16720 (B) Neural Networks for Recognition - Assignment 3

```
    Instructor: Kris Kitani                          TAs: Qichen(Lead), Parito
  sh, Rawal, Yan, Zen, Wen-Hsuan
```

Submission Instructions:

1. Submit the PDF version of `theory.ipynb` to HW3:PDF. The `theory.ipynb` should include **ALL the writeup answers AND ALL the screenshots of code** specifically required in questions **from Q1 to Q7**. This section will be manually Graded.
2. Submit `q2.ipynb` , `q3.ipynb` , `q5.ipynb` to HW3:Code. Please do not submit other jupyter notebooks as they will not be autograded. Submitting them may cause running time out. (`q5.ipynb` is optional for extra credits)

The Appendix section at the end of this file would help you on questions P1 and P2.

Q1 Theory Questions (45 points)

Q1.1 (4 Points WriteUp)

Prove that softmax is invariant to translation, that is

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

Softmax is defined as below, 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 $c = -\max x_i$)

With the input change from x to x+c, both numerator and denominator of the function $softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$ will be multiplied by e^c.

Thus, softmax if invariant to translation.

With $c = -\max x_i$, $e^{x_max-c} = 1$, which will avoid zero denominator.

Q1.2

Softmax can be written as a three step processes, with $s_i = e^{x_i}$, $S = \sum_i s_i$ and $softmax(x_i) = \frac{1}{S} s_i$.

Q1.2.1 (1 point WriteUp)

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

The range of each element will be between 0 to 1. And the sum of all element will be 1.

Q1.2.2 (1 point WriteUp)

One could say that ”softmax takes an arbitrary real valued vector x and turns it into a **_”. Please think about a short phrase to fill in _**.

Probablity diatribution.

Q1.2.3 (1 point WriteUp)

Can you see the role of each step in the multi-step process now? Explain them.

The first step turns all values into positive values.

The second step computes the sum of all values.

The third step normalize each values.

Q1.3 (3 points WriteUp)

Show that multi-layer neural networks without a non-linear activation function are equivalent to linear regression.

With $y_1 = W_1 * x_1 + b_1$ and $y_2 = W_2 * y_1 + b_2$,

$y_2 = W_2 * W_1 * x_1 + W_2 * b_1 + b_2 = W_3 * x_1 + b_3$

Q1.4 (4 points WriteUp)

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 as a function of $\sigma(x)$ (without having access to x directly)

$-(1 + e^{-x})^{-2} * e^{-x} * (-1)$

$= (1 + e^{-x})^{-2} * e^{-x}$

$= (1 + e^{-x})^{-1} * (1 + e^{-x})^{-1} * e^{-x}$

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

Q1.5 (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 notional suggestions.

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

As $y_j = \sum_{i=1}^{d} x_i W_{ji} + b_j$,

we have $\frac{\partial y_j}{\partial W_j i} = x_i$, and $\frac{\partial y_j}{\partial x_i} = W_j i$.

Thus,

$$\frac{\partial J}{\partial W_j i} = \frac{\partial J}{\partial y_j} * \frac{\partial y_j}{\partial W_j i} = \delta j * x_i$$

$$\frac{\partial J}{\partial x_i} = \frac{\partial J}{\partial y_j} * \frac{\partial y_j}{\partial x_i} = \delta j * W_j i$$

$$\frac{\partial J}{\partial b_j} = \frac{\partial J}{\partial y_j} * \frac{\partial y_j}{\partial b_j} = \delta j$$

Then,

$$\frac{\partial J}{\partial W} = \delta x^T$$

$$\frac{\partial J}{\partial x} = W^T * \delta$$

$$\frac{\partial J}{\partial b} = \delta$$

Q1.6 (15 points WriteUp)

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 (https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html#torch.nn.Conv2d) and Tensorflow (https://www.tensorflow.org/api_docs/python/tf/nn/convolution)), 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

With $y_{d,i,j} = \sum_c \sum_p \sum_q (W_{d,c,p,q} * X_{c,i+p,j+q}) + b_d$,

we can see the matirx $W_{d,c,p,q}$ is fixed and will be multiplied by $X_{c,i+p,j+q}$ for all i and j,

Thus, we have $\frac{\partial J}{\partial W_{d,c,p,q}} = \frac{\partial J}{\partial y} \frac{\partial y}{\partial W_{d,c,p,q}} = \sum_i \sum_j \delta_{d,i,j} X_{c,i+p,j+q}$,

where X_c, k, l is fixed and multiplied by $W_{d,c,k-i,l-j}$ for all i, j and d.

Thus, we have $\frac{\partial J}{\partial X_{c,k,l}} = \frac{\partial J}{\partial y} \frac{\partial y}{\partial X_{c,k,l}} = \sum_d \sum_i \sum_j \delta_{d,i,j} W_{d,c,k-i,l-j}$.

Then, with d fixed, we can calculate $\frac{\partial J}{\partial b_d}$,

and sum over all δ,

$\frac{\partial J}{\partial b_d} = \frac{\partial J}{\partial y_d} \frac{\partial y_d}{\partial b_d} = \sum_c \sum_i \sum_j \delta_{d,i,j}$

Q1.7

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, $\frac{d}{dx} f(g(x)) = f'(g(x))g'(x)$.

Q1.7.1 (1 point WriteUp)

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 derivatives of sigmoid function is close to 0.

Thus, the when working on multiple-layer models, the gradient will reduce very quickly.

Q1.7.2 (1 point WriteUp)

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 $\tanh$?

The output range of $tanh$ is -1 to 1.

And the output range of $sigmoid$ is 0 to 1.

Since $tanh$ function is larger absolute values at most of the range than the $sigmoid$,

it tends to solve the vanishing gradient problem.

Q1.7.3 (1 point WriteUp)

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

The $tanh$ function has larger values around 0,

which will lead to larger gradient values when working on multi-layer models.

Q1.7.4 (1 point WriteUp)

$\tanh$ 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*)

$tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$,

$= \frac{1-e^{-2x}}{1+e^{-2x}} + \frac{1+e^{-2x}}{1+e^{-2x}} - 1$,

$= \frac{2}{1+e^{-2x}} - 1$,

$= 2\sigma(2x) - 1$.

For the following questions, please find the instructions in the corresponding jupyter notebooks.

Q2 Implement a Fully Connected Network (65 points + 10 Extra Credit)

Q2.1.1 (3 points WriteUp)

If all network are initialized with zero, it will results in zero or close to zero gradients, which will lead to small or negligible update to weight and bias.

The output will be very close to the input of the training.

Q2.1.3 (2 points WriteUp)

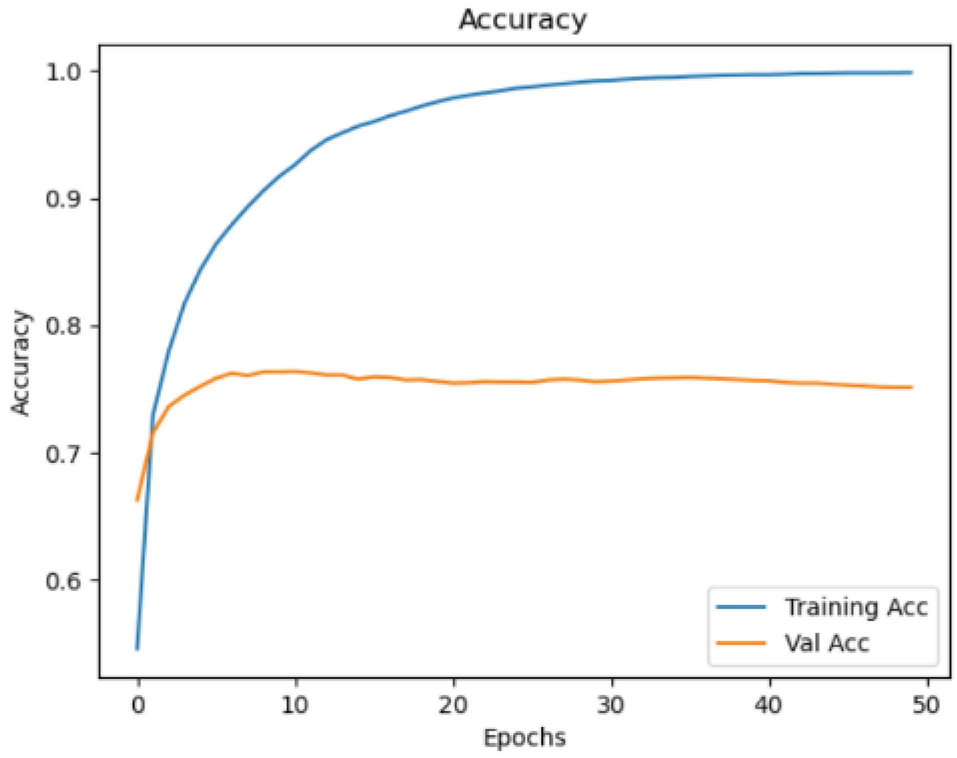
We initialized with random numbers to have the hidden layers be updated in different scales.

