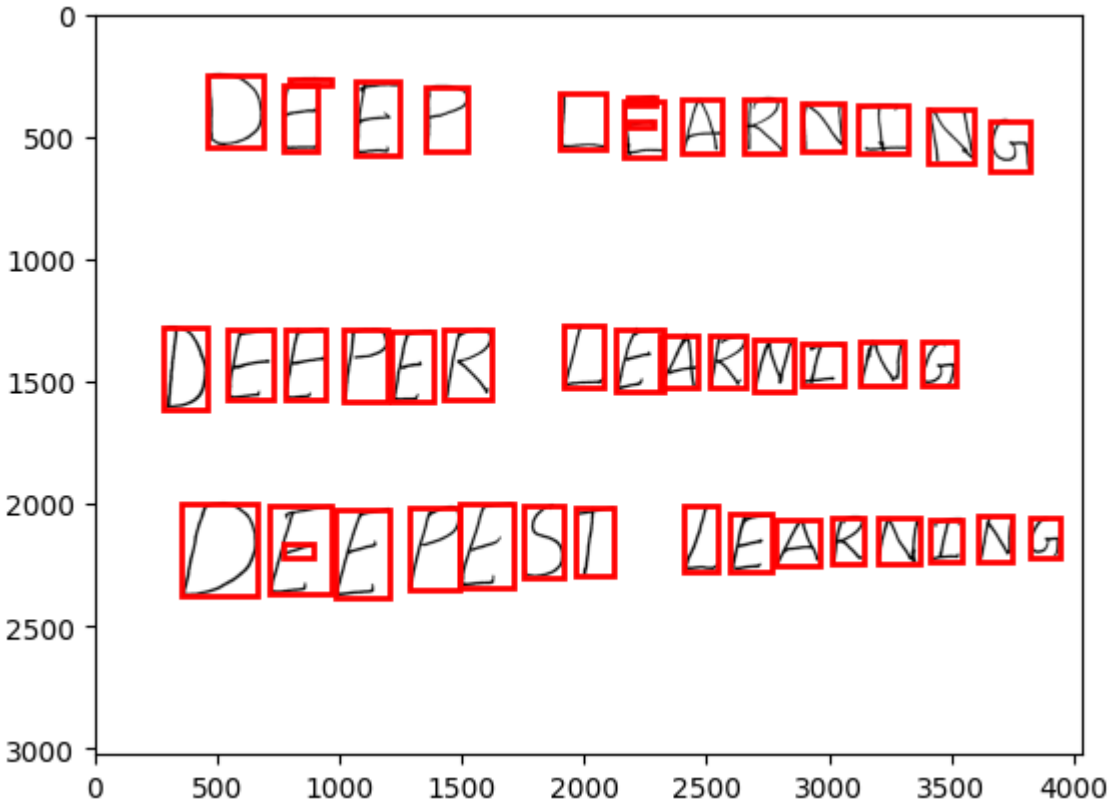
```
    # denoised_img = denoise_bilateral(image, sigma_color=0.05, sigma_spatial=

    denoised_img = skimage.morphology.erosion(denoised_img, square(3))
    denoised_img = skimage.morphology.erosion(denoised_img, square(3))
    denoised_img = skimage.morphology.dilation(denoised_img)



    bboxes = []
    bw = None
    # insert processing in here
    # one idea estimate noise -> denoise -> greyscale -> threshold -> morpholo
    # this can be 10 to 15 lines of code using skimage functions
    # YOUR CODE HERE

    # apply threshold
    thresh = threshold_otsu(denoised_img)
    bw = denoised_img > thresh

    # remove artifacts connected to image border
    # cleared = clear_border(bw)

    # label image regions
    label_image = label(1-bw)
    # to make the background transparent, pass the value of `bg_label`,
    # and leave `bg_color` as `None` and `kind` as `overlay`
    # image_label_overlay = label2rgb(label_image, image=denoised_img, bg_labe

    # fig, ax = plt.subplots(figsize=(10, 6))
    # ax.imshow(image_label_overlay, cmap="gray")

    for region in regionprops(label_image):
        # take regions with large enough areas
        if region.area >= 50:
            # draw rectangle around segmented coins
            minr, minc, maxr, maxc = region.bbox
            temp = [minr, minc, maxr, maxc]
            # print(temp)
            bboxes.append(temp)
    # # raise NotImplementedError()
    return bboxes, bw
```

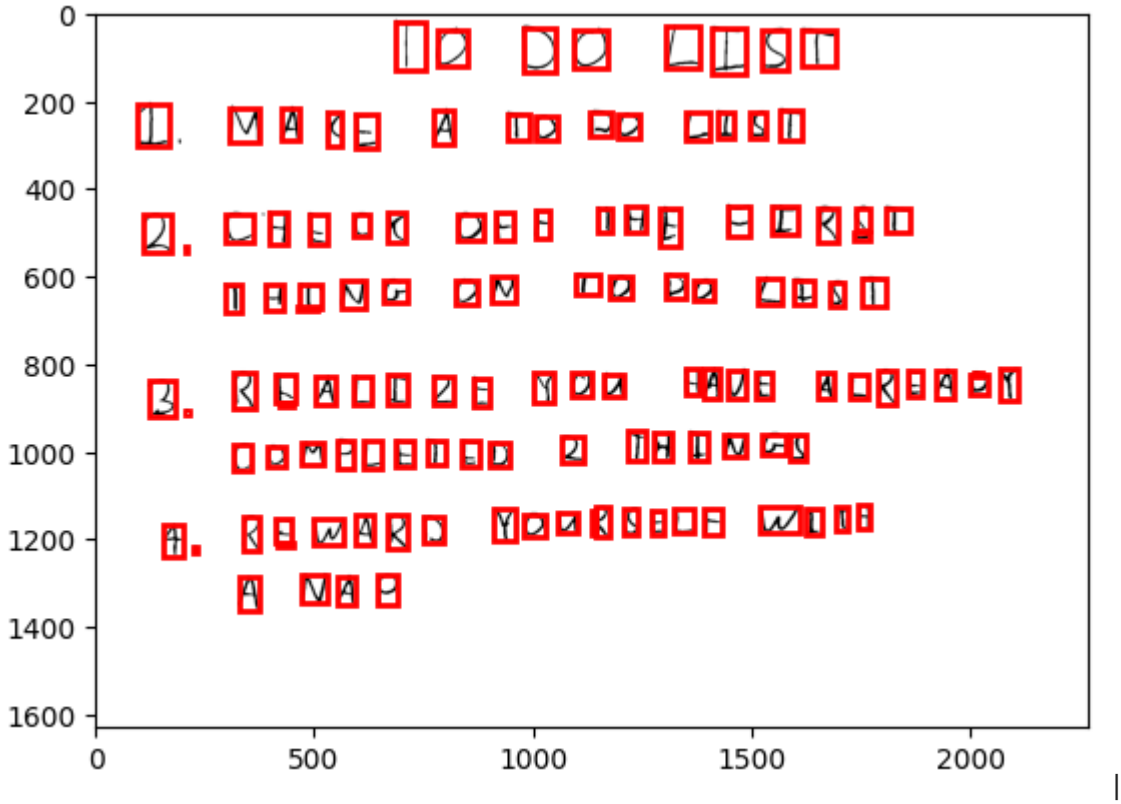
In [ ]:

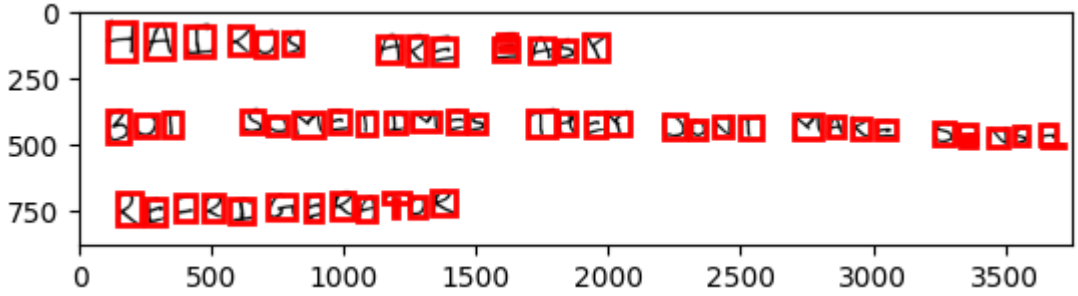
```
### Q3.3
import skimage
import os
import matplotlib.pyplot as plt
import matplotlib.patches

from ipynb.fs.defs.q1 import *

# do not include any more libraries here!
# no opencv, no sklearn, etc!
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=UserWarning)

for img in os.listdir('images'):
```

```
    im1 = skimage.img_as_float(skimage.io.imread(os.path.join('images',img)))
    bboxes, bw = findLetters(im1)

    plt.imshow(bw, cmap="gray")
    for bbox in bboxes:
        minr, minc, maxr, maxc = bbox
        rect = matplotlib.patches.Rectangle((minc, minr), maxc - minc, maxr - r
                                fill=False, edgecolor='red', linewidth=2)
        plt.gca().add_patch(rect)
    plt.show()
    # find the rows using..RANSAC, counting, clustering, etc.
    # YOUR CODE HERE

    # raise NotImplementedError()
    # crop the bounding boxes
    # note.. before you flatten, transpose the image (that's how the dataset i
    # consider doing a square crop, and even using np.pad() to get your images
    # YOUR CODE HERE
    # raise NotImplementedError()
```

