16-720A — Spring 2021 — Homework 5

Jen-Hung Ho
jenhungh@andrew.cmu.edu

April 25, 2021

Part 1

Q1.1

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{d} e^{x_i}} \tag{1}$$

Apply translation c to the softmax function:

$$softmax(x_i + c) = \frac{e^{x_i+c}}{\sum_{j=1}^{d} e^{x_i+c}} = \frac{e^c e^{x_i}}{e^c \sum_{j=1}^{d} e^{x_i}} = \frac{e^{x_i}}{\sum_{j=1}^{d} e^{x_i}} = softmax(x_i) \tag{2}$$

Equation 2 proves that softmax function is invariant to translation. If we set c to 0, the range of values of the numerator e^{x_i} is $(0, +\infty)$. If we set c to -max(x_i), the range of values of the numerator e^{x_i} is (0, 1]. This prevents the "overflow" of the softmax function.

Q1.2

For softmax function, the range of each element is (0, 1], and the sum over all element is 1. One could say that "softmax takes an arbitrary real valued vector x and turns it into a **probability distribution**."
Three step process of softmax:
(1) $s_i = e^{x_i}$: Compute the exponential value of each element, which represents the outcome frequency of x_i.
(2) $S = \sum s_i$: Compute the sum of the exponential values of every elements, which represents the total frequency.
(3) Divide s_i by S to compute the probability of each element x_i.

Q1.3

A 2-layer MLP without activation function:

$$f_1(x) = A_1 x + b_1 \tag{3}$$
$$f_2(f_1(x)) = A_2(f_1(x)) + b_2 = A_2(A_1 x + b_1) + b_2 = (A_2 A_1)x + (A_2 b_1 + b_2) = A^* x + b^* \tag{4}$$
$$where \;\; A^* = A_2 A_1 \;\; and \;\; b^* = A_2 b_1 + b_2 \tag{5}$$

From equation 4 and 5, we prove that a 2-layer MLP without activation function is still a linear classifier. Since the linearity applies to more layers by induction, we prove that a MLP without activation function is still a linear classifier.

Q1.4

$$\sigma(y) = \frac{1}{1+e^{-y}} \tag{6}$$

$$\frac{d\sigma(y)}{dy} = \frac{d}{dy}(\frac{1}{1+e^{-y}}) = \frac{0-(-e^{-y})}{(1+e^{-y})^2} = \frac{(1+e^{-y})-1}{1+e^{-y}}\frac{1}{1+e^{-y}} \tag{7}$$

$$= (1-\frac{1}{1+e^{-y}})\frac{1}{1+e^{-y}} = (1-\sigma(y))\sigma(y) \tag{8}$$

Q1.5

From the notional suggestions, we know that

$$\frac{\partial J}{\partial y} = \delta \in \mathbb{R}^{k\times 1} \tag{9}$$

$$W \in \mathbb{R}^{k\times d} \tag{10}$$

$$x \in \mathbb{R}^{d\times 1} \tag{11}$$

$$b \in \mathbb{R}^{k\times 1} \tag{12}$$

Given y = Wx + b, we know that y_i = W_{ij} x_j + b_j. First compute

$$\frac{\partial J}{\partial x_j} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial x_j} = \delta W_{ij} \tag{13}$$

$$\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial W_{ij}} = \delta x_j \tag{14}$$

$$\frac{\partial J}{\partial b_i} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial b_i} = \delta * 1 = \delta \tag{15}$$

Then derive the matrix form

$$\frac{\partial J}{\partial x} = W^T\delta \in \mathbb{R}^{d\times 1} \tag{16}$$

$$\frac{\partial J}{\partial W} = \delta x^T \in \mathbb{R}^{k\times d} \tag{17}$$

$$\frac{\partial J}{\partial b} = \delta \in \mathbb{R}^{k\times 1} \tag{18}$$

Q1.6

(1) As shown in Figure 1, sigmoid function is in range (0, 1) and its derivative is in range (0, 0.25]. Thus, when the neural network is deep, and we keep doing back-propagation (multiply gradients that are less than 1),there will be a "vanishing gradient" problem.

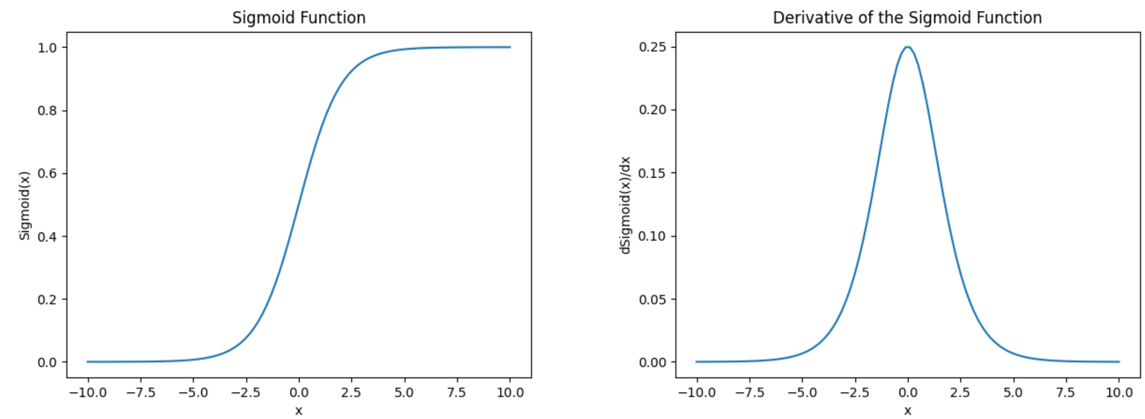

Figure 1: Sigmoid Function and its Derivative

(2) As shown in Figure 1 and 2, sigmoid function is in range (0, 1) and tanh function is in range (-1, 1). We prefer tanh function because it could reach the negative part (when $x < 0$, $tanh(x) < 0$) and it is also zero-centered.

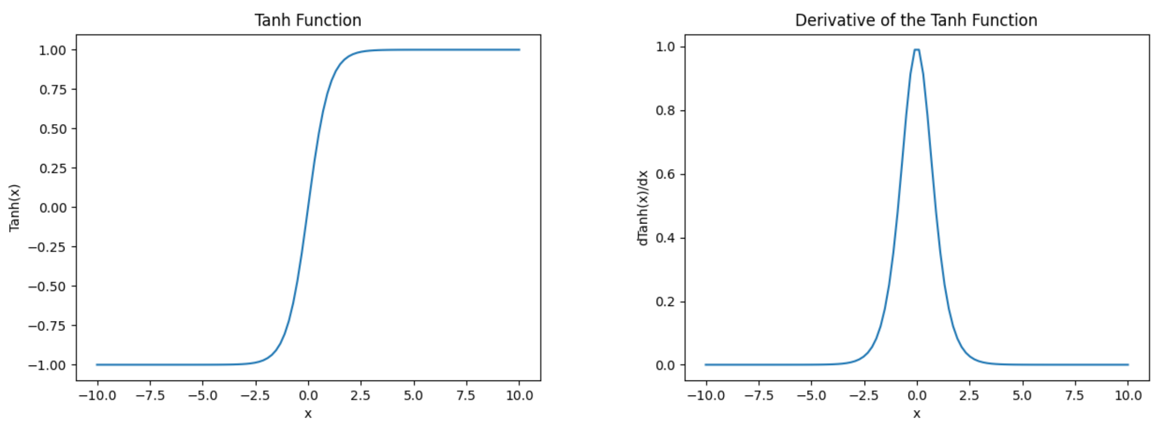

Figure 2: Tanh Function and its Derivative

(3) As shown in Figure 1 and 2, the derivativeof the sigmoid function is in range (0, 0.25] and the derivative of the tanh function is in range (0, 1]. Since the range of the gradients of tanh function is larger than sigmoid function, it has less of a vanishing gradient problem.
(4)

$$\sigma(2y) = \frac{1}{1+e^{-2y}} \tag{19}$$

$$tanh(y) = \frac{1-e^{-2y}}{1+e^{-2y}} = \frac{1+e^{-2y}-2-2e^{-2y}+2}{1+e^{-2y}} = 2(\frac{1}{1+e^{-2y}}) - 1 = 2\sigma(2y) - 1 \tag{20}$$

(5) ReLU is the Rectified Linear Unit (as shown in Figure 3).

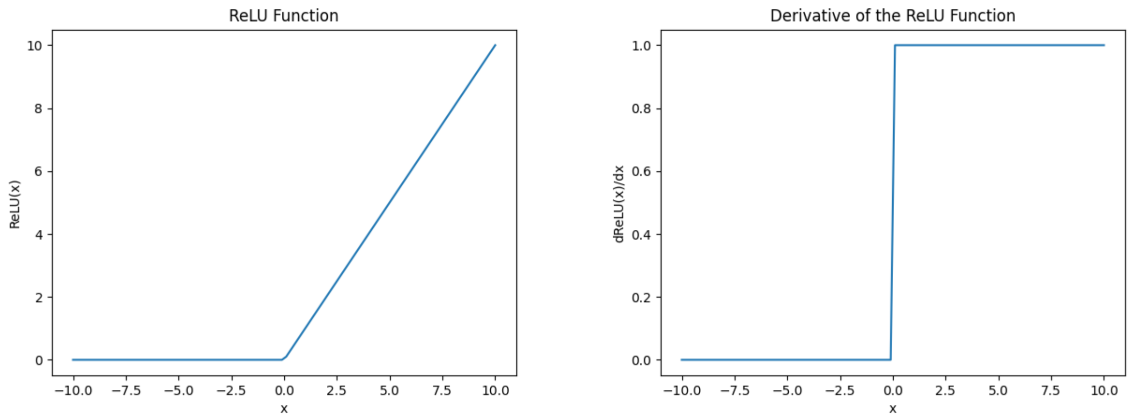

Figure 3: ReLU Function and its Derivative

$$ReLU(x) = max(0, x) \tag{21}$$

Some benefits of the ReLU function:
1. ReLU solves the vanishing gradient problem that Sigmoid and Tanh have.
2. The computational cost of ReLU (Rectified Linear Unit) is lower than Sigmoid and Tanh since it doesn't require exponential computation.

Part 2

Q2.1.1

If we initialize the network with all zeros, the model will generate all outputs as 0 after the forward propagation. When doing back propagation, the gradients of each of the perceptron are the same. Thus, all parameters will have same updates. This makes the neural network symmetric and unable to train.
If we initialize the weight and bias to the same constant value instead of zero, the training result will still be poor. Since the updates of all parameters are the same, the model fails to break the symmetry.

Q2.1.3

The random initialization based on input and output size creates randomness for each layer and thus break the symmetry of the model. Scaling the random initialization based on input and output size works as a normalization method and keeps the variance of the gradients in a certain range, which prevents gradient vanishing/explosion and increases the training efficiency.

