

16-720A — Spring 2021 — Homework 5

Jen-Hung Ho
jenhungh@andrew.cmu.edu

April 25, 2021

Part 1

Q1.1

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

Apply translation c to the softmax function:

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

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:

$$f_1(x) = A_1 x + b_1 \quad (3)$$

$$f_2(f_1(x)) = A_2(f_1(x)) + b_2 = A_2(A_1 x + b_1) + b_2 = (A_2 A_1)x + (A_2 b_1 + b_2) = A^* x + b^* \quad (4)$$

$$\text{where } A^* = A_2 A_1 \text{ and } b^* = A_2 b_1 + b_2 \quad (5)$$

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

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

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

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

Q1.5

From the notional suggestions, we know that

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

$$W \in \mathbb{R}^{k \times d} \quad (10)$$

$$x \in \mathbb{R}^{d \times 1} \quad (11)$$

$$b \in \mathbb{R}^{k \times 1} \quad (12)$$

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

$$\frac{\partial J}{\partial x_j} = \frac{\partial J}{\partial y} \frac{\partial y}{\partial x_j} = \delta W_{ij} \quad (13)$$

$$\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial y} \frac{\partial y}{\partial W_{ij}} = \delta x_j \quad (14)$$

$$\frac{\partial J}{\partial b_i} = \frac{\partial J}{\partial y} \frac{\partial y}{\partial b_i} = \delta * 1 = \delta \quad (15)$$

Then derive the matrix form

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

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

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

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.

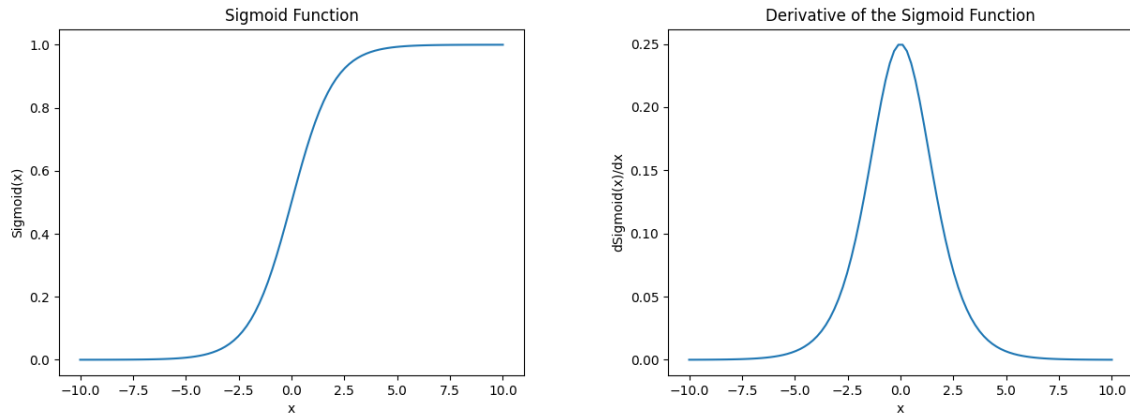


Figure 1: Sigmoid Function and its Derivative

(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.

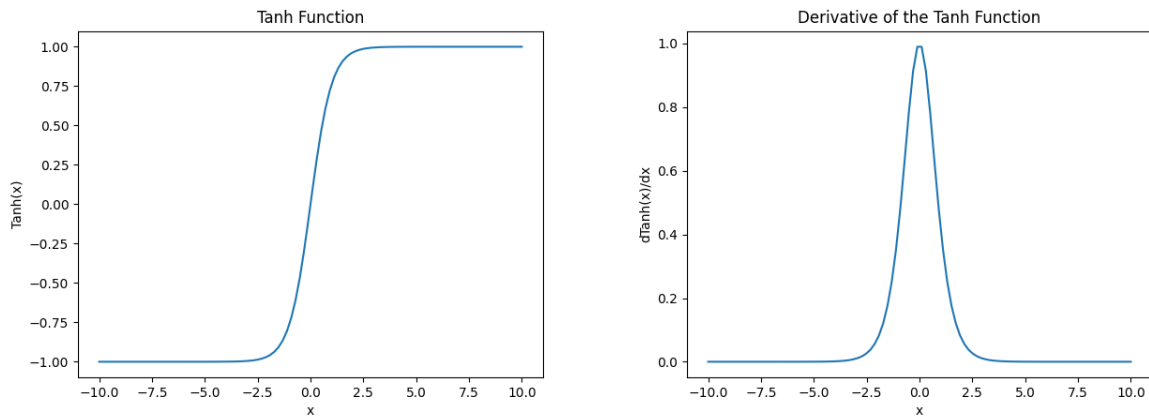


Figure 2: Tanh Function and its Derivative

(3) As shown in Figure 1 and 2, the derivative of 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)

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

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

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

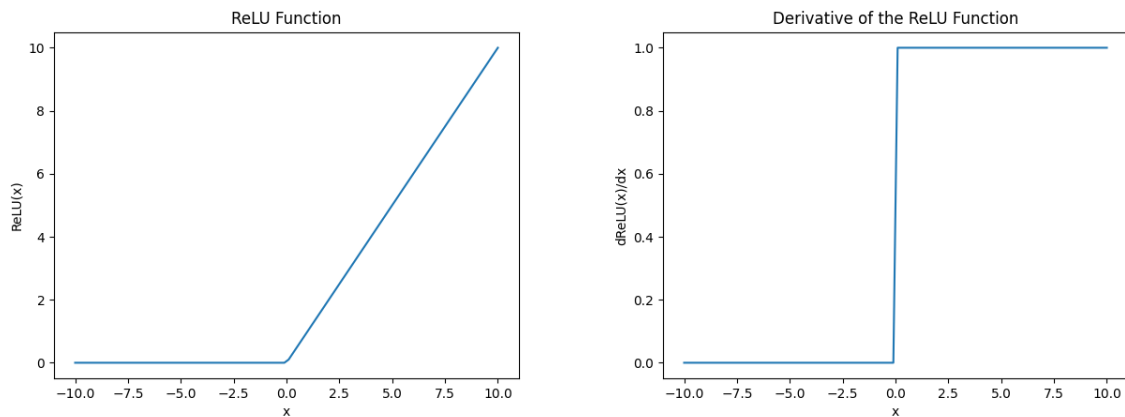


Figure 3: ReLU Function and its Derivative

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

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

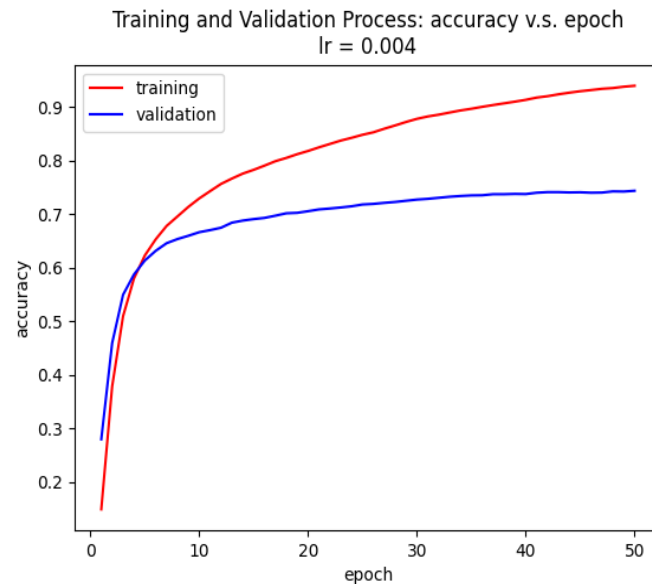


Figure 6: Training and Validation Process (Accuracy vs Epoch)