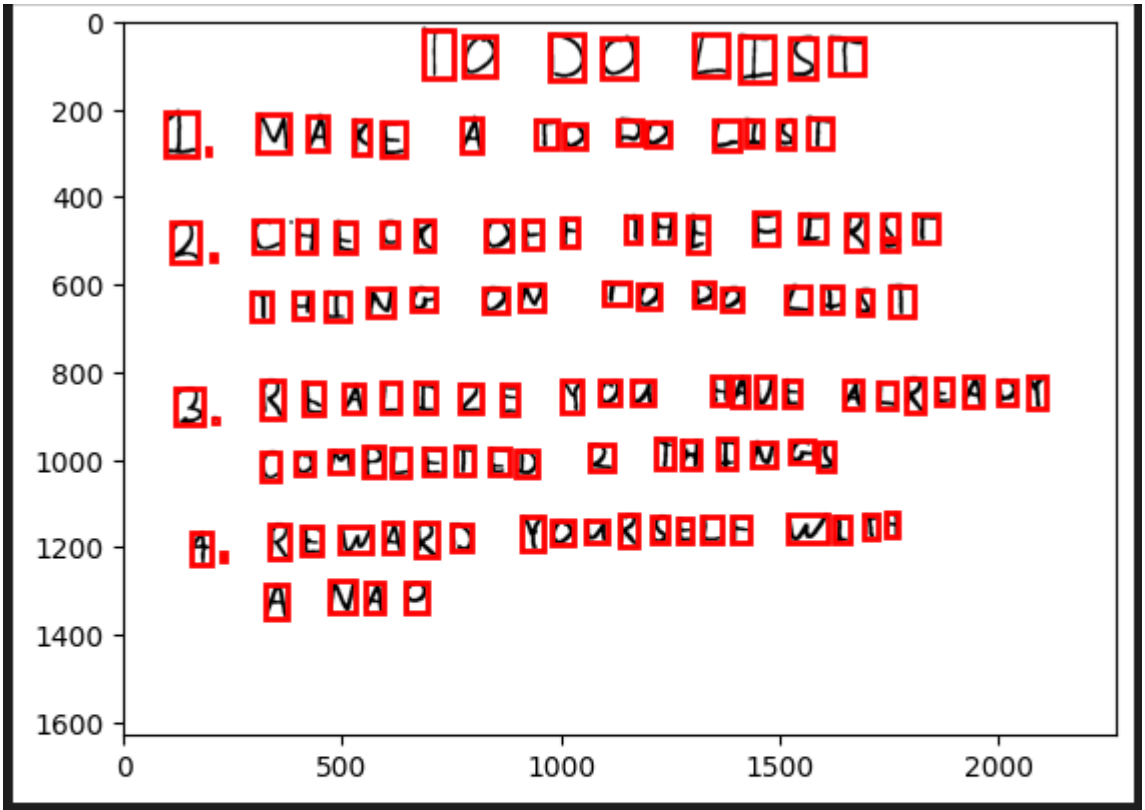



0
500
1000
1500
2000
2500
3000
0
500
1000
1500
2000
2500
3000
3500
4000

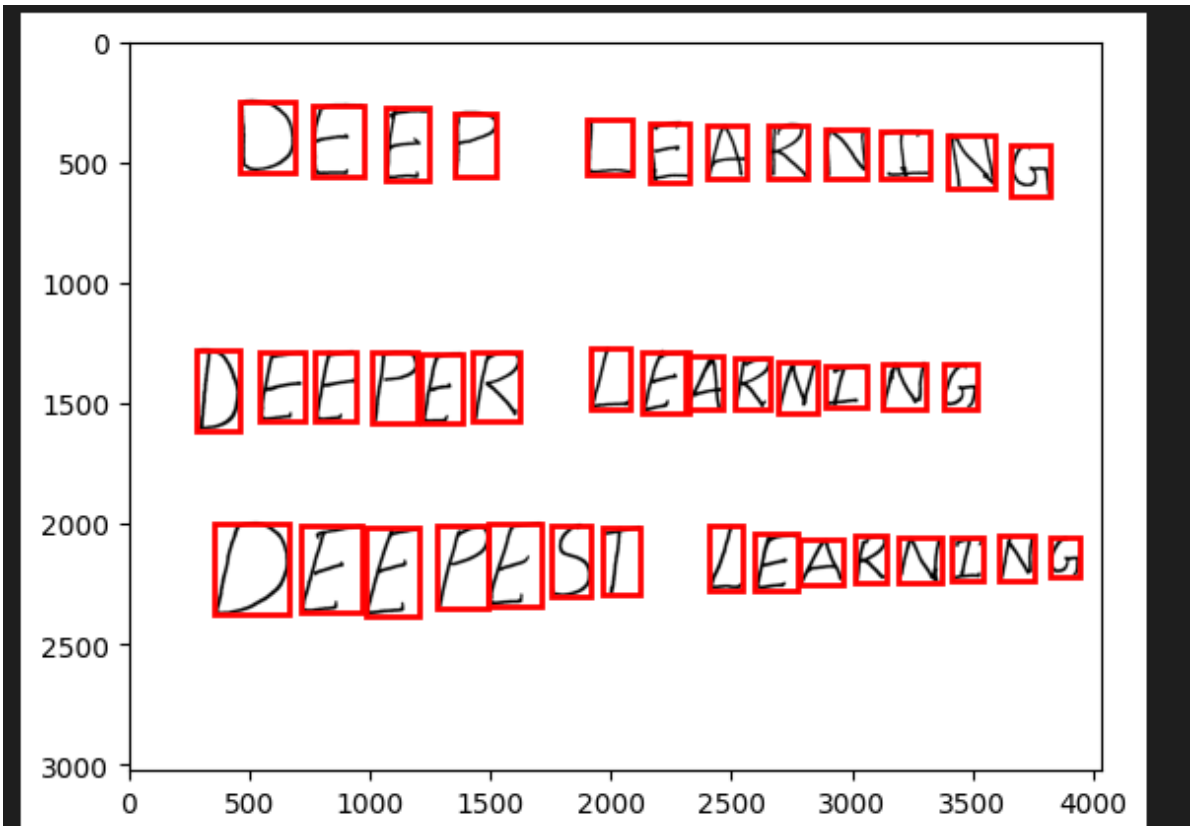

0
200
400
600
800
1000
1200
1400
1600
0
500
1000
1500
2000



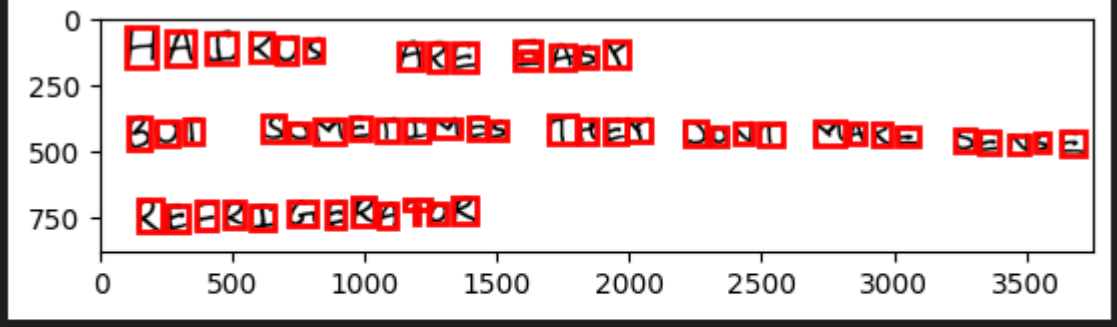
Q3.4



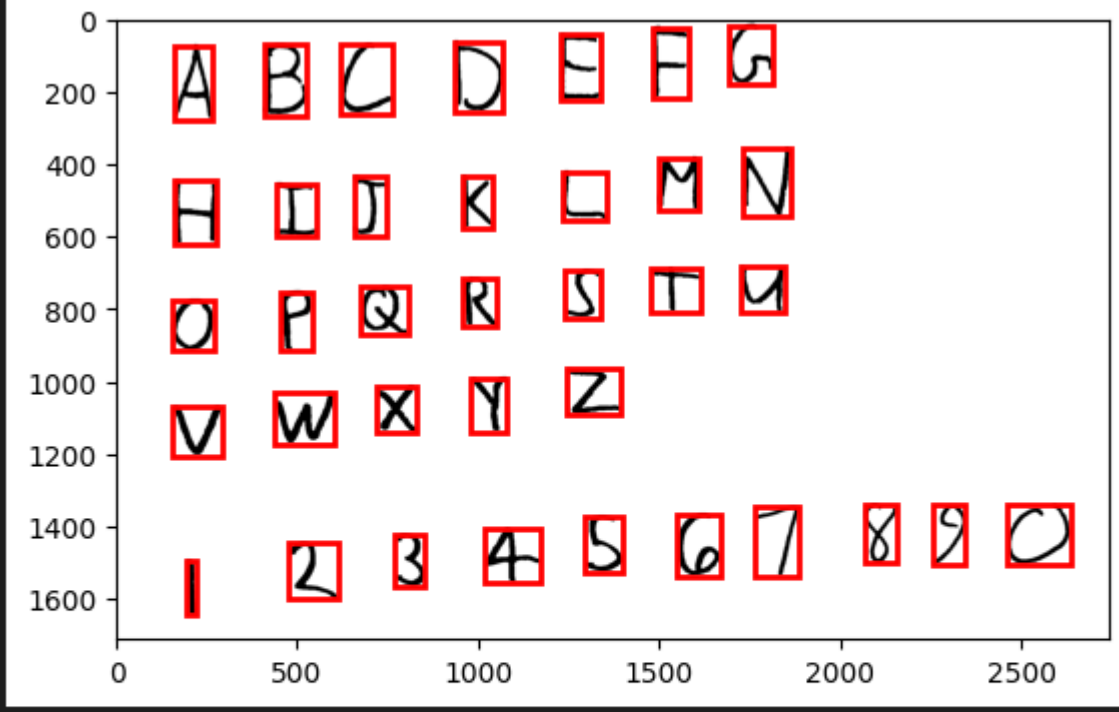
```
T0C0LIS
IXM1XEAT0D0LIST
IXTCHHIENCKG0OENETJOHIO0FEIXRSTHTT
3XCRO8MAYEEIETZ2EDY2OUT4HIAVNEFSALREADY
5X8EWXRDY0URSELFWITA
ANAP
```



```
EF9LEARMING
CEP9ERLEAR5ING
C5CPEST1EARNING
```




```
HAIKUSAREEASY
EUTSOMETIMESTREYDONTMAK2SENSE
REFRIGERA1M0R
```




```
2BCDEP
HIIKLMN
0PQRSTU
VWXYZ
1Z3FSG787C
```

In [ ]:
```
### Q3.4
# load the weights
```

```
# run the crops through your neural network and print them out
import skimage
import skimage.io
import pickle
import string
import numpy as np

letters = np.array([_ for _ in string.ascii_uppercase[:26]] + [str(_) for _ in
params = pickle.load(open('q2_weights.pickle','rb'))
# YOUR CODE HERE
for img in os.listdir('images'):
    im1 = skimage.img_as_float(skimage.io.imread(os.path.join('images',img)))
    bboxes, bw = findLetters(im1)

    plt.imshow(bw, cmap="gray")
    for bbox in bboxes:
        minr, minc, maxr, maxc = bbox
        rect = matplotlib.patches.Rectangle((minc, minr), maxc - minc, maxr - r
                                fill=False, edgecolor='red', linewidth=2)
        plt.gca().add_patch(rect)
    plt.show()

    sorted_temp = sorted(bboxes, key = lambda x:x[2])
    lines = 1
    max_chars_in_line = 0
    chars_in_line = 0
    # for i in range(1,len(sorted_temp)):
    #     if abs(sorted_temp[i-1][2] - sorted_temp[i][2]) > 100:
    #         # Append in the next row of sorted y
    #         lines += 1
    #         max_chars_in_line = max(chars_in_line, max_chars_in_line)
    #         chars_in_line = 0
    #     # sorted_y[x][y] = sorted_temp[i]
    #     chars_in_line += 1

    sorted_lines = []

    # print("lines = ", lines)
    # print("max cchars in line = ", max_chars_in_line)
    # sorted_lines = np.ones((lines, max_chars_in_line, 4))*np.nan
    curr_line = 0
    curr_char_idx = 0
    chars_in_one_line = []
    for j in range(1,len(sorted_temp)):
        if abs(sorted_temp[j-1][2] - sorted_temp[j][2]) > 100:
            sorted_lines.append(chars_in_one_line)
            chars_in_one_line = []
            curr_line += 1
            curr_char_idx = 0
        # print("currline = ", curr_line)
        # print("len = ", len(sorted_temp))
        chars_in_one_line.append(sorted_temp[j])
        curr_char_idx += 1

    sorted_lines.append(chars_in_one_line)

    for k in range(len(sorted_lines)):
```

```
        sorted_lines[k] = sorted(sorted_lines[k], key = lambda x:x[3])

    for i in sorted_lines:
        for j in i:
            patch = bw[j[0]:j[2],j[1]:j[3]]
            # plt.imshow(patch, cmap='gray')
            patch_ht = patch.shape[0]
            patch_wdt = patch.shape[1]
            pad_with = patch_ht-patch_wdt
            if pad_with > 0:
                patch = np.pad(patch, ((0,0),(pad_with//2, pad_with//2)), mode=
            elif pad_with < 0:
                patch = np.pad(patch, ((-pad_with//2, -pad_with//2),(0,0)), mo

            patch = np.pad(patch, ((25,25),(25,25)), mode='constant', constant_
            patch = skimage.morphology.erosion(patch)
            patch = skimage.transform.resize(patch, (32,32))
            # plt.imshow(patch, cmap='gray')
            patch = patch.transpose()

            # plt.show()
            patch = patch.reshape(1,1024)
            post_act = forward(patch,params,'layer1',sigmoid)
            pred_output = forward(post_act,params,'output',softmax)
            pred_idx = np.argmax(pred_output[0])
            detected_char = letters[pred_idx]
            print(detected_char, end='')
            # break
        # break
        print('')
    # break


# raise NotImplementedError()
```

Q4 (4 points)

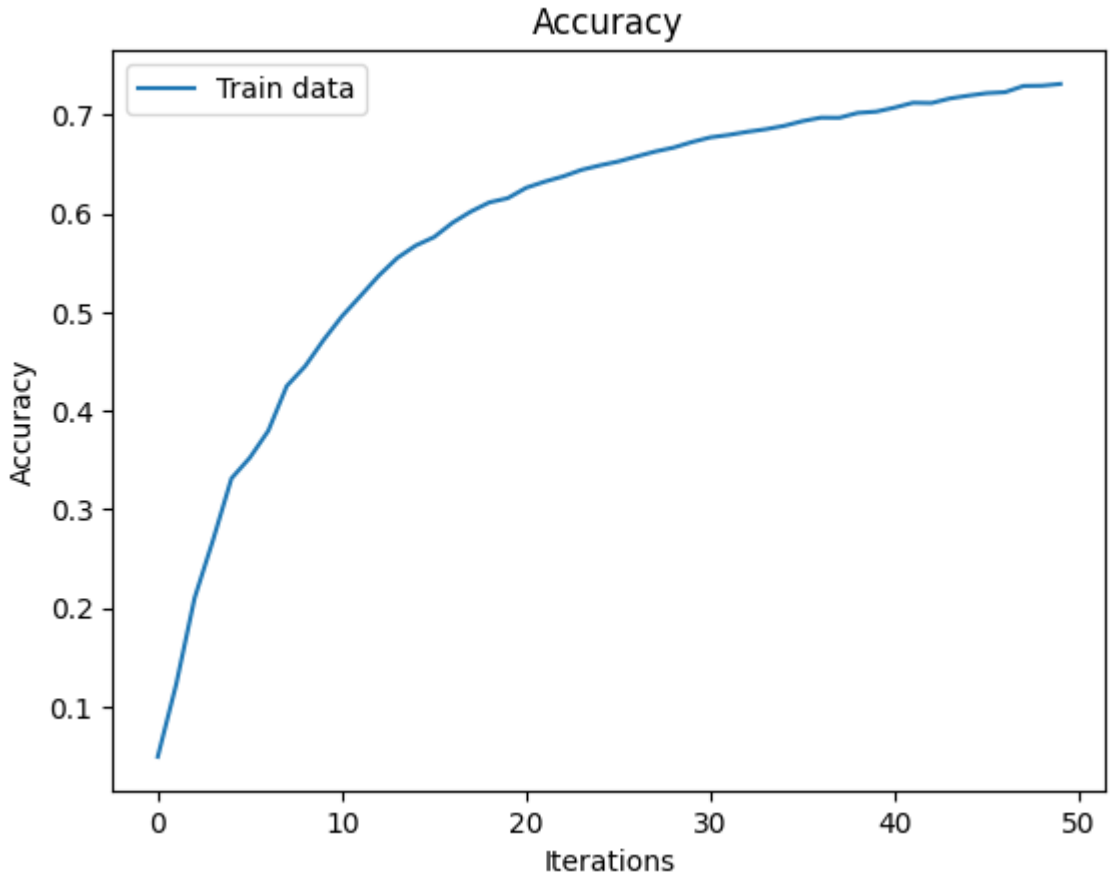
Given the sigmoid activation function $\sigma(x) = \frac{1}{1+e^{-x}}$, derive the gradient of the sigmoid function and show that it can be written solely as a function of $\sigma(x)$.

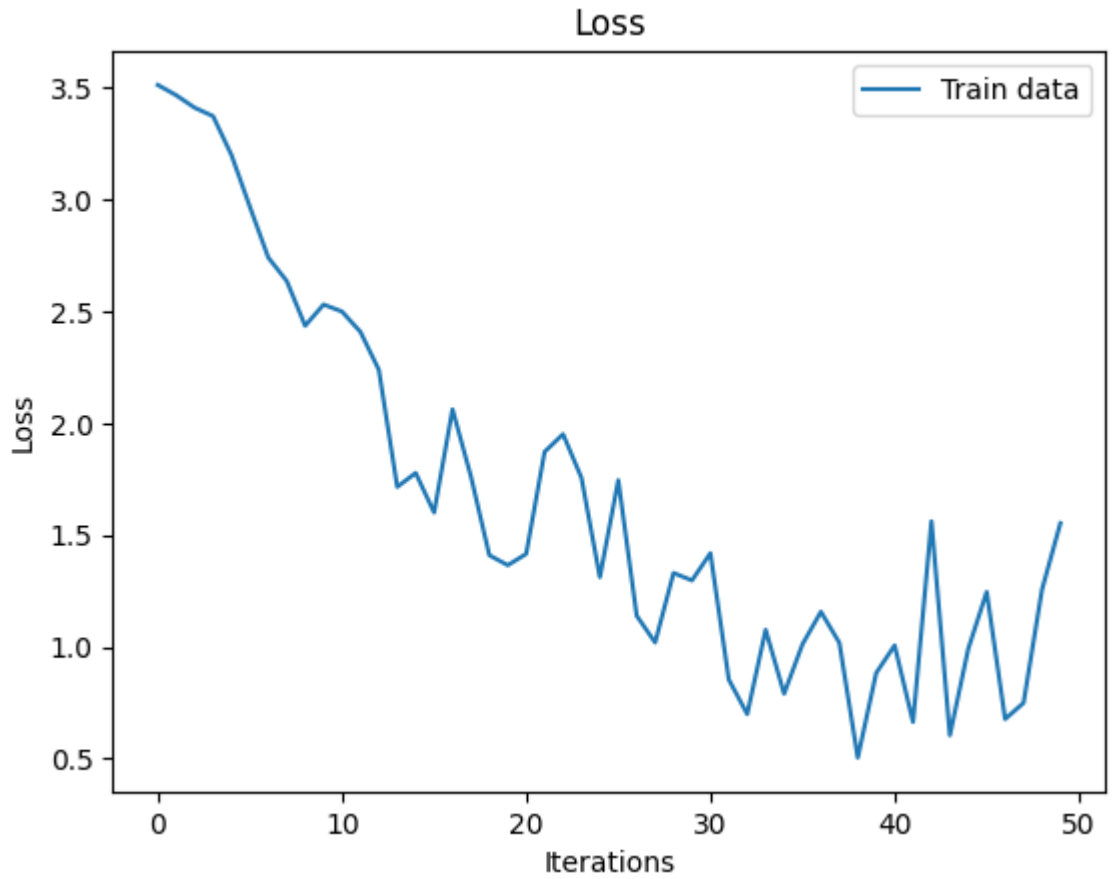
$$\frac{\partial \sigma(x)}{\partial x} = \frac{-(-e^{-x})}{(1+e^{-x})^2} = \sigma(x) * (1 - \sigma(x))$$

Therefore, the derivative of $\sigma(x)$ can be expressed as a function of $\sigma(x)$ itself!

Q4.1.1

Accuracy | Loss :----------:|:----------:





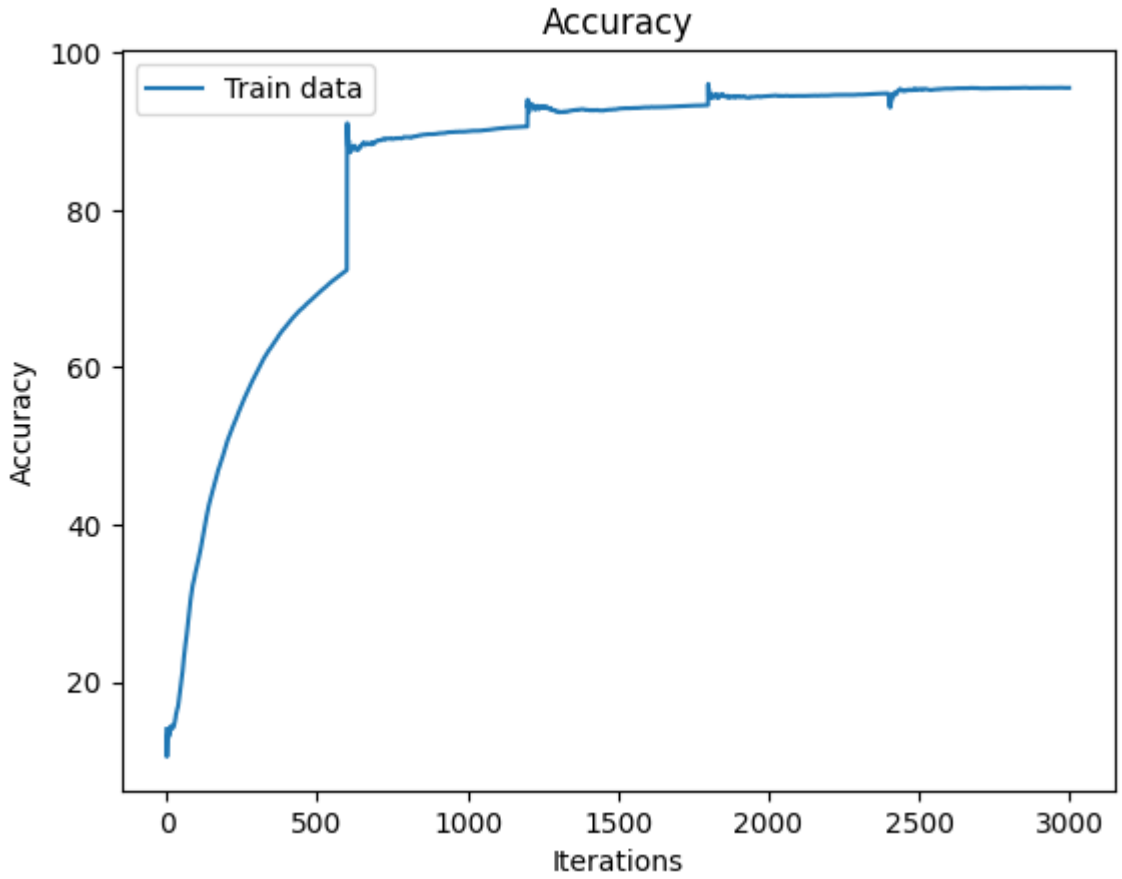
In [ ]:
```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
from torchvision import datasets
from torchvision.transforms import ToTensor
from ipynb.fs.defs.q1 import *
import scipy.io
import torch.nn.functional as F

training_data = scipy.io.loadmat('data/nist36_train.mat')
train_x, train_y = training_data['train_data'], training_data['train_labels']
train_x, train_y = torch.from_numpy(train_x).float(),torch.from_numpy(train_y)

# load_train_data = DataLoader(TensorDataset(train_x, train_y))

batch_size = 8

# Create data loaders.
train_dataloader = DataLoader(TensorDataset(train_x, train_y), batch_size=batcl

# Get cpu or gpu device for training.
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using {device} device")

# Define model
# class NeuralNetwork(nn.Module):
#     def __init__(self):
#         super().__init__()
#         self.flatten = nn.Flatten()
#         self.linear_relu_stack = nn.Sequential(
#             nn.Linear(1024, 64),
#             nn.Sigmoid(),
#             nn.Linear(64, 36),
#             nn.Softmax(dim=1),
#             # nn.Linear(512, 10)
#         )

#     def forward(self, x):
#         x = self.flatten(x)
#         logits = self.linear_relu_stack(x)
#         return logits

# model = NeuralNetwork().to(device)

class Network(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(1024, 64)
        self.output = nn.Linear(64, 36)

    def forward(self, x):
        x = self.hidden(x)
        x = torch.sigmoid(x)
        x = self.output(x)
        x = F.log_softmax(x,dim=1)

        return x
```