Q2.4

```
####################################################
######              Training Loop       ##########
####################################################

itr: 00          loss: 69.07     acc : 0.00
itr: 100         loss: 38.67     acc : 0.67
itr: 200         loss: 30.92     acc : 0.67
itr: 300         loss: 26.28     acc : 0.67
itr: 400         loss: 23.26     acc : 0.72
itr: 500         loss: 20.87     acc : 0.85

####################################################
######              Training Loop End   ##########
####################################################
```

Figure 4: Screenshot of the Output of the Training Loop

Q2.5

```
####################################################
#####      Numerical Gradient Checker    ##########
####################################################

grad_Woutput 1.11e-06
grad_boutput 2.12e-09
grad_Wlayer1 2.64e-06
grad_blayer1 1.99e-06
total 5.75e-06

####################################################
#####     Numerical Gradient Checker END ##########
####################################################
```

Figure 5: Screenshot of the Output of the Numerical Gradient Checker

Q3.1

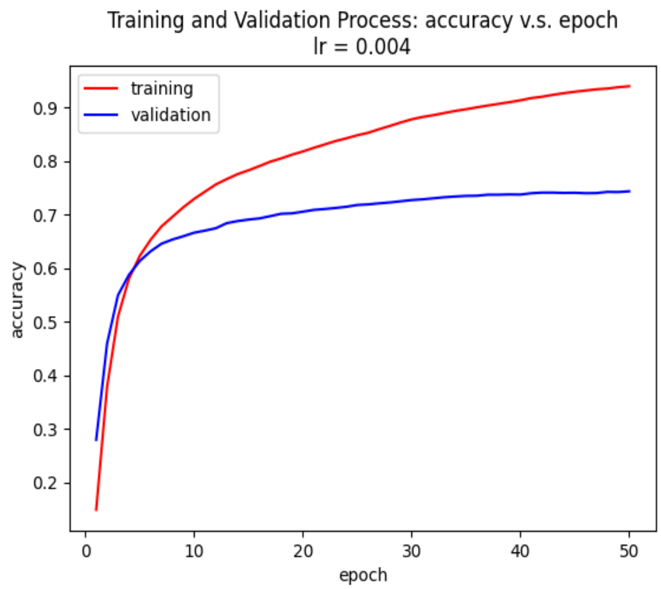

Figure 6: Training and Validation Process (Accuracy vs Epoch)

Q3.2

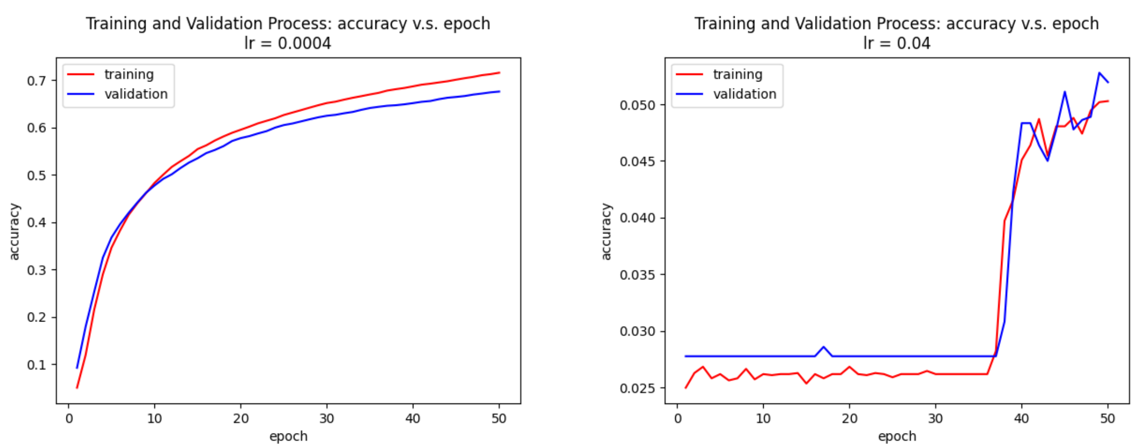

Figure 7: Training and Validation Process for the Two New Learning Rate

Learning rate is the **step size** of every updates, changing the learning rate will affect the convergence and the learning speed. When learning rate is 10 times larger (4e-2), the accuracy decreases and the training process is not smooth (oscillation occurs) since the step size is too big. When learning rate is one tenth of the default setting (4e-4), the accuracy decreases since the step size is too small, so it requires more than 50 epochs to converge to the optimum.

Q3.3

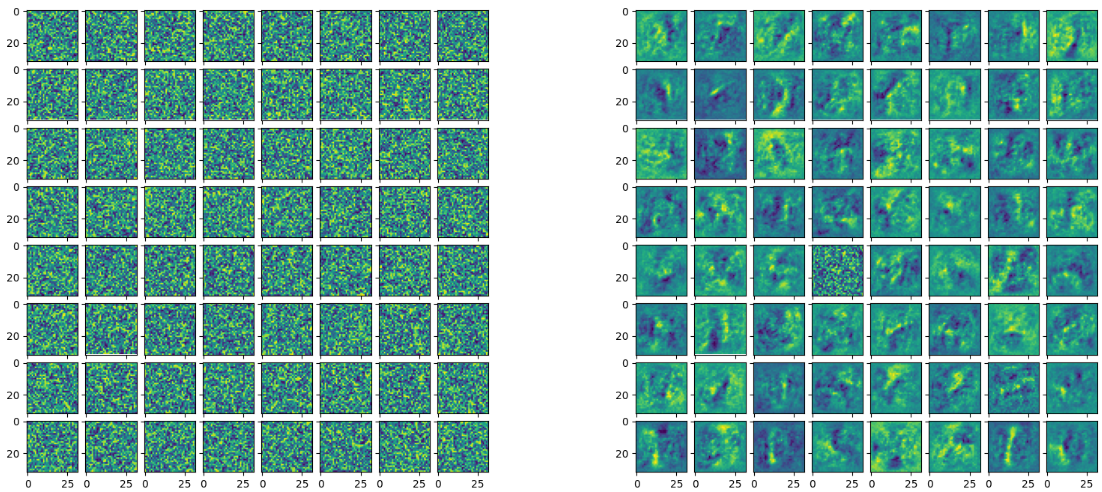

Figure 8: Visualization of the First Layer Weights (left: right after Xavier Initialization, right: after Training Loop)

As shown in Figure 8, the first layer weights right after the Xavier Initialization are just random patterns (initialize randomly). After the whole training process (50 epochs), the first layer weights show some specific patterns.

Q3.4

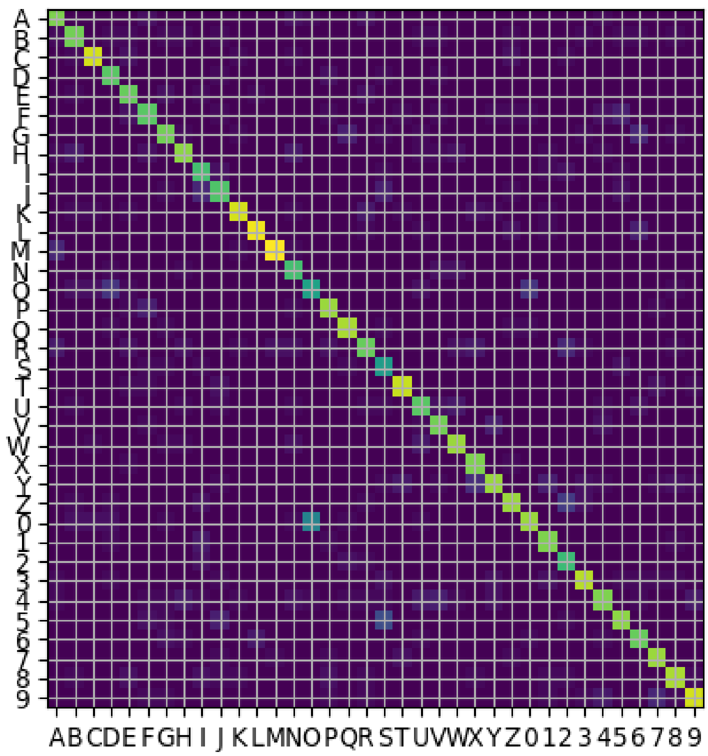

Figure 9: The Confusion Matrix on the Test Set

From Figure 9, we can see that 0 (zero) and O (alphabet O), 5 (five) and S (alphabet S) are mostly commonly confused since they are brighter than the other grids. The reason is that they look similar, so the model might misclassify the characters.

Q4.1

The two big assumptions that the sample method makes are:
(1) Every characters are isolated and don't have connection with the neighbor characters since the sample method is based on the extraction of the connected area.
(2) Every characters need to be fully-connected since the sample method extracts connected pixels.

As shown in Figure 10, if the characters are not fully-connected (ex. TO DO LIST) or some characters are connected together (ex. HAIKUS ARE EASY), the characters detection might fail.

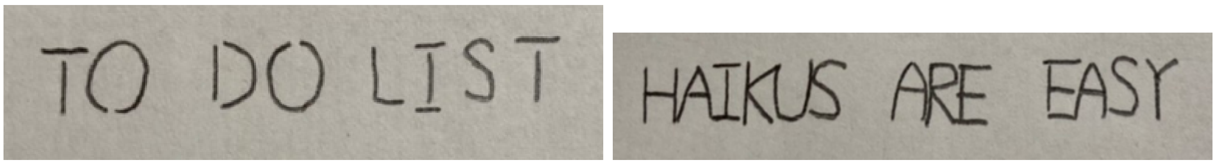

Figure 10: Example Images of Possible Characters Detection Failure

Q4.2

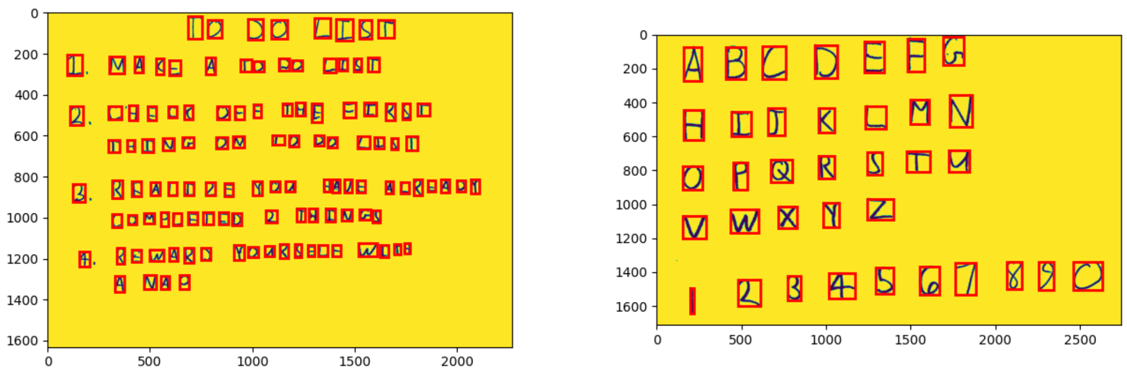

Figure 11: Visualization of the Bounding Boxes for 01_list and 02_letters

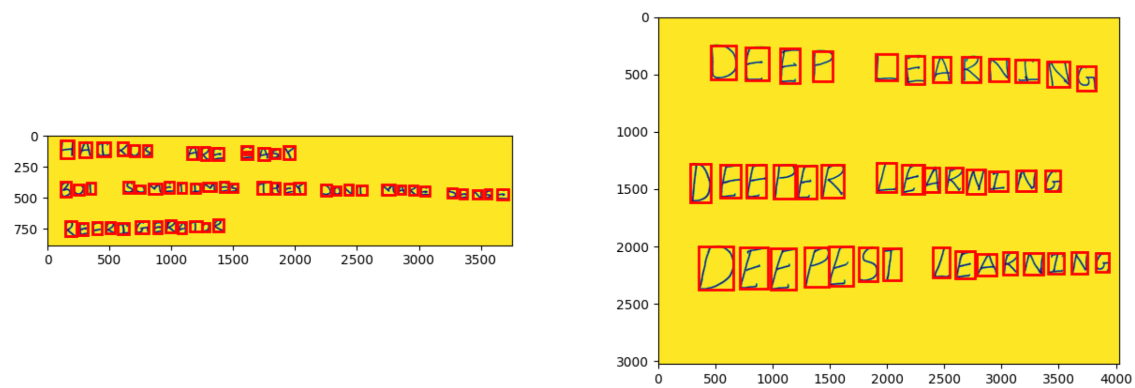

Figure 12: Visualization of the Bounding Boxes for 03_haiku and 04_deep

Q4.3

Figure 13: Screenshot of the Extracted Text for 01_list and 02_letters

Figure 14: Screenshot of the Extracted Text for 03_haiku and 04_deep

Q5.2

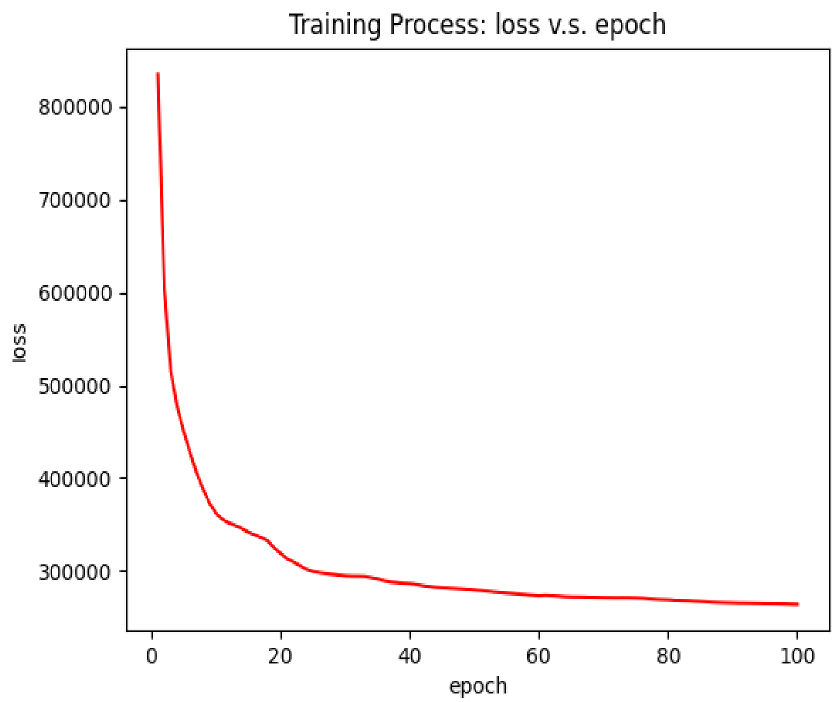

Figure 15: Training Process (Loss vs Epoch)

Using the provided default setting, the training loss curve converges to the optimum. The final total squared loss is around 260000.