Q3.2

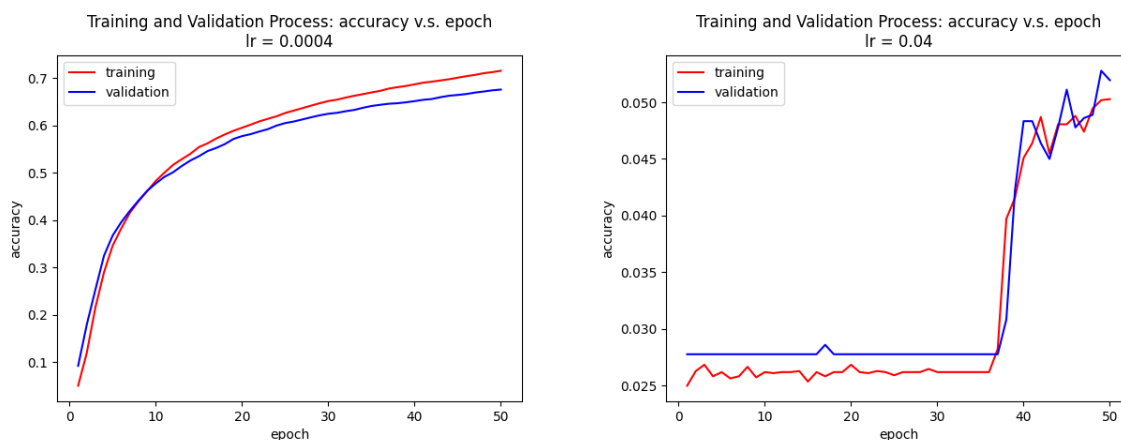


Figure 7: Training and Validation Process for the Two New Learning Rate

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.

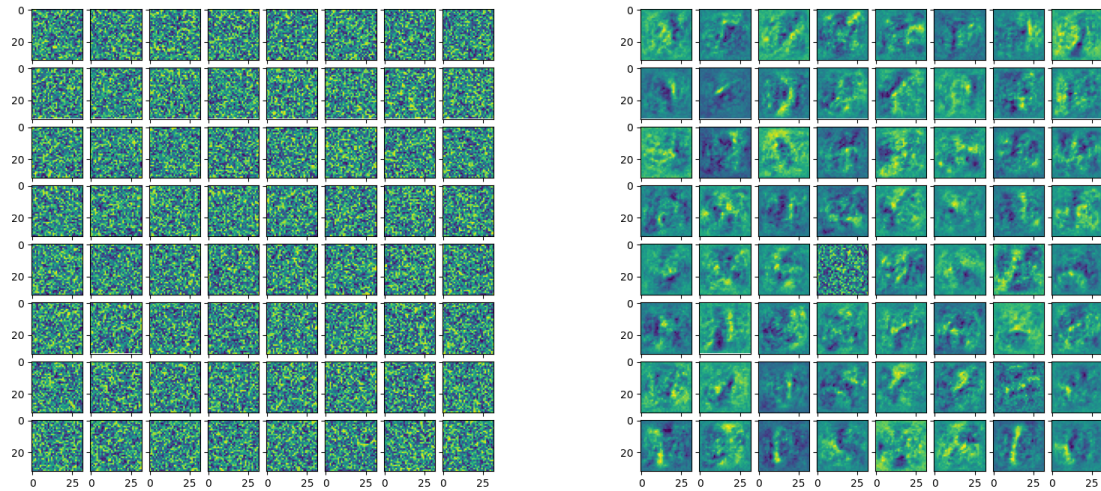
Q3.3

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

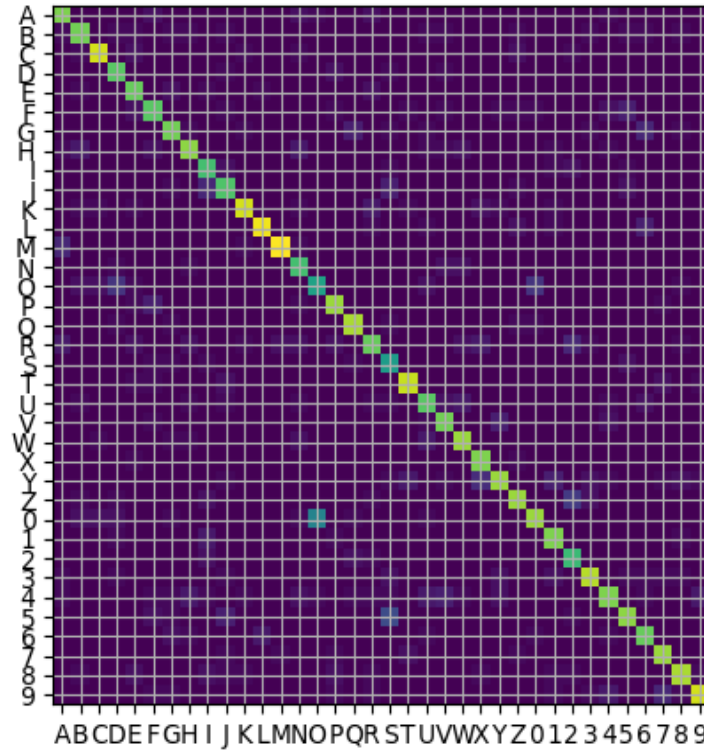


Figure 9: The Confusion Matrix on the Test Set

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.