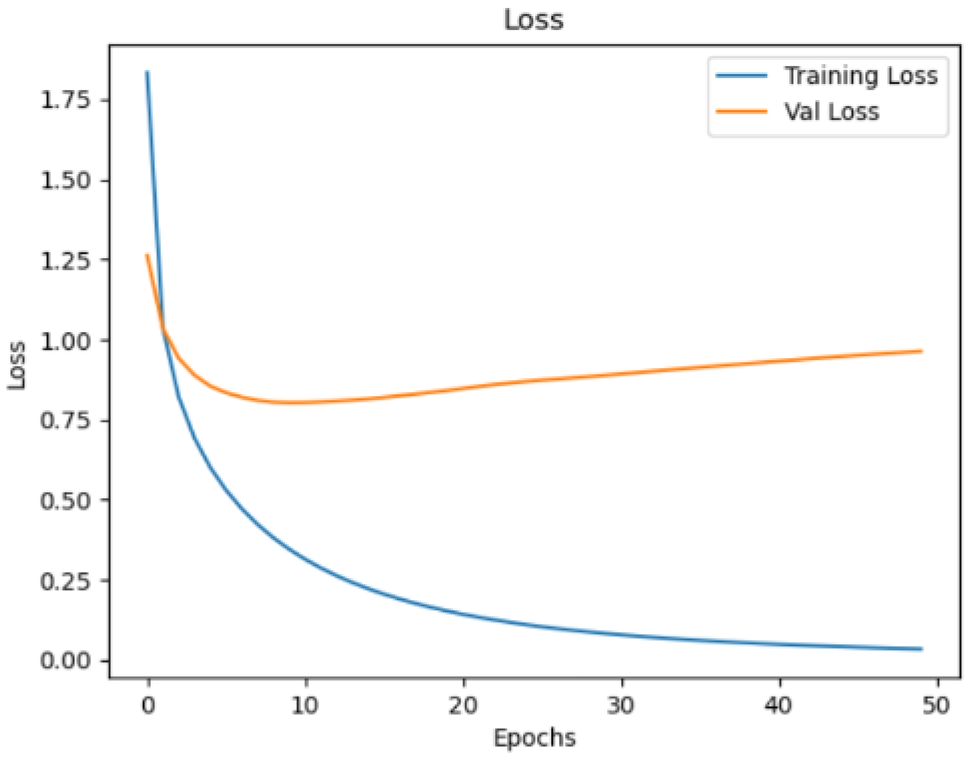
We scale the initialization depanding on layer size to maintain the variance for layers with different sizes.

Q3 Training Models (20 Points)

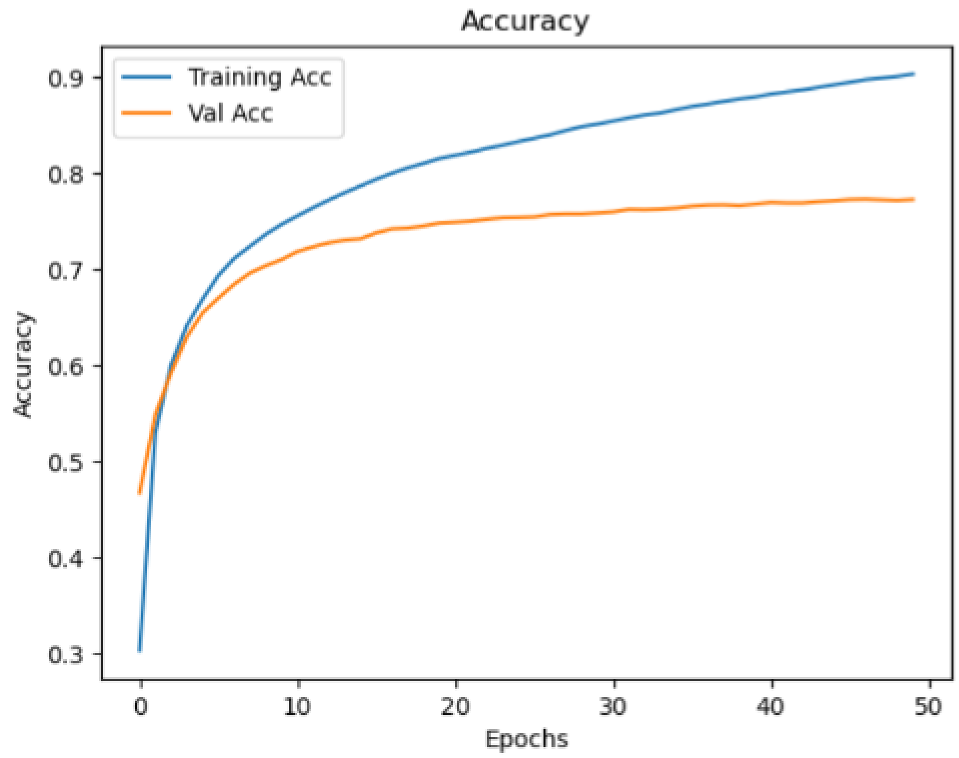
Q3.2 (3 points Code+WriteUp)