Q5.3.1

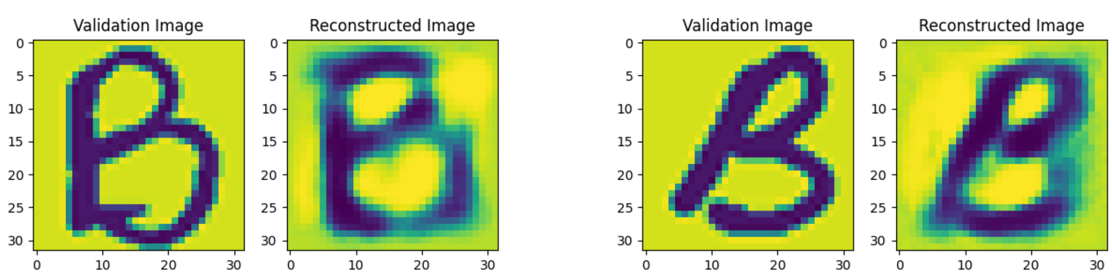

Figure 16: 2 Validation and Reconstructed Images of Class B

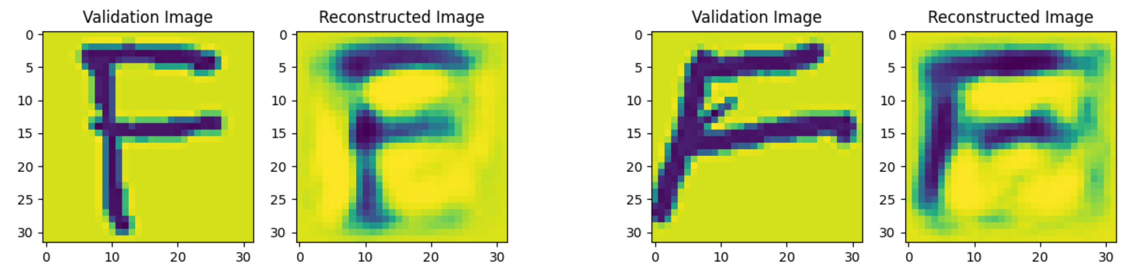

Figure 17: 2 Validation and Reconstructed Images of Class F

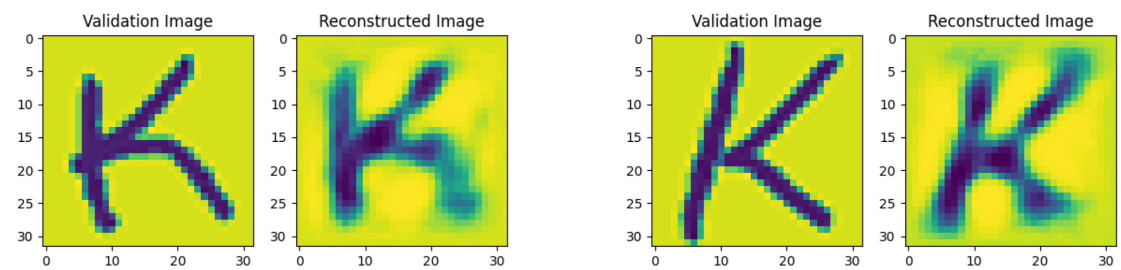

Figure 18: 2 Validation and Reconstructed Images of Class K

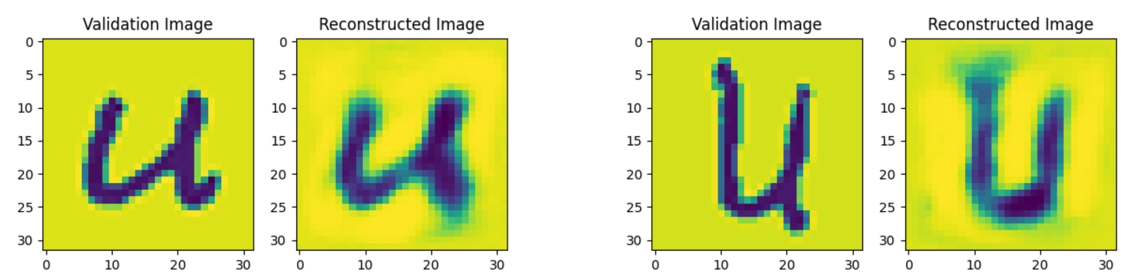

Figure 19: 2 Validation and Reconstructed Images of Class u

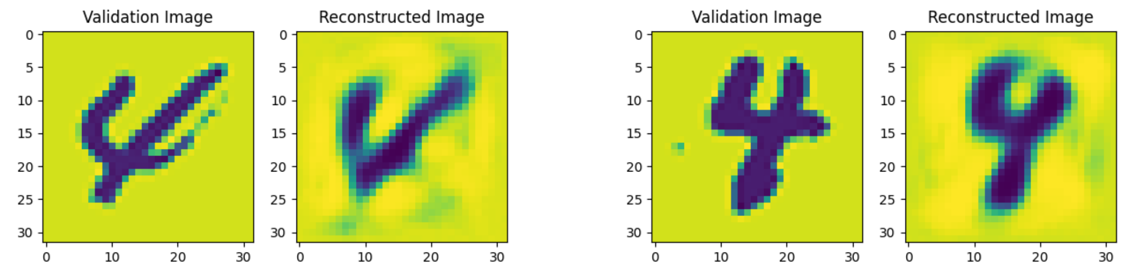

Figure 20: 2 Validation and Reconstructed Images of Class 4

Compared to the original validation images, the reconstructed images are blurred since it is impossible for the autoencoder to fully reconstruct the input images.

Q5.3.2

The average PSNR across all validation images = 16.397221387836495

Q6.1.1

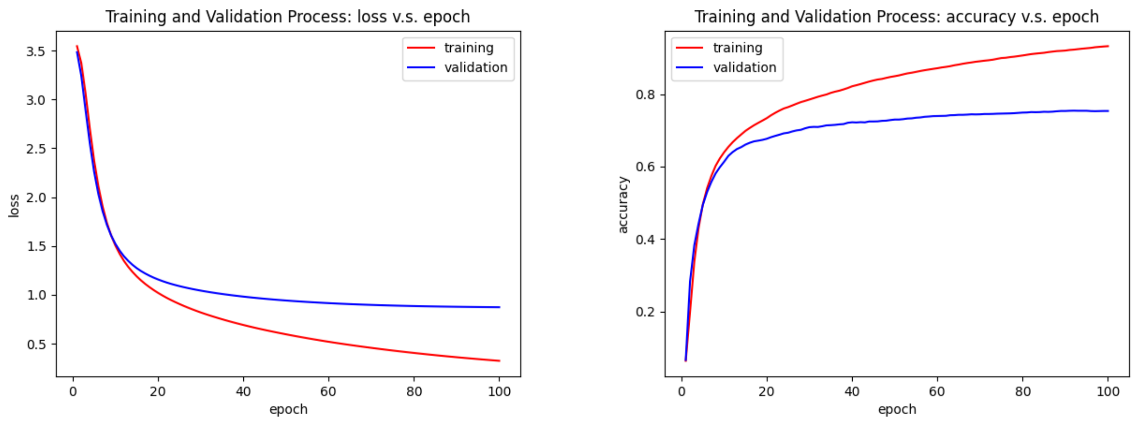

Figure 21: Training and Validation Process of the FC Network on the NIST36 dataset (Loss and Accuracy)

According to Figure 21, the loss of the training and validation dataset are 0.32 and 0.87, and the accuracy of the training and validation dataset are 93% and 75%. The accuracy of the test dataset is 76%.

Q6.1.2

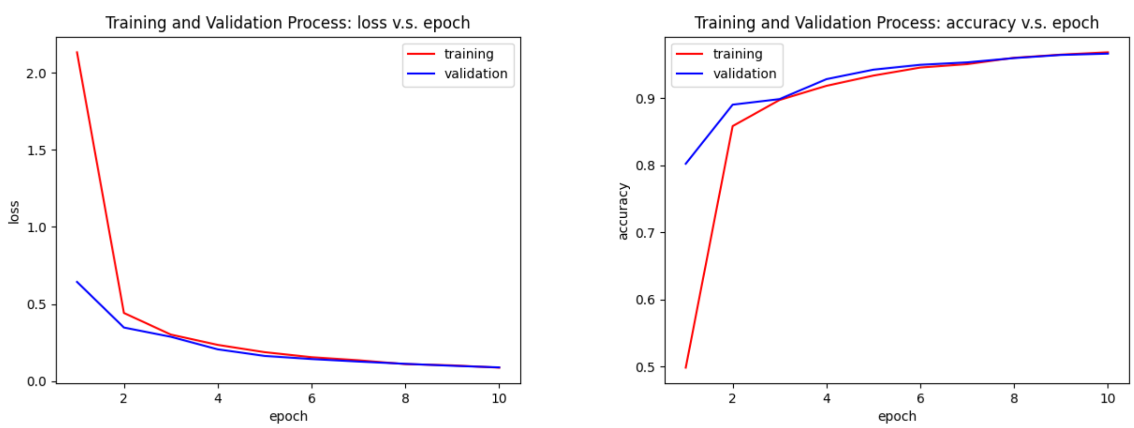

Figure 22: Training and Validation Process of the CNN on the NIST36 dataset (Loss and Accuracy)

According to Figure 22, the loss of the training and validation dataset are both 0.09, and the accuracy of the training and validation dataset are both 97%. The accuracy of the test dataset is 88%. Compared the performance of the Fully-connected Network and the Convolutional Neural Network, we know that the Convolutional Neural Network has higher accuracy and lower computational cost.

Q6.1.3

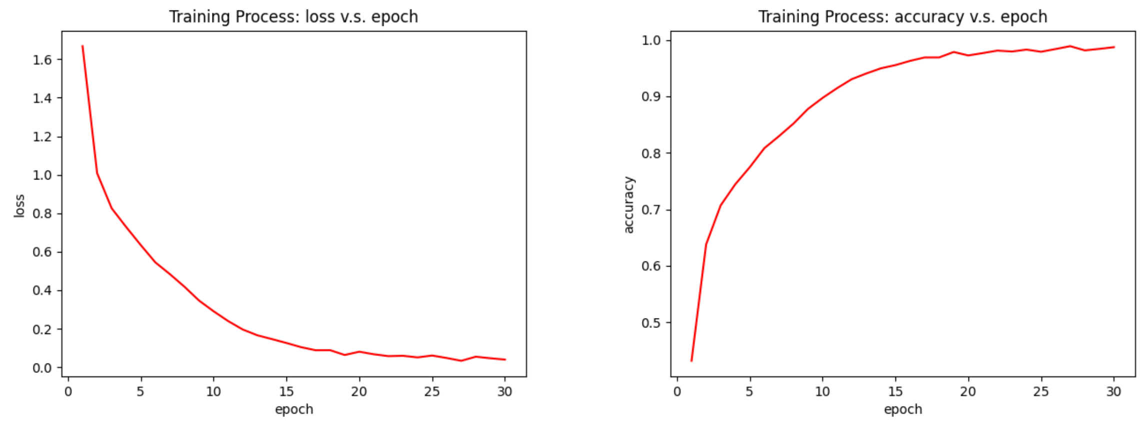

Figure 23: Training Process of the CNN on the CIFAR-10 dataset (Loss and Accuracy)

According to Figure 23, the loss of the training dataset is 0.04, and the accuracy of the training dataset is 99%. The accuracy of the test dataset is 77%.