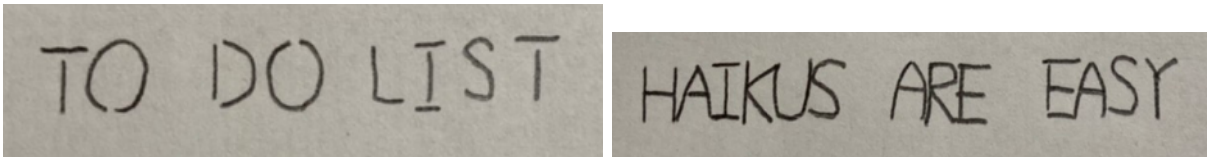


Figure 10: Example Images of Possible Characters Detection Failure

Q4.2

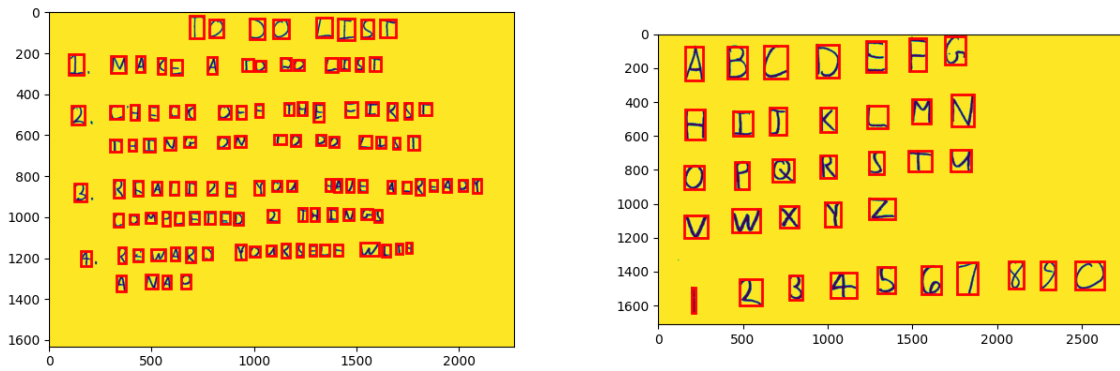


Figure 11: Visualization of the Bounding Boxes for 01_list and 02_letters

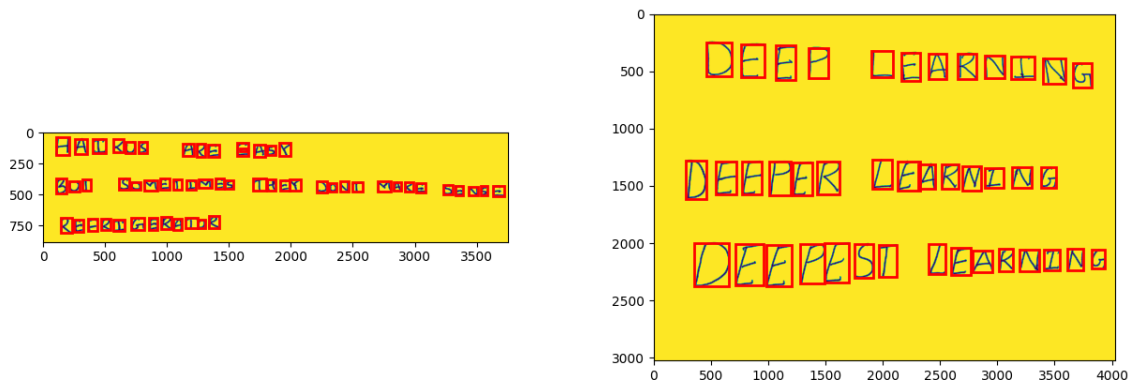


Figure 12: Visualization of the Bounding Boxes for 03_haiku and 04_deep

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

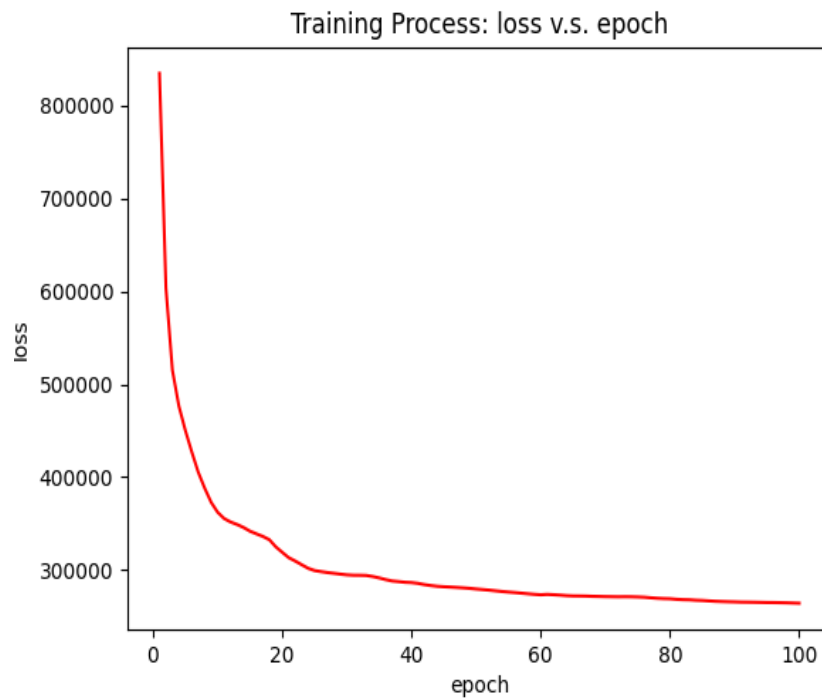


Figure 15: Training Process (Loss vs Epoch)