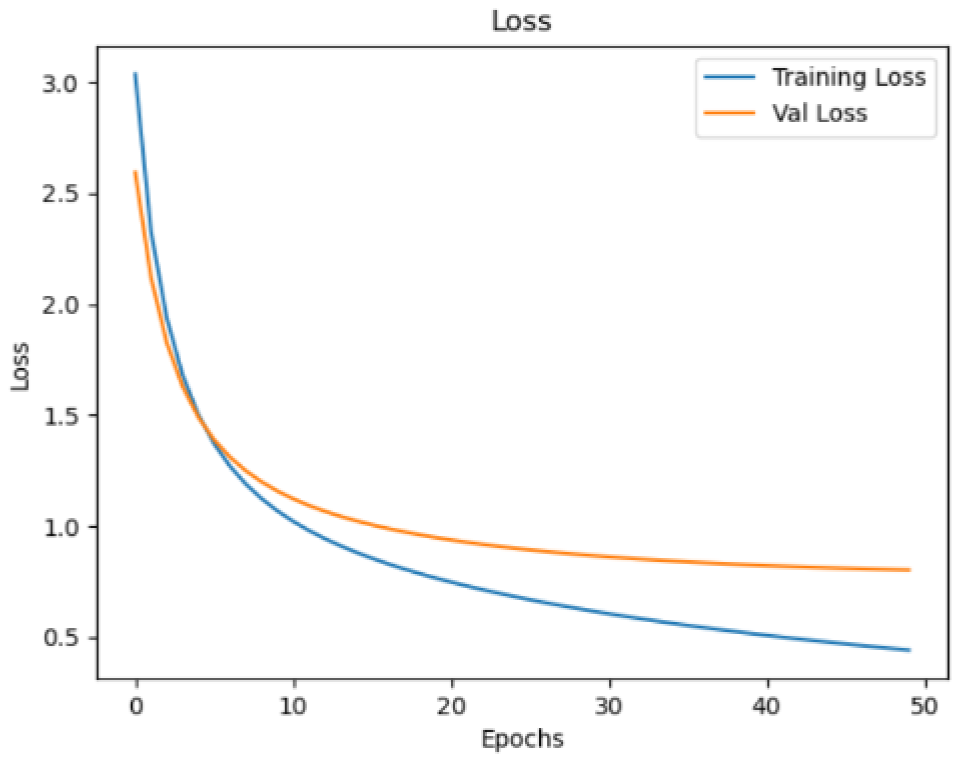
With learning rate = 1e-2



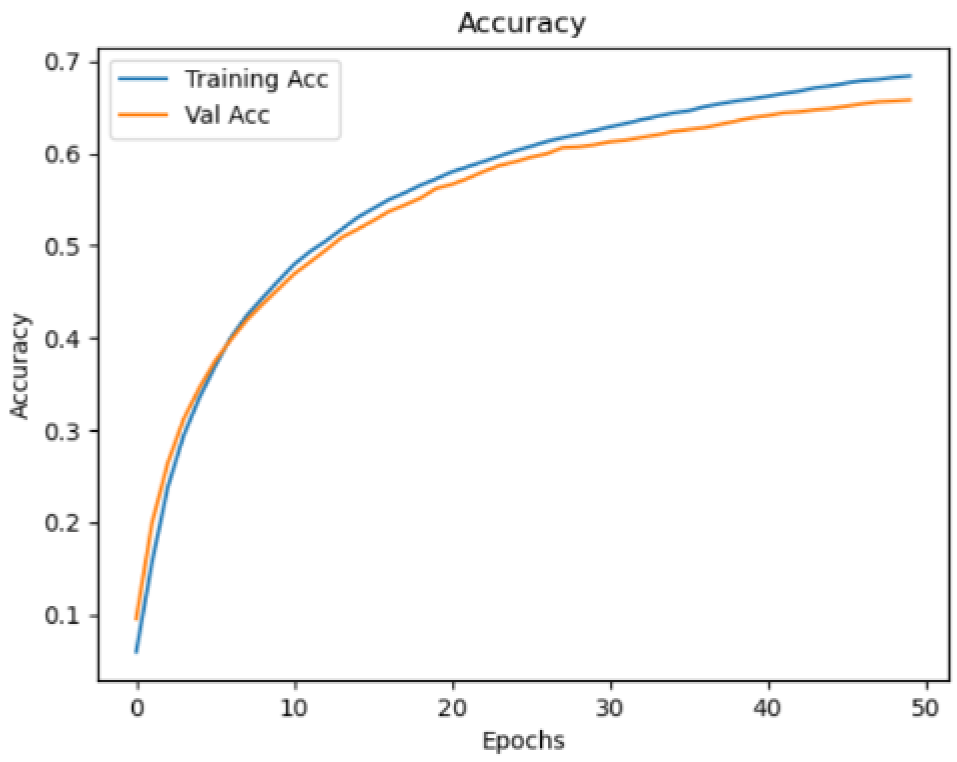


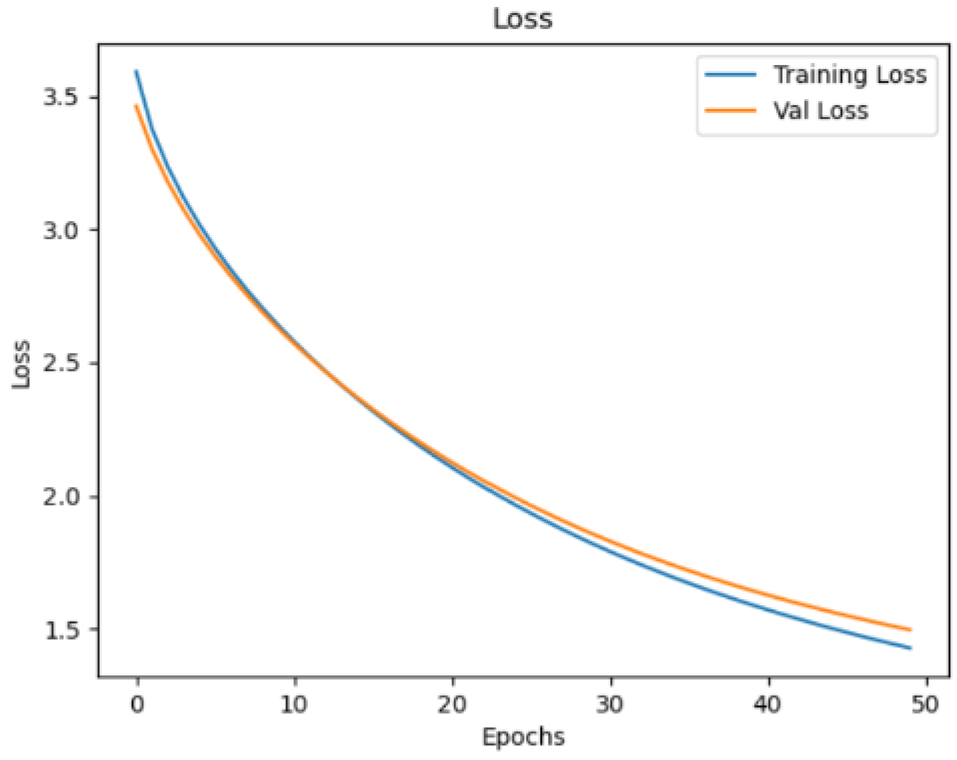
With learning rate = 1e-3





With learning rate = 1e-4





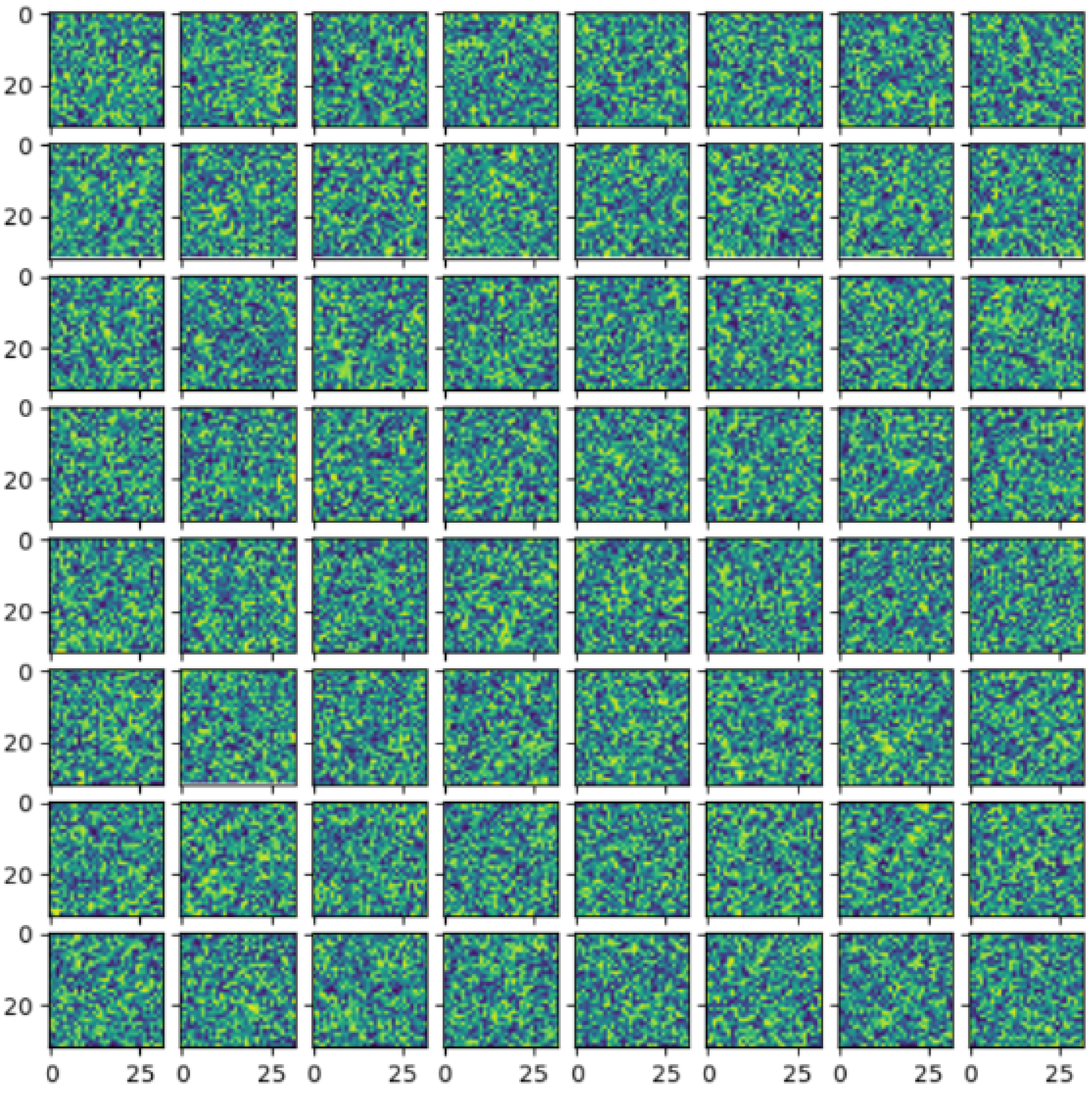
The best validating accuracy is at around 1e-3 learning rate.

At learning rate = 1e-2, the accuracy start to drop after 10 epoches,

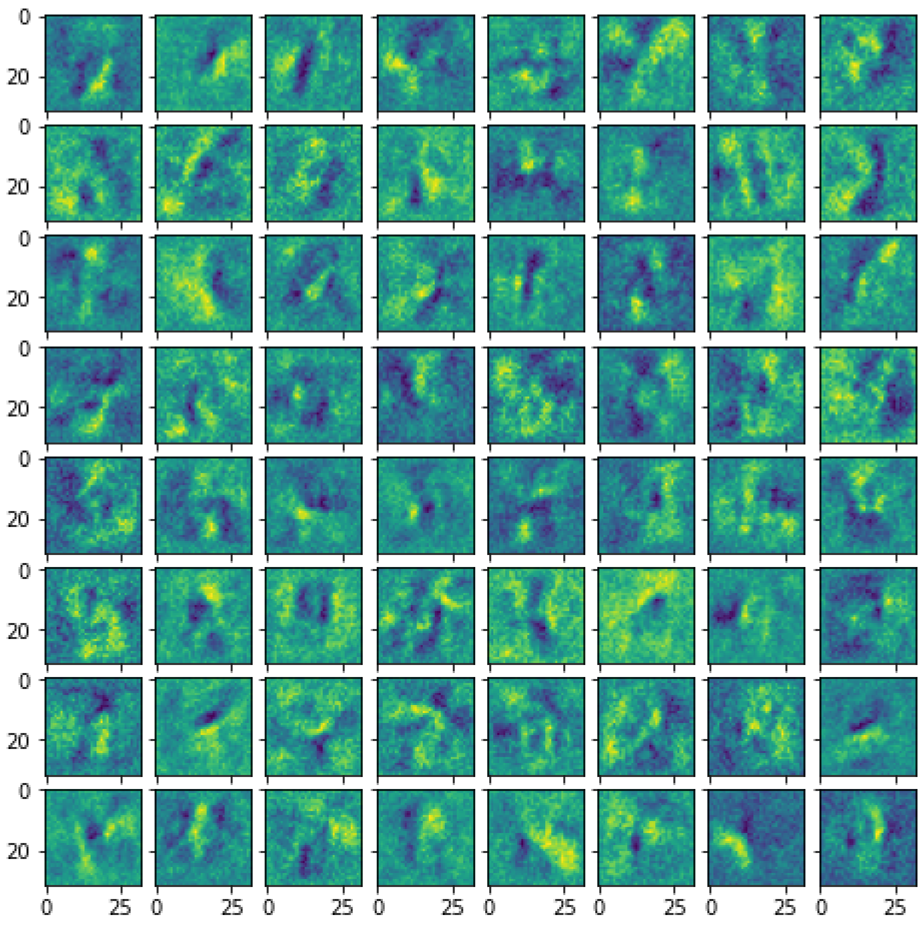
and at learning rate = 1e-4, the accuracy iters very slow.

Q3.3 (2 points Code+WriteUp)

Initial weight:

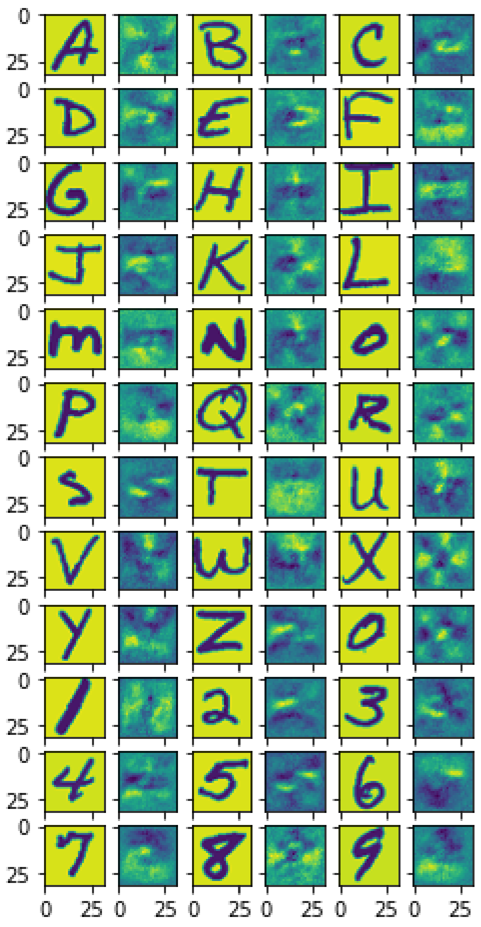


Weight after training:



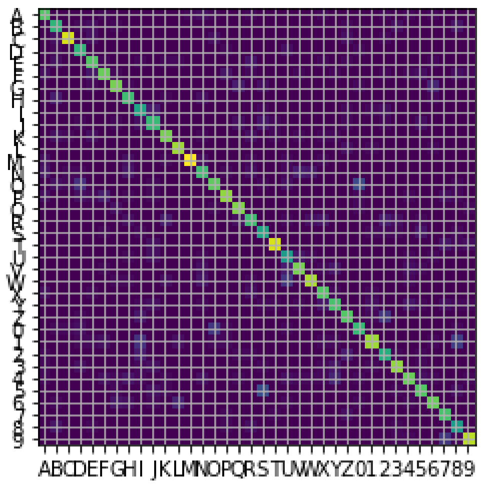
After the training, the weight shows some patterns.