Q6.1.4

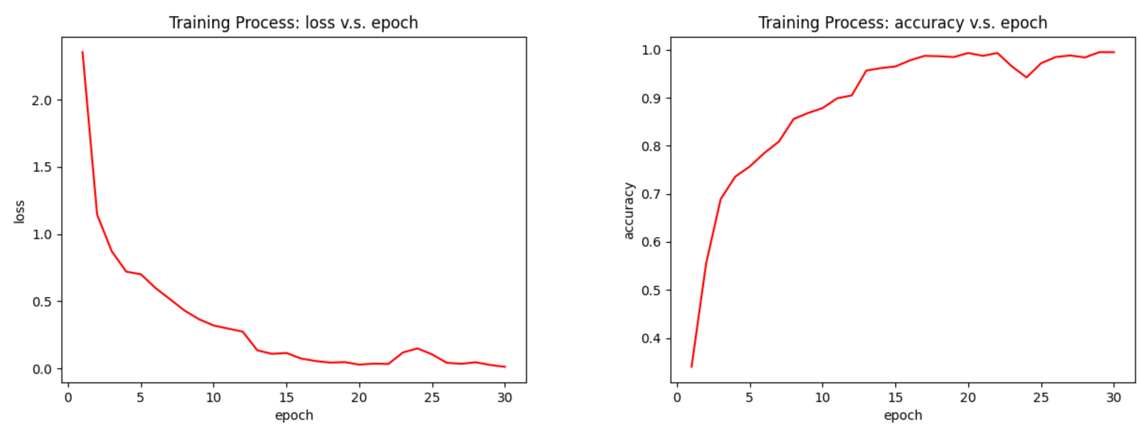

Figure 24: Training Process of the CNN on the SUN dataset (Loss and Accuracy)

According to Figure 24, the loss of the training dataset is 0.01, and the accuracy of the training dataset is 99%. The accuracy of the test dataset is 74%. Compared the result with the one from Assignment 01 (The accuracy of the BoW method on test dataset is 64.75%), we know that the Convolutional Neural Network has better performance (higher accuracy).

Q6.2

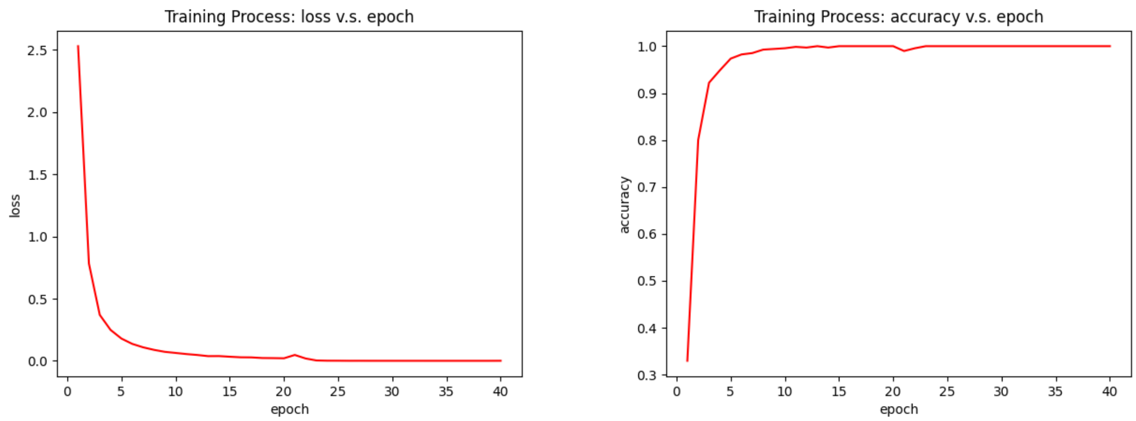

Figure 25: Training Process of the Fine-tune Single Layer Classifier on Oxford-Flower17 dataset (Loss and Accuracy)

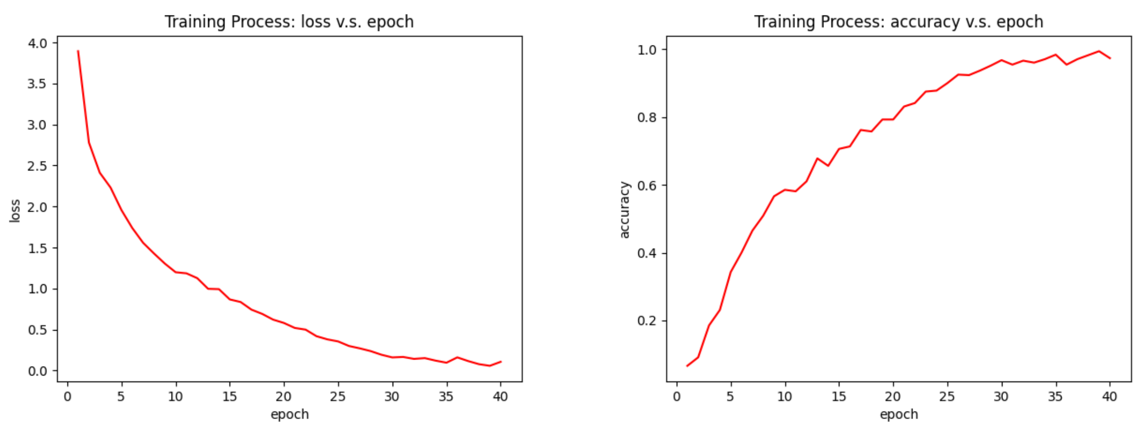

Figure 26: Training Process of the Self-defined CNN on Oxford-Flower17 dataset (Loss and Accuracy)

According to Figure 25 and 26, the loss of the training dataset for the Fine-tune Single Layer Classifier and the Self-defined CNN trained from scratch are 0 and 0.11, and the accuracy are 100% and 97%. The accuracy of the test dataset for the Fine-tune Single Layer Classifier and the Self-defined CNN trained from scratch are 91% and 49%. Compared the performance of the Fine-tune model and the Self-designed model, we know that the Fine-tune model converges faster and has higher accuracy.

Code Appendix

0.1 nn.py

```
import numpy as np
from util import *
# do not include any more libraries here!
# do not put any code outside of functions!

############################## Q 2.1 ##############################
# initialize b to 0 vector
# b should be a 1D array, not a 2D array with a singleton dimension
# we will do XW + b.
# X be [Examples, Dimensions]
def initialize_weights(in_size, out_size, params,name=''):
    W, b = None, None

    ##########################
    ##### your code here #####
    ##########################
    value = np.sqrt(6 / (in_size + out_size))
    W = np.random.uniform(-value, value, size = (in_size, out_size))
    b = np.zeros(out_size)

    params['W' + name] = W
    params['b' + name] = b

############################## Q 2.2.1 ##############################
# x is a matrix
# a sigmoid activation function
def sigmoid(x):
    res = None

    ##########################
    ##### your code here #####
    ##########################
    res = 1 / (1 + np.exp(-x))

    return res

############################## Q 2.2.1 ##############################
def forward(X, params, name='', activation=sigmoid):
    """
    Do a forward pass

    Keyword arguments:
    X -- input vector [Examples x D]
    params -- a dictionary containing parameters
```

```
    name -- name of the layer
    activation -- the activation function (default is sigmoid)
    """
    pre_act, post_act = None, None
    # get the layer parameters
    W = params['W' + name]
    b = params['b' + name]


    ##########################
    ##### your code here #####
    ##########################
    pre_act = X @ W + b
    post_act = activation(pre_act)

    # store the pre-activation and post-activation values
    # these will be important in backprop
    params['cache_' + name] = (X, pre_act, post_act)

    return post_act

############################## Q 2.2.2  ##############################
# x is [examples,classes]
# softmax should be done for each row
def softmax(x):
    res = None

    ##########################
    ##### your code here #####
    ##########################
    # Add translation c for numerical stability
    c = -np.max(x, axis = 1).reshape(-1, 1)
    x += c
    # Compute softmax
    numerator = np.exp(x)
    denominator = np.sum(numerator, axis = 1).reshape(-1, 1)
    res = numerator / denominator

    return res

############################## Q 2.2.3 ##############################
# compute total loss and accuracy
# y is size [examples,classes]
# probs is size [examples,classes]
def compute_loss_and_acc(y, probs):
    loss, acc = None, None

```

```
    ##########################
    ##### your code here #####
    ##########################
    # Compute the cross-entropy loss
    loss = -np.sum(y * np.log(probs))
    # Compute the accuracy
    examples = y.shape[0]
    count = 0
    for i in range(examples):
        pred_label = np.argmax(probs[i, :])
        true_label = np.argmax(y[i, :])
        if pred_label == true_label:
            count += 1
    acc = count / examples

    return loss, acc

############################## Q 2.3 ##############################
# we give this to you
# because you proved it
# it's a function of post_act
def sigmoid_deriv(post_act):
    res = post_act*(1.0-post_act)
    return res

def backwards(delta, params, name='', activation_deriv=sigmoid_deriv):
    """
    Do a backwards pass

    Keyword arguments:
    delta -- errors to backprop
    params -- a dictionary containing parameters
    name -- name of the layer
    activation_deriv -- the derivative of the activation_func
    """
    # everything you may need for this layer
    W = params['W' + name]
    b = params['b' + name]
    X, pre_act, post_act = params['cache_' + name]

    grad_X, grad_W, grad_b = None, None, None
    # do the derivative through activation first
    # then compute the derivative W, b, and X
    ##########################
    ##### your code here #####
    ##########################
    # Derivative through activation
    d_activation = activation_deriv(post_act)
```

```
    dJ_dy = delta * d_activation
    # Compute the derivative W, b, and X
    grad_X = dJ_dy @ (W.T)
    grad_W = (X.T) @ dJ_dy
    grad_b = (dJ_dy.T @ np.ones((dJ_dy.shape[0], 1))).reshape(-1)

    # store the gradients
    params['grad_W' + name] = grad_W
    params['grad_b' + name] = grad_b

    return grad_X

############################## Q 2.4 ##############################
# split x and y into random batches
# return a list of [(batch1_x,batch1_y)...]
def get_random_batches(x, y, batch_size):
    batches = []

    ##########################
    ##### your code here #####
    ##########################
    # Get the basic information for splitting
    num_examples = x.shape[0]
    num_batches = int(num_examples / batch_size)
    # Split x and y into random batches
    for i in range(num_batches):
        random_index = np.random.choice(num_examples, size = batch_size,
                                                               replace = False)
        batch_x = x[random_index, :]
        batch_y = y[random_index, :]
        batches.append((batch_x, batch_y))

    return batches
```