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

Q5.3.1

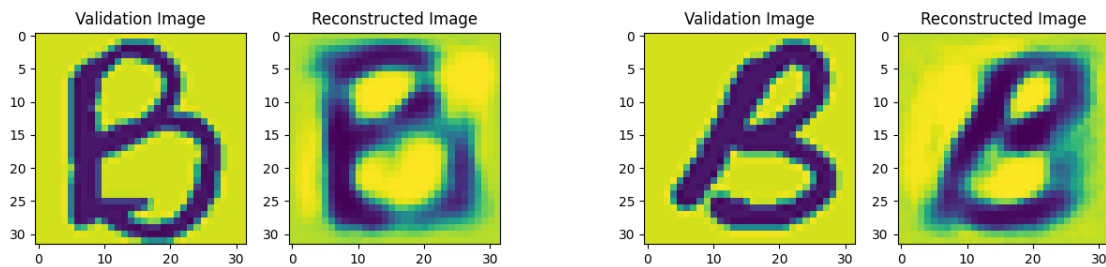


Figure 16: 2 Validation and Reconstructed Images of Class B

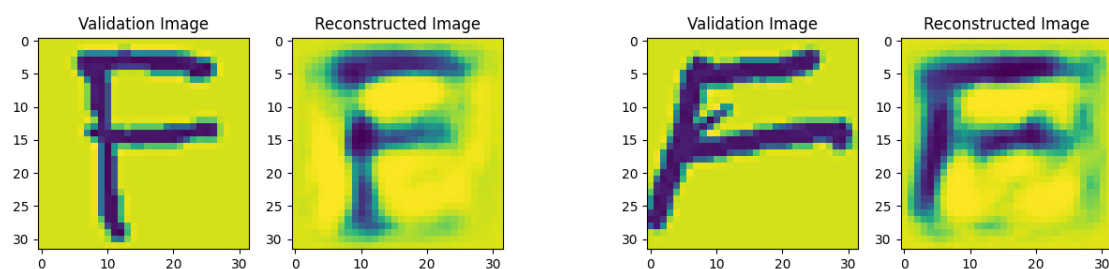


Figure 17: 2 Validation and Reconstructed Images of Class F

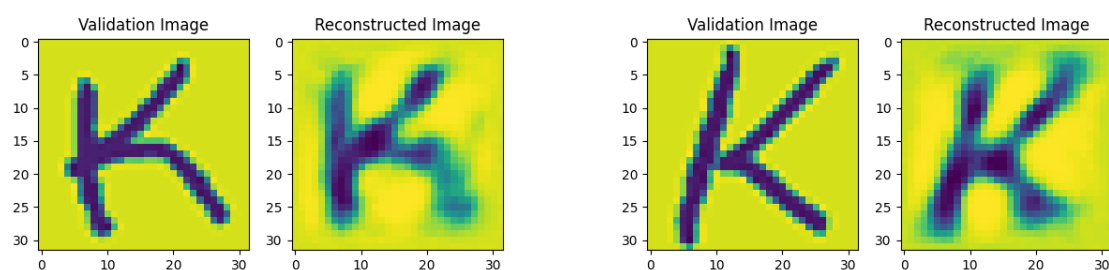


Figure 18: 2 Validation and Reconstructed Images of Class K

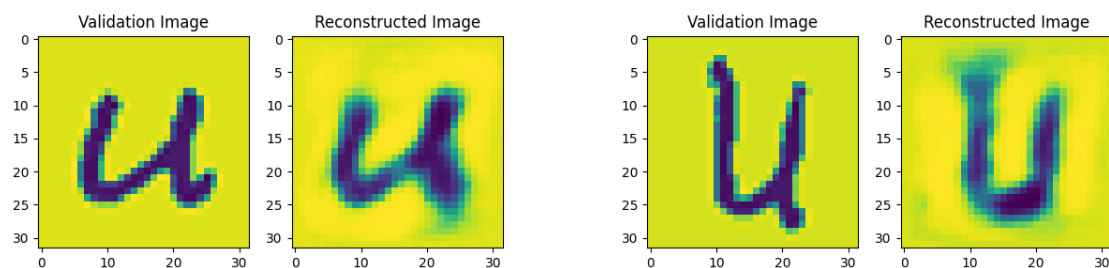


Figure 19: 2 Validation and Reconstructed Images of Class u

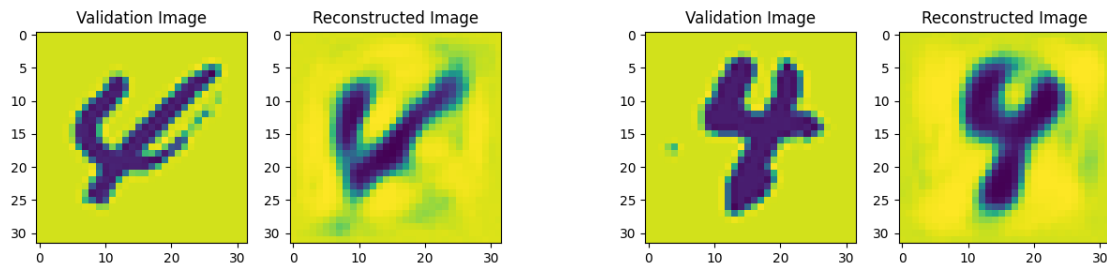


Figure 20: 2 Validation and Reconstructed Images of Class 4

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

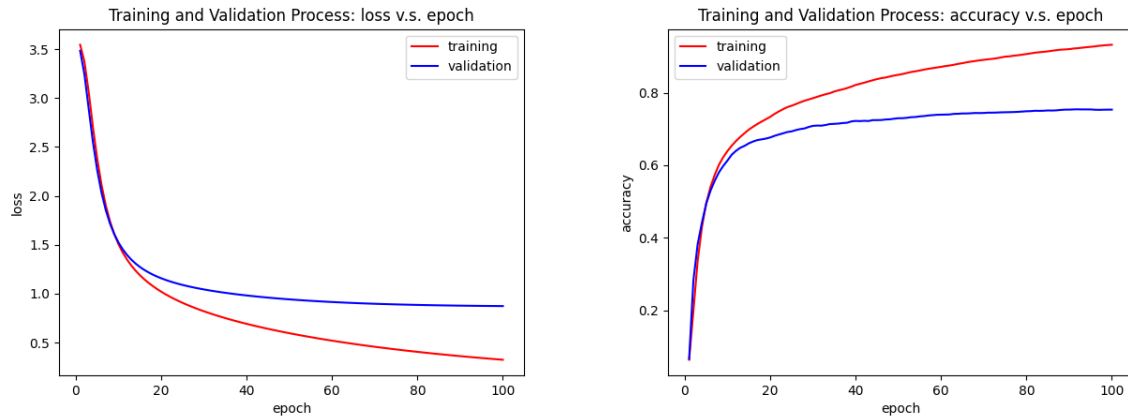


Figure 21: Training and Validation Process of the FC Network on the NIST36 dataset (Loss and Accuracy)

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

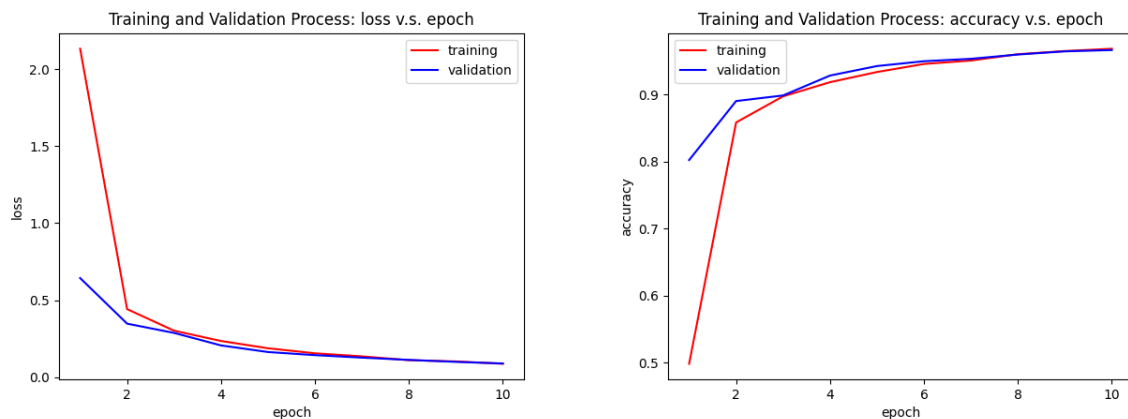


Figure 22: Training and Validation Process of the CNN on the NIST36 dataset (Loss and Accuracy)

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.