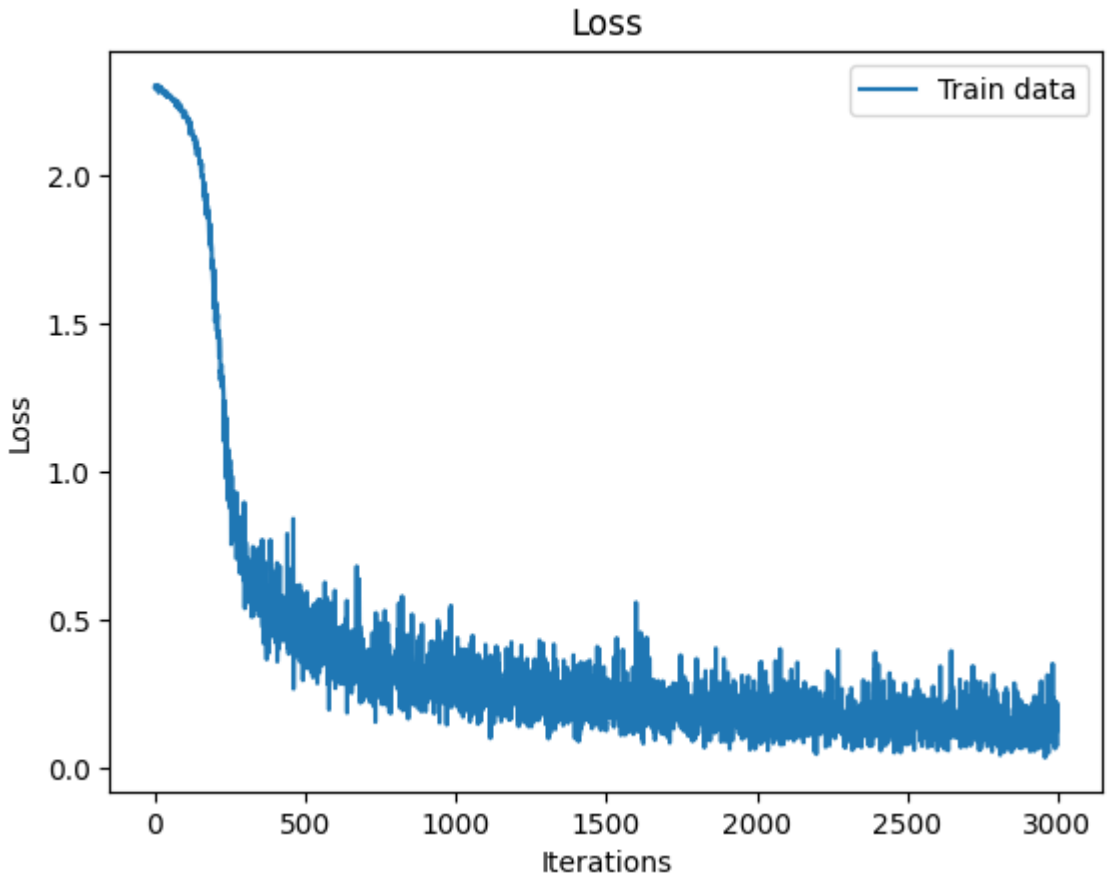
```
model = Network()

print(model)

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=5e-3)


def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    avg_acc = 0
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = nn.functional.cross_entropy(pred, y)
        _, acc = compute_loss_and_acc(y.detach().numpy(), pred.detach().numpy(
        avg_acc += acc/len(train_dataloader)
        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()


    print("Loss = ", loss)
    print("Acc = ", avg_acc)
    return avg_acc, loss.detach().numpy()


epochs = 50
acc_arr = []
loss_arr = []
itr_arr = []
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    acc, l = train(train_dataloader, model, loss_fn, optimizer)
    acc_arr.append(acc)
    loss_arr.append(l)
    itr_arr.append(t)
print("Done!")

import matplotlib.pyplot as plt
plt.plot(itr_arr,acc_arr, label = "Train data")
plt.xlabel('Iterations')
plt.ylabel('Accuracy')
plt.legend()
plt.title("Accuracy")
plt.show()

plt.plot(itr_arr,loss_arr, label = "Train data")
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
```

```
plt.title("Loss")
plt.show()
```

Q4.1.2

Accuracy | Loss :----------:|:----------:





In [ ]:
```
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.datasets as dsets
from torch.autograd import Variable

train_dataset = dsets.MNIST(root='./data',
                            train=True,
                            transform=transforms.ToTensor(),
                            download=True)

test_dataset = dsets.MNIST(root='./data',
                           train=False,
                           transform=transforms.ToTensor())

batch_size = 100
n_iters = 3000
num_epochs = n_iters / (len(train_dataset) / batch_size)
# num_epochs = int(num_epochs)
num_epochs = 5

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

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

```
class CNNModel(nn.Module):
    def __init__(self):
        super(CNNModel, self).__init__()

        # Convolution 1
        self.cnn1 = nn.Conv2d(1, 16, kernel_size=5, stride=1, padding=0)
        self.relu1 = nn.ReLU()

        # Max pool 1
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)

        # Convolution 2
        self.cnn2 = nn.Conv2d(16, 32, kernel_size=5, stride=1, padding=0)
        self.relu2 = nn.ReLU()

        # Max pool 2
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)

        # Fully connected 1 (readout)
        self.fc1 = nn.Linear(32 * 4 * 4, 10)
    # 100, 32, 4, 4
    def forward(self, x):
        # Convolution 1
        out = self.cnn1(x)
        out = self.relu1(out)

        # Max pool 1
        out = self.maxpool1(out)

        # Convolution 2
        out = self.cnn2(out)
        out = self.relu2(out)

        # Max pool 2
        out = self.maxpool2(out)

        # Resize
        # Original size: (100, 32, 7, 7)
        # out.size(0): 100
        # New out size: (100, 32*7*7)
        out = out.view(out.size(0), -1)

        # Linear function (readout)
        out = self.fc1(out)

        return out


model = CNNModel()

if torch.cuda.is_available():
    model.cuda()

loss_fn = nn.CrossEntropyLoss()

learning_rate = 0.01
```

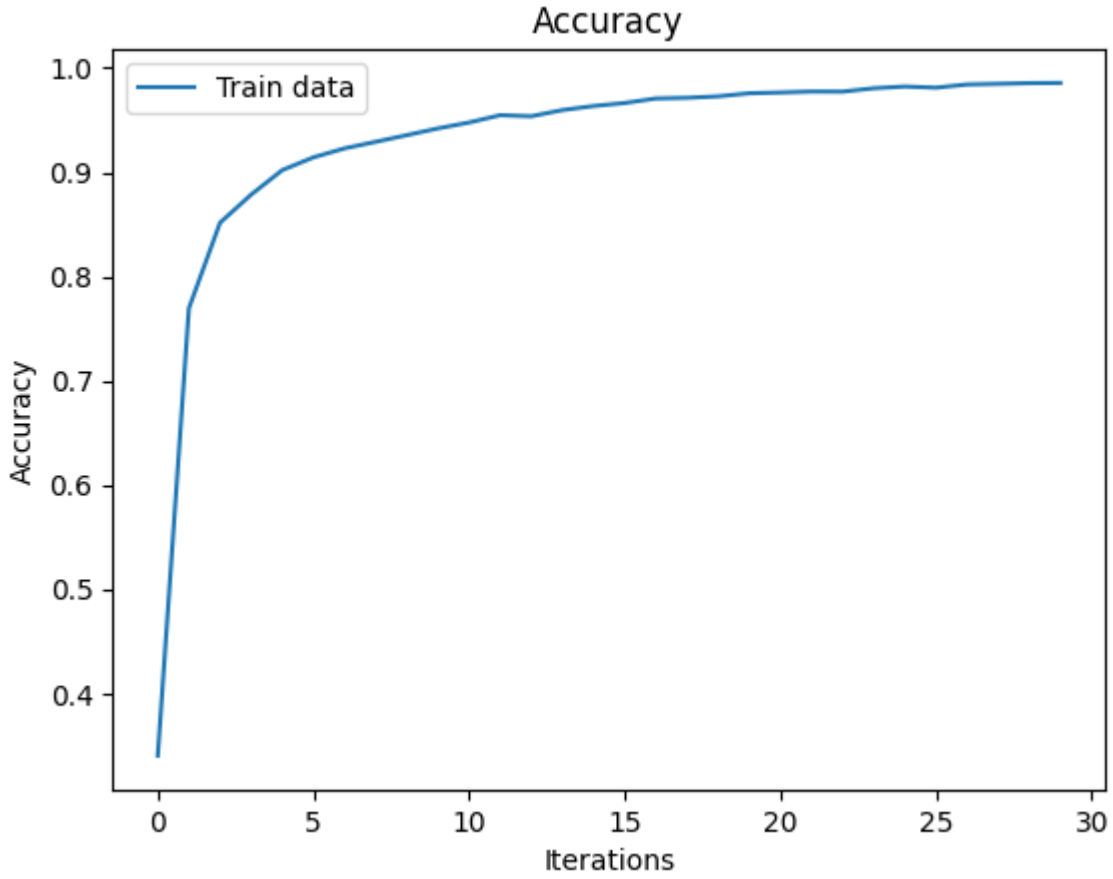
```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)


def train(train_loader, model, loss_fn, optimizer):
    # iter = 0
    # for epoch in range(num_epochs):
    train_correct = 0
    train_total = 0
    for i, (images, labels) in enumerate(train_loader):

        if torch.cuda.is_available():
            images = Variable(images.cuda())
            labels = Variable(labels.cuda())
        else:
            images = Variable(images)
            labels = Variable(labels)
        print("image shape = ",images.size())
        print("labels shape = ",labels.size())
        # print("images size = ", images)
        # print("labesl size = ", labels)
        # Clear gradients w.r.t. parameters
        optimizer.zero_grad()

        # Forward pass to get output/logits
        outputs = model(images)
        # print("outputs  = ", outputs)
        # Calculate Loss: softmax --> cross entropy loss
        train_loss = loss_fn(outputs, labels)

        # Getting gradients w.r.t. parameters
        train_loss.backward()

        # Updating parameters
        optimizer.step()

            # Get predictions from the maximum value
        _, predicted = torch.max(outputs.data, 1)

        # Total number of labels
        train_total += labels.size(0)

        # Total correct predictions
        if torch.cuda.is_available():
            train_correct += (predicted.cpu() == labels.cpu()).sum()
        else:
            train_correct += (predicted == labels).sum()

        train_accuracy = 100 * train_correct / train_total
        train_acc_arr.append(train_accuracy)
        train_loss_arr.append(train_loss.detach().numpy())
        # itr_arr.append(passes)
        # passes += 1
        print("Loss = ", train_loss)
        print("Acc = ", train_accuracy)
    # return train_acc_arr.detach().numpy(), train_loss_arr.detach().numpy()
```

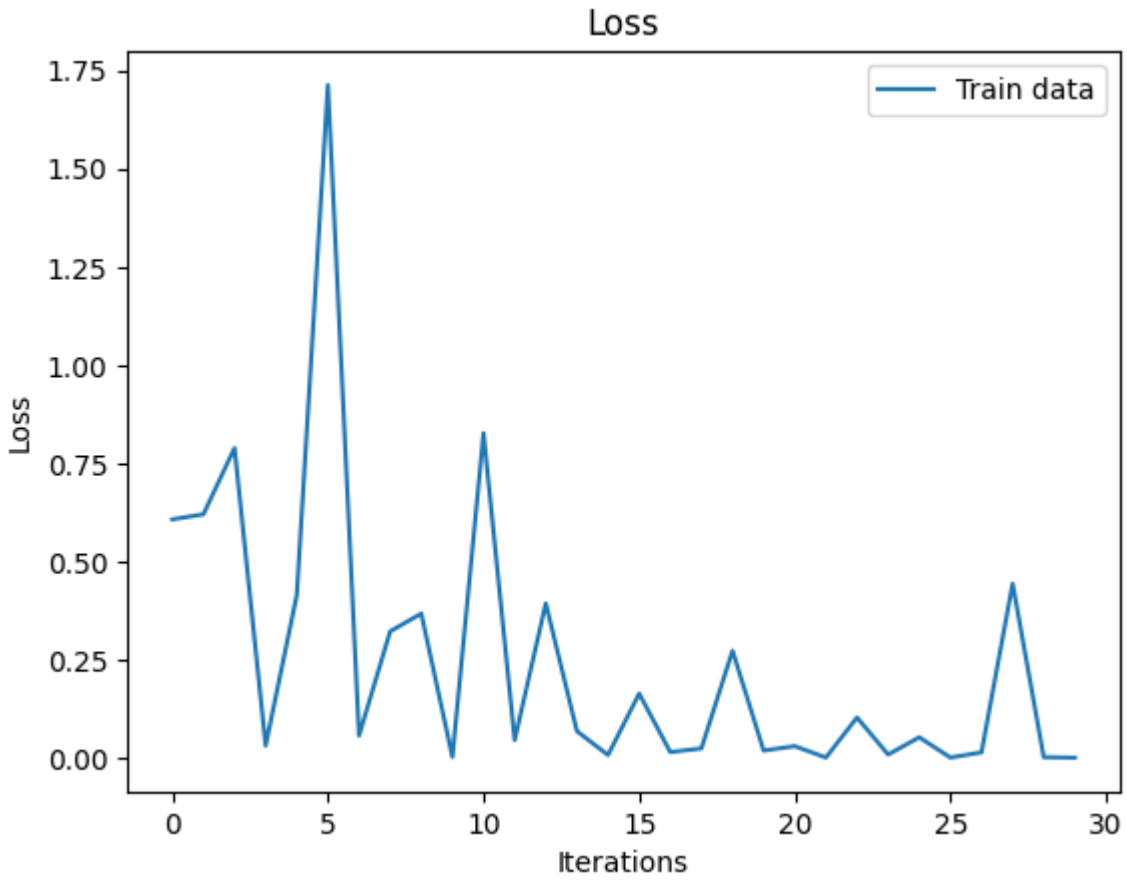
```
# def test(test_loader, model, loss_fn):
# # Calculate Accuracy
#     test_correct = 0
#     test_total = 0
#     # Iterate through test dataset
#     for images, labels in test_loader:

#         if torch.cuda.is_available():
#             images = Variable(images.cuda())
#         else:
#             images = Variable(images)

#         # Forward pass only to get logits/output
#         outputs = model(images)

#         # Loss
#         test_loss = loss_fn(outputs, labels)

#         # Get predictions from the maximum value
#         _, predicted = torch.max(outputs.data, 1)

#         # Total number of labels
#         test_total += labels.size(0)

#         # Total correct predictions
#         if torch.cuda.is_available():
#             test_correct += (predicted.cpu() == labels.cpu()).sum()
#         else:
#             test_correct += (predicted == labels).sum()

#     test_accuracy = 100 * test_correct / test_total


            # Print Loss
            # print('Iteration: {}'.format(iter))
            # print('Loss: {}'.format(loss.item()))
            # print('Accuracy: {}'.format(accuracy.item()))

train_acc_arr = []
train_loss_arr = []
itr_arr = []

for t in range(num_epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(train_loader, model, loss_fn, optimizer)
    # acc, l = train(train_loader, model, loss_fn, optimizer)
    # train_acc_arr.append(acc)
    # train_loss_arr.append(l)
    # itr_arr.append(t)
print("Done!")

import matplotlib.pyplot as plt
import numpy as np
plt.plot(np.arange(len(train_acc_arr)),train_acc_arr, label = "Train data")
plt.xlabel('Iterations')
plt.ylabel('Accuracy')
plt.legend()
```