0.2 run_q2.py

```
import numpy as np
# you should write your functions in nn.py
from nn import *
from util import *


# fake data
# feel free to plot it in 2D
# what do you think these 4 classes are?
g0 = np.random.multivariate_normal([3.6,40],[[0.05,0],[0,10]],10)
g1 = np.random.multivariate_normal([3.9,10],[[0.01,0],[0,5]],10)
g2 = np.random.multivariate_normal([3.4,30],[[0.25,0],[0,5]],10)
g3 = np.random.multivariate_normal([2.0,10],[[0.5,0],[0,10]],10)
x = np.vstack([g0,g1,g2,g3])
# we will do XW + B
# that implies that the data is N x D

# create labels
y_idx = np.array([0 for _ in range(10)] + [1 for _ in range(10)] +
                              [2 for _ in range(10)] + [3 for _ in range(10)])
# turn to one_hot
y = np.zeros((y_idx.shape[0],y_idx.max()+1))
y[np.arange(y_idx.shape[0]),y_idx] = 1

# parameters in a dictionary
params = {}

# Q 2.1
# initialize a layer
initialize_weights(2,25,params,'layer1')
initialize_weights(25,4,params,'output')
assert(params['Wlayer1'].shape == (2,25))
assert(params['blayer1'].shape == (25,))

#expect 0, [0.05 to 0.12]
print("{}, {:.2f}".format(params['blayer1'].sum(),params['Wlayer1'].std()**2))
print("{}, {:.2f}".format(params['boutput'].sum(),params['Woutput'].std()**2))

# Q 2.2.1
# implement sigmoid
test = sigmoid(np.array([-100,100]))
print('should be zero and one\t',test.min(),test.max())
# implement forward
h1 = forward(x,params,'layer1')
print(h1.shape)
# Q 2.2.2
```

```python
# implement softmax
probs = forward(h1,params,'output',softmax)
# make sure you understand these values!
# positive, ~1, ~1, (40,4)
print(probs.min(),min(probs.sum(1)),max(probs.sum(1)),probs.shape)

# Q 2.2.3
# implement compute_loss_and_acc
loss, acc = compute_loss_and_acc(y, probs)
# should be around -np.log(0.25)*40 [~55] and 0.25
# if it is not, check softmax!
print("{}, {:.2f}".format(loss,acc))

# here we cheat for you
# the derivative of cross-entropy(softmax(x)) is probs - 1[correct actions]
delta1 = probs
delta1[np.arange(probs.shape[0]),y_idx] -= 1

# we already did derivative through softmax
# so we pass in a linear_deriv, which is just a vector of ones
# to make this a no-op
delta2 = backwards(delta1,params,'output',linear_deriv)
# Implement backwards!
backwards(delta2,params,'layer1',sigmoid_deriv)

# W and b should match their gradients sizes
for k,v in sorted(list(params.items())):
    if 'grad' in k:
        name = k.split('_')[1]
        print(name,v.shape, params[name].shape)

# Q 2.4
batches = get_random_batches(x,y,5)
# print batch sizes
print([_[0].shape[0] for _ in batches])
batch_num = len(batches)

print()
print("##########################################################")
print("######                  Training Loop         ##########")
print("##########################################################")
print()

max_iters = 501
learning_rate = 1e-3
# with default settings, you should get loss < 35 and accuracy > 75%
# Here we also cheat for you and write the forward + backward loop for you
for itr in range(max_iters):
```

```
    total_loss = 0
    avg_acc = 0
    for xb,yb in batches:
        # forward
        yb_idx = np.argmax(yb,1)
        h1 = forward(xb,params,'layer1')
        probs = forward(h1,params,'output',softmax)

        # loss
        # be sure to add loss and accuracy to epoch totals
        loss, acc = compute_loss_and_acc(yb, probs)
        total_loss += loss
        avg_acc += acc/batch_num

        # backward
        delta1 = probs
        delta1[np.arange(probs.shape[0]),yb_idx] -= 1

        delta2 = backwards(delta1,params,'output',linear_deriv)
        backwards(delta2,params,'layer1',sigmoid_deriv)

        # apply gradient
        names = ['layer1','output']
        for k in names:
            params['W'+k] -= learning_rate * params['grad_W' + k]
            params['b'+k] -= learning_rate * params['grad_b' + k]



    if itr % 100 == 0:
        print("itr: {:02d} \t loss: {:.2f} \t acc :
                                        {:.2f}".format(itr,total_loss,avg_acc))


print()
print("##################################################")
print("######              Training Loop End    ##########")
print("##################################################")
print()

print()
print("##################################################")
print("#####       Numerical Gradient Checker    ##########")
print("##################################################")
print()

h1 = forward(x,params,'layer1')
probs = forward(h1,params,'output',softmax)
```

```
delta1 = probs
delta1[np.arange(probs.shape[0]),y_idx] -= 1

delta2 = backwards(delta1,params,'output',linear_deriv)
backwards(delta2,params,'layer1',sigmoid_deriv)

# save the old params
import copy
params_orig = copy.deepcopy(params)

eps = 1e-6
for k,v in params.items():
    if '_' in k:
        continue
    # we have a real parameter!
    # for each value inside the parameter
    #   add epsilon
    #   run the network
    #   get the loss
    #   compute derivative with central diffs

    def f(p):
        h1 = forward(x,p,'layer1')
        probs = forward(h1,p,'output',softmax)
        new_loss, _ = compute_loss_and_acc(y, probs)
        return new_loss
    grad_v = np.zeros_like(v)
    it = np.nditer(params[k], flags=['multi_index'], op_flags=['readwrite'])
    while not it.finished:

        # evaluate function at x+h
        ix = it.multi_index
        og = v[ix]
        v[ix] += eps # increment by h
        fxh = f(params) # evalute f(x + h)
        v[ix] -= 2*eps # restore to previous value (very important!)
        fx = f(params) # evalute f(x + h)

        # compute the partial derivative
        grad_v[ix] = (fxh - fx) / (2*eps) # the slope
        v[ix] = og
        it.iternext() # step to next dimension
    params['grad_' + k] = grad_v

total_error = 0
for k in params.keys():
    if 'grad_' in k:
        # relative error
```

```
        err = np.abs(params[k] - params_orig[k])/np.maximum(np.abs(params[k]),
                                                            np.abs(params_orig[k]))
        err = err.sum()
        print('{} {:.2e}'.format(k, err))
        total_error += err
# should be less than 1e-4
print('total {:.2e}'.format(total_error))
print()
print("#####################################################")
print("#####      Numerical Gradient Checker END ##########")
print("#####################################################")
print()
```

0.3 run_q3.py

```
import numpy as np
import scipy.io
from nn import *
import pickle
import matplotlib.pyplot as plt

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 = train_data['train_data'], train_data['train_labels']
valid_x, valid_y = valid_data['valid_data'], valid_data['valid_labels']
test_x, test_y = test_data['test_data'], test_data['test_labels']

max_iters = 50  # For q4, max_iters = 100
batch_size = 72
learning_rate = 4e-3   # For q4, lr = 5e-3
hidden_size = 64

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

params = {}

# We have initialized the single layer network for you here
# Do not change the layer names
initialize_weights(train_x.shape[1],hidden_size,params,'layer1')
initialize_weights(hidden_size,train_y.shape[1],params,'output')

# Store the intialized weights for Q3.3
initial_w = np.copy(params['Wlayer1'])

list_of_train_acc_per_iter = []
list_of_val_acc_per_iter = []
for itr in range(max_iters):
    total_train_loss = 0
    total_train_acc = 0
    for xb,yb in batches:
        ##########################
        ##### your code here #####
        ##########################
        # Forward propagation
        h1 = forward(xb, params, 'layer1')
        probs = forward(h1, params, 'output', softmax)

        # Compute and Update loss and accuracy
```

```
        loss, acc = compute_loss_and_acc(yb, probs)
        total_train_loss += loss
        total_train_acc += acc

        # Back propagation
        # Derivative of cross-entropy(softmax(x))
        delta1 = probs - yb
        delta2 = backwards(delta1,params,'output',linear_deriv)
        # Implement back propagation
        backwards(delta2,params,'layer1',sigmoid_deriv)

        # Apply gradient descent
        # Ouput layer
        params['W' + 'output'] -= learning_rate * params['grad_W' + 'output']
        params['b' + 'output'] -= learning_rate * params['grad_b' + 'output']
        # Hidden layer
        params['W' + 'layer1'] -= learning_rate * params['grad_W' + 'layer1']
        params['b' + 'layer1'] -= learning_rate * params['grad_b' + 'layer1']

    # Average training loss and accuracy
    total_train_loss = total_train_loss / batch_num
    total_train_acc = total_train_acc / batch_num

    val_acc = None
    # compute the validation accuracy here, make sure there is little
    # overfitting issue
    ##########################
    ##### your code here #####
    ##########################
    # Forward propagation
    h1 = forward(valid_x, params, 'layer1')
    probs = forward(h1, params, 'output', softmax)

    # Compute loss and accuracy
    val_loss, val_acc = compute_loss_and_acc(valid_y, probs)

    # Update training and validation accuracy
    list_of_train_acc_per_iter.append(total_train_acc)
    list_of_val_acc_per_iter.append(val_acc)

    print("itr: {:02d} \t train loss: {:.2f} \t train acc : {:.2f} \t
           validation acc : {:.2f}".format(itr+1,total_train_loss,
           total_train_acc, val_acc))

# In a single plot, plot the training v.s. validation accuracy per iteration
##########################
##### your code here #####
##########################
```

```
95  # Plot the training and validation process
96  plt.figure()
97  plt.plot(np.arange(1, max_iters+1), list_of_train_acc_per_iter, color = 'r',
98                                                             label = 'training')
99  plt.plot(np.arange(1, max_iters+1), list_of_val_acc_per_iter, color = 'b',
100                                                          label = 'validation')
101 plt.title(f"Training and Validation Process: accuracy v.s. epoch\nlr =
102                                                               {learning_rate}")
103 plt.xlabel('epoch')
104 plt.ylabel('accuracy')
105 plt.legend()
106 plt.show()
107
108 # run on test set and report accuracy! should be around 75%
109 test_acc = None
110 h1 = forward(test_x,params,'layer1')
111 probs = forward(h1,params,'output',softmax)
112 loss, test_acc = compute_loss_and_acc(test_y, probs)
113
114 print('Test accuracy: ',test_acc)
115 saved_params = {k:v for k,v in params.items() if '_' not in k}
116 with open('q3_weights.pickle', 'wb') as handle:
117     pickle.dump(saved_params, handle, protocol=pickle.HIGHEST_PROTOCOL)
118
119 # Q3.3
120 import matplotlib.pyplot as plt
121 from mpl_toolkits.axes_grid1 import ImageGrid
122
123 # The weights after training loop are visualized here.
124 # You may use the same visualization script to visualize the layer right after
125 # Xavier initialization
126 fig = plt.figure(1, (8., 8.))
127 if hidden_size < 128:
128     grid = ImageGrid(fig, 111,  # similar to subplot(111)
129                     nrows_ncols=(8, 8),  # creates 2x2 grid of axes
130                     axes_pad=0.1,  # pad between axes in inch.
131                     )
132     img_w = params['Wlayer1'].reshape((32,32,hidden_size))
133     for i in range(hidden_size):
134         grid[i].imshow(img_w[:,:,i])  # The AxesGrid object work as a list of
135                                       # axes.
136     plt.show()
137
138 # Visualize the layer right after Xavier initialization
139 # Line 30-31 stores the Xavier Initialization
140 fig = plt.figure(1, (8., 8.))
141 if hidden_size < 128:
142     grid = ImageGrid(fig, 111,  # similar to subplot(111)
```

```python
                    nrows_ncols=(8, 8),  # creates 2x2 grid of axes
                    axes_pad=0.1,  # pad between axes in inch.
                    )
    initial_w = initial_w.reshape((32,32,hidden_size))
    for i in range(hidden_size):
        grid[i].imshow(initial_w[:,:,i])  # The AxesGrid object work as a list
                                          # of axes.
    plt.show()


# Q3.4
confusion_matrix = np.zeros((train_y.shape[1],train_y.shape[1]))

# compute confusion matrix here
for i in range(test_y.shape[0]):
    i1 = np.argmax(probs[i])
    i2 = np.argmax(test_y[i])
    confusion_matrix[i1,i2] += 1

import string
plt.imshow(confusion_matrix,interpolation='nearest')
plt.grid(True)
plt.xticks(np.arange(36),string.ascii_uppercase[:26] + ''.join([str(_) for _
                                                               in range(10)]))
plt.yticks(np.arange(36),string.ascii_uppercase[:26] + ''.join([str(_) for _
                                                               in range(10)]))
plt.show()
```

0.4 run_q4.py

```python
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches

import skimage
import skimage.measure
import skimage.color
import skimage.restoration
import skimage.io
import skimage.filters
import skimage.morphology
import skimage.segmentation

from nn import *
from q4 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)
    for bbox in bboxes:
        minr, minc, maxr, maxc = bbox
        rect = matplotlib.patches.Rectangle((minc, minr), maxc - minc,
                               maxr - minr, fill=False, edgecolor='red', linewidth=2)
        plt.gca().add_patch(rect)
    plt.show()

    # find the rows using..RANSAC, counting, clustering, etc.
    ##########################
    ##### your code here #####
    ##########################
    # Split the rows based on min_row and max_row
    # Sort the bboxes based on max_row
    bboxes.sort(key = lambda x : x[2])

    # Initialize rows and single_row
    all_rows, single_row = [], []

    # Split the characters into rows
```

```
    boundary = bboxes[0][2]
    for bbox in bboxes:
        # Extract min_row and max_row
        min_row, max_row = bbox[0], bbox[2]
        # Check for new rows
        if min_row > boundary:
            # Sort the previos row based on min_column
            single_row.sort(key = lambda x : x[1])
            # Update rows
            all_rows.append(single_row)
            # Update boundary and Reset single_row
            boundary = max_row
            single_row = []
        # Update single_row
        single_row.append(bbox)
    # Sort and Update the final single_row into rows
    single_row.sort(key = lambda x : x[1])
    all_rows.append(single_row)

    # crop the bounding boxes
    # note.. before you flatten, transpose the image (that's how the dataset is!)
    # consider doing a square crop, and even using np.pad() to get your
    # images looking more like the dataset
    ##########################
    ##### your code here #####
    ##########################
    # Crop the bounding box
    crop_bboxes, row_crop_bboxes = [], []
    for r in range(len(all_rows)):
        for i, bbox in enumerate(all_rows[r]):
            # Crop the bbox
            min_row, min_col, max_row, max_col = bbox
            crop_bbox = bw[min_row:max_row, min_col:max_col]
            # Padding
            crop_bbox = np.pad(crop_bbox, ((30, 30), (30, 30)), 'constant',
                                                           constant_values = (1, 1))
            # Resize and Preprocessing (erosion)
            crop_bbox = skimage.transform.resize(crop_bbox, (32, 32))
            crop_bbox = skimage.morphology.erosion(crop_bbox,
                                    np.array([[0, 1, 0], [1, 1, 1], [0, 1, 0]]))
            # Transpose, Flatten, and Update
            crop_bbox = crop_bbox.T
            crop_bbox = crop_bbox.reshape(1, -1)
            row_crop_bboxes.append(crop_bbox)
        # Update crop_bboxes and Reset row_crop_bboxes
        crop_bboxes.append(row_crop_bboxes)
        row_crop_bboxes = []