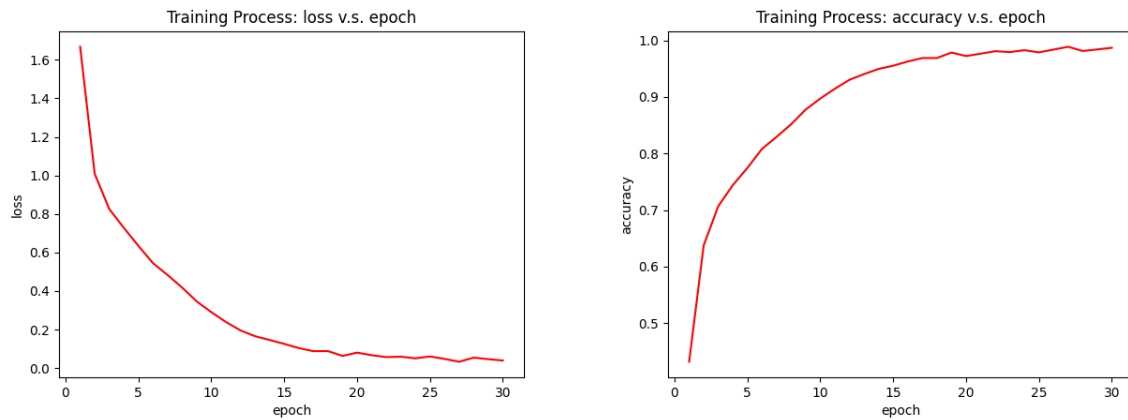
Q6.1.3

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%.

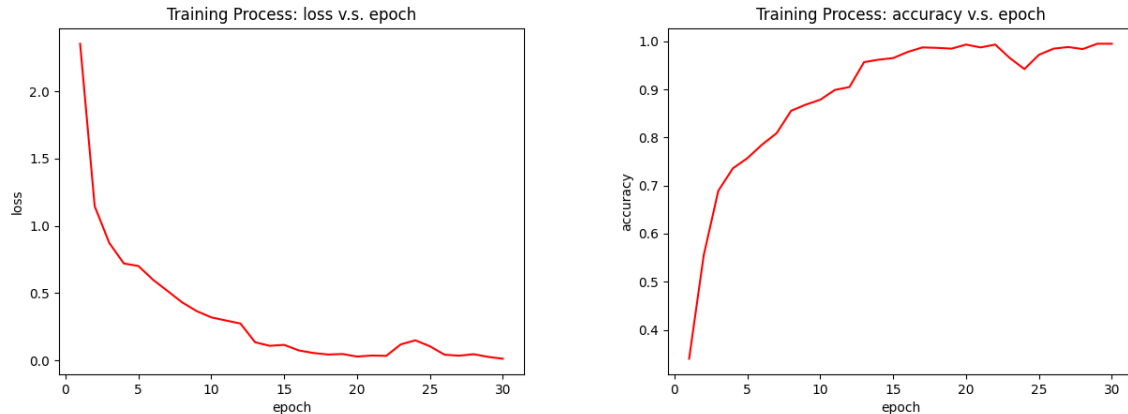
Q6.1.4

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

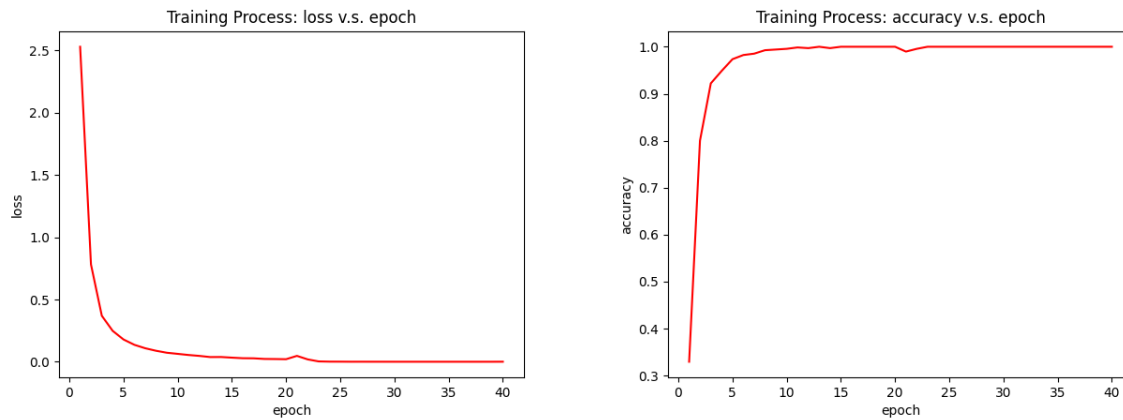


Figure 25: Training Process of the Fine-tune Single Layer Classifier on Oxford-Flower17 dataset (Loss and Accuracy)

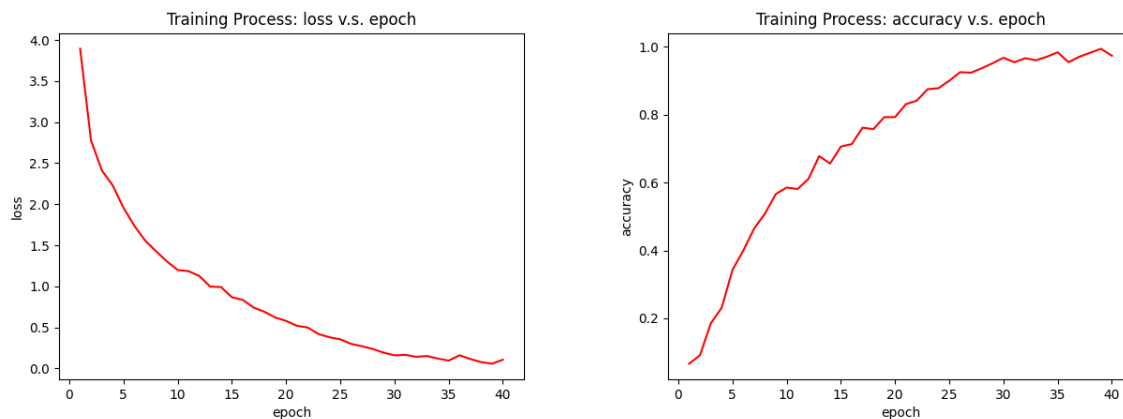


Figure 26: Training Process of the Self-defined CNN on Oxford-Flower17 dataset (Loss and Accuracy)

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

```

1  import numpy as np
2  from util import *
3  # do not include any more libraries here!
4  # do not put any code outside of functions!
5
6  ##### Q 2.1 #####
7  # initialize b to 0 vector
8  # b should be a 1D array, not a 2D array with a singleton dimension
9  # we will do  $XW + b$ .
10 # X be [Examples, Dimensions]
11 def initialize_weights(in_size, out_size, params, name=''):
12     W, b = None, None
13
14     #####
15     ##### your code here #####
16     #####
17     value = np.sqrt(6 / (in_size + out_size))
18     W = np.random.uniform(-value, value, size = (in_size, out_size))
19     b = np.zeros(out_size)
20
21     params['W' + name] = W
22     params['b' + name] = b
23
24     ##### Q 2.2.1 #####
25     # x is a matrix
26     # a sigmoid activation function
27     def sigmoid(x):
28         res = None
29
30         #####
31         ##### your code here #####
32         #####
33         res = 1 / (1 + np.exp(-x))
34
35         return res
36
37     ##### Q 2.2.1 #####
38     def forward(X, params, name='', activation=sigmoid):
39         """
40         Do a forward pass
41
42         Keyword arguments:
43         X -- input vector [Examples x D]
44         params -- a dictionary containing parameters