```
plt.title("Accuracy")
plt.show()

plt.plot(np.arange(len(train_acc_arr)),train_loss_arr, label = "Train data")
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
plt.title("Loss")
plt.show()
```

Q4.1.3

Accuracy | Loss :----------:|:----------:





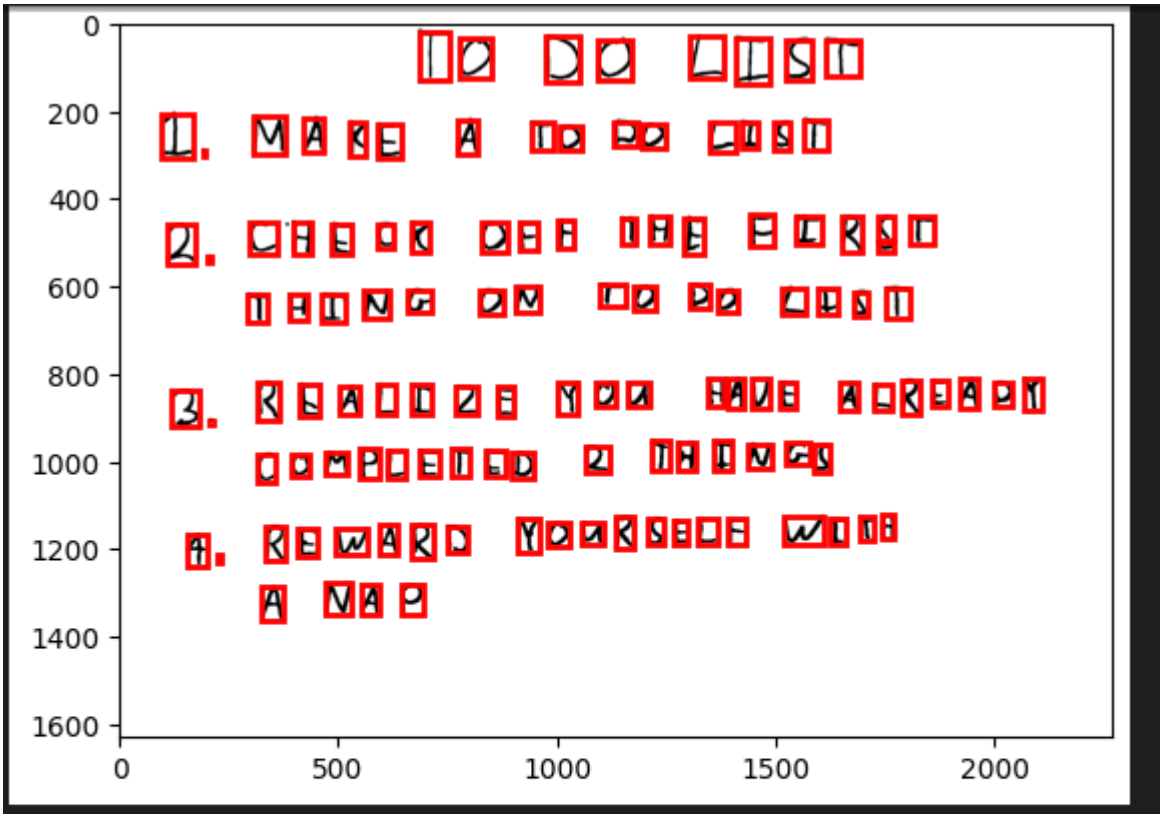
In [ ]:
```
### Q4.1.3
# YOUR CODE HERE
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
# from torchvision import datasets
from torchvision.transforms import ToTensor
from ipynb.fs.defs.q1 import *
import scipy.io
import torch.nn.functional as F
import torchvision.datasets as dsets
from torch.autograd import Variable


training_data = scipy.io.loadmat('data/nist36_train.mat')
train_x, train_y = training_data['train_data'], training_data['train_labels']
train_x, train_y = torch.from_numpy(train_x).float(),torch.from_numpy(train_y)

# load_train_data = DataLoader(TensorDataset(train_x, train_y))

batch_size = 8

# Create data loaders.
train_dataloader = DataLoader(TensorDataset(train_x, train_y), batch_size=batcl

class CNNModel(nn.Module):
    def __init__(self):
        super(CNNModel, self).__init__()
```

```
        # Convolution 1
        self.cnn1 = nn.Conv2d(1, 16, kernel_size=5, stride=1, padding=0)
        self.relu1 = nn.ReLU()

        # Max pool 1
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)

        # Convolution 2
        self.cnn2 = nn.Conv2d(16, 32, kernel_size=5, stride=1, padding=0)
        self.relu2 = nn.ReLU()

        # Max pool 2
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)

        # Fully connected 1 (readout)
        self.fc1 = nn.Linear(32*5*5, 36)
    # 32, 5, 5
    def forward(self, x):
        out = x.view(-1,32,32)
        out = torch.unsqueeze(out,1)
        # Convolution 1
        out = self.cnn1(out)
        out = self.relu1(out)

        # Max pool 1
        out = self.maxpool1(out)

        # Convolution 2
        out = self.cnn2(out)
        out = self.relu2(out)

        # Max pool 2
        out = self.maxpool2(out)

        # Resize
        # Original size: (100, 32, 7, 7)
        # out.size(0): 100
        # New out size: (100, 32*7*7)
        # print("maxpool shape = ", out.size())
        # out = out.view(out.size(0), -1)
        out = torch.flatten(out,1)

        # Linear function (readout)
        out = self.fc1(out)

        return out

model = CNNModel()

if torch.cuda.is_available():
    model.cuda()

loss_fn = nn.CrossEntropyLoss()

learning_rate = 0.01

optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

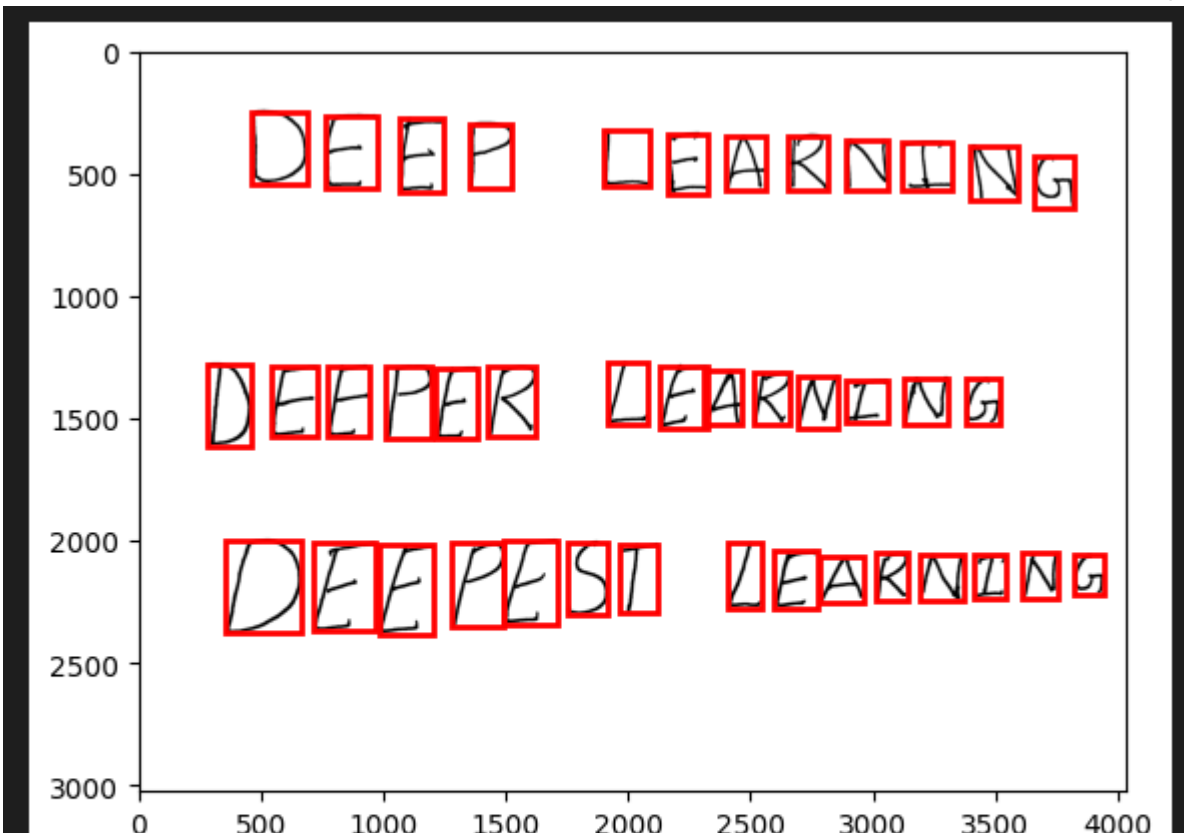
```
def train(train_loader, model, loss_fn, optimizer):
    # iter = 0
    # for epoch in range(num_epochs):
    train_correct = 0
    train_total = 0
    train_accuracy = 0
    for i, (images, labels) in enumerate(train_loader):

        if torch.cuda.is_available():
            images = Variable(images.cuda())
            labels = Variable(labels.cuda())
        else:
            images = Variable(images)
            labels = Variable(labels)

        # Clear gradients w.r.t. parameters
        optimizer.zero_grad()

        # Forward pass to get output/logits
        outputs = model(images)

        # Calculate Loss: softmax --> cross entropy loss
        train_loss = loss_fn(outputs, labels)

        # Getting gradients w.r.t. parameters
        train_loss.backward()

        # Updating parameters
        optimizer.step()

        _, acc = compute_loss_and_acc(labels.detach().numpy(), outputs.detach(
        train_accuracy += acc/len(train_loader)

            # Get predictions from the maximum value
        # _, predicted = torch.max(outputs.data, 1)

        # # Total number of labels
        # train_total += labels.size(0)

        # # Total correct predictions
        # if torch.cuda.is_available():
        #     train_correct += (predicted.cpu() == labels.cpu()).sum()
        # else:
        #     train_correct += (predicted == labels).sum()

        # train_accuracy = 100 * train_correct / train_total
        # train_acc_arr.append(train_accuracy)
        # train_loss_arr.append(train_loss.detach().numpy())
        # itr_arr.append(passes)
        # passes += 1
        print("Loss = ", train_loss)
        print("Acc = ", train_accuracy)
    return train_accuracy, train_loss.detach().numpy()
```

```
train_acc_arr = []
train_loss_arr = []
itr_arr = []
num_epochs = 30
for t in range(num_epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    # train(train_dataloader, model, loss_fn, optimizer)
    acc, l = train(train_dataloader, model, loss_fn, optimizer)
    train_acc_arr.append(acc)
    train_loss_arr.append(l)
    itr_arr.append(t)
print("Done!")

import matplotlib.pyplot as plt
import numpy as np
plt.plot(np.arange(len(train_acc_arr)),train_acc_arr, label = "Train data")
plt.xlabel('Iterations')
plt.ylabel('Accuracy')
plt.legend()
plt.title("Accuracy")
plt.show()

plt.plot(np.arange(len(train_acc_arr)),train_loss_arr, label = "Train data")
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
plt.title("Loss")
plt.show()
# raise NotImplementedError()
```

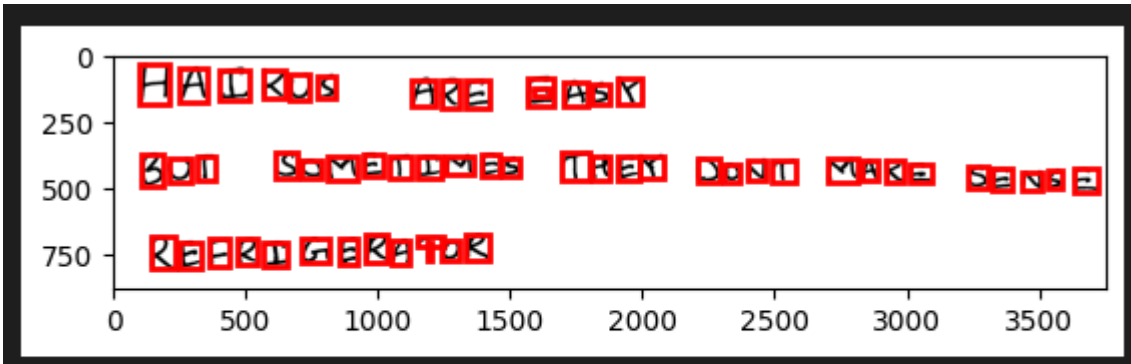
Q4.1.4



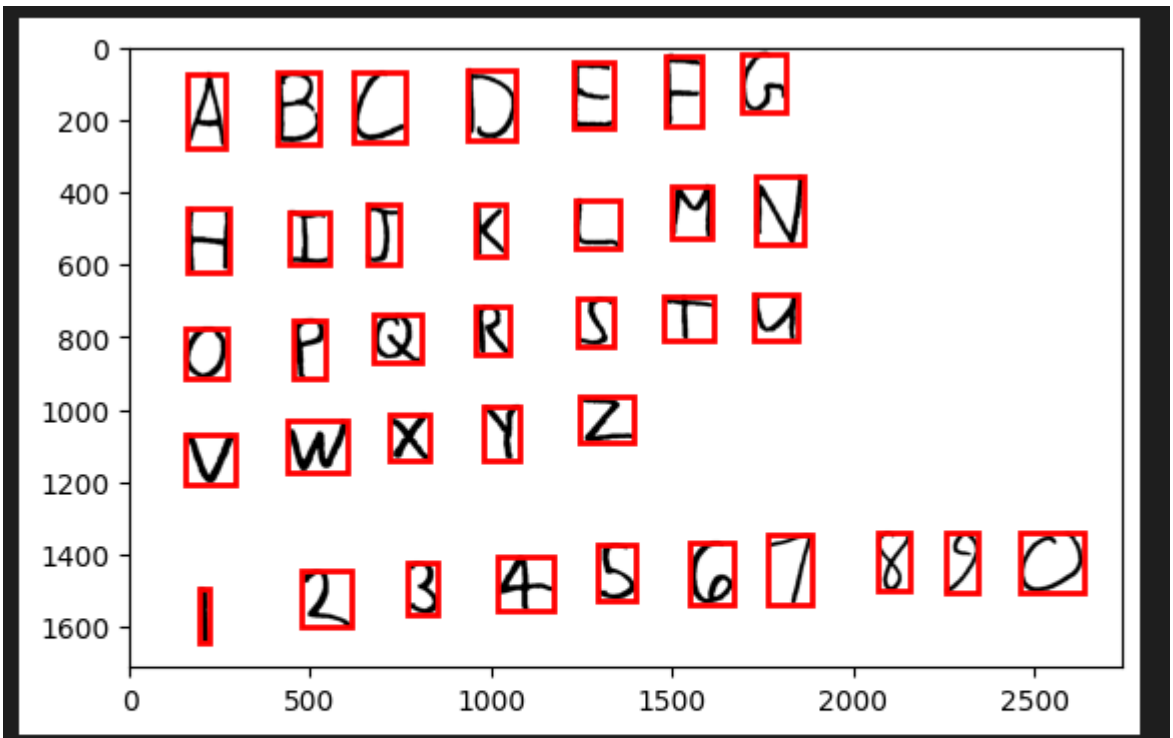

```
T0D0LIS
IIMhKEhT0D0LIST
2IICHHIENCKE00FNFrI0HED0FLItRSrtTT
3ICR0FHhPLLIET2tEDY20UIHHIhVNEEShLREADY
4IREWhRDY0URSELFWIIH
ANhP
```



```
FFPWEARNING
DFFPFKLEARNING
NFFPFST1EAKNING
```



```
HAIKUSAREEASX
BUTSOMET2MESTREXDDNTMAKESENSE
REFR2GERAIM0R
```



```
ABCDEF
HIJKLMN
0PQRSTV
VWXYZ
1234S6789Q
```

In [ ]:
```
# YOUR CODE HERE
import torch
```

```
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.datasets as dsets
from torch.autograd import Variable

train_dataset = dsets.EMNIST(root='./data',
                             train=True,
                             transform=transforms.ToTensor(),
                             download=True,
                             split="balanced")

# test_dataset = dsets.EMNIST(root='./data',
#                             train=False,
#                             transform=transforms.ToTensor())


batch_size = 100
n_iters = 3000
num_epochs = n_iters / (len(train_dataset) / batch_size)
# num_epochs = int(num_epochs)
num_epochs = 5

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

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

class CNNModel(nn.Module):
    def __init__(self):
        super(CNNModel, self).__init__()

        # Convolution 1
        self.cnn1 = nn.Conv2d(1, 16, kernel_size=5, stride=1, padding=0)
        self.relu1 = nn.ReLU()

        # Max pool 1
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)

        # Convolution 2
        self.cnn2 = nn.Conv2d(16, 32, kernel_size=5, stride=1, padding=0)
        self.relu2 = nn.ReLU()

        # Max pool 2
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)

        # Fully connected 1 (readout)
        self.fc1 = nn.Linear(32 * 4 * 4, 47)
    # 100, 32, 4, 4
    def forward(self, x):
        # Convolution 1
        out = self.cnn1(x)
        out = self.relu1(out)

        # Max pool 1
```

```
        out = self.maxpool1(out)

        # Convolution 2
        out = self.cnn2(out)
        out = self.relu2(out)

        # Max pool 2
        out = self.maxpool2(out)
        # print("maxpool shape = ", out.size())
        # Resize
        # Original size: (100, 32, 7, 7)
        # out.size(0): 100
        # New out size: (100, 32*7*7)
        out = out.view(out.size(0), -1)

        # Linear function (readout)
        out = self.fc1(out)

        return out


model = CNNModel()

if torch.cuda.is_available():
    model.cuda()

loss_fn = nn.CrossEntropyLoss()

learning_rate = 0.01

optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)


def train(train_loader, model, loss_fn, optimizer):
    # iter = 0
    # for epoch in range(num_epochs):
    train_correct = 0
    train_total = 0
    for i, (images, labels) in enumerate(train_loader):

        if torch.cuda.is_available():
            images = Variable(images.cuda())
            labels = Variable(labels.cuda())
        else:
            images = Variable(images)
            labels = Variable(labels)
        # print("images size = ", images.size())
        # print("labesl size = ", labels.size())
        # Clear gradients w.r.t. parameters
        optimizer.zero_grad()

        # Forward pass to get output/logits
        outputs = model(images)
        # print("outputs  = ", outputs)
        # Calculate Loss: softmax --> cross entropy loss
        train_loss = loss_fn(outputs, labels)
```

```
        # Getting gradients w.r.t. parameters
        train_loss.backward()

        # Updating parameters
        optimizer.step()

            # Get predictions from the maximum value
        _, predicted = torch.max(outputs.data, 1)

        # Total number of labels
        train_total += labels.size(0)

        # Total correct predictions
        if torch.cuda.is_available():
            train_correct += (predicted.cpu() == labels.cpu()).sum()
        else:
            train_correct += (predicted == labels).sum()

        train_accuracy = 100 * train_correct / train_total
        train_acc_arr.append(train_accuracy)
        train_loss_arr.append(train_loss.detach().numpy())
        # itr_arr.append(passes)
        # passes += 1
        # print("Loss = ", train_loss)
        # print("Acc = ", train_accuracy)
    # return train_acc_arr.detach().numpy(), train_loss_arr

train_acc_arr = []
train_loss_arr = []
itr_arr = []

for t in range(num_epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(train_loader, model, loss_fn, optimizer)
    # acc, l = train(train_loader, model, loss_fn, optimizer)
    # train_acc_arr.append(acc)
    # train_loss_arr.append(l)
    # itr_arr.append(t)
print("Done!")

import matplotlib.pyplot as plt
import numpy as np
plt.plot(np.arange(len(train_acc_arr)),train_acc_arr, label = "Train data")
plt.xlabel('Iterations')
plt.ylabel('Accuracy')
plt.legend()
plt.title("Accuracy")
plt.show()

plt.plot(np.arange(len(train_acc_arr)),train_loss_arr, label = "Train data")
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
plt.title("Loss")
plt.show()
# raise NotImplementedError()
```