```

```
    # load the weights
    # run the crops through your neural network and print them out
    import pickle
    import string
    letters = np.array([_ for _ in string.ascii_uppercase[:26]]
                                                              + [str(_) for _ in range(10)])
    params = pickle.load(open('q3_weights.pickle','rb'))
    ##########################
    ##### your code here #####
    ##########################
    # Print the image name
    print("\n" + img.split('.')[0] + "\n")

    # Classify the letter in crop_bboxes
    for r in range(len(crop_bboxes)):
        for x in crop_bboxes[r]:
            # Forward propagation
            h1 = forward(x, params, 'layer1')
            probs = forward(h1, params, 'output', softmax)
            # Evaluate the one-hot vector of the letter
            letter_index = np.argmax(probs, axis = 1)
            # Transfer index into letters
            letter = letters[letter_index][0]
            # Print out the letters
            print(f"{letter} ", end = '')
        print("\n")
```

0.5 run_q5.py

```
import numpy as np
import scipy.io
from nn 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, learning rate
batch_size = 36
learning_rate =  3e-5
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()

# Q5.1.1 & Q5.1.2
# initialize layers here
##########################
##### your code here #####
##########################
# Initialize input, hidden, and output layer
layer_names = ['input', 'hlayer1', 'hlayer2', 'output']
initialize_weights(train_x.shape[1], hidden_size, params, layer_names[0])
initialize_weights(hidden_size, hidden_size, params, layer_names[1])
initialize_weights(hidden_size, hidden_size, params, layer_names[2])
initialize_weights(hidden_size, train_x.shape[1], params, layer_names[3])

# Initialize momentum accumulators with zeros
layer_names = ['input', 'hlayer1', 'hlayer2', 'output']
for layer in layer_names:
    params['m_W' + layer] = np.zeros_like(params['W' + layer])
    params['m_b' + layer] = np.zeros_like(params['b' + layer])

# Initialize training loss list
list_of_train_loss_per_iter = []

# 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 missing
        #   so you should be able to write your loop without any special
        #   conditions

        ##########################
        ##### your code here #####
        ##########################
        # Forward propagation
        h1 = forward(xb, params, layer_names[0], relu)
        h2 = forward(h1, params, layer_names[1], relu)
        h3 = forward(h2, params, layer_names[2], relu)
        probs = forward(h3, params, layer_names[3], sigmoid)

        # Compute and Update loss
        loss = np.sum((probs - xb) ** 2)
        total_loss += loss

        # Back propagation
        # delta1 = derivative of (x-y)^2
        delta1 = 2 * (probs - xb)
        delta2 = backwards(delta1, params, layer_names[3], sigmoid_deriv)
        delta3 = backwards(delta2, params, layer_names[2], relu_deriv)
        delta4 = backwards(delta3, params, layer_names[1], relu_deriv)
        backwards(delta4, params, layer_names[0], relu_deriv)

        # Apply gradient descent with momentum
        for layer in layer_names:
            # Update weights
            params['m_W' + layer] = 0.9 * params['m_W' + layer]
                                    - learning_rate * params['grad_W' + layer]
            params['W' +  layer] += params['m_W' +  layer]
            # Update biases
            params['m_b' + layer] = 0.9 * params['m_b' + layer]
                                    - learning_rate * params['grad_b' + layer]
            params['b' +  layer] += params['m_b' +  layer]

    # Update training loss
    list_of_train_loss_per_iter.append(total_loss)

    if itr % 2 == 0:
```

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

# Q5.2
import matplotlib.pyplot as plt
# Plot the training process
plt.figure()
plt.plot(np.arange(1, max_iters+1), list_of_train_loss_per_iter, color = 'r')
plt.title(f"Training Process: loss v.s. epoch")
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()

# Q5.3.1
import matplotlib.pyplot as plt
# visualize some results
##########################
##### your code here #####
##########################
# Forward propagation for the validation data
h1 = forward(valid_x, params, layer_names[0], relu)
h2 = forward(h1, params, layer_names[1], relu)
h3 = forward(h2, params, layer_names[2], relu)
valid_probs = forward(h3, params, layer_names[3], sigmoid)

# Extract 5 classes from the total 36 classes
class_index = [1, 5, 10, 20, 30]
valid_y = valid_data['valid_labels']
extracted_classes, single_class = [], []
for index in class_index:
    for i in range(len(valid_y)):
        if np.argmax(valid_y, axis = 1)[i] == index:
            single_class.append(i)
    extracted_classes.append(single_class)
    single_class = []

# Visualize 2 validation and reconstructed images for each class
for i in range(len(extracted_classes)):
    for j in range(2):
        index = extracted_classes[i][j]
        # Plot the validation and reconstructed images
        fig, [ax1, ax2] = plt.subplots(1, 2)
        ax1.imshow(valid_x[index].reshape(32,32).T)
        ax1.set_title("Validation Image")
        ax2.imshow(valid_probs[index].reshape(32,32).T)
        ax2.set_title("Reconstructed Image")
        plt.show()
```

```python



# Q5.3.2
# skimage version == 0.18.1
from skimage.metrics import peak_signal_noise_ratio as psnr
# evaluate PSNR
##########################
##### your code here #####
##########################
# Forward propagation for the validation data
h1 = forward(valid_x, params, layer_names[0], relu)
h2 = forward(h1, params, layer_names[1], relu)
h3 = forward(h2, params, layer_names[2], relu)
valid_probs = forward(h3, params, layer_names[3], sigmoid)

# Compute the average psnr for all validation images
total_psnr = 0
for i in range(len(valid_x)):
    total_psnr += psnr(valid_x[i], valid_probs[i])
average_psnr = total_psnr / len(valid_x)
print(f"Average PSNR = {average_psnr}")
```

0.6 run_q6_1.py

```python
# Import package
import numpy as np
import matplotlib.pyplot as plt
import scipy.io
import torch
import torch.nn as nn
import torch.nn.functional as func
import torchvision
import torchvision.transforms as transforms
import torch.optim as optim
from nn import get_random_batches

# Load the NIST36 dataset
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 = train_data['train_data'], train_data['train_labels']
valid_x, valid_y = valid_data['valid_data'], valid_data['valid_labels']
test_x, test_y = test_data['test_data'], test_data['test_labels']

# Turn the dataset (numpy) into tensor
train_x, train_y = torch.from_numpy(train_x).float(),  torch.from_numpy(train_y)
valid_x, valid_y = torch.from_numpy(valid_x).float(),  torch.from_numpy(valid_y)
test_x, test_y = torch.from_numpy(test_x).float(),  torch.from_numpy(test_y)

# Q6.1.1 Fully-connected Neural Network
# Set up the hyperparameters
max_iters = 100
batch_size = 72
learning_rate = 0.1
hidden_size = 64
batches = get_random_batches(train_x, train_y, batch_size)
batch_num = len(batches)

# Build the fully-connected network
class FullyConnectedNetwork(nn.Module):
    def __init__(self):
        super(FullyConnectedNetwork, self).__init__()
        self.fc_layer_set = nn.Sequential(
            nn.Linear(train_x.shape[1], hidden_size),
            nn.Sigmoid(),
            nn.Linear(hidden_size, train_y.shape[1]),
        )

    def forward(self, x):
        # Fully-connected layer
```

```
        x = self.fc_layer_set(x)
        return x

# Train the fully-connected layer
# Load the model and Set up the criterion and optimizer
fcn_model = FullyConnectedNetwork()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(fcn_model.parameters(), lr = learning_rate)

# Initialize training/validation loss and accuracy list
list_of_train_loss_per_iter = []
list_of_train_acc_per_iter = []
list_of_valid_loss_per_iter = []
list_of_valid_acc_per_iter = []

# Train the model
for iter in range(max_iters):
    total_train_loss = 0
    total_train_acc = 0
    for xb, yb in batches:
        # Turn yb into labelb
        labelb = torch.argmax(yb, dim = 1)

        # Forward propagation
        optimizer.zero_grad()
        probs = fcn_model(xb)

        # Back propagation + Optimize
        loss = criterion(probs, labelb)
        loss.backward()
        optimizer.step()

        # Compute loss and accuaracy
        total_train_loss += loss.item()
        _, pred_label = torch.max(probs.data, 1)
        acc = torch.eq(pred_label, labelb).float().sum()
        total_train_acc += acc.item()

    # Average and Update training loss and accuracy
    average_train_loss = total_train_loss / batch_num
    average_train_acc = total_train_acc / train_x.shape[0]
    list_of_train_loss_per_iter.append(average_train_loss)
    list_of_train_acc_per_iter.append(average_train_acc)

    # Validation process
    # Turn valid_y into valid_label
    valid_label = torch.argmax(valid_y, dim = 1)

```

```
    # Forward propagation
    valid_probs = fcn_model(valid_x)

    # Compute loss and accuracy
    valid_loss = criterion(valid_probs, valid_label)
    _, pred_valid_label = torch.max(valid_probs.data, 1)
    valid_acc = torch.eq(pred_valid_label, valid_label).float().mean()

    # Update validation loss and accuracy
    list_of_valid_loss_per_iter.append(valid_loss)
    list_of_valid_acc_per_iter.append(valid_acc)

    # Print the training and validation process
    print("iter: {:02d}".format(iter+1))
    print("train loss: {:.2f} \t train acc : {:.2f}".format(average_train_loss,
                                                              average_train_acc))
    print("validation loss : {:.2f} \t validation acc : {:.2f}"
                                                  .format(valid_loss, valid_acc))

# Test the model with the test dataset
# Turn test_y into test_label
test_label = torch.argmax(test_y, dim = 1)
# Forward propagation
test_probs = fcn_model(test_x)
# Compute loss and accuracy
test_loss = criterion(test_probs, test_label)
_, pred_test_label = torch.max(test_probs.data, 1)
test_acc = torch.eq(pred_test_label, test_label).float().mean()
print("Test\ntest loss : {:.2f} \t test accuracy : {:.2f}".format(test_loss,
                                                                    test_acc))

# Plot the training and validation process (loss)
plt.figure()
plt.plot(np.arange(1, max_iters+1), list_of_train_loss_per_iter, color = 'r',
                                                              label = 'training')
plt.plot(np.arange(1, max_iters+1), list_of_valid_loss_per_iter, color = 'b',
                                                            label = 'validation')
plt.title(f"Training and Validation Process: loss v.s. epoch")
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.show()

# Plot the training and validation process (accuracy)
plt.figure()
plt.plot(np.arange(1, max_iters+1), list_of_train_acc_per_iter, color = 'r',
                                                              label = 'training')
plt.plot(np.arange(1, max_iters+1), list_of_valid_acc_per_iter, color = 'b',
```

```
                                                            label = 'validation')
plt.title(f"Training and Validation Process: accuracy v.s. epoch")
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.legend()
plt.show()

# Q6.1.2 Convolutional Neural Network
# Set up the hyperparameters
max_iters = 10
batch_size = 72
learning_rate = 4e-3
train_batches = get_random_batches(train_x, train_y, batch_size)
train_batch_num = len(train_batches)
valid_batches = get_random_batches(valid_x, valid_y, batch_size)
valid_batch_num = len(valid_batches)

# Build the CNN
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        # Convolutional block
        self. conv_layer_set = nn.Sequential(
            nn.Conv2d(in_channels = 1, out_channels = 64, kernel_size = 3,
                                                                    padding = 1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace = True),
            nn.MaxPool2d(kernel_size = 2, stride = 2),
            nn.Dropout(p = 0.1),
            nn.Conv2d(in_channels = 64, out_channels = 32, kernel_size = 3,
                                                                    padding = 1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace = True),
            nn.MaxPool2d(kernel_size = 2, stride = 2),
            nn.Dropout(p = 0.1)
        )
        # Fully-connected block
        self. fc_layer_set = nn.Sequential(
            nn.Linear(32 * 8 * 8, 1024),
            nn.ReLU(inplace = True),
            nn.Linear(1024, 36),
        )

    def forward(self, x):
        # Convolution layer block
        x = self.conv_layer_set(x)
        x = x.view(-1, 32 * 8 * 8)
        # Fully-connected layer block
```

```
        x = self.fc_layer_set(x)
        return x

# Train the CNN
# Choose the device to use
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Load the model and Set up the criterion and optimizer
cnn_model = CNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(cnn_model.parameters(), lr = learning_rate)

# Initialize training/validation loss and accuracy list
list_of_train_loss_per_iter = []
list_of_train_acc_per_iter = []
list_of_valid_loss_per_iter = []
list_of_valid_acc_per_iter = []