```

```

45     name -- name of the layer
46     activation -- the activation function (default is sigmoid)
47     """
48     pre_act, post_act = None, None
49     # get the layer parameters
50     W = params['W' + name]
51     b = params['b' + name]
52
53
54     #####
55     ##### your code here #####
56     #####
57     pre_act = X @ W + b
58     post_act = activation(pre_act)
59
60     # store the pre-activation and post-activation values
61     # these will be important in backprop
62     params['cache_' + name] = (X, pre_act, post_act)
63
64     return post_act
65
66     ##### Q 2.2.2 #####
67     # x is [examples, classes]
68     # softmax should be done for each row
69     def softmax(x):
70         res = None
71
72         #####
73         ##### your code here #####
74         #####
75         # Add translation c for numerical stability
76         c = -np.max(x, axis = 1).reshape(-1, 1)
77         x += c
78         # Compute softmax
79         numerator = np.exp(x)
80         denominator = np.sum(numerator, axis = 1).reshape(-1, 1)
81         res = numerator / denominator
82
83         return res
84
85     ##### Q 2.2.3 #####
86     # compute total loss and accuracy
87     # y is size [examples, classes]
88     # probs is size [examples, classes]
89     def compute_loss_and_acc(y, probs):
90         loss, acc = None, None
91

```

```

92     #####
93     ##### your code here #####
94     #####
95     # Compute the cross-entropy loss
96     loss = -np.sum(y * np.log(probs))
97     # Compute the accuracy
98     examples = y.shape[0]
99     count = 0
100    for i in range(examples):
101        pred_label = np.argmax(probs[i, :])
102        true_label = np.argmax(y[i, :])
103        if pred_label == true_label:
104            count += 1
105    acc = count / examples
106
107    return loss, acc
108
109    ##### Q 2.3 #####
110    # we give this to you
111    # because you proved it
112    # it's a function of post_act
113    def sigmoid_deriv(post_act):
114        res = post_act*(1.0-post_act)
115        return res
116
117    def backwards(delta, params, name='', activation_deriv=sigmoid_deriv):
118        """
119        Do a backwards pass
120
121        Keyword arguments:
122        delta -- errors to backprop
123        params -- a dictionary containing parameters
124        name -- name of the layer
125        activation_deriv -- the derivative of the activation_func
126        """
127        # everything you may need for this layer
128        W = params['W' + name]
129        b = params['b' + name]
130        X, pre_act, post_act = params['cache_' + name]
131
132        grad_X, grad_W, grad_b = None, None, None
133        # do the derivative through activation first
134        # then compute the derivative W, b, and X
135        #####
136        ##### your code here #####
137        #####
138        # Derivative through activation
139        d_activation = activation_deriv(post_act)

```

```

140     dJ_dy = delta * d_activation
141     # Compute the derivative W, b, and X
142     grad_X = dJ_dy @ (W.T)
143     grad_W = (X.T) @ dJ_dy
144     grad_b = (dJ_dy.T @ np.ones((dJ_dy.shape[0], 1))).reshape(-1)
145
146     # store the gradients
147     params['grad_W' + name] = grad_W
148     params['grad_b' + name] = grad_b
149
150     return grad_X
151
152     ##### Q 2.4 #####
153     # split x and y into random batches
154     # return a list of [(batch1_x, batch1_y)...]
155     def get_random_batches(x, y, batch_size):
156         batches = []
157
158         #####
159         ##### your code here #####
160         #####
161         # Get the basic information for splitting
162         num_examples = x.shape[0]
163         num_batches = int(num_examples / batch_size)
164         # Split x and y into random batches
165         for i in range(num_batches):
166             random_index = np.random.choice(num_examples, size = batch_size,
167                                             replace = False)
168             batch_x = x[random_index, :]
169             batch_y = y[random_index, :]
170             batches.append((batch_x, batch_y))
171
172     return batches

```

0.2 run_q2.py

```

1  import numpy as np
2  # you should write your functions in nn.py
3  from nn import *
4  from util import *
5
6
7  # fake data
8  # feel free to plot it in 2D
9  # what do you think these 4 classes are?
10 g0 = np.random.multivariate_normal([3.6,40],[[0.05,0],[0,10]],10)
11 g1 = np.random.multivariate_normal([3.9,10],[[0.01,0],[0,5]],10)
12 g2 = np.random.multivariate_normal([3.4,30],[[0.25,0],[0,5]],10)
13 g3 = np.random.multivariate_normal([2.0,10],[[0.5,0],[0,10]],10)
14 x = np.vstack([g0,g1,g2,g3])
15 # we will do  $XW + B$ 
16 # that implies that the data is  $N \times D$ 
17
18 # create labels
19 y_idx = np.array([0 for _ in range(10)] + [1 for _ in range(10)] +
20                  [2 for _ in range(10)] + [3 for _ in range(10)])
21 # turn to one_hot
22 y = np.zeros((y_idx.shape[0],y_idx.max()+1))
23 y[np.arange(y_idx.shape[0]),y_idx] = 1
24
25 # parameters in a dictionary
26 params = {}
27
28 # Q 2.1
29 # initialize a layer
30 initialize_weights(2,25,params,'layer1')
31 initialize_weights(25,4,params,'output')
32 assert(params['Wlayer1'].shape == (2,25))
33 assert(params['blayer1'].shape == (25,))
34
35 #expect 0, [0.05 to 0.12]
36 print("{}, {:.2f}".format(params['blayer1'].sum(),params['Wlayer1'].std()2))
37 print("{}, {:.2f}".format(params['boutput'].sum(),params['Woutput'].std()2))
38
39 # Q 2.2.1
40 # implement sigmoid
41 test = sigmoid(np.array([-100,100]))
42 print('should be zero and one\t',test.min(),test.max())
43 # implement forward
44 h1 = forward(x,params,'layer1')
45 print(h1.shape)
46 # Q 2.2.2