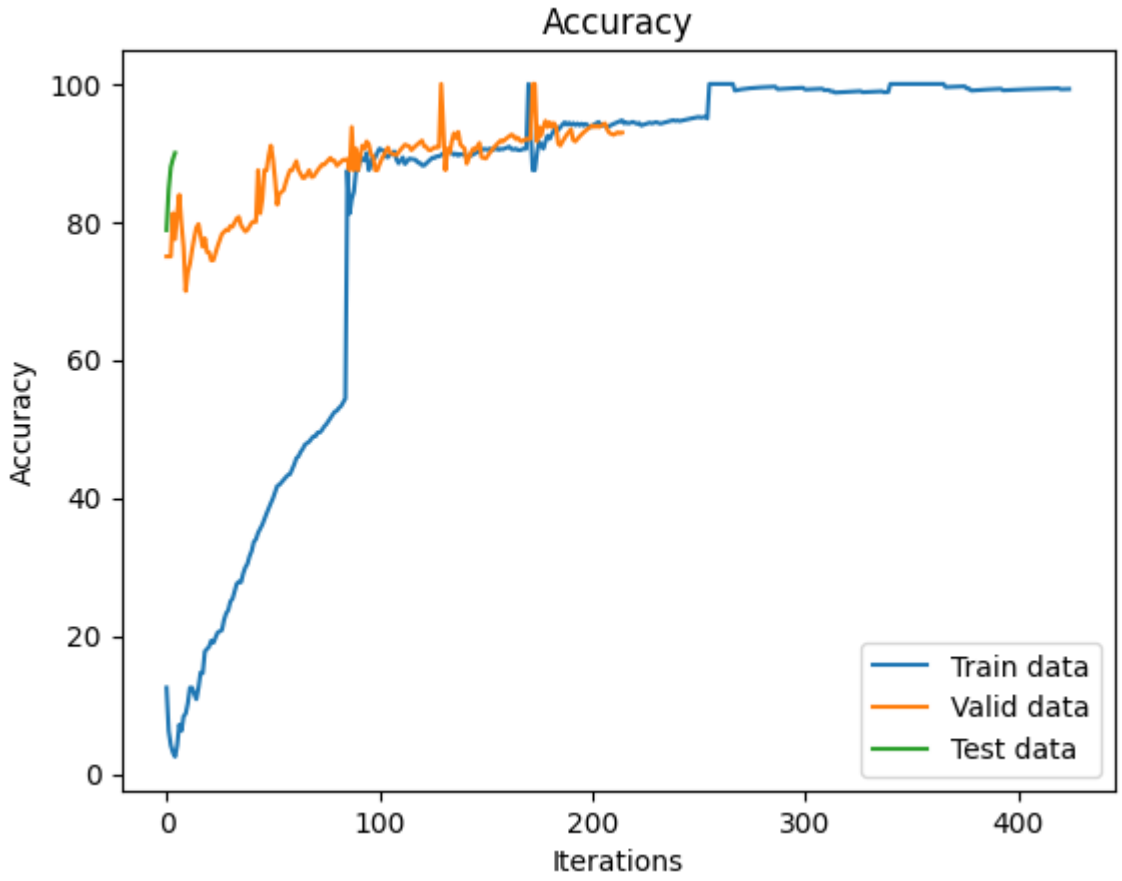
In [ ]:
```
# load the weights
# run the crops through your neural network and print them out
import skimage
import skimage.io
import pickle
import string
import numpy as np
from ipynb.fs.defs.q3 import *

# letters = np.array([_ for _ in string.ascii_uppercase[:26]] + [str(_) for _ 
letters = train_dataset.classes
params = pickle.load(open('q2_weights.pickle','rb'))
# YOUR CODE HERE
for img in os.listdir('images'):
    im1 = skimage.img_as_float(skimage.io.imread(os.path.join('images',img)))
    bboxes, bw = findLetters(im1)

    plt.imshow(bw, cmap="gray")
    for bbox in bboxes:
        minr, minc, maxr, maxc = bbox
        rect = matplotlib.patches.Rectangle((minc, minr), maxc - minc, maxr - m
                                fill=False, edgecolor='red', linewidth=2)
        plt.gca().add_patch(rect)
    plt.show()

    sorted_temp = sorted(bboxes, key = lambda x:x[2])
    lines = 1
    max_chars_in_line = 0
    chars_in_line = 0
    # for i in range(1,len(sorted_temp)):
    #     if abs(sorted_temp[i-1][2] - sorted_temp[i][2]) > 100:
    #         # Append in the next row of sorted y
    #         lines += 1
    #         max_chars_in_line = max(chars_in_line, max_chars_in_line)
    #         chars_in_line = 0
    #     # sorted_y[x][y] = sorted_temp[i]
    #     chars_in_line += 1

    sorted_lines = []

    # print("lines = ", lines)
    # print("max cchars in line = ", max_chars_in_line)
    # sorted_lines = np.ones((lines, max_chars_in_line, 4))*np.nan
    curr_line = 0
    curr_char_idx = 0
    chars_in_one_line = []
    for j in range(1,len(sorted_temp)):
        if abs(sorted_temp[j-1][2] - sorted_temp[j][2]) > 100:
            sorted_lines.append(chars_in_one_line)
            chars_in_one_line = []
            curr_line += 1
            curr_char_idx = 0
        # print("currline = ", curr_line)
        # print("len = ", len(sorted_temp))
        chars_in_one_line.append(sorted_temp[j])
        curr_char_idx += 1
```

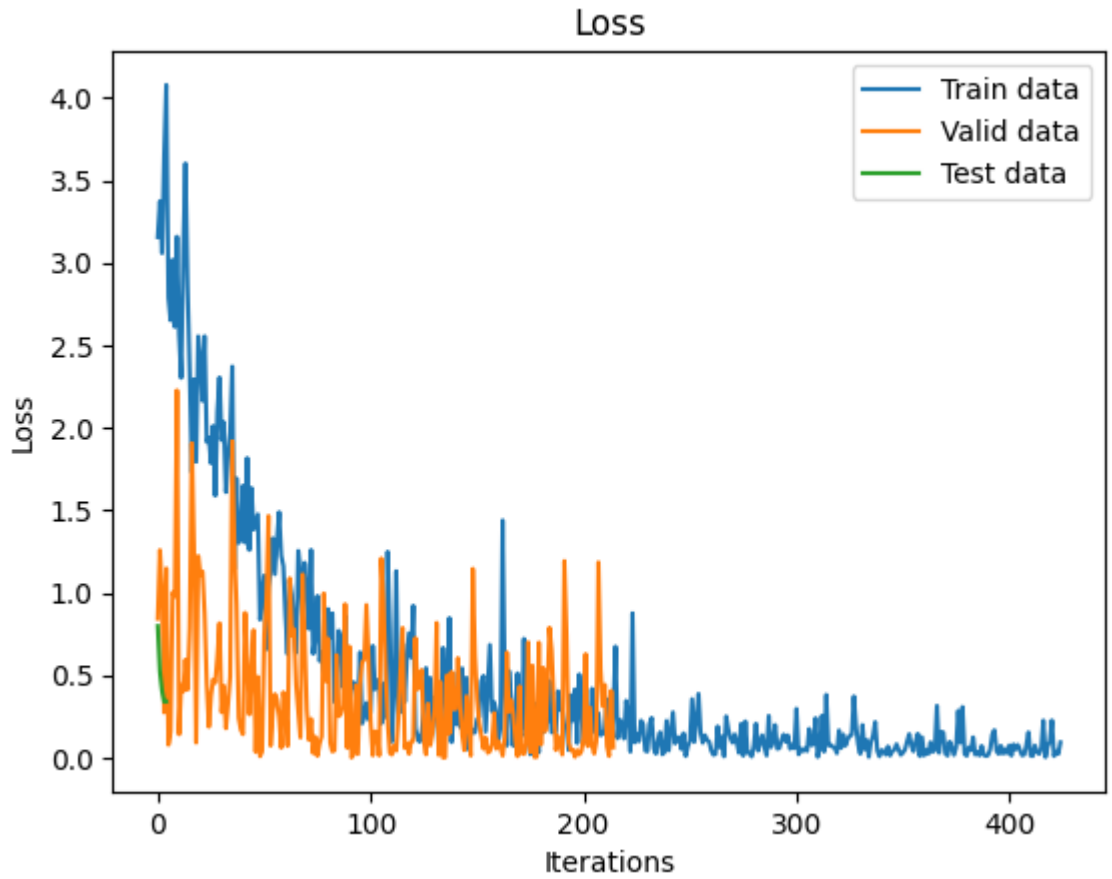
```
    sorted_lines.append(chars_in_one_line)

    for k in range(len(sorted_lines)):
        sorted_lines[k] = sorted(sorted_lines[k], key = lambda x:x[3])

    for i in sorted_lines:
        for j in i:
            patch = bw[j[0]:j[2],j[1]:j[3]]
            # plt.imshow(patch, cmap='gray')
            patch_ht = patch.shape[0]
            patch_wdt = patch.shape[1]
            pad_with = patch_ht-patch_wdt
            if pad_with > 0:
                patch = np.pad(patch, ((0,0),(pad_with//2, pad_with//2)), mode=
            elif pad_with < 0:
                patch = np.pad(patch, ((-pad_with//2, -pad_with//2),(0,0)), mo

            patch = np.pad(patch, ((25,25),(25,25)), mode='constant', constant_
            patch = skimage.morphology.erosion(patch)
            patch = skimage.transform.resize(patch, (32,32))
            # plt.imshow(patch, cmap='gray')
            patch = patch.transpose()

            # plt.show()
            patch = patch.reshape(1,1,32,32)
            patch = patch[:,:,2:-2,2:-2]
            patch = 1-patch
            # post_act = forward(patch,params,'layer1',sigmoid)
            # pred_output = forward(post_act,params,'output',softmax)
            patch = torch.from_numpy(patch).float()
            output = model(patch)
            output = output.detach().numpy()
            pred_idx = np.argmax(output[0])
            detected_char = letters[pred_idx]
            print(detected_char, end='')
            # break
        # break
        print('')
    # break


# raise NotImplementedError()
```

Q4.2

For squeezenet:

Accuracy | Loss :----------:|:----------:





In [ ]:
```
# YOUR CODE HERE
import torch
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.datasets as dsets
from torch.autograd import Variable

model = torch.hub.load('pytorch/vision:v0.10.0', 'squeezenet1_1', pretrained=T

final_conv = nn.Conv2d(512, 17, kernel_size=1)
model.classifier = nn.Sequential(
        nn.Dropout(p=0.5), final_conv, nn.ReLU(inplace=True), nn.AdaptiveAvgPo
    )

for param in model.parameters():
    param.requires_grad = False
for param in model.classifier.parameters():
    param.requires_grad = True

# Construct an Optimizer object for updating the last layer only.
optimizer = torch.optim.Adam(model.classifier.parameters(), lr=1e-3)

from PIL import Image
from torchvision import transforms
# input_image = Image.open(filename)
preprocess = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]
])
# input_tensor = preprocess(input_image)
# input_batch = input_tensor.unsqueeze(0)
batch_size = 8
train_dataloader = torch.utils.data.DataLoader(dsets.ImageFolder('./data/oxfor
valid_dataloader = torch.utils.data.DataLoader(dsets.ImageFolder('./data/oxfor
test_dataloader = torch.utils.data.DataLoader(dsets.ImageFolder('./data/oxford
# print("len of data set = ", valid_dataloader.size())

loss_fn = nn.CrossEntropyLoss()

def train(train_loader, model, loss_fn, optimizer):
    # iter = 0
    # for epoch in range(num_epochs):
    train_correct = 0
    train_total = 0
    for i, (images, labels) in enumerate(train_loader):

        if torch.cuda.is_available():
            images = Variable(images.cuda())
            labels = Variable(labels.cuda())
        else:
            images = Variable(images)
            labels = Variable(labels)
        # print("images size = ", images.size())
        # print("labesl size = ", labels.size())
```

```
        # Clear gradients w.r.t. parameters
        optimizer.zero_grad()

        # Forward pass to get output/logits
        outputs = model(images)
        # print("outputs  = ", outputs)
        # Calculate Loss: softmax --> cross entropy loss
        train_loss = loss_fn(outputs, labels)

        # Getting gradients w.r.t. parameters
        train_loss.backward()

        # Updating parameters
        optimizer.step()

            # Get predictions from the maximum value
        _, predicted = torch.max(outputs.data, 1)

        # Total number of labels
        train_total += labels.size(0)

        # Total correct predictions
        if torch.cuda.is_available():
            train_correct += (predicted.cpu() == labels.cpu()).sum()
        else:
            train_correct += (predicted == labels).sum()

        train_accuracy = 100 * train_correct / train_total
        train_acc_arr.append(train_accuracy)
        train_loss_arr.append(train_loss.detach().numpy())
        # itr_arr.append(passes)
        # passes += 1
        print("Loss = ", train_loss)
        print("Acc = ", train_accuracy)
    # return train_acc_arr.detach().numpy(), train_loss_arr.detach().numpy()

def valid(valid_loader, model, loss_fn):
    # iter = 0
    # for epoch in range(num_epochs):
    valid_correct = 0
    valid_total = 0
    for i, (images, labels) in enumerate(valid_loader):

        if torch.cuda.is_available():
            images = Variable(images.cuda())
            labels = Variable(labels.cuda())
        else:
            images = Variable(images)
            labels = Variable(labels)
        # print("images size = ", images.size())
        # print("labesl size = ", labels.size())
        # Clear gradients w.r.t. parameters
        # optimizer.zero_grad()

        # Forward pass to get output/logits
        outputs = model(images)
        # print("outputs  = ", outputs)
```