# Train the model
for iter in range(max_iters):
    total_train_loss = 0
    total_train_acc = 0
    total_valid_loss = 0
    total_valid_acc = 0
    for xb, yb in train_batches:
        # Reshape xb and Turn yb into labelb
        xb = xb.reshape(batch_size, 1, 32, 32).to(device)
        labelb = torch.argmax(yb, dim = 1).to(device)

        # Forward propagation
        optimizer.zero_grad()
        probs = cnn_model(xb)

        # Back propagation + Optimize
        loss = criterion(probs, labelb)
        loss.backward()
        optimizer.step()

        # Compute loss and accuaracy
        total_train_loss += loss.item()
        _, pred_label = torch.max(probs.data, 1)
        acc = torch.eq(pred_label, labelb).float().sum()
        total_train_acc += acc.item()

    # Average and Update training loss and accuracy
    average_train_loss = total_train_loss / train_batch_num
    average_train_acc = total_train_acc / train_x.shape[0]
    list_of_train_loss_per_iter.append(average_train_loss)
```

```
    list_of_train_acc_per_iter.append(average_train_acc)

    # Validation process
    for valid_xb, valid_yb in valid_batches:
        # Reshape xb and Turn yb into labelb
        valid_xb = valid_xb.reshape(batch_size, 1, 32, 32).to(device)
        valid_labelb = torch.argmax(valid_yb, dim = 1).to(device)

        # Forward propagation
        optimizer.zero_grad()
        valid_probs = cnn_model(valid_xb)

        # Back propagation + Optimize
        loss = criterion(valid_probs, valid_labelb)
        loss.backward()
        optimizer.step()

        # Compute loss and accuaracy
        total_valid_loss += loss.item()
        _, pred_valid_label = torch.max(valid_probs.data, 1)
        valid_acc = torch.eq(pred_valid_label, valid_labelb).float().sum()
        total_valid_acc += valid_acc.item()

    # Average and Update validation loss and accuracy
    average_valid_loss = total_valid_loss / valid_batch_num
    average_valid_acc = total_valid_acc / valid_x.shape[0]
    list_of_valid_loss_per_iter.append(average_valid_loss)
    list_of_valid_acc_per_iter.append(average_valid_acc)

    # Print the training and validation process
    print("iter: {:02d}".format(iter+1))
    print("train loss: {:.2f} \t train acc : {:.2f}".format(average_train_loss,
                                                            average_train_acc))
    print("validation loss : {:.2f} \t validation acc :
                        {:.2f}".format(average_valid_loss, average_valid_acc))

# Test the model with the test dataset
# Reshape test_x and Turn test_y into test_label
test_x = test_x.reshape(test_x.shape[0], 1, 32, 32).to(device)
test_label = torch.argmax(test_y, dim = 1).to(device)
# Forward propagation
test_probs = cnn_model(test_x)
# Compute loss and accuracy
test_loss = criterion(test_probs, test_label)
_, pred_test_label = torch.max(test_probs.data, 1)
test_acc = torch.eq(pred_test_label, test_label).float().mean()
print("Test\ntest loss : {:.2f} \t test accuracy : {:.2f}".format(test_loss,
                                                                  test_acc))
```

```
# Plot the training and validation process (loss)
plt.figure()
plt.plot(np.arange(1, max_iters+1), list_of_train_loss_per_iter, color = 'r',
                                                              label = 'training')
plt.plot(np.arange(1, max_iters+1), list_of_valid_loss_per_iter, color = 'b',
                                                            label = 'validation')
plt.title(f"Training and Validation Process: loss v.s. epoch")
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.show()

# Plot the training and validation process (accuracy)
plt.figure()
plt.plot(np.arange(1, max_iters+1), list_of_train_acc_per_iter, color = 'r',
                                                              label = 'training')
plt.plot(np.arange(1, max_iters+1), list_of_valid_acc_per_iter, color = 'b',
                                                            label = 'validation')
plt.title(f"Training and Validation Process: accuracy v.s. epoch")
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.legend()
plt.show()

# Q6.1.3 Convolutional Neural Network on CIFAR-10 dataset
# Set up the hyperparameters
max_iters = 30
batch_size = 200
learning_rate = 0.001

# Load the training and testing datatset
# Set up the transform for training and testing dataset
transform = transforms.Compose([transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))])
# Load the dataset
train_set = torchvision.datasets.CIFAR10(root = './data', train = True,
                                         download = True, transform = transform)
train_loader = torch.utils.data.DataLoader(train_set, batch_size = batch_size,
                                                                 shuffle = True)
test_set = torchvision.datasets.CIFAR10(root = './data', train = False,
                                         download = True, transform = transform)
test_loader = torch.utils.data.DataLoader(test_set, batch_size = batch_size,
                                                                shuffle = False)

# Build the CNN
class CNN_CIFAR10(nn.Module):
    def __init__(self):
```

```
        super(CNN_CIFAR10, self).__init__()
        # Convolutional block
        self. conv_layer_set = nn.Sequential(
            nn.Conv2d(in_channels = 3, out_channels = 128, kernel_size = 3,
                                                                padding = 1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace = True),
            nn.MaxPool2d(kernel_size = 2, stride = 2),
            nn.Conv2d(in_channels = 128, out_channels = 256, kernel_size = 3,
                                                                padding = 1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace = True),
            nn.MaxPool2d(kernel_size = 2, stride = 2),
            nn.Dropout(p = 0.1)
        )
        # Fully-connected block
        self. fc_layer_set = nn.Sequential(
            nn.Linear(256 * 8 * 8, 1024),
            nn.ReLU(inplace = True),
            nn.Linear(1024, 512),
            nn.ReLU(inplace = True),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        # Convolution layer block
        x = self.conv_layer_set(x)
        x = x.view(-1, 256 * 8 * 8)
        # Fully-connected layer block
        x = self.fc_layer_set(x)
        return x

# Train the CNN
# Choose the device to use
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Load the model and Set up the criterion and optimizer
cnn_cifar10_model = CNN_CIFAR10().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(cnn_cifar10_model.parameters(), lr = learning_rate)

# Initialize training/validation loss and accuracy list
list_of_train_loss_per_iter = []
list_of_train_acc_per_iter = []
list_of_valid_loss_per_iter = []
list_of_valid_acc_per_iter = []

# Train the model
```

```
for iter in range(max_iters):
    total_train_loss = 0
    total_train_acc = 0
    total_valid_loss = 0
    total_valid_acc = 0
    for train_data in train_loader:
        # Extract xb and labelb
        xb, labelb = train_data
        xb, labelb = xb.to(device), labelb.to(device)

        # Forward propagation
        optimizer.zero_grad()
        probs = cnn_cifar10_model(xb)

        # Back propagation + Optimize
        loss = criterion(probs, labelb)
        loss.backward()
        optimizer.step()

        # Compute loss and accuaracy
        total_train_loss += loss.item()
        _, pred_label = torch.max(probs.data, 1)
        acc = torch.eq(pred_label, labelb).float().sum()
        total_train_acc += acc.item()

    # Average and Update training loss and accuracy
    average_train_loss = total_train_loss / len(train_loader)
    average_train_acc = total_train_acc / len(train_set)
    list_of_train_loss_per_iter.append(average_train_loss)
    list_of_train_acc_per_iter.append(average_train_acc)

    # Print the training process
    print("iter: {:02d}".format(iter+1))
    print("train loss: {:.2f} \t train acc : {:.2f}".format(average_train_loss,
                                                              average_train_acc))

# Test the model with the test dataset
test_loss = 0
test_acc = 0
for test_data in test_loader:
        # Extract test_xb and test_labelb
        test_xb, test_labelb = test_data
        test_xb, test_labelb = test_xb.to(device), test_labelb.to(device)

        # Forward propagation
        optimizer.zero_grad()
        test_probs = cnn_cifar10_model(test_xb)

```

```
        # Compute and Update loss and accuracy
        loss = criterion(test_probs, test_labelb)
        test_loss += loss.item()
        _, pred_test_label = torch.max(test_probs.data, 1)
        acc = torch.eq(pred_test_label, test_labelb).float().sum()
        test_acc += acc.item()

# Print the test loss and accuracy
test_loss = test_loss / len(test_loader)
test_acc = test_acc / len(test_set)
print("Test\ntest loss : {:.2f} \t test accuracy : {:.2f}".format(test_loss,
                                                                  test_acc))

# Plot the training process (loss)
plt.figure()
plt.plot(np.arange(1, max_iters+1), list_of_train_loss_per_iter, color = 'r')
plt.title(f"Training Process: loss v.s. epoch")
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()

# Plot the training and validation process (accuracy)
plt.figure()
plt.plot(np.arange(1, max_iters+1), list_of_train_acc_per_iter, color = 'r')
plt.title(f"Training Process: accuracy v.s. epoch")
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()

# Q6.1.4 Convolutional Neural Network on SUN dataset
# Set up the hyperparameters
max_iters = 30
batch_size = 50
learning_rate = 0.001

# Load the SUN dataset
transform = transforms.Compose([transforms.Resize((32, 32)),
    transforms.ToTensor(),
    transforms.Normalize((0.425, 0.425, 0.425), (0.225, 0.225, 0.225))])
sun_data = torchvision.datasets.ImageFolder(root = '../SUN', transform =
                                                                  transform)
# Split into training and testing dataset
train_set, test_set = torch.utils.data.random_split(sun_data, [1177, 400])
train_loader = torch.utils.data.DataLoader(train_set, batch_size = batch_size,
                                                              shuffle = True)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=batch_size,
                                                             shuffle = False)
```

```
# Build the CNN
class CNN_SUN(nn.Module):
    def __init__(self):
        super(CNN_SUN, self).__init__()
        # Convolutional block
        self. conv_layer_set = nn.Sequential(
            nn.Conv2d(in_channels = 3, out_channels = 64, kernel_size = 3,
                                                                 padding = 1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace = True),
            nn.MaxPool2d(kernel_size = 2, stride = 2),
            nn.Conv2d(in_channels = 64, out_channels = 128, kernel_size = 3,
                                                                 padding = 1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace = True),
            nn.MaxPool2d(kernel_size = 2, stride = 2),
            nn.Dropout(p = 0.1)
        )
        # Fully-connected block
        self. fc_layer_set = nn.Sequential(
            nn.Linear(128 * 8 * 8, 1024),
            nn.ReLU(inplace = True),
            nn.Linear(1024, 512),
            nn.ReLU(inplace = True),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        # Convolution layer block
        x = self.conv_layer_set(x)
        x = x.view(-1, 128 * 8 * 8)
        # Fully-connected layer block
        x = self.fc_layer_set(x)
        return x

# Train the CNN
# Choose the device to use
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Load the model and Set up the criterion and optimizer
cnn_sun_model = CNN_SUN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(cnn_sun_model.parameters(), lr = learning_rate)

# Initialize training/validation loss and accuracy list
list_of_train_loss_per_iter = []
list_of_train_acc_per_iter = []
list_of_valid_loss_per_iter = []
```

```
list_of_valid_acc_per_iter = []

# Train the model
for iter in range(max_iters):
    total_train_loss = 0
    total_train_acc = 0
    total_valid_loss = 0
    total_valid_acc = 0
    for train_data in train_loader:
        # Extract xb and labelb
        xb, labelb = train_data
        xb, labelb = xb.to(device), labelb.to(device)

        # Forward propagation
        optimizer.zero_grad()
        probs = cnn_sun_model(xb)

        # Back propagation + Optimize
        loss = criterion(probs, labelb)
        loss.backward()
        optimizer.step()

        # Compute loss and accuaracy
        total_train_loss += loss.item()
        _, pred_label = torch.max(probs.data, 1)
        acc = torch.eq(pred_label, labelb).float().sum()
        total_train_acc += acc.item()

    # Average and Update training loss and accuracy
    average_train_loss = total_train_loss / len(train_loader)
    average_train_acc = total_train_acc / len(train_set)
    list_of_train_loss_per_iter.append(average_train_loss)
    list_of_train_acc_per_iter.append(average_train_acc)

    # Print the training process
    print("iter: {:02d}".format(iter+1))
    print("train loss: {:.2f} \t train acc : {:.2f}".format(average_train_loss,
                                                             average_train_acc))

# Test the model with the test dataset
test_loss = 0
test_acc = 0
for test_data in test_loader:
        # Extract test_xb and test_labelb
        test_xb, test_labelb = test_data
        test_xb, test_labelb = test_xb.to(device), test_labelb.to(device)