Q3.4 (3 points Code+WriteUp)



From the image comparision, we can see the weight shows similar patterns with the actual letters.

Q3.5 (4 points Code+WriteUp)



From the plot, we can see that "O" and "0", "5" and "S" are the most commly confused pairs.

Q4 Extract Text from Images (35 points)

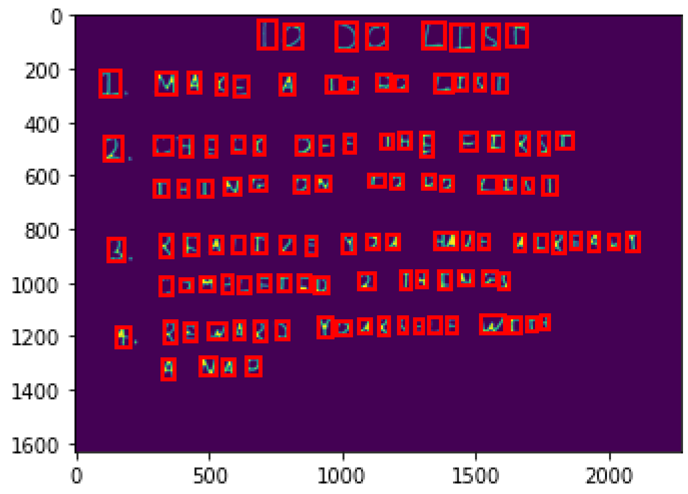
Q4.1 (3 points WriteUp)

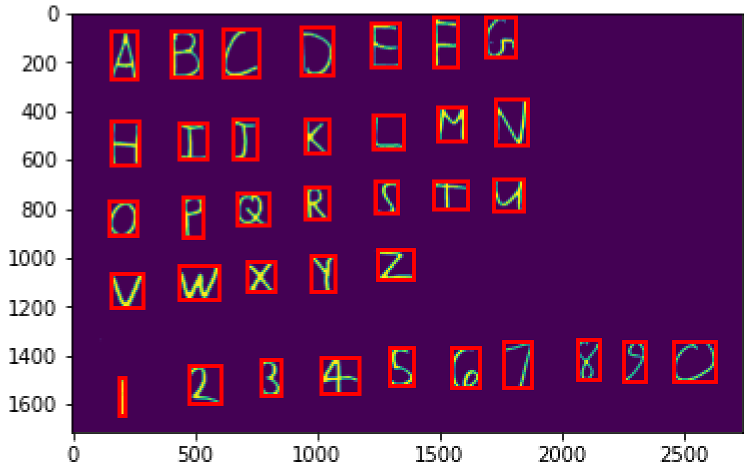
1. All characters must be seperate.
2. All components of a single character must be connected.

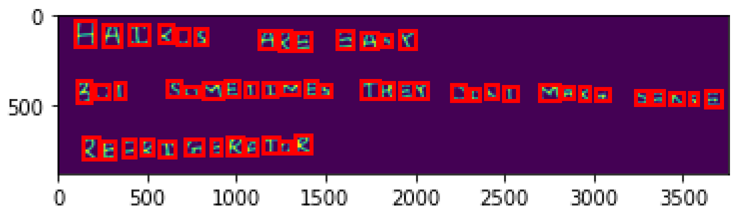
Q4.2 (13 points Code+WriteUp)

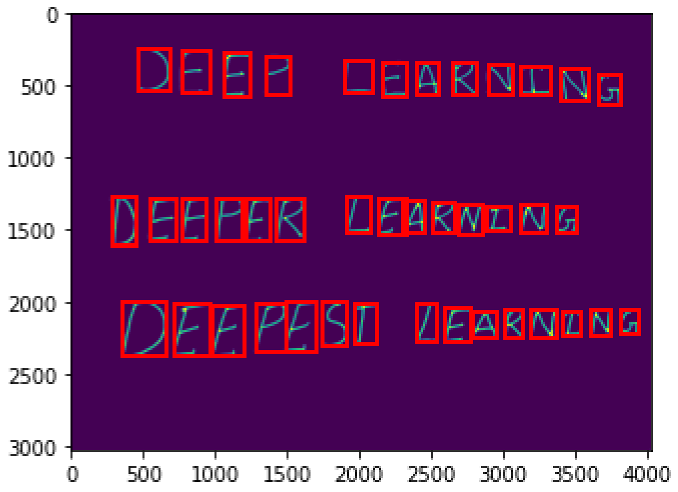
```
import numpy as np
import skimage
import skimage.measure
import skimage.color
import skimage.restoration
import skimage.io
import skimage.filters
import skimage.morphology
import skimage.segmentation

# takes a color image
# returns a list of bounding boxes and black_and_white image
def findLetters(image):
    bboxes = []
    bw = None
    # insert processing in here
    # one idea estimate noise -> denoise -> greyscale -> threshold -> mo
rphology -> label -> skip small boxes
    # this can be 10 to 15 lines of code using skimage functions
    # YOUR CODE HERE

    img = skimage.color.rgb2gray(image)
    thr = skimage.filters.threshold_otsu(img)
    bw = skimage.morphology.closing(img <= thr, skimage.morphology.squar
e(10))
    clr = skimage.segmentation.clear_border(bw)
    label_img = skimage.measure.label(clr)

    for reg in skimage.measure.regionprops(label_img):
        if (reg.area >= 100):
            bboxes.append(reg.bbox)

    # raise NotImplementedError()
    return bboxes, bw
```

Q4.3 (6 points WriteUp)









Q4.4 (13 points Code+WriteUp)

```
01_list.jpg
7QDOLIQT
INAREATDDDLIQ8
2LH2ERQEE8HER8RQ8
8H8NGQNTQDQEIQ7
3R2AE82EX0UM6EA6R6ADX
EQMP6886DZ1H8NGQ
5R8WARDX0URQE6EWD8B
ANAR

02_letters.jpg
AB6DE8G
HI8RLRN
08QRQTU
VWXXZ
8Z3QS6383Q

03_haiku.jpg
HAIR6QAREGASX
BQ7SQMETIMESTREXDDNTMAKESENQE
RBFRIGERA8QR

04_deep.jpg
DEE8LEARBING
DEEBERLBARNING
DEERES8LEARRING
```

Q5 Image Compression with Autoencoders [Extra Credit] (25 points)

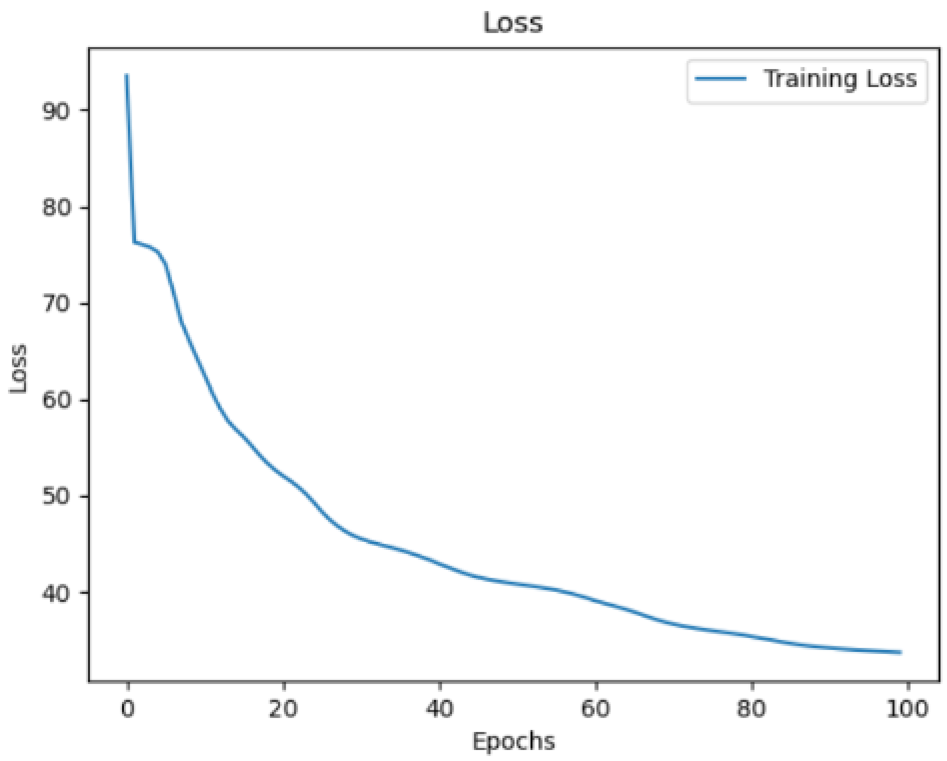
Q5.1.1 [Extra Credit](10 points Code)

```
import numpy as np
import scipy.io

# from ipynb.fs.defs.q2 import *
import import_ipynb
from q2 import *

from collections import Counter

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 = None
learning_rate = None
# YOUR CODE HERE

batch_size = 64
learning_rate = 4e-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, 'output')

params['m_layer1'] = np.zeros((1024, hidden_size))
params['m_layer2'] = np.zeros((hidden_size, hidden_size))
params['m_layer3'] = np.zeros((hidden_size, hidden_size))
params['m_output'] = np.zeros((hidden_size, 1024))

# raise NotImplementedError()
```