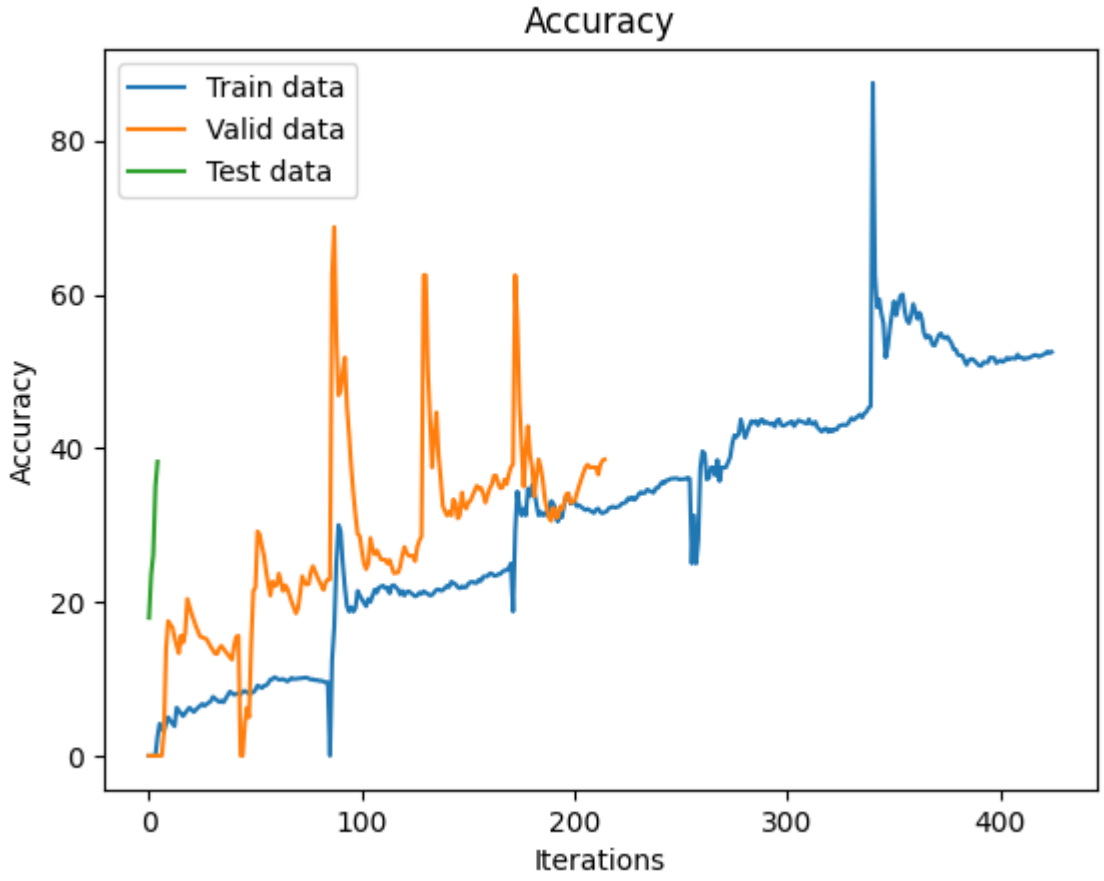
```
        # Calculate Loss: softmax --> cross entropy loss
        valid_loss = loss_fn(outputs, labels)

        # Getting gradients w.r.t. parameters
        # valid_loss.backward()

        # Updating parameters
        # optimizer.step()

            # Get predictions from the maximum value
        _, predicted = torch.max(outputs.data, 1)

        # Total number of labels
        valid_total += labels.size(0)

        # Total correct predictions
        if torch.cuda.is_available():
            valid_correct += (predicted.cpu() == labels.cpu()).sum()
        else:
            valid_correct += (predicted == labels).sum()

        valid_accuracy = 100 * valid_correct / valid_total
        valid_acc_arr.append(valid_accuracy)
        valid_loss_arr.append(valid_loss.detach().numpy())
        # itr_arr.append(passes)
        # passes += 1
        print("Loss = ", valid_loss)
        print("Acc = ", valid_accuracy)
    # return valid_acc_arr.detach().numpy(), valid_loss_arr.detach().numpy()


def test(test_loader, model, loss_fn):
# Calculate Accuracy
    test_correct = 0
    test_total = 0
    # Iterate through test dataset
    for images, labels in test_loader:

        if torch.cuda.is_available():
            images = Variable(images.cuda())
        else:
            images = Variable(images)

        # Forward pass only to get logits/output
        outputs = model(images)

        # Loss
        test_loss = loss_fn(outputs, labels)

        # Get predictions from the maximum value
        _, predicted = torch.max(outputs.data, 1)

        # Total number of labels
        test_total += labels.size(0)

        # Total correct predictions
        if torch.cuda.is_available():
```

```
            test_correct += (predicted.cpu() == labels.cpu()).sum()
        else:
            test_correct += (predicted == labels).sum()

        test_accuracy = 100 * test_correct / test_total
        test_acc_arr.append(test_accuracy)
        test_loss_arr.append(test_loss.detach().numpy())



            # Print Loss
            # print('Iteration: {}'.format(iter))
            # print('Loss: {}'.format(loss.item()))
            # print('Accuracy: {}'.format(accuracy.item()))

train_acc_arr = []
valid_acc_arr = []
test_acc_arr = []
train_loss_arr = []
test_loss_arr =[]
valid_loss_arr = []
itr_arr = []

num_epochs = 5
for t in range(num_epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(train_dataloader, model, loss_fn, optimizer)
    valid(valid_dataloader, model, loss_fn)
    test(test_dataloader, model, loss_fn)
    # acc, l = train(train_loader, model, loss_fn, optimizer)
    # train_acc_arr.append(acc)
    # train_loss_arr.append(l)
    # itr_arr.append(t)
print("Done!")

import matplotlib.pyplot as plt
import numpy as np
plt.plot(np.arange(len(train_acc_arr)),train_acc_arr, label = "Train data")
plt.plot(np.arange(len(valid_acc_arr)),valid_acc_arr, label = "Valid data")
plt.plot(np.arange(len(test_acc_arr)),test_acc_arr, label = "Test data")
plt.xlabel('Iterations')
plt.ylabel('Accuracy')
plt.legend()
plt.title("Accuracy")
plt.show()

plt.plot(np.arange(len(train_acc_arr)),train_loss_arr, label = "Train data")
plt.plot(np.arange(len(valid_acc_arr)),valid_loss_arr, label = "Valid data")
plt.plot(np.arange(len(test_acc_arr)),test_loss_arr, label = "Test data")
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
plt.title("Loss")
plt.show()
# raise NotImplementedError()
# raise NotImplementedError()
```

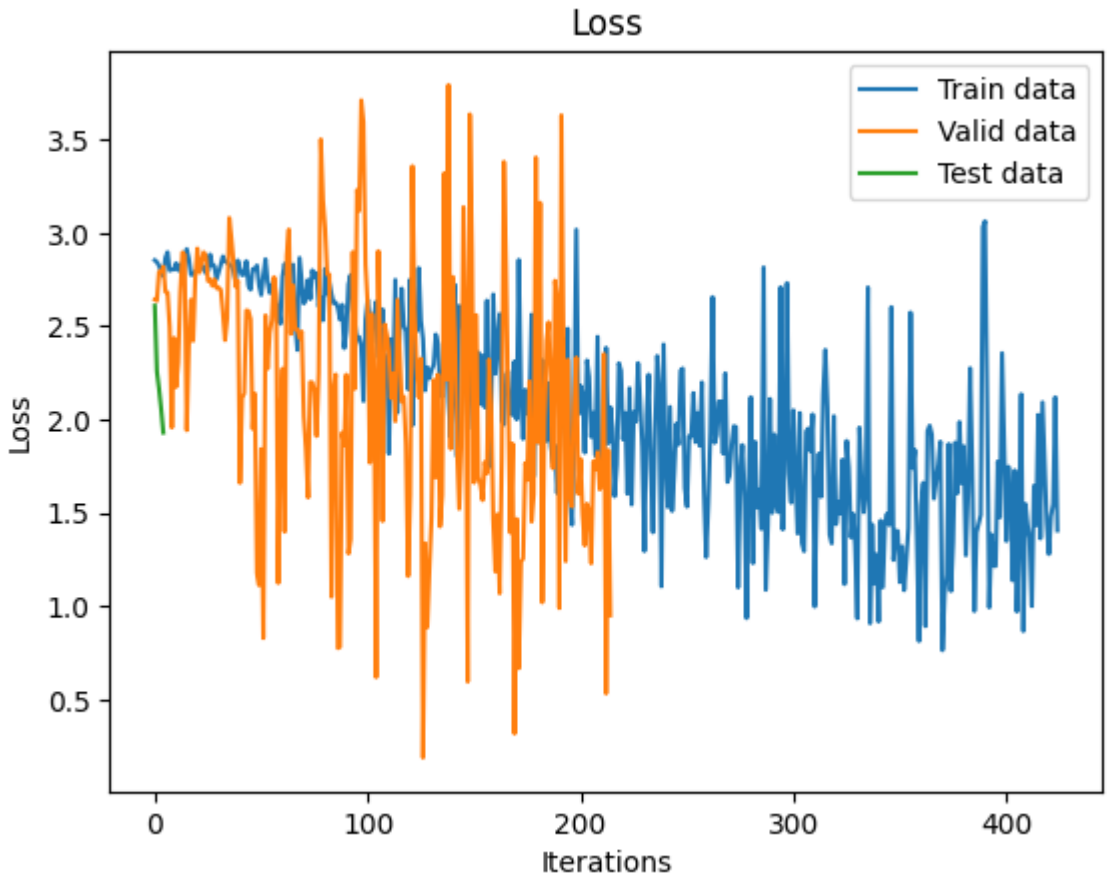
Q4.2

For my network:

3 conv layers, followed 2 fc layers Accuracy | Loss :----------:|:----------:



|



In [ ]:
```
# YOUR CODE HERE
# YOUR CODE HERE
import torch
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torchvision.datasets as dsets
from torch.autograd import Variable

# model = torch.hub.load('pytorch/vision:v0.10.0', 'squeezenet1_1', pretrained

# final_conv = nn.Conv2d(512, 17, kernel_size=1)
# model.classifier = nn.Sequential(
#         nn.Dropout(p=0.5), final_conv, nn.ReLU(inplace=True), nn.AdaptiveAvg


# for param in model.parameters():
#     param.requires_grad = False
# for param in model.classifier.parameters():
#     param.requires_grad = True

# Construct an Optimizer object for updating the last layer only.

# optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

from PIL import Image
from torchvision import transforms
# input_image = Image.open(filename)
preprocess = transforms.Compose([
```

```
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]
])
# input_tensor = preprocess(input_image)
# input_batch = input_tensor.unsqueeze(0)

batch_size = 8
train_dataloader = torch.utils.data.DataLoader(dsets.ImageFolder('./data/oxfor
valid_dataloader = torch.utils.data.DataLoader(dsets.ImageFolder('./data/oxfor
test_dataloader = torch.utils.data.DataLoader(dsets.ImageFolder('./data/oxford
# print("len of data set = ", valid_dataloader.size())

class CNNModel(nn.Module):
    def __init__(self):
        super(CNNModel, self).__init__()

        # Convolution 1
        self.cnn1 = nn.Conv2d(3, 64, kernel_size=5, stride=1, padding=0)
        self.relu1 = nn.ReLU()

        # Max pool 1
        self.maxpool1 = nn.MaxPool2d(kernel_size=2)

        # Convolution 2
        self.cnn2 = nn.Conv2d(64, 128, kernel_size=5, stride=1, padding=0)
        self.relu2 = nn.ReLU()

        # Max pool 2
        self.maxpool2 = nn.MaxPool2d(kernel_size=2)

        self.cnn3 = nn.Conv2d(128, 64, kernel_size=5, stride=1, padding=0)
        self.relu3 = nn.ReLU()
        # Fully connected 1 (readout)
        self.fc1 = nn.Linear(153664, 47)
        self.fc2 = nn.Linear(47, 17)
    # 100, 32, 4, 4
    def forward(self, x):
        # Convolution 1
        out = self.cnn1(x)
        out = self.relu1(out)

        # Max pool 1
        out = self.maxpool1(out)

        # Convolution 2
        out = self.cnn2(out)
        out = self.relu2(out)

        # Max pool 2
        out = self.maxpool2(out)

        out = self.cnn3(out)
        out = self.relu3(out)
        # print("maxpool shape = ", out.size())
        # Resize
```

```
        # Original size: (100, 32, 7, 7)
        # out.size(0): 100
        # New out size: (100, 32*7*7)
        out = out.view(out.size(0), -1)

        # Linear function (readout)
        out = self.fc1(out)
        # print("first conv shape = ", out.size())
        out = self.fc2(out)

        return out

'''
STEP 4: INSTANTIATE MODEL CLASS
'''

model = CNNModel()
# optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
loss_fn = nn.CrossEntropyLoss()

def train(train_loader, model, loss_fn, optimizer):
    # iter = 0
    # for epoch in range(num_epochs):
    train_correct = 0
    train_total = 0
    for i, (images, labels) in enumerate(train_loader):

        if torch.cuda.is_available():
            images = Variable(images.cuda())
            labels = Variable(labels.cuda())
        else:
            images = Variable(images)
            labels = Variable(labels)
        # print("images size = ", images.size())
        # print("labesl size = ", labels.size())
        # Clear gradients w.r.t. parameters
        optimizer.zero_grad()

        # Forward pass to get output/logits
        outputs = model(images)
        # print("outputs  = ", outputs)
        # Calculate Loss: softmax --> cross entropy loss
        train_loss = loss_fn(outputs, labels)

        # Getting gradients w.r.t. parameters
        train_loss.backward()

        # Updating parameters
        optimizer.step()

            # Get predictions from the maximum value
        _, predicted = torch.max(outputs.data, 1)

        # Total number of labels
        train_total += labels.size(0)
```

```
        # Total correct predictions
        if torch.cuda.is_available():
            train_correct += (predicted.cpu() == labels.cpu()).sum()
        else:
            train_correct += (predicted == labels).sum()

        train_accuracy = 100 * train_correct / train_total
        train_acc_arr.append(train_accuracy)
        train_loss_arr.append(train_loss.detach().numpy())
        # itr_arr.append(passes)
        # passes += 1
        # print("Loss = ", train_loss)
        # print("Acc = ", train_accuracy)
    # return train_acc_arr.detach().numpy(), train_loss_arr.detach().numpy()

def valid(valid_loader, model, loss_fn):
    # iter = 0
    # for epoch in range(num_epochs):
    valid_correct = 0
    valid_total = 0
    for i, (images, labels) in enumerate(valid_loader):

        if torch.cuda.is_available():
            images = Variable(images.cuda())
            labels = Variable(labels.cuda())
        else:
            images = Variable(images)
            labels = Variable(labels)
        # print("images size = ", images.size())
        # print("labesl size = ", labels.size())
        # Clear gradients w.r.t. parameters
        # optimizer.zero_grad()

        # Forward pass to get output/logits
        outputs = model(images)
        # print("outputs  = ", outputs)
        # Calculate Loss: softmax --> cross entropy loss
        valid_loss = loss_fn(outputs, labels)

        # Getting gradients w.r.t. parameters
        # valid_loss.backward()

        # Updating parameters
        # optimizer.step()

            # Get predictions from the maximum value
        _, predicted = torch.max(outputs.data, 1)

        # Total number of labels
        valid_total += labels.size(0)

        # Total correct predictions
        if torch.cuda.is_available():
            valid_correct += (predicted.cpu() == labels.cpu()).sum()
        else:
            valid_correct += (predicted == labels).sum()
```

```
        valid_accuracy = 100 * valid_correct / valid_total
        valid_acc_arr.append(valid_accuracy)
        valid_loss_arr.append(valid_loss.detach().numpy())
        # itr_arr.append(passes)
        # passes += 1
        # print("Loss = ", valid_loss)
        # print("Acc = ", valid_accuracy)
    # return valid_acc_arr.detach().numpy(), valid_loss_arr.detach().numpy()


def test(test_loader, model, loss_fn):
# Calculate Accuracy
    test_correct = 0
    test_total = 0
    # Iterate through test dataset
    for images, labels in test_loader:

        if torch.cuda.is_available():
            images = Variable(images.cuda())
        else:
            images = Variable(images)

        # Forward pass only to get logits/output
        outputs = model(images)

        # Loss
        test_loss = loss_fn(outputs, labels)

        # Get predictions from the maximum value
        _, predicted = torch.max(outputs.data, 1)

        # Total number of labels
        test_total += labels.size(0)

        # Total correct predictions
        if torch.cuda.is_available():
            test_correct += (predicted.cpu() == labels.cpu()).sum()
        else:
            test_correct += (predicted == labels).sum()

        test_accuracy = 100 * test_correct / test_total
        test_acc_arr.append(test_accuracy)
        test_loss_arr.append(test_loss.detach().numpy())
    print("test accuracy = ", test_accuracy)


            # Print Loss
            # print('Iteration: {}'.format(iter))
            # print('Loss: {}'.format(loss.item()))
            # print('Accuracy: {}'.format(accuracy.item()))

train_acc_arr = []
valid_acc_arr = []
test_acc_arr = []
train_loss_arr = []
test_loss_arr =[]
valid_loss_arr = []
```

```
itr_arr = []

num_epochs = 5
for t in range(num_epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(train_dataloader, model, loss_fn, optimizer)
    valid(valid_dataloader, model, loss_fn)
    test(test_dataloader, model, loss_fn)
    # acc, l = train(train_loader, model, loss_fn, optimizer)
    # train_acc_arr.append(acc)
    # train_loss_arr.append(l)
    # itr_arr.append(t)
print("Done!")

import matplotlib.pyplot as plt
import numpy as np
plt.plot(np.arange(len(train_acc_arr)),train_acc_arr, label = "Train data")
plt.plot(np.arange(len(valid_acc_arr)),valid_acc_arr, label = "Valid data")
plt.plot(np.arange(len(test_acc_arr)),test_acc_arr, label = "Test data")
plt.xlabel('Iterations')
plt.ylabel('Accuracy')
plt.legend()
plt.title("Accuracy")
plt.show()

plt.plot(np.arange(len(train_acc_arr)),train_loss_arr, label = "Train data")
plt.plot(np.arange(len(valid_acc_arr)),valid_loss_arr, label = "Valid data")
plt.plot(np.arange(len(test_acc_arr)),test_loss_arr, label = "Test data")
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
plt.title("Loss")
plt.show()
# raise NotImplementedError()
# raise NotImplementedError()
# raise NotImplementedError()
```

The performance of squeeznet is much better than my own network. (However I couldnt train my network using high number of hidden layer size due to not good enough compute.)