```

```

47 # implement softmax
48 probs = forward(h1,params,'output',softmax)
49 # make sure you understand these values!
50 # positive, ~1, ~1, (40,4)
51 print(probs.min(),min(probs.sum(1)),max(probs.sum(1)),probs.shape)
52
53 # Q 2.2.3
54 # implement compute_loss_and_acc
55 loss, acc = compute_loss_and_acc(y, probs)
56 # should be around -np.log(0.25)*40 [~55] and 0.25
57 # if it is not, check softmax!
58 print("{}, {:.2f}".format(loss,acc))
59
60 # here we cheat for you
61 # the derivative of cross-entropy(softmax(x)) is probs - 1[correct actions]
62 delta1 = probs
63 delta1[np.arange(probs.shape[0]),y_idx] -= 1
64
65 # we already did derivative through softmax
66 # so we pass in a linear_deriv, which is just a vector of ones
67 # to make this a no-op
68 delta2 = backwards(delta1,params,'output',linear_deriv)
69 # Implement backwards!
70 backwards(delta2,params,'layer1',sigmoid_deriv)
71
72 # W and b should match their gradients sizes
73 for k,v in sorted(list(params.items())):
74     if 'grad' in k:
75         name = k.split('_')[1]
76         print(name,v.shape, params[name].shape)
77
78 # Q 2.4
79 batches = get_random_batches(x,y,5)
80 # print batch sizes
81 print([_[0].shape[0] for _ in batches])
82 batch_num = len(batches)
83
84 print()
85 print("#####")
86 print("#####      Training Loop      #####")
87 print("#####")
88 print()
89
90 max_iters = 501
91 learning_rate = 1e-3
92 # with default settings, you should get loss < 35 and accuracy > 75%
93 # Here we also cheat for you and write the forward + backward loop for you
94 for itr in range(max_iters):

```



```

95     total_loss = 0
96     avg_acc = 0
97     for xb,yb in batches:
98         # forward
99         yb_idx = np.argmax(yb,1)
100        h1 = forward(xb,params,'layer1')
101        probs = forward(h1,params,'output',softmax)
102
103        # loss
104        # be sure to add loss and accuracy to epoch totals
105        loss, acc = compute_loss_and_acc(yb, probs)
106        total_loss += loss
107        avg_acc += acc/batch_num
108
109        # backward
110        delta1 = probs
111        delta1[np.arange(probs.shape[0]),yb_idx] -= 1
112
113        delta2 = backwards(delta1,params,'output',linear_deriv)
114        backwards(delta2,params,'layer1',sigmoid_deriv)
115
116        # apply gradient
117        names = ['layer1','output']
118        for k in names:
119            params['W'+k] -= learning_rate * params['grad_W' + k]
120            params['b'+k] -= learning_rate * params['grad_b' + k]
121
122
123
124        if itr % 100 == 0:
125            print("itr: {:02d} \t loss: {:.2f} \t acc :
126                  {:.2f}".format(itr,total_loss,avg_acc))
127
128
129    print()
130    print("#####")
131    print("#####          Training Loop End          #####")
132    print("#####")
133    print()
134
135    print()
136    print("#####")
137    print("#####          Numerical Gradient Checker          #####")
138    print("#####")
139    print()
140
141    h1 = forward(x,params,'layer1')
142    probs = forward(h1,params,'output',softmax)

```

```

143 delta1 = probs
144 delta1[np.arange(probs.shape[0]),y_idx] -= 1
145
146 delta2 = backwards(delta1,params,'output',linear_deriv)
147 backwards(delta2,params,'layer1',sigmoid_deriv)
148
149 # save the old params
150 import copy
151 params_orig = copy.deepcopy(params)
152
153 eps = 1e-6
154 for k,v in params.items():
155     if '_' in k:
156         continue
157     # we have a real parameter!
158     # for each value inside the parameter
159     # add epsilon
160     # run the network
161     # get the loss
162     # compute derivative with central diffs
163
164     def f(p):
165         h1 = forward(x,p,'layer1')
166         probs = forward(h1,p,'output',softmax)
167         new_loss, _ = compute_loss_and_acc(y, probs)
168         return new_loss
169     grad_v = np.zeros_like(v)
170     it = np.nditer(params[k], flags=['multi_index'], op_flags=['readwrite'])
171     while not it.finished:
172
173         # evaluate function at x+h
174         ix = it.multi_index
175         og = v[ix]
176         v[ix] += eps # increment by h
177         fxh = f(params) # evalute f(x + h)
178         v[ix] -= 2*eps # restore to previous value (very important!)
179         fx = f(params) # evalute f(x + h)
180
181         # compute the partial derivative
182         grad_v[ix] = (fxh - fx) / (2*eps) # the slope
183         v[ix] = og
184         it.iternext() # step to next dimension
185     params['grad_' + k] = grad_v
186
187 total_error = 0
188 for k in params.keys():
189     if 'grad_' in k:
190         # relative error

```

```
191         err = np.abs(params[k] - params_orig[k])/np.maximum(np.abs(params[k]),
192                                                                np.abs(params_orig[k]))
193         err = err.sum()
194         print('{} {:.2e}'.format(k, err))
195         total_error += err
196         # should be less than 1e-4
197         print('total {:.2e}'.format(total_error))
198         print()
199         print("#####")
200         print("#####      Numerical Gradient Checker END      #####")
201         print("#####")
202         print()
```

0.3 run_q3.py

```

1  import numpy as np
2  import scipy.io
3  from nn import *
4  import pickle
5  import matplotlib.pyplot as plt
6
7  train_data = scipy.io.loadmat('../data/nist36_train.mat')
8  valid_data = scipy.io.loadmat('../data/nist36_valid.mat')
9  test_data = scipy.io.loadmat('../data/nist36_test.mat')
10
11 train_x, train_y = train_data['train_data'], train_data['train_labels']
12 valid_x, valid_y = valid_data['valid_data'], valid_data['valid_labels']
13 test_x, test_y = test_data['test_data'], test_data['test_labels']
14
15 max_iters = 50  # For q4, max_iters = 100
16 batch_size = 72
17 learning_rate = 4e-3  # For q4, lr = 5e-3
18 hidden_size = 64
19
20 batches = get_random_batches(train_x, train_y, batch_size)
21 batch_num = len(batches)
22
23 params = {}
24
25 # We have initialized the single layer network for you here
26 # Do not change the layer names
27 initialize_weights(train_x.shape[1], hidden_size, params, 'layer1')
28 initialize_weights(hidden_size, train_y.shape[1], params, 'output')
29
30 # Store the intialized weights for Q3.3
31 initial_w = np.copy(params['Wlayer1'])
32
33 list_of_train_acc_per_iter = []
34 list_of_val_acc_per_iter = []
35 for itr in range(max_iters):
36     total_train_loss = 0
37     total_train_acc = 0
38     for xb, yb in batches:
39         #####
40         ##### your code here #####
41         #####
42         # Forward propagation
43         h1 = forward(xb, params, 'layer1')
44         probs = forward(h1, params, 'output', softmax)
45
46         # Compute and Update loss and accuracy