Q5.2 [Extra Credit](3 points Code+WriteUp)

```
# should look like your previous training loops
for itr in range(max_iters):
    total_loss = 0
    for xb,_ in batches:

        # training loop can be exactly the same as q2!
        # your loss is now squared error
        # delta is the d/dx of (x-y)^2
        # to implement momentum
        #   just use 'm_'+name variables
        #   to keep a saved value over timestamps
        #   params is a Counter(), which returns a 0 if an element is mi
ssing
        #   so you should be able to write your loop without any special
conditions
        # YOUR CODE HERE

        h1 = forward(xb, params, 'layer1', relu)
        h2 = forward(h1, params, 'layer2', relu)
        h3 = forward(h2, params, 'layer3', relu)
        output = forward(h3, params, 'output', sigmoid)
        loss = np.sum((xb - output) ** 2)
        total_loss += loss / train_x.shape[0]

        delta1 = 2 * (output - xb)
        delta2 = backwards(delta1, params, 'output', sigmoid_deriv)
        delta3 = backwards(delta2, params, 'layer3', relu_deriv)
        delta4 = backwards(delta3, params, 'layer2', relu_deriv)
        backwards(delta3, params, 'layer1', relu_deriv)

        params['m_layer1'] = 0.9 * params['m_layer1'] - learning_rate *
 params['grad_Wlayer1']
        params['m_layer2'] = 0.9 * params['m_layer2'] - learning_rate *
 params['grad_Wlayer2']
        params['m_layer3'] = 0.9 * params['m_layer3'] - learning_rate *
 params['grad_Wlayer3']
        params['m_output'] = 0.9 * params['m_output'] - learning_rate *
 params['grad_Woutput']

        params['Wlayer1'] += params['m_layer1']
        params['blayer1'] -= learning_rate * params['grad_blayer1']
        params['Wlayer2'] += params['m_layer2']
        params['blayer2'] -= learning_rate * params['grad_blayer2']
        params['Wlayer3'] += params['m_layer3']
        params['blayer3'] -= learning_rate * params['grad_blayer3']
        params['Woutput'] += params['m_output']
        params['boutput'] -= learning_rate * params['grad_boutput']

        # raise NotImplementedError()
    if itr % 2 == 0:
        print("itr: {:02d} \t loss: {:.2f}".format(itr,total_loss))
    if itr % lr_rate == lr_rate-1:
        learning_rate *= 0.9
```



The loss curve decreases very quick at first 10 epoches and gradually reduces the decrease speed.

Q5.3.1 [Extra Credit](4 points Code+WriteUp)

N/A

Q5.3.2 [Extra Credit](3 points Code+WriteUp)

N/A

Q6 Comparing against PCA [Extra Credit](15 Points)

Q6.1 [Extra Credit](4 points Code+WriteUp)

N/A

Q6.2 [Extra Credit](4 points Code+WriteUp)

N/A

Q6.3 [Extra Credit](4 points Code+WriteUp)

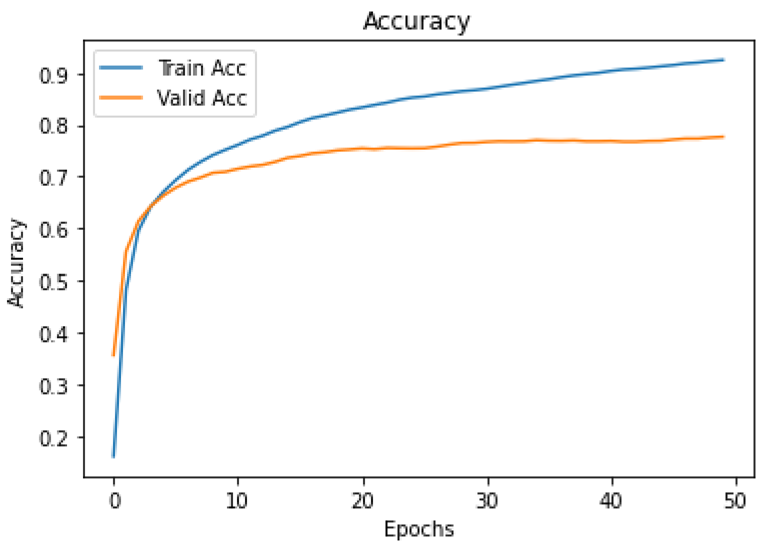
N/A

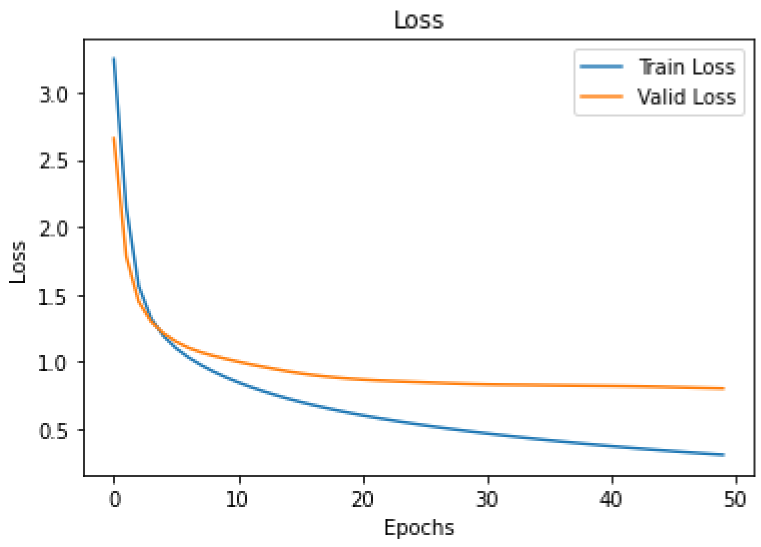
Q6.4 [Extra Credit](3 points Code+WriteUp)

N/A

Q7 PyTorch (40 points)

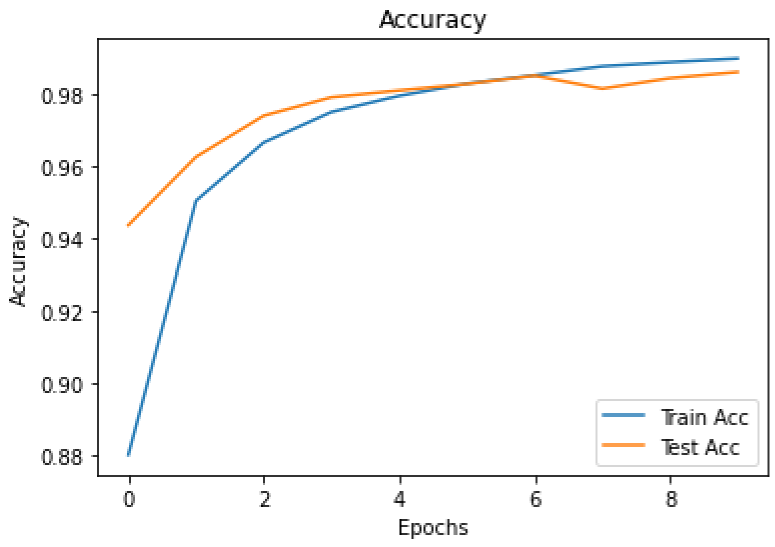
Q7.1.1 (10 points Code+WriteUp)

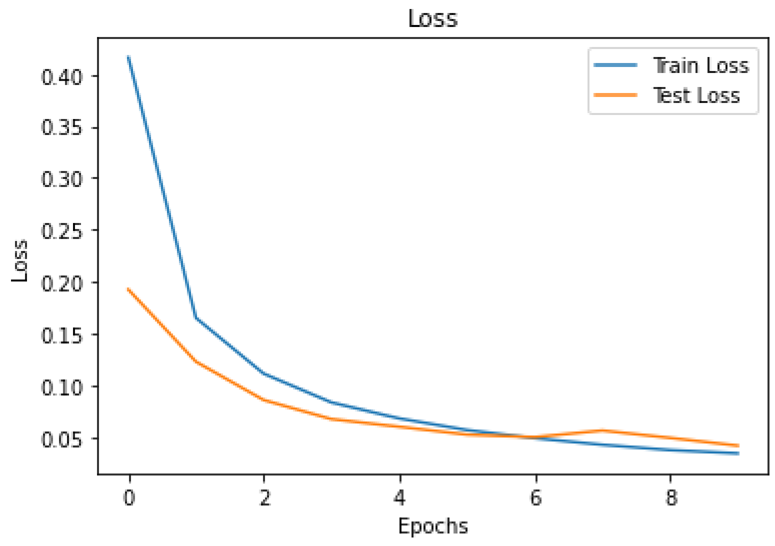




```
import numpy as np
import scipy.io

#from ipynb.fs.defs.q2 import *
import import_ipynb
from q2 import *

import torch
import torchvision.datasets
import matplotlib.pyplot as plt

# YOUR CODE HERE

train_data = scipy.io.loadmat('data/nist36_train.mat')
valid_data = scipy.io.loadmat('data/nist36_valid.mat')
test_data = scipy.io.loadmat('data/nist36_test.mat')

train_x, train_y = torch.from_numpy(train_data['train_data']), torch.fro
m_numpy(train_data['train_labels'])
valid_x, valid_y = torch.from_numpy(valid_data['valid_data']), torch.fro
m_numpy(valid_data['valid_labels'])
test_x, test_y = torch.from_numpy(test_data['test_data']), torch.from_nu
mpy(test_data['test_labels'])

class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = torch.nn.Linear(train_x.shape[1], 64)
        self.fc2 = torch.nn.Linear(64, train_y.shape[1])
        self.sigmoid = torch.nn.Sigmoid()

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

model = Net()
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

train_batches = get_random_batches(train_x, train_y, 16)

max_iters = 50
train_loss_mem = []
train_acc_mem = []
valid_loss_mem = []
valid_acc_mem = []

for iters in range(max_iters):
    for phase in ['train', 'valid']:
        total_loss = 0
        total = 0
        correct = 0

        if phase == 'train':
            batches = train_batches
            model.train()
        else:
            batches = [(valid_x, valid_y)]
            model.eval()

        for xb, yb in batches:
            inputs, labels = xb.float(), np.argmax(yb.float(), axis=1)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            if phase == 'train':
                loss.backward()
                optimizer.step()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            total_loss += loss.item() * inputs.size(0)

        if phase == 'train':
            print("Train Acc:", correct/total, " Train Loss:", total_los
s/total)
            train_acc_mem.append(correct/total)
            train_loss_mem.append(total_loss/total)
        else:
            print("Valid Acc:", correct/total, " Valid Loss:", total_los
s/total)
            valid_acc_mem.append(correct/total)
            valid_loss_mem.append(total_loss/total)

print('Validation accuracy: ', correct/total)

plt.title('Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.plot(train_acc_mem, label='Train Acc')
plt.plot(valid_acc_mem, label='Valid Acc')
plt.legend()
plt.show()

plt.title('Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.plot(train_loss_mem, label='Train Loss')
plt.plot(valid_loss_mem, label='Valid Loss')
plt.legend()
plt.show()

# raise NotImplementedError()
```