Q5 (12 points WriteUp)

Given $y = Wx + b$ (or $y_j = \sum_{i=1}^{d} x_i W_{ji} + b_j$), and the gradient of some loss J with respect y, show how to get $\frac{\partial J}{\partial W}$, $\frac{\partial J}{\partial x}$ and $\frac{\partial J}{\partial b}$. Be sure to do the derivatives with scalars and re-form the matrix form afterwards. Here are some helpful notations.

$$\frac{\partial J}{\partial y} = \delta \in \mathbb{R}^{k \times 1} \quad W \in \mathbb{R}^{k \times d} \quad x \in \mathbb{R}^{d \times 1} \quad b \in \mathbb{R}^{k \times 1}$$

By chain rule:

$$\frac{\partial J}{\partial W} = \sum_{i}^{k} \frac{\partial y_i}{\partial W} * \frac{\partial J}{\partial y_i}$$

$$Now, \frac{\partial y_i}{\partial W_{rc}} = \frac{\partial \sum_{j=1}^{d} W_{ij} x_j + b}{\partial W_{rc}}$$

$$\implies \frac{\partial y_i}{\partial W_{rc}} = x_c for(i == r)$$

Since for every row in x we are getting the values in the columns first, we can express it as the transpose matrix. Therefore, $\frac{\partial y_i}{\partial W_{rc}} = x^T$ And

$$\frac{\partial J}{\partial W} = \delta x^T which \in \mathbb{R}^{k \times d}$$

Now for: $\frac{\partial J}{\partial x}$

$$\frac{\partial J}{\partial x} = \sum_{i}^{k} \frac{\partial y_i}{\partial x} * \frac{\partial J}{\partial y_i}$$

$$Now, \frac{\partial y_i}{\partial x_c} = \frac{\partial \sum_{j=1}^{d} W_{ij} x_j + b}{\partial x_c}$$

$$\implies \frac{\partial y_i}{\partial x_c} = W_{ic}$$

Therefore,

$$\frac{\partial J}{\partial x} = \sum_{i}^{k} \sum_{c}^{d} W_{ic} * \frac{\partial J}{\partial y_i}$$

Now similarly, if we check the matrix dimensonality we can figure that this is equivalent to multiplying by W^T.

$$\implies \frac{\partial J}{\partial x} = W^T * \delta \in \mathbb{R}^{d \times 1}$$

Finally for $\frac{\partial J}{\partial b}$

$$\frac{\partial J}{\partial b} = \sum_{i}^{k} \frac{\partial y_i}{\partial b} * \frac{\partial J}{\partial y_i}$$

Since it is just a scalar quantity getting added up, the derivative will be = 1.

$$\implies \frac{\partial y_i}{\partial b_j} = 1 for(i == j)$$

Therefore,

$$\frac{\partial J}{\partial b} = \delta \in \mathbb{R}^{k \times 1}$$

Q6 (15 points)

We will find the derivatives for Conv layers now. Since most Deep Learning frameworks such as Pytorch, Tensorflow use cross-correlation in their respective "convolution" functions (Pytorch and Tensorflow), we will continue this abuse of notation. So the operation performed with the Conv Layer weights will be cross-correlation.

The input, x is of shape $M \times N$ with C channels. This will be *convolved* (actually cross-correlation) with D number of $K \times K$ filters, each with a bias term. The stride is 1 and there will be no padding. We know the gradient of some loss J with respect to the output y, which will have D channels. Show how to get $\frac{\partial J}{\partial W}$, $\frac{\partial J}{\partial x}$ and $\frac{\partial J}{\partial b}$.

The dimensions and notation are as follows:

$$\frac{\partial J}{\partial y} = \delta \in \mathbb{R}^{D \times M_o \times N_o} \quad M_o = M - K + 1 \quad N_o = N - K + 1$$

$$x \in \mathbb{R}^{C \times M \times N} \quad W \in \mathbb{R}^{D \times C \times K \times K} \quad b \in \mathbb{R}^{D}$$

$x_{c,i,j}$: The element at the i^{th} row, the j^{th} column and the c^{th} channel of the input

$y_{c,i,j}$: The element at the i^{th} row, the j^{th} column and the c^{th} channel of the output

$W_{d,c,i,j}$: The element at the i^{th} row, the j^{th} column, the c^{th} channel of the kernel of the d^{th} filter

For this question, you may compute the derivatives with scalars only. You don't need to re-form the matrix

$$y_{d,i,j} = \sum_{i'=0}^{K-1} \sum_{j'=0}^{K-1} \sum_{d=1}^{d} w_{d,c,g,h} x_{c,i+i',j+j'} + b$$

$$\left(\frac{\partial J}{\partial W}\right) d, c, i, j = \sum g = 1^{M_0} \sum_{h=1}^{N_0} \left(\frac{\partial J}{\partial y_{d,g,h}}\right) \left(\frac{\partial y_{d,g,h}}{\partial W_{d,c,i,j}}\right)$$

$$= \sum_{g=1}^{M_0} \sum_{h=1}^{N_0} \delta_{d,g,h} \frac{\partial y_{d,g,h}}{\partial W_{d,c,i,j}} = \sum_{g=1}^{M_0} \sum_{h=1}^{N_0} \delta_{d,g,h} x_{c,g+i,h+j}$$

$$\left(\frac{\partial J}{\partial x}\right) c, i, j = \sum g = 0^{M_0} \sum_{h=0}^{N_0} \sum_{d=1}^{D} \left(\frac{\partial J}{\partial y_{d,g,h}}\right) \left(\frac{\partial y_{d,g,h}}{\partial x_{c,i,j}}\right)$$
$$= \sum_{g=0}^{M_0} \sum_{h=0}^{N_0} \sum_{d=1}^{D} \delta_{d,g,h} W_{d,i-g,j-h}$$

Wherever i-g and j-h aare negative, take $W_{d,i-g,j-h}$ to be 0.

$$\left(\frac{\partial J}{\partial b}\right) d = \sum g = 1^{M_0} \sum_{h=1}^{N_0} \left(\frac{\partial J}{\partial y_{d,g,h}}\right) \left(\frac{\partial y_{d,g,h}}{\partial b_d}\right)$$
$$= \sum_{g=1}^{M_0} \sum_{h=1}^{N_0} \delta_{d,g,h}$$

Q7 (4 points)

When the neural network applies the elementwise activation function (such as sigmoid), the gradient of the activation function scales the back-propagation update. This is directly from the chain rule, $\frac{d}{dx} f(g(x)) = f'(g(x))g'(x)$.

Q7.1 (1 point)

Consider the sigmoid activation function for deep neural networks. Why might it lead to a "vanishing gradient" problem if it is used for many layers (consider plotting the $\sigma'(x)$ in Q1.4)?

The major "active area" of the sigmoid function is very close to zero. So if we use this function multiple times, while back-propogating, we are multiplying very small numbers many times which will result in a quantity that is too small, thus eventuaally leading to vanishing of the gradient. After a point, the numerical underflow will occur which will make the gradient negligible.

Q7.2 (1 point)

Often it is replaced with $\tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$. What are the output ranges of both $\tanh$ and sigmoid? Why might we prefer $\tanh$?

Output range of tanh: [-1,1]
Output range of sigmoid: [0,1]
Convergence is usually faster if the average of each input variable over the training set is close to zero. And the outputs using tanh centre around 0 rather than sigmoid's 0.5, and this "makes learning for the next layer a little bit easier. Theefore we might prefre tanh.

Q7.3 (1 point)

Why does $\tanh(x)$ have less of a vanishing gradient problem? (plotting the derivatives helps! for reference: $\tanh'(x) = 1 - \tanh(x)^2$)

The response of tanh derivative as the input value approaches zero is larger than that of sigmoid's derivative. Therefore, the gradient will be magnified to a greater extent in case of tanh than sigmoid and hence vanshing gradient will be less of a problem if we use tanh.

Q7.4 (1 point)

$\tanh$ is a scaled and shifted version of the sigmoid. Show how $\tanh(x)$ can be written in terms of $\sigma(x)$. (*Hint: consider how to make it have the same range*)

$\tanh(x) = \frac{1-e^{-x}}{1+e^{-x}}$
$\sigma(x) = \frac{1}{1+e^{-x}}$

$tanh(x) = 2 * \sigma(x) - 1$

Q5.1.1 code

In [ ]:
```
import torch
import numpy as np
import scipy.io
from ipynb.fs.defs.q1 import *
from collections import Counter
from torch import nn

train_data = scipy.io.loadmat('data/nist36_train.mat')
valid_data = scipy.io.loadmat('data/nist36_valid.mat')

# we don't need labels now!
train_x = train_data['train_data']
valid_x = valid_data['valid_data']

max_iters = 100
# pick a batch size, initial learning rate
batch_size = 3
learning_rate = 1e-3

# raise NotImplementedError()
hidden_size = 32
lr_rate = 20

batches = get_random_batches(train_x,np.ones((train_x.shape[0],1)),batch_size)
batch_num = len(batches)

params = Counter()

# initialize layers here
# YOUR CODE HERE
initialize_weights(1024,hidden_size,params,'layer1')
initialize_weights(hidden_size,hidden_size,params,'layer2')
initialize_weights(hidden_size,hidden_size,params,'layer3')
```

```
initialize_weights(hidden_size,1024,params,'layer4')
# raise NotImplementedError()
```

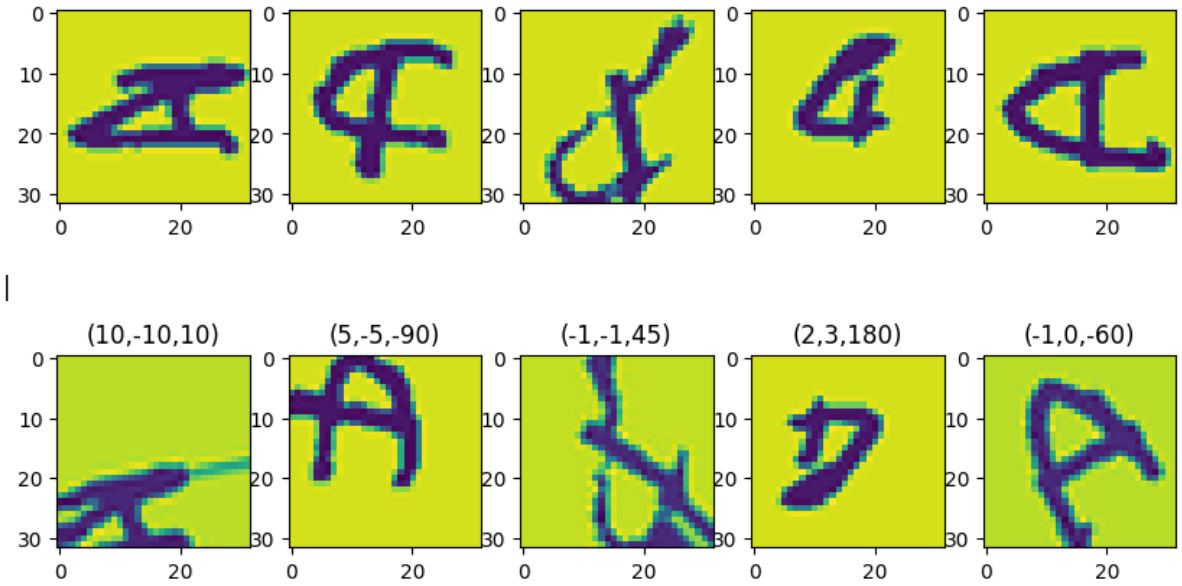
Q6.1.1 code

In [ ]:
```
def apply_random_translation(im, dx, dy):
    '''
    Applies a random translation to the image, described by dx, and dy.

    [input]
    * im -- image to be translation
    * dx -- the number of pixels the image should be translated in the x direc
    * dy -- the number of pixesl the image should be translated in the y direc
    [output]
    * im -- the translated image
    '''
    # YOUR CODE HERE
    sequence = np.array([dx, dy, 3])
    im_translated = ndimage.shift(im, sequence, mode='nearest')
    # raise NotImplementedError

    return im_translated
```

Q6.1.2 code

In [ ]:
```
def apply_random_rotation(im, angle):
    '''
    Applies a random rotation to the image of angle degrees.

    [input]
    * im -- image to be rotation
    * angle -- the number of degrees for the image to be rotated.

    [output]
    * im -- rotated image.
    '''
    # YOUR CODE HERE
    from PIL import Image
    from scipy import misc

    # rotated_im = misc.ascent()
    rotated_im = ndimage.rotate(im, angle, reshape=False, mode='nearest')
    # raise NotImplementedError

    return rotated_im
```