        # Forward propagation
```

```
        optimizer.zero_grad()
        test_probs = cnn_sun_model(test_xb)

        # Compute and Update loss and accuracy
        loss = criterion(test_probs, test_labelb)
        test_loss += loss.item()
        _, pred_test_label = torch.max(test_probs.data, 1)
        acc = torch.eq(pred_test_label, test_labelb).float().sum()
        test_acc += acc.item()

# Print the test loss and accuracy
test_loss = test_loss / len(test_loader)
test_acc = test_acc / len(test_set)
print("Test\ntest loss : {:.2f} \t test accuracy : {:.2f}".format(test_loss,
                                                                   test_acc))

# Plot the training process (loss)
plt.figure()
plt.plot(np.arange(1, max_iters+1), list_of_train_loss_per_iter, color = 'r')
plt.title(f"Training Process: loss v.s. epoch")
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()

# Plot the training and validation process (accuracy)
plt.figure()
plt.plot(np.arange(1, max_iters+1), list_of_train_acc_per_iter, color = 'r')
plt.title(f"Training Process: accuracy v.s. epoch")
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```

0.7 run_q6_2.py

```
# Import package
import numpy as np
import matplotlib.pyplot as plt
import scipy.io
import torch
import torch.nn as nn
import torch.nn.functional as func
import torchvision
import torchvision.transforms as transforms
import torch.optim as optim


# Fine-tune a single layer classifier with SqueezeNet
# Set up the hyperparameters
batch_size = 32
max_iters1 = 20
max_iters2 = 20
learning_rate1 = 0.001
learning_rate2 = 0.0001

# Load flowers17 dataset
# For fine tuning
finetune_transform = transforms.Compose([transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize((0.425, 0.425, 0.425), (0.225, 0.225, 0.225))])
finetune_train_set = torchvision.datasets.ImageFolder(root =
               '../data/oxford-flowers17/train', transform = finetune_transform)
finetune_test_set = torchvision.datasets.ImageFolder(root =
                '../data/oxford-flowers17/test', transform = finetune_transform)
# Load the dataset into DataLoader
finetune_train_loader = torch.utils.data.DataLoader(finetune_train_set,
                                        batch_size = batch_size, shuffle = True)
finetune_test_loader = torch.utils.data.DataLoader(finetune_test_set,
                                       batch_size = batch_size, shuffle = False)

# Load the SqueezeNet
squeeze_model = torchvision.models.squeezenet1_1(pretrained = True)
# Replace the classifier layer into 102 classes
squeeze_model.classifier[1] = nn.Conv2d(in_channels = 512, out_channels = 102,
                                                                 kernel_size = 1)

# Train the SqueezeNet : 2 Steps
# Choose the device to use
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

```

```python
# Load the SqueezeNet
squeeze_model = torchvision.models.squeezenet1_1(pretrained = True).to(device)
# Replace the classifier layer into 102 classes
squeeze_model.classifier[1] = nn.Conv2d(in_channels = 512, out_channels = 102,
                                        kernel_size = 1).to(device)

# 1st Step : train the sinlgle classifier layer
for param in squeeze_model.parameters():
    param.requires_grad = False
for param in squeeze_model.classifier.parameters():
    param.requires_grad = True

# Set up the criterion and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(squeeze_model.classifier.parameters(), lr =
                                                            learning_rate1)

# Initialize training loss and accuracy list
list_of_train_loss_per_iter = []
list_of_train_acc_per_iter = []

# Train the classifier layer
print("Train the single layer classifier")
for iter in range(max_iters1):
    total_train_loss = 0
    total_train_acc = 0
    for train_data in finetune_train_loader:
        # Extract xb and labelb
        xb, labelb = train_data
        xb, labelb = xb.to(device), labelb.to(device)

        # Forward propagation
        optimizer.zero_grad()
        probs = squeeze_model(xb)

        # Back propagation + Optimize
        loss = criterion(probs, labelb)
        loss.backward()
        optimizer.step()

        # Compute loss and accuaracy
        total_train_loss += loss.item()
        _, pred_label = torch.max(probs.data, 1)
        acc = torch.eq(pred_label, labelb).float().sum()
        total_train_acc += acc.item()

    # Average and Update training loss and accuracy
    average_train_loss = total_train_loss / len(finetune_train_loader)
```

```
    average_train_acc = total_train_acc / len(finetune_train_set)
    list_of_train_loss_per_iter.append(average_train_loss)
    list_of_train_acc_per_iter.append(average_train_acc)

    # Print the training process
    print("iter: {:02d}".format(iter+1))
    print("train loss: {:.2f} \t train acc : {:.2f}".format(average_train_loss,
                                                            average_train_acc))

# 2nd Step : finetune the SqueezeNet
for param in squeeze_model.parameters():
    param.requires_grad = True

# Update the optimizer
optimizer = optim.Adam(squeeze_model.parameters(), lr = learning_rate2)

# Train the SqueezeNet
print("Train the SqueezeNet")
for iter in range(max_iters2):
    total_train_loss = 0
    total_train_acc = 0
    for train_data in finetune_train_loader:
        # Extract xb and labelb
        xb, labelb = train_data
        xb, labelb = xb.to(device), labelb.to(device)

        # Forward propagation
        optimizer.zero_grad()
        probs = squeeze_model(xb)

        # Back propagation + Optimize
        loss = criterion(probs, labelb)
        loss.backward()
        optimizer.step()

        # Compute loss and accuaracy
        total_train_loss += loss.item()
        _, pred_label = torch.max(probs.data, 1)
        acc = torch.eq(pred_label, labelb).float().sum()
        total_train_acc += acc.item()

    # Average and Update training loss and accuracy
    average_train_loss = total_train_loss / len(finetune_train_loader)
    average_train_acc = total_train_acc / len(finetune_train_set)
    list_of_train_loss_per_iter.append(average_train_loss)
    list_of_train_acc_per_iter.append(average_train_acc)

    # Print the training process
```

```
    print("iter: {:02d}".format(iter+1))
    print("train loss: {:.2f} \t train acc : {:.2f}".format(average_train_loss,
                                                           average_train_acc))

# Test the model with the test dataset
test_loss = 0
test_acc = 0
for test_data in finetune_test_loader:
        # Extract test_xb and test_labelb
        test_xb, test_labelb = test_data
        test_xb, test_labelb = test_xb.to(device), test_labelb.to(device)

        # Forward propagation
        optimizer.zero_grad()
        test_probs = squeeze_model(test_xb)

        # Compute and Update loss and accuracy
        loss = criterion(test_probs, test_labelb)
        test_loss += loss.item()
        _, pred_test_label = torch.max(test_probs.data, 1)
        acc = torch.eq(pred_test_label, test_labelb).float().sum()
        test_acc += acc.item()

# Print the test loss and accuracy
test_loss = test_loss / len(finetune_test_loader)
test_acc = test_acc / len(finetune_test_set)
print("Test\ntest loss : {:.2f} \t test accuracy : {:.2f}".format(test_loss,
                                                                  test_acc))

# Plot the training process (loss)
plt.figure()
plt.plot(np.arange(1, max_iters1+max_iters2+1), list_of_train_loss_per_iter,
                                                                 color = 'r')
plt.title(f"Training Process: loss v.s. epoch")
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()

# Plot the training and validation process (accuracy)
plt.figure()
plt.plot(np.arange(1, max_iters1+max_iters2+1), list_of_train_acc_per_iter,
                                                                 color = 'r')
plt.title(f"Training Process: accuracy v.s. epoch")
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()

# Self-defined CNN
```

```
# Set up the hyperparameters
batch_size = 32
max_iters = 40
learning_rate = 0.001

# Load flowers17 dataset
# For self-defined CNN
transform = transforms.Compose([transforms.Resize((32, 32)),
    transforms.ToTensor(),
    transforms.Normalize((0.425, 0.425, 0.425), (0.225, 0.225, 0.225))])
train_set = torchvision.datasets.ImageFolder(root =
                        '../data/oxford-flowers17/train', transform = transform)
test_set = torchvision.datasets.ImageFolder(root =
                         '../data/oxford-flowers17/test', transform = transform)
# Load the dataset into DataLoader
train_loader = torch.utils.data.DataLoader(train_set, batch_size = batch_size,
                                                                  shuffle = True)
test_loader = torch.utils.data.DataLoader(test_set, batch_size = batch_size,
                                                                 shuffle = False)

# Build the self-defined CNN (LeNet)
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        # Convolutional block : 3 Convolutional Layers
        self. conv_layer_set = nn.Sequential(
            nn.Conv2d(in_channels = 3, out_channels = 6, kernel_size = 5,
                                                                    stride = 1),
            nn.ReLU(inplace = True),
            nn.MaxPool2d(kernel_size = 2, stride = 2),
            nn.Conv2d(in_channels = 6, out_channels = 16, kernel_size = 5,
                                                                    stride = 1),
            nn.ReLU(inplace = True),
            nn.MaxPool2d(kernel_size = 2, stride = 2),
            nn.Conv2d(in_channels = 16, out_channels = 120, kernel_size = 5,
                                                                    stride = 1),
            nn.ReLU(inplace = True),
        )
        # Fully-connected block : 2 FC Layers
        self. fc_layer_set = nn.Sequential(
            nn.Linear(120, 84),
            nn.ReLU(inplace = True),
            nn.Linear(84, 102),
        )

    def forward(self, x):
        # Convolution layer block
        x = self.conv_layer_set(x)
```

```
        x = x.view(-1, 120)
        # Fully-connected layer block
        x = self.fc_layer_set(x)
        return x

# Train the self-defined CNN
# Choose the device to use
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Load the model and Set up the criterion and optimizer
cnn_model = CNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(cnn_model.parameters(), lr = learning_rate)

# Initialize training loss and accuracy list
list_of_train_loss_per_iter = []
list_of_train_acc_per_iter = []

# Train the model
for iter in range(max_iters):
    total_train_loss = 0
    total_train_acc = 0
    for train_data in train_loader:
        # Extract xb and labelb
        xb, labelb = train_data
        xb, labelb = xb.to(device), labelb.to(device)

        # Forward propagation
        optimizer.zero_grad()
        probs = cnn_model(xb)

        # Back propagation + Optimize
        loss = criterion(probs, labelb)
        loss.backward()
        optimizer.step()

        # Compute loss and accuaracy
        total_train_loss += loss.item()
        _, pred_label = torch.max(probs.data, 1)
        acc = torch.eq(pred_label, labelb).float().sum()
        total_train_acc += acc.item()

    # Average and Update training loss and accuracy
    average_train_loss = total_train_loss / len(train_loader)
    average_train_acc = total_train_acc / len(train_set)
    list_of_train_loss_per_iter.append(average_train_loss)
    list_of_train_acc_per_iter.append(average_train_acc)

```

```
    # Print the training process
    print("iter: {:02d}".format(iter+1))
    print("train loss: {:.2f} \t train acc : {:.2f}".format(average_train_loss,
                                                            average_train_acc))

# Test the model with the test dataset
test_loss = 0
test_acc = 0
for test_data in test_loader:
        # Extract test_xb and test_labelb
        test_xb, test_labelb = test_data
        test_xb, test_labelb = test_xb.to(device), test_labelb.to(device)

        # Forward propagation
        optimizer.zero_grad()
        test_probs = cnn_model(test_xb)

        # Compute and Update loss and accuracy
        loss = criterion(test_probs, test_labelb)
        test_loss += loss.item()
        _, pred_test_label = torch.max(test_probs.data, 1)
        acc = torch.eq(pred_test_label, test_labelb).float().sum()
        test_acc += acc.item()

# Print the test loss and accuracy
test_loss = test_loss / len(test_loader)
test_acc = test_acc / len(test_set)
print("Test\ntest loss : {:.2f} \t test accuracy : {:.2f}".format(test_loss,
                                                                  test_acc))

# Plot the training process (loss)
plt.figure()
plt.plot(np.arange(1, max_iters+1), list_of_train_loss_per_iter, color = 'r')
plt.title(f"Training Process: loss v.s. epoch")
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()

# Plot the training and validation process (accuracy)
plt.figure()
plt.plot(np.arange(1, max_iters+1), list_of_train_acc_per_iter, color = 'r')
plt.title(f"Training Process: accuracy v.s. epoch")
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```

0.8 util.py

```
import numpy as np

# use for a "no activation" layer
def linear(x):
    return x

def linear_deriv(post_act):
    return np.ones_like(post_act)

def tanh(x):
    return np.tanh(x)

def tanh_deriv(post_act):
    return 1-post_act**2

def relu(x):
    return np.maximum(x,0)

def relu_deriv(x):
    return (x > 0).astype(np.float)
```