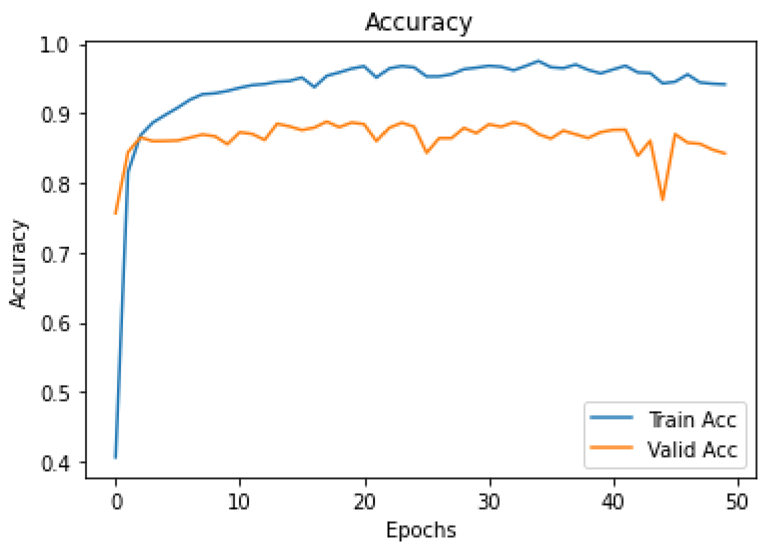
Q7.1.2 (3 points Code+WriteUp)

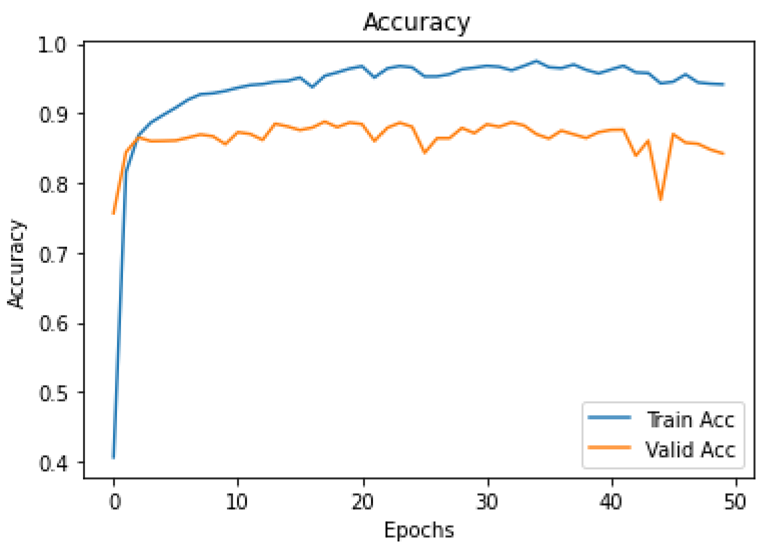




```
import numpy as np
import scipy.io
import torch
import torchvision
from torch import nn, optim
import torch.nn.functional as F  # a lower level (compared to torch.nn)
 interface
from torch.utils.data import Dataset, DataLoader
from torchvision.datasets import MNIST
import torchvision.transforms as transforms
from time import time
import copy

# Use GPU if available, otherwise stick with cpu
use_cuda = torch.cuda.is_available()
torch.manual_seed(123)
device = torch.device("cuda" if use_cuda else "cpu")
print("device = {}".format(device))

print("Get dataset")
mnist_train = MNIST(root="data", train=True, download=True, transform=tr
ansforms.ToTensor())
trainset_loader = DataLoader(mnist_train, batch_size=16, shuffle=True, n
um_workers=1)

mnist_test = MNIST(root="data", train=False, download=True, transform=tr
ansforms.ToTensor())
testset_loader = DataLoader(mnist_test, batch_size=16, shuffle=True, num
_workers=1)

print("dataset size train, test")
print(trainset_loader.dataset.data.shape)
print(testset_loader.dataset.data.shape)

# YOUR CODE HERE

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv = nn.Conv2d(1, 16, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(16*12*12, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv(x)))
        x = x.view(-1, 16*12*12)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

Best_state = copy.deepcopy(model.state_dict())
best_acc = 0

max_iters = 10
train_loss_mem = []
train_acc_mem = []
valid_loss_mem = []
valid_acc_mem = []

for itr in range(max_iters):
    for phase in ['train', 'valid']:
        if phase == 'train':
            model.train()
        else:
            model.eval()

        total_loss = 0
        total = 0
        total_cor = 0

        if phase == 'train':
            dataset_loader = trainset_loader
        else:
            dataset_loader = testset_loader

        for i, data in enumerate(dataset_loader, 0):
            inputs, labels = data
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            if phase == 'train':
                loss.backward()
                optimizer.step()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            total_cor += (predicted == labels).sum().item()
            total_loss += loss.item() * inputs.size(0)

        if phase == 'train':
            print("Train Acc:", total_cor/total, " Train Loss:", total_l
oss/total)
            train_acc_mem.append(total_cor/total)
            train_loss_mem.append(total_loss/total)
        else:
            acc = total_cor/total
            if acc > best_acc:
                best_acc = acc
                best_model_state = copy.deepcopy(model.state_dict())

            print("Valid Acc:", total_cor/total, " Valid Loss:", total_l
oss/total)
            valid_acc_mem.append(total_cor/total)
            valid_loss_mem.append(total_loss/total)

print('Testing accuracy: ',best_acc)

plt.title('Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.plot(train_acc_mem, label='Train Acc')
plt.plot(valid_acc_mem, label='Test Acc')
plt.legend()
plt.show()

plt.title('Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.plot(train_loss_mem, label='Train Loss')
plt.plot(valid_loss_mem, label='Test Loss')
plt.legend()
plt.show()

# raise NotImplementedError()
```

Q7.1.3 (2 points Code+WriteUp)




Accuracy
Accuracy
Epochs
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0
10
20
30
40
50
Train Acc
Valid Acc

```
import numpy as np
import scipy.io
import torch
from torch import nn, optim
import torch.nn.functional as F  # a lower level (compared to torch.nn)
 interface
from torch.utils.data import Dataset, DataLoader
import torchvision
import torchvision.transforms as transforms
from time import time
import matplotlib.pyplot as plt

# Use GPU if available, otherwise stick with cpu
use_cuda = torch.cuda.is_available()
torch.manual_seed(123)
device = torch.device("cuda" if use_cuda else "cpu")
print("device = {}".format(device))

print("Get dataset")
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'].astype(np.float32), train_
data['train_labels'].astype(np.int)
# valid_x, valid_y = valid_data['valid_data'].astype(np.float32), valid_
data['valid_labels'].astype(np.int)

# To do

train_x, train_y = torch.from_numpy(train_data['train_data']), torch.fro
m_numpy(train_data['train_labels'])
valid_x, valid_y = torch.from_numpy(valid_data['valid_data']), torch.fro
m_numpy(valid_data['valid_labels'])

# YOUR CODE HERE

train_batches = get_random_batches(train_x, train_y, 16)

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(16 * 5 * 5, 64)
        self.fc2 = nn.Linear(64, train_y.shape[1])

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

max_iters = 50
train_loss_mem = []
train_acc_mem = []
valid_loss_mem = []
valid_acc_mem = []

for itr in range(max_iters):
    for phase in ['train', 'valid']:
        running_loss = 0.0
        total = 0
        correct = 0

        if phase == 'train':
            batches = train_batches
            model.train()
        else:
            batches = [(valid_x, valid_y)]
            model.eval()

        for xb, yb in batches:
            inputs, labels = xb.float(), np.argmax(yb.float(), axis=1)
            inputs = inputs.view(inputs.shape[0], 1, 32, 32)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            if phase == 'train':
                loss.backward()
                optimizer.step()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            running_loss += loss.item() * inputs.size(0)

        if phase == 'train':
            print("Train Acc:", correct / total, " Train Loss:", running
_loss / total)
            train_acc_mem.append(correct / total)
            train_loss_mem.append(running_loss / total)
        else:
            print("Valid Acc:", correct / total, " Valid Loss:", running
_loss / total)
            valid_acc_mem.append(correct / total)
            valid_loss_mem.append(running_loss / total)

print('validate accuracy: ',best_acc)

plt.title('Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.plot(train_acc_mem, label='Train Acc')
plt.plot(valid_acc_mem, label='Valid Acc')
plt.legend()
plt.show()

plt.title('Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.plot(train_loss_mem, label='Train Loss')
plt.plot(valid_loss_mem, label='Valid Loss')
plt.legend()
plt.show()

# raise NotImplementedError()
```