Q6.1.3 code

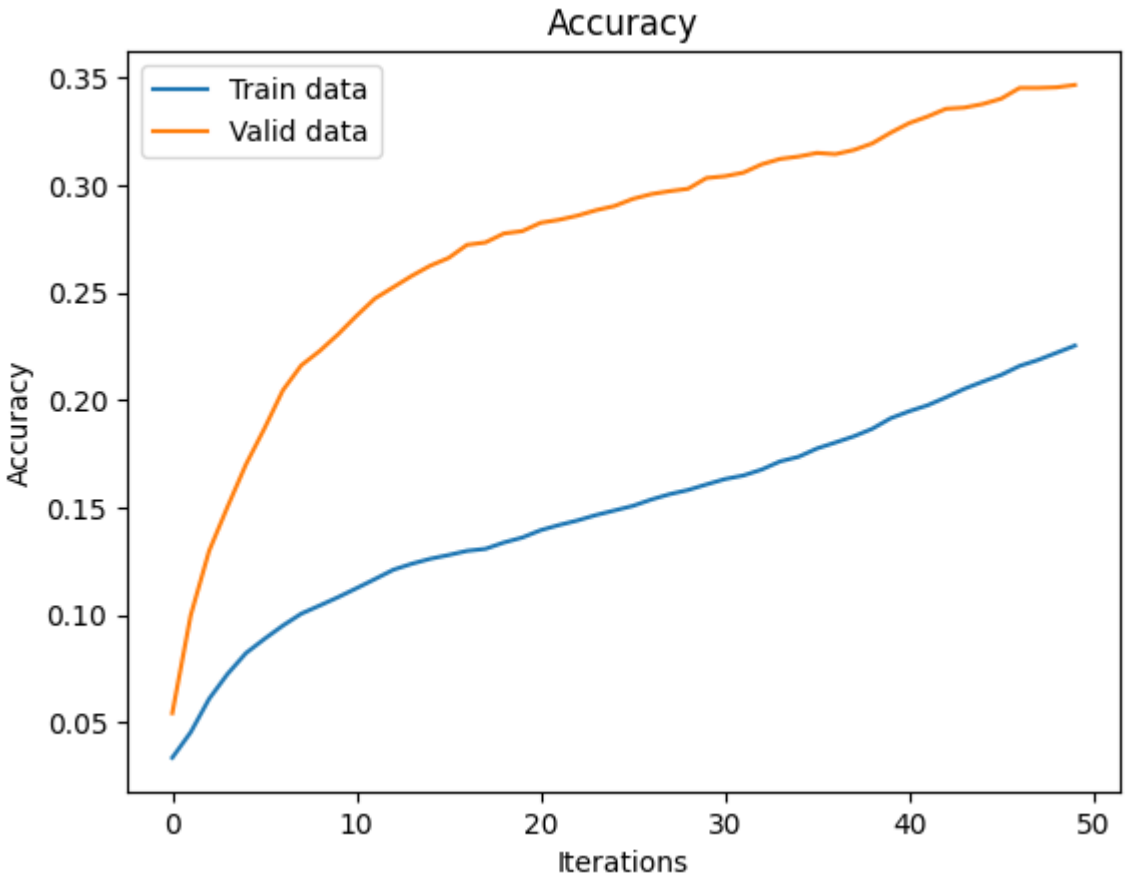


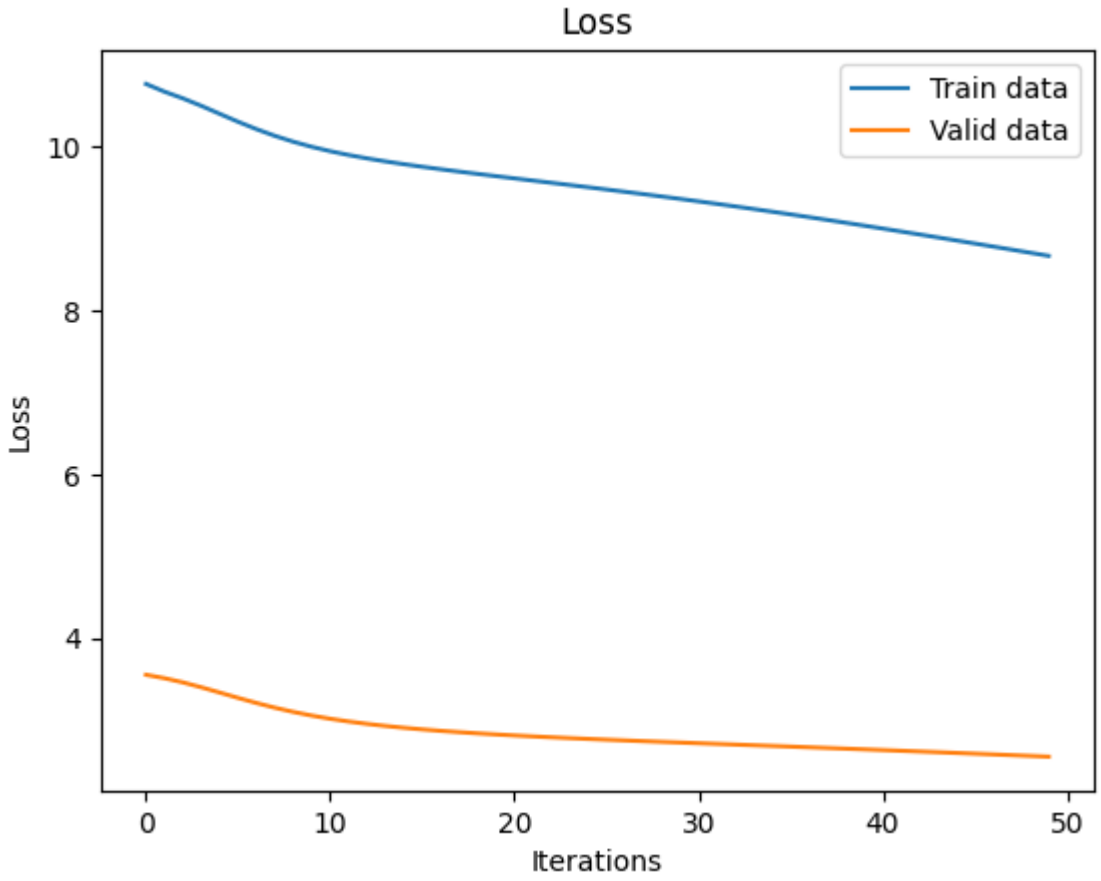
In [ ]:
```
train_data = scipy.io.loadmat('data/nist36_train.mat')
valid_data = scipy.io.loadmat('data/nist36_valid.mat')

train_x, train_y = train_data['train_data'], train_data['train_labels']
valid_x, valid_y = valid_data['valid_data'], valid_data['valid_labels']

# YOUR VISUALIZATION CODE HERE
import matplotlib.pyplot as plt
dx1 = 10
dy1 = -10
dx2 = 5
dy2 = -5
dx3 = -1
dy3 = -1
dx4 = 2
dy4 = 3
dx5 = -1
dy5 = 0
angle1 = 10
angle2 = -90
angle3 = 45
angle4 = 180
angle5 = -60

print("input shape = ", train_x[0].shape)

image1 = train_x[0].reshape(32,32, -1)
image2 = train_x[1].reshape(32,32, -1)
image3 = train_x[2].reshape(32,32, -1)
image4 = train_x[3].reshape(32,32, -1)
image5 = train_x[4].reshape(32,32, -1)

fig2 = plt.figure(figsize=(10, 5))
ax1, ax2, ax3, ax4, ax5 = fig2.subplots(1, 5)

ax1.imshow(image1)
ax2.imshow(image2)
ax3.imshow(image3)
```

```
ax4.imshow(image4)
ax5.imshow(image5)
plt.show()

image1mod = apply_random_translation(image1, dx1, dy1)
image1mod = apply_random_rotation(image1mod, angle1)
image2mod = apply_random_translation(image2, dx2, dy2)
image2mod = apply_random_rotation(image2mod, angle2)
image3mod = apply_random_translation(image3, dx3, dy3)
image3mod = apply_random_rotation(image3mod, angle3)
image4mod = apply_random_translation(image4, dx4, dy4)
image4mod = apply_random_rotation(image4mod, angle4)
image5mod = apply_random_translation(image5, dx5, dy5)
image5mod = apply_random_rotation(image5mod, angle5)

fig = plt.figure(figsize=(10, 5))
ax1, ax2, ax3, ax4, ax5 = fig.subplots(1, 5)

ax1.imshow(image1mod)
ax1.title.set_text('(10,-10,10)')
ax2.imshow(image2mod)
ax2.title.set_text('(5,-5,-90)')
ax3.imshow(image3mod)
ax3.title.set_text('(-1,-1,45)')
ax4.imshow(image4mod)
ax4.title.set_text('(2,3,180)')
ax5.imshow(image5mod)
ax5.title.set_text('(-1,0,-60)')
# fig.set_layout_engine('tight')
# image1mod = np.expand_dims(image1mod, axis=0)
# plt.imshow(image1mod)

plt.show()
```

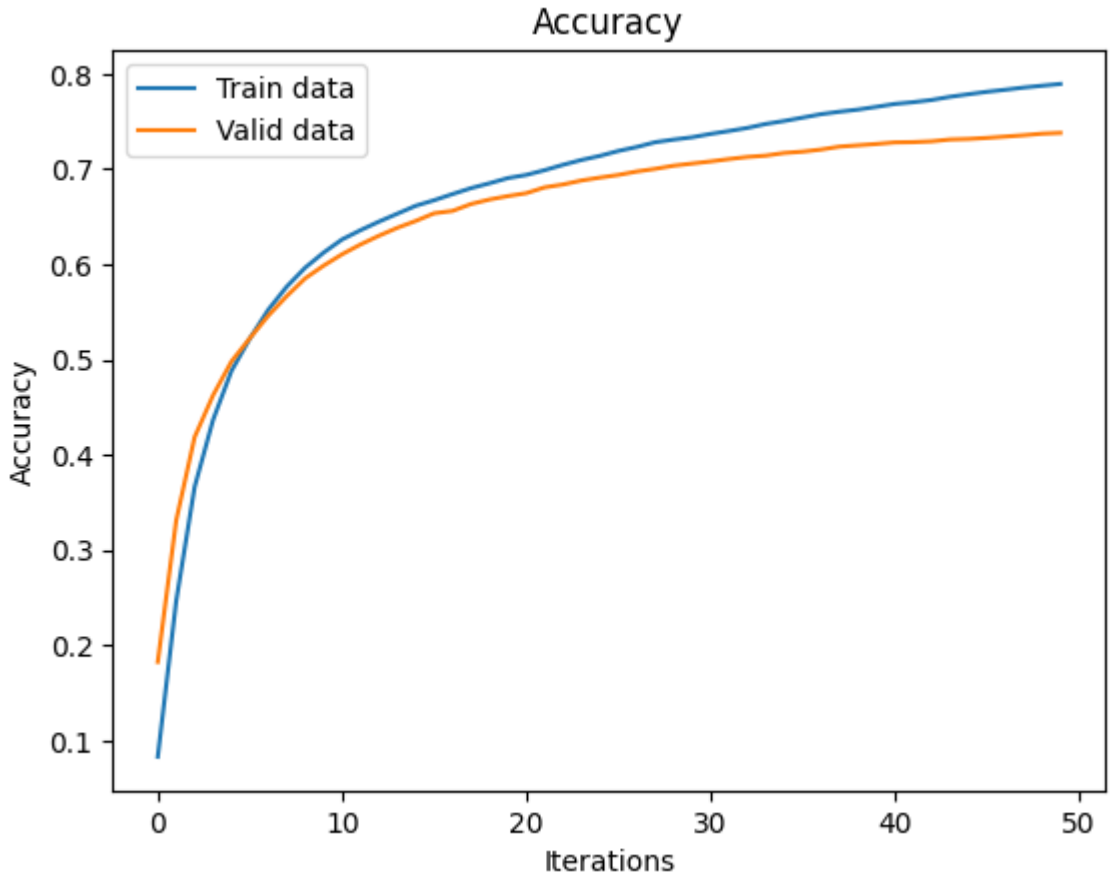
Q6.2 code

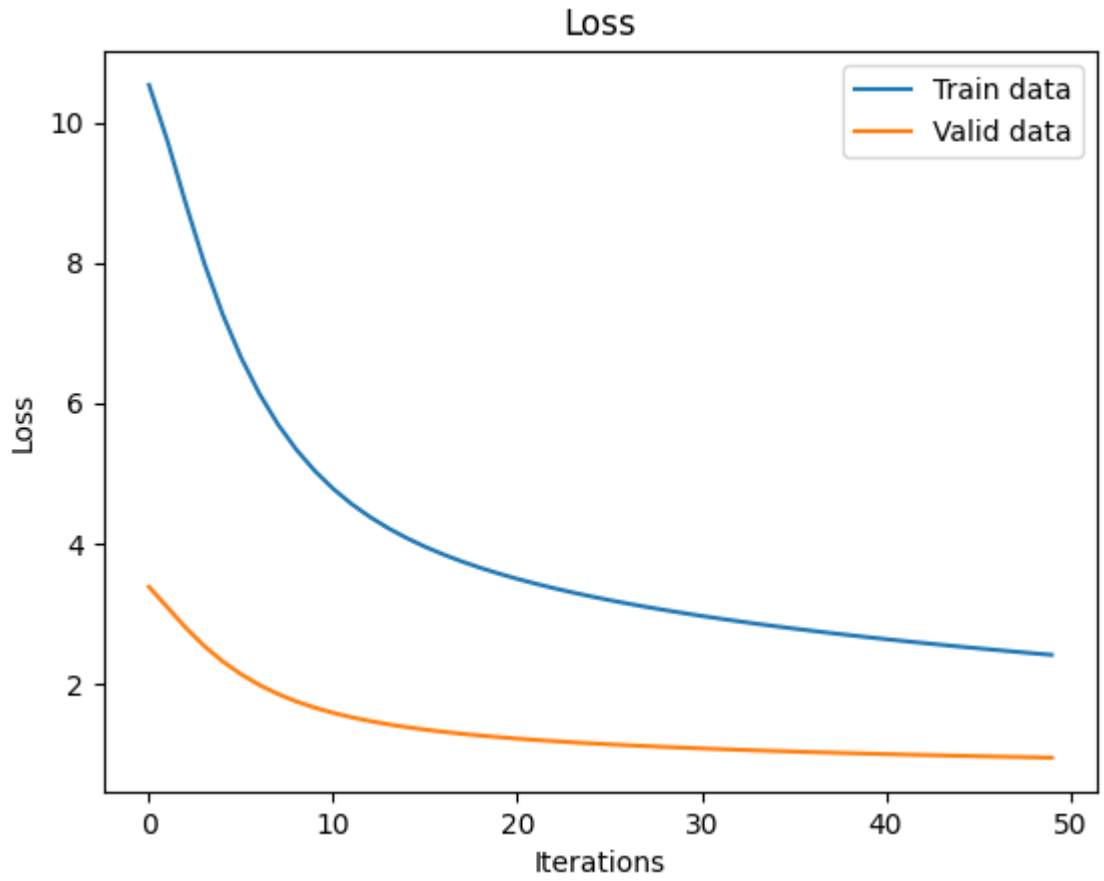
The accuracy for augmented data is less than that compared to non-augmented data. This is because we are modifying the original data for input but asking the network to learn it using the original data as the ground truth. Therefore, it takes more time for the network to learn the augmentation as well! And hence if we run it for the same parameters as that of the non-augmented data the accuracy will be less. Augmented Accuracy | Augmented Loss :----------:|:----------:|:----------:


Accuracy
Train data
Valid data
Accuracy
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0
10
20
30
40
50
Iterations


Loss
Train data
Valid data
Loss
10
8
6
4
0
10
20
30
40
50
Iterations

Non-Augmented Accuracy | Non-Augmented Loss :----------:|:----------:





In [ ]:
```
import math
train_data = scipy.io.loadmat('data/nist36_train.mat')
valid_data = scipy.io.loadmat('data/nist36_valid.mat')

train_x, train_y = train_data['train_data'], train_data['train_labels']
valid_x, valid_y = valid_data['valid_data'], valid_data['valid_labels']

for i in range(len(train_x)):
    x = np.random.randint(-6,6)
    y = np.random.randint(-6,6)
    ang = np.random.randint(-45,45)
    image = train_x[i].reshape(32,32, -1)
    image = apply_random_translation(image, x, y)
    train_x[i] = apply_random_rotation(image, ang).ravel()
# apply_random_translation(image1, dx1, dy1)
# image1mod = apply_random_rotation(image1mod, angle1)

max_iters = 50
# pick a batch size, learning rate
batch_size = 3
learning_rate = 1e-3
# YOUR CODE HERE

# raise NotImplementedError()
hidden_size = 64

batches = get_random_batches(train_x,train_y,batch_size)
batch_num = len(batches)

params = {}

# initialize layers (named "layer1" and "output") here
# YOUR CODE HERE
initialize_weights(1024,hidden_size,params,'layer1')
initialize_weights(hidden_size,36,params,'output')
# raise NotImplementedError()
train_acc_arr = []
train_loss_arr = []
valid_acc_arr = []
valid_loss_arr = []
itr_arr = []
# with default settings, you should get loss < 150 and accuracy > 80%
for itr in range(max_iters):
    itr_arr.append(itr)
    total_loss = 0
    total_acc = 0
    valid_loss = 0
    valid_acc = 0
    for xb,yb in batches:

        # print("xb shape = ", xb.shape)
        # training loop can be exactly the same as q2!
        # YOUR CODE HERE

        post_act = forward(xb,params,'layer1',sigmoid)
        pred_output = forward(post_act,params,'output',softmax)
        # raise NotImplementedError()
```

```
        # loss
        # be sure to add loss and accuracy to epoch totals
        # YOUR CODE HERE
        loss, acc = compute_loss_and_acc(yb, pred_output)
        total_loss += loss/len(batches)
        total_acc += acc/len(batches)

        # raise NotImplementedError()

        # backward
        # YOUR CODE HERE
        last_layer_backprop = backwards(pred_output - yb, params, 'output', li
        hidden_layer_backprop  = backwards(last_layer_backprop, params, 'layer
        # raise NotImplementedError()

        # apply gradient
        # YOUR CODE HERE
        params['Woutput'] = params['Woutput'] - learning_rate*params['grad_Wou
        params['boutput'] = params['boutput'] - learning_rate*params['grad_bou
        params['Wlayer1'] = params['Wlayer1'] - learning_rate*params['grad_Wla
        params['blayer1'] = params['blayer1'] - learning_rate*params['grad_bla

        # raise NotImplementedError()
    train_acc_arr.append(total_acc)
    train_loss_arr.append(total_loss)
    if itr % 2 == 0:
        print("itr: {:02d} \t loss: {:.2f} \t acc : {:.2f}".format(itr,total_l

# run on validation set and report accuracy! should be above 70%

    post_act = forward(valid_x,params,'layer1',sigmoid)
    pred_output = forward(post_act,params,'output',softmax)
    # raise NotImplementedError()

    # loss
    # be sure to add loss and accuracy to epoch totals
    # YOUR CODE HERE
    loss, acc = compute_loss_and_acc(valid_y, pred_output)
    valid_loss += loss/len(batches)
    valid_acc += acc
    valid_acc_arr.append(valid_acc)
    valid_loss_arr.append(valid_loss)
    # raise NotImplementedError()
    print('Validation accuracy: ',valid_acc)


import matplotlib.pyplot as plt
plt.plot(itr_arr,train_acc_arr, label = "Train data")
plt.plot(itr_arr,valid_acc_arr, label = "Valid data")
plt.xlabel('Iterations')
plt.ylabel('Accuracy')
plt.legend()
plt.title("Accuracy")
plt.show()

plt.plot(itr_arr,train_loss_arr, label = "Train data")
```

```
plt.plot(itr_arr,valid_loss_arr, label = "Valid data")
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
plt.title("Loss")
plt.show()
```

In [ ]:

```
import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset
from torchvision import datasets
from torchvision.transforms import ToTensor
from ipynb.fs.defs.q1 import *
import scipy.io
import torch.nn.functional as F

training_data = scipy.io.loadmat('data/nist36_train.mat')
train_x, train_y = training_data['train_data'], training_data['train_labels']
train_x, train_y = torch.from_numpy(train_x).float(),torch.from_numpy(train_y)

# load_train_data = DataLoader(TensorDataset(train_x, train_y))

batch_size = 8

# Create data loaders.
train_dataloader = DataLoader(TensorDataset(train_x, train_y), batch_size=batcl

# Get cpu or gpu device for training.
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using {device} device")

# Define model
# class NeuralNetwork(nn.Module):
#     def __init__(self):
#         super().__init__()
#         self.flatten = nn.Flatten()
#         self.linear_relu_stack = nn.Sequential(
#             nn.Linear(1024, 64),
#             nn.Sigmoid(),
#             nn.Linear(64, 36),
#             nn.Softmax(dim=1),
#             # nn.Linear(512, 10)
#         )

#     def forward(self, x):
#         x = self.flatten(x)
#         logits = self.linear_relu_stack(x)
#         return logits

# model = NeuralNetwork().to(device)

class Network(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(1024, 64)
        self.output = nn.Linear(64, 36)
```

```
    def forward(self, x):
        x = self.hidden(x)
        x = torch.sigmoid(x)
        x = self.output(x)
        x = F.log_softmax(x,dim=1)

        return x


model = Network()

print(model)

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=5e-3)


def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    avg_acc = 0
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = nn.functional.cross_entropy(pred, y)
        _, acc = compute_loss_and_acc(y.detach().numpy(), pred.detach().numpy(
        avg_acc += acc/len(train_dataloader)
        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()


    print("Loss = ", loss)
    print("Acc = ", avg_acc)
    return avg_acc, loss.detach().numpy()


epochs = 50
acc_arr = []
loss_arr = []
itr_arr = []
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    acc, l = train(train_dataloader, model, loss_fn, optimizer)
    acc_arr.append(acc)
    loss_arr.append(l)
    itr_arr.append(t)
print("Done!")

import matplotlib.pyplot as plt
plt.plot(itr_arr,acc_arr, label = "Train data")
plt.xlabel('Iterations')
plt.ylabel('Accuracy')
plt.legend()
```

```
plt.title("Accuracy")
plt.show()

plt.plot(itr_arr,loss_arr, label = "Train data")
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
plt.title("Loss")
plt.show()
```