```

```

47     loss, acc = compute_loss_and_acc(yb, probs)
48     total_train_loss += loss
49     total_train_acc += acc
50
51     # Back propagation
52     # Derivative of cross-entropy(softmax(x))
53     delta1 = probs - yb
54     delta2 = backwards(delta1, params, 'output', linear_deriv)
55     # Implement back propagation
56     backwards(delta2, params, 'layer1', sigmoid_deriv)
57
58     # Apply gradient descent
59     # Output layer
60     params['W' + 'output'] -= learning_rate * params['grad_W' + 'output']
61     params['b' + 'output'] -= learning_rate * params['grad_b' + 'output']
62     # Hidden layer
63     params['W' + 'layer1'] -= learning_rate * params['grad_W' + 'layer1']
64     params['b' + 'layer1'] -= learning_rate * params['grad_b' + 'layer1']
65
66     # Average training loss and accuracy
67     total_train_loss = total_train_loss / batch_num
68     total_train_acc = total_train_acc / batch_num
69
70     val_acc = None
71     # compute the validation accuracy here, make sure there is little
72     # overfitting issue
73     #####
74     ##### your code here #####
75     #####
76     # Forward propagation
77     h1 = forward(valid_x, params, 'layer1')
78     probs = forward(h1, params, 'output', softmax)
79
80     # Compute loss and accuracy
81     val_loss, val_acc = compute_loss_and_acc(valid_y, probs)
82
83     # Update training and validation accuracy
84     list_of_train_acc_per_iter.append(total_train_acc)
85     list_of_val_acc_per_iter.append(val_acc)
86
87     print("itr: {:02d} \t train loss: {:.2f} \t train acc : {:.2f} \t
88           validation acc : {:.2f}".format(itr+1, total_train_loss,
89           total_train_acc, val_acc))
90
91     # In a single plot, plot the training v.s. validation accuracy per iteration
92     #####
93     ##### your code here #####
94     #####

```

```

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)

```

[illegible]

0.4 run_q4.py

```

1  import os
2  import numpy as np
3  import matplotlib.pyplot as plt
4  import matplotlib.patches
5
6  import skimage
7  import skimage.measure
8  import skimage.color
9  import skimage.restoration
10 import skimage.io
11 import skimage.filters
12 import skimage.morphology
13 import skimage.segmentation
14
15 from nn import *
16 from q4 import *
17 # do not include any more libraries here!
18 # no opencv, no sklearn, etc!
19 import warnings
20 warnings.simplefilter(action='ignore', category=FutureWarning)
21 warnings.simplefilter(action='ignore', category=UserWarning)
22
23 for img in os.listdir('../images'):
24     im1 = skimage.img_as_float(skimage.io.imread(os.path.join('../images',img)))
25     bboxes, bw = findLetters(im1)
26
27     plt.imshow(bw)
28     for bbox in bboxes:
29         minr, minc, maxr, maxc = bbox
30         rect = matplotlib.patches.Rectangle((minc, minr), maxc - minc,
31                                             maxr - minr, fill=False, edgecolor='red', linewidth=2)
32         plt.gca().add_patch(rect)
33     plt.show()
34
35     # find the rows using..RANSAC, counting, clustering, etc.
36     #####
37     ##### your code here #####
38     #####
39     # Split the rows based on min_row and max_row
40     # Sort the bboxes based on max_row
41     bboxes.sort(key = lambda x : x[2])
42
43     # Initialize rows and single_row
44     all_rows, single_row = [], []
45
46     # Split the characters into rows

```



```

47     boundary = bboxes[0][2]
48     for bbox in bboxes:
49         # Extract min_row and max_row
50         min_row, max_row = bbox[0], bbox[2]
51         # Check for new rows
52         if min_row > boundary:
53             # Sort the previous row based on min_column
54             single_row.sort(key = lambda x : x[1])
55             # Update rows
56             all_rows.append(single_row)
57             # Update boundary and Reset single_row
58             boundary = max_row
59             single_row = []
60             # Update single_row
61             single_row.append(bbox)
62     # Sort and Update the final single_row into rows
63     single_row.sort(key = lambda x : x[1])
64     all_rows.append(single_row)
65
66     # crop the bounding boxes
67     # note.. before you flatten, transpose the image (that's how the dataset is!)
68     # consider doing a square crop, and even using np.pad() to get your
69     # images looking more like the dataset
70     #####
71     ##### your code here #####
72     #####
73     # Crop the bounding box
74     crop_bboxes, row_crop_bboxes = [], []
75     for r in range(len(all_rows)):
76         for i, bbox in enumerate(all_rows[r]):
77             # Crop the bbox
78             min_row, min_col, max_row, max_col = bbox
79             crop_bbox = bw[min_row:max_row, min_col:max_col]
80             # Padding
81             crop_bbox = np.pad(crop_bbox, ((30, 30), (30, 30)), 'constant',
82                                constant_values = (1, 1))
83             # Resize and Preprocessing (erosion)
84             crop_bbox = skimage.transform.resize(crop_bbox, (32, 32))
85             crop_bbox = skimage.morphology.erosion(crop_bbox,
86                                                     np.array([[0, 1, 0], [1, 1, 1], [0, 1, 0]]))
87             # Transpose, Flatten, and Update
88             crop_bbox = crop_bbox.T
89             crop_bbox = crop_bbox.reshape(1, -1)
90             row_crop_bboxes.append(crop_bbox)
91     # Update crop_bboxes and Reset row_crop_bboxes
92     crop_bboxes.append(row_crop_bboxes)
93     row_crop_bboxes = []
94

```

```

95     # load the weights
96     # run the crops through your neural network and print them out
97     import pickle
98     import string
99     letters = np.array([_ for _ in string.ascii_uppercase[:26]]
100                        + [str(_) for _ in range(10)])
101     params = pickle.load(open('q3_weights.pickle', 'rb'))
102     #####
103     ##### your code here #####
104     #####
105     # Print the image name
106     print("\n" + img.split('.')[0] + "\n")
107
108     # Classify the letter in crop_bboxes
109     for r in range(len(crop_bboxes)):
110         for x in crop_bboxes[r]:
111             # Forward propagation
112             h1 = forward(x, params, 'layer1')
113             probs = forward(h1, params, 'output', softmax)
114             # Evaluate the one-hot vector of the letter
115             letter_index = np.argmax(probs, axis = 1)
116             # Transfer index into letters
117             letter = letters[letter_index][0]
118             # Print out the letters
119             print(f"{letter} ", end = '')
120         print("\n")

```

0.5 run_q5.py

```

1  import numpy as np
2  import scipy.io
3  from nn import *
4  from collections import Counter
5
6  train_data = scipy.io.loadmat('../data/nist36_train.mat')
7  valid_data = scipy.io.loadmat('../data/nist36_valid.mat')
8
9  # we don't need labels now!
10 train_x = train_data['train_data']
11 valid_x = valid_data['valid_data']
12
13 max_iters = 100
14 # pick a batch size, learning rate
15 batch_size = 36
16 learning_rate = 3e-5
17 hidden_size = 32
18 lr_rate = 20
19 batches = get_random_batches(train_x, np.ones((train_x.shape[0], 1)), batch_size)
20 batch_num = len(batches)
21
22 params = Counter()
23
24 # Q5.1.1 & Q5.1.2
25 # initialize layers here
26 #####
27 ##### your code here #####
28 #####
29 # Initialize input, hidden, and output layer
30 layer_names = ['input', 'hlayer1', 'hlayer2', 'output']
31 initialize_weights(train_x.shape[1], hidden_size