Q7.1.4 (15 points Code+WriteUp)

```
01_list.jpg
rOOO7HWr
4EtTMtrOOO7YNr
UnInnTOTT4InTYnnr
7IHSQOSrOOOgMNf
WrnKTnnraDZZZn27aMeOX
OOET7MrrOYFZHSQN
4Ir3eaOXO2rnn7T3HrI
eStO
Accuracy: 0.09649122807017543

02_letters.jpg
tOODMTQ
IHPT7ES
OrQrNtZ
D3XXn
tgW4nOCnWQ
Accuracy: 0.1388888888888889
```

```
03_haiku.jpg
ICHXDNC0MMCNr
ODrNOEMrHEMNrZM2OOSrECTMNMSN9
TnTOHQMaCFOa
Accuracy: 0.05555555555555555

04_deep.jpg
OMTr7MtnSHIQ
GrnLrJnngTSrSQ
OrnKrNfGRKJSHSR
Accuracy: 0.0
```

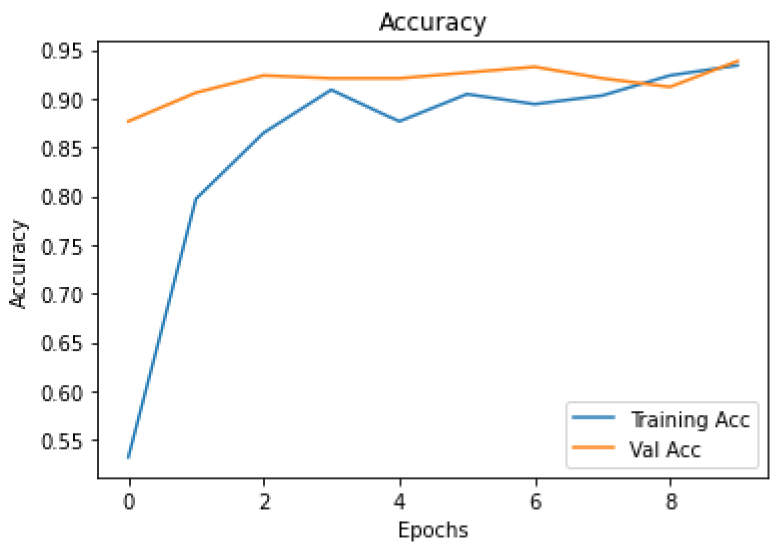
```
import numpy as np
import scipy.io
import torch
from torch import nn, optim
import torchvision
import torch.nn.functional as F  # a lower level (compared to torch.nn)
 interface
from torch.utils.data import Dataset, DataLoader
from torchvision.datasets import EMNIST
import torchvision.transforms as transforms
from time import time
import os
import skimage
import matplotlib
# from q4.ipynb import *

# Use GPU if available, otherwise stick with cpu
use_cuda = torch.cuda.is_available()
torch.manual_seed(123)
device = torch.device("cuda" if use_cuda else "cpu")
print("device = {}".format(device))

print("Get dataset")

EMNIST.url = 'http://www.itl.nist.gov/iaui/vip/cs_links/EMNIST/gzip.zip'
# Reference for transform function
# https://stackoverflow.com/a/54513835
transform=torchvision.transforms.Compose([
    lambda img: torchvision.transforms.functional.rotate(img, -90),
    lambda img: torchvision.transforms.functional.hflip(img),
    torchvision.transforms.ToTensor()
])
emnist_train = EMNIST(root="data", split='balanced', train=True, downloa
d=True, transform=transform)
trainset_loader = DataLoader(emnist_train, batch_size=20, shuffle=True,
num_workers=1)

emnist_test = EMNIST(root="data", split='balanced', train=False, downloa
d=True, transform=transform)
testset_loader = DataLoader(emnist_test, batch_size=20, shuffle=True, nu
m_workers=1)

# Ref: https://github.com/gaurav0651/emnist/blob/master/train_emnist.ipy
nb
label_map = np.array(['0','1','2','3','4','5','6','7','8','9',
       'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P',
'Q','R','S','T','U','V','W','X','Y','Z',
       'a','b','d','e','f','g','h','n','q','r','t'])
```

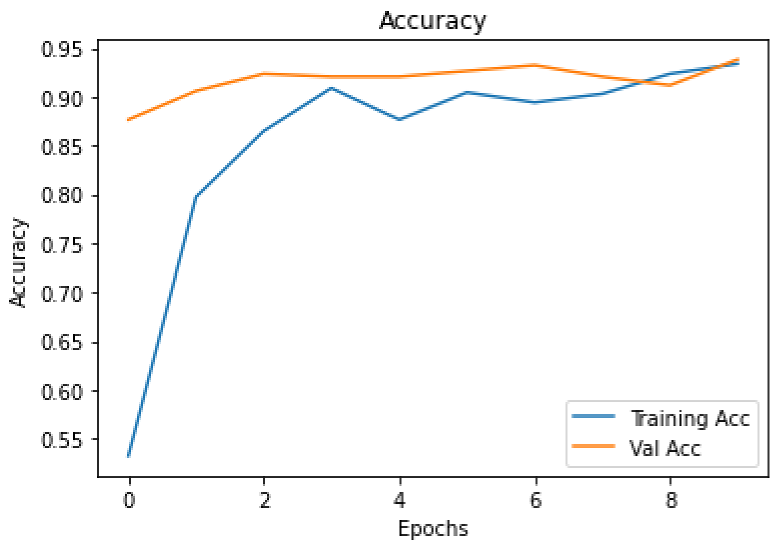
```
print(trainset_loader.dataset.data.shape)
print(trainset_loader.dataset.data.shape)
print(testset_loader.dataset.data.shape)

# YOUR CODE HERE

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv = nn.Conv2d(1, 16, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(16 * 12 * 12, 128)
        self.fc2 = nn.Linear(128, 47)

    def forward(self, x):
        x = self.pool(F.relu(self.conv(x)))
        x = x.view(-1, 16 * 12 * 12)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.005, momentum=0.9)

max_iters = 5

for itr in range(max_iters):
    loss_sum = 0
    for i, data in enumerate(trainset_loader):
        inputs, labels = data

        batch_imgs = inputs[:, 0, :, :].detach().numpy()
        batch_imgs = batch_imgs.reshape(batch_imgs.shape[0], -1)
        batch_imgs = -(batch_imgs - np.mean(batch_imgs, axis=1).reshape(
-1, 1))/np.std(batch_imgs, axis=1).reshape(-1, 1)
        inputs = torch.from_numpy(batch_imgs.reshape(batch_imgs.shape[0
], 1, 28, 28))
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        loss_sum +=loss.item() * inputs.shape[0]

    if itr % 1 == 0:
        print("valid itr: {:02d} \t loss: {:.2f}".format(itr, loss_sum/l
en(trainset_loader.dataset)))
```

```
print('Finished Training')
torch.save(model.state_dict(), "q7_1_4.pth")

model.load_state_dict(torch.load("q7_1_4.pth"))
model.eval()

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)
    for bbox in bboxes:
        minr, minc, maxr, maxc = bbox

        rect = matplotlib.patches.Rectangle((minc, minr), maxc - minc, m
axr - minr,
                                            fill=False, edgecolor='red',
linewidth=2)
        plt.gca().add_patch(rect)
    plt.show()

    bboxes = sorted(bboxes, key=lambda x: x[0])
    prev_y1, prev_x1, prev_y2, prev_x2 = bboxes[0]
    rows = [[bboxes[0]]]
    for i in range(1, len(bboxes)):
        y1, x1, y2, x2 = bboxes[i]
        if y1 > prev_y2:
            rows.append([bboxes[i]])
        else:
            rows[-1].append(bboxes[i])

        prev_y1, prev_x1, prev_y2, prev_x2 = y1, x1, y2, x2

    for i in range(0, len(rows)):
        rows[i] = sorted(rows[i], key=lambda x: x[1])

    crop_rows = []
    for row in rows:
        crops = []
        for bbox in row:
            y1, x1, y2, x2 = bbox
            side = max(y2-y1, x2-x1)
            center_y, center_x = (y2+y1) / 2, (x2+x1) / 2
            crop = bw[int(center_y-side/2): int(center_y+side/2), int(ce
nter_x-side/2):int(center_x+side/2)] * (-1.)
            crop = skimage.transform.resize(crop, (20, 20))
            crop = np.pad(crop, ((4, 4), (4, 4)), constant_values=crop.m
ax())
```

```
                crop = crop.T.flatten()
                crops.append(crop)

            crops = np.array(crops)
            crops = (crops - np.mean(crops, axis=1).reshape(-1, 1)) / np.std
(crops, axis=1).reshape(-1, 1)
            crop_rows.append(crops)

        print(img)

        correct = None
        if img == '02_letters.jpg':
            correct = ["ABCDEFG", "HIJKLMN", "OPQRSTU", "VWXYZ", "123456789
0"]
        elif img == '04_deep.jpg':
            correct = ["DEEPLEARNING", "DEEPERLEARNING", "DEEPESTLEARNING"]
        elif img == '01_list.jpg':
            correct = ["TODOLIST", "1MAKEATODOLIST", "2CHECKOFFTHEFIRST", "T
HINGONTODOLIST", "3REALIZEYOUHVEALREADY",
                       "COMPLETED2THINGS", "4REWARDYOURSELFWITH", "ANAP"]
        elif img == '03_haiku.jpg':
            correct = ["HAIKUSAREEASY", "BUTSOMETIMESTHEYDONTMAKESENSE", "RE
FRIGERATOR"]

        count = 0
        total = 0
        for idx, crops in enumerate(crop_rows):
            crops = crops.reshape(-1, 1, 28, 28)
            outputs = model(torch.from_numpy(crops).float())
            _, predicted = torch.max(outputs.data, 1)
            output = label_map[predicted]
            output = ''.join(output)
            print(output)

            total += len(output)
            if correct is not None:
                for i in range(len(output)):
                    if output[i] == correct[idx][i]:
                        count += 1

        print("Accuracy:", count / total)
        print()

    # raise NotImplementedError()
```

Q7.2.1 (10 points WriteUp)


Accuracy
Accuracy
Epochs
Training Acc
Val Acc
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0
2
4
6
8


Accuracy
Accuracy
Epochs
Training Acc
Val Acc
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0
2
4
6
8

```
# Code for fine-tune squeezenet1_1
# YOUR CODE HERE

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader

# from nn import *
# from q4 import *

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches
import torch
from torchvision import transforms, datasets
import torchvision
import copy



train_transform = transforms.Compose([transforms.RandomResizedCrop(224),
transforms.RandomHorizontalFlip(), transforms.ToTensor(), transforms.Nor
malize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])
val_transform = transforms.Compose([transforms.Resize(256), transforms.C
enterCrop(224), transforms.ToTensor(), transforms.Normalize(mean=[0.485,
0.456, 0.406], std=[0.229, 0.224, 0.225])])

flower17_trainset = datasets.ImageFolder(root='data/oxford-flowers17/tra
in', transform=train_transform)
train_loader = torch.utils.data.DataLoader(flower17_trainset, batch_size
=8, shuffle=True, num_workers=4)
flower17_valset = datasets.ImageFolder(root='data/oxford-flowers17/val',
transform=val_transform)
val_loader = torch.utils.data.DataLoader(flower17_valset, batch_size=8,
shuffle=False, num_workers=4)
flower17_testset = datasets.ImageFolder(root='data/oxford-flowers17/tes
t', transform=val_transform)
test_loader = torch.utils.data.DataLoader(flower17_testset, batch_size=8
, shuffle=False, num_workers=4)


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.conv3 = nn.Conv2d(16, 32, 5)
        self.fc1 = nn.Linear(32 * 24 * 24, 512)
        self.fc2 = nn.Linear(512, 17)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = x.view(-1, 32 * 24 * 24)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x


def main():
    model = torchvision.models.squeezenet1_1(pretrained=True)
    for param in model.parameters():
        param.requires_grad = False
    model.classifier[1] = nn.Conv2d(512, 17, kernel_size=(1, 1), stride=
(1, 1))
    model.num_classes = 17

    params_to_update = []
    for name, param in model.named_parameters():
        if param.requires_grad:
            params_to_update.append(param)

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(params_to_update, lr=0.001, momentum=0.9)

    best_model_state = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    max_iters = 10
    train_acc_history = []
    train_loss_history = []
    val_acc_history = []
    val_loss_history = []

    for iter in range(max_iters):
        for phase in ['train', 'valid']:
            if phase == 'train':
                model.train()
            else:
                model.eval()

            running_loss = 0.
            running_total = 0.
            running_correct = 0.
```

```
                if phase == 'train':
                    dataset_loader = train_loader
                else:
                    dataset_loader = val_loader

                for i, data in enumerate(dataset_loader, 0):
                    inputs, labels = data

                    optimizer.zero_grad()

                    outputs = model(inputs)
                    loss = criterion(outputs, labels)
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                    _, predicted = torch.max(outputs.data, 1)
                    running_total += labels.size(0)
                    running_correct += (predicted == labels).sum().item()
                    running_loss += loss.item() * inputs.size(0)

                if phase == 'train':
                    print("Train Acc:", running_correct / running_total, " T
rain Loss:", running_loss / running_total)
                    train_acc_history.append(running_correct / running_total
)
                    train_loss_history.append(running_loss / running_total)
                else:
                    acc = running_correct / running_total
                    if acc > best_acc:
                        best_acc = acc
                        best_model_state = copy.deepcopy(model.state_dict())

                    print("Valid Acc:", running_correct / running_total, " V
alid Loss:", running_loss / running_total)
                    val_acc_history.append(running_correct / running_total)
                    val_loss_history.append(running_loss / running_total)

        plt.title('Accuracy')
        plt.xlabel('Epochs')
        plt.ylabel('Accuracy')
        plt.plot(train_acc_history, label='Training Acc')
        plt.plot(val_acc_history, label='Val Acc')
        plt.legend()
        plt.show()

        plt.title('Loss')
        plt.xlabel('Epochs')
        plt.ylabel('Loss')
        plt.plot(train_loss_history, label='Training Loss')
        plt.plot(val_loss_history, label='Val Loss')
        plt.legend()
        plt.show()

        torch.save(best_model_state, "q7_2_fine_tuned.pth")
        model.load_state_dict(torch.load("q7_2_fine_tuned.pth"))

        model.eval()
        total = 0
        correct = 0
        for i, data in enumerate(test_loader, 0):
            inputs, labels = data
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        print()
        print("Test Accuracy:", correct / total)

    # raise NotImplementedError()
```

Appendix: Neural Network Overview

Deep learning has quickly become one of the most applied machine learning techniques in computer vision. Convolutional neural networks have been applied to many different computer vision problems such as image classification, recognition, and segmentation with great success. In this assignment, you will first implement a fully connected feed forward neural network for hand written character classification. Then in the second part, you will implement a system to locate characters in an image, which you can then classify with your deep network. The end result will be a system that, given an image of hand written text, will output the text contained in the image.

Basic Use

Here we will give a brief overview of the math for a single hidden layer feed forward network. For a more detailed look at the math and derivation, please see the class slides.

A fully-connected network $\mathbf{f}$, for classification, applies a series of linear and non-linear functions to an input data vector $\mathbf{x}$ of size $N \times 1$ to produce an output vector $\mathbf{f}(\mathbf{x})$ of size $C \times 1$, where each element i of the output vector represents the probability of $\mathbf{x}$ belonging to the class i. Since the data samples are of dimensionality N, this means the input layer has N input units. To compute the value of the output units, we must first compute the values of all the hidden layers. The first hidden layer *pre-activation* $\mathbf{a}^{(1)}(\mathbf{x})$ is given by

$$\mathbf{a}^{(1)}(\mathbf{x}) = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$

Then the *post-activation* values of the first hidden layer $\mathbf{h}^{(1)}(\mathbf{x})$ are computed by applying a non-linear activation function $\mathbf{g}$ to the *pre-activation* values

$$\mathbf{h}^{(1)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(1)}(\mathbf{x})) = \mathbf{g}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$$

Subsequent hidden layer ($1 < t \leq T$) pre- and post activations are given by:

$$\mathbf{a}^{(t)}(\mathbf{x}) = \mathbf{W}^{(t)}\mathbf{h}^{(t-1)} + \mathbf{b}^{(t)}$$

$$\mathbf{h}^{(t)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(t)}(\mathbf{x}))$$

The output layer *pre-activations* $\mathbf{a}^{(T)}(\mathbf{x})$ are computed in a similar way

$$\mathbf{a}^{(T)}(\mathbf{x}) = \mathbf{W}^{(T)}\mathbf{h}^{(T-1)}(\mathbf{x}) + \mathbf{b}^{(T)}$$

and finally the \emph{post-activation} values of the output layer are computed with

$$\mathbf{f}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(T)}(\mathbf{x})) = \mathbf{o}(\mathbf{W}^{(T)}\mathbf{h}^{(T-1)}(\mathbf{x}) + \mathbf{b}^{(T)})$$

where $\mathbf{o}$ is the output activation function. Please note the difference between $\mathbf{g}$ and $\mathbf{o}$! For this assignment, we will be using the sigmoid activation function for the hidden layer, so:

$$\mathbf{g}(y) = \frac{1}{1 + \exp(-y)}$$

where when $\mathbf{g}$ is applied to a vector, it is applied element wise across the vector.

Since we are using this deep network for classification, a common output activation function to use is the softmax function. This will allow us to turn the real value, possibly negative values of $\mathbf{a}^{(T)}(\mathbf{x})$ into a set of probabilities (vector of positive numbers that sum to 1). Letting $\mathbf{x}_i$ denote the i^{th} element of the vector $\mathbf{x}$, the softmax function is defined as:

$$\mathbf{o}_i(\mathbf{y}) = \frac{\exp(\mathbf{y}_i)}{\sum_j \exp(\mathbf{y}_j)}$$

Samples from NIST Special 19 dataset

Gradient descent is an iterative optimisation algorithm, used to find the local optima. To find the local minima, we start at a point on the function and move in the direction of negative gradient (steepest descent) till some stopping criteria is met.

Backprop

The update equation for a general weight $W_{ij}^{(t)}$ and bias $b_i^{(t)}$ is

$$W_{ij}^{(t)} = W_{ij}^{(t)} - \alpha * \frac{\partial L_{\mathbf{f}}}{\partial W_{ij}^{(t)}}(\mathbf{x}) \qquad b_i^{(t)} = b_i^{(t)} - \alpha * \frac{\partial L_{\mathbf{f}}}{\partial b_i^{(t)}}(\mathbf{x})$$

α is the learning rate. Please refer to the back-propagation slides for more details on how to derive the gradients. Note that here we are using softmax loss (which is different from the least square loss in the slides).

References

[1] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. 2010. http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf (http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf).

[2] P. J. Grother. Nist special database 19 – handprinted forms and characters database. https://www.nist.gov/srd/nist-special-database-19 (https://www.nist.gov/srd/nist-special-database-19), 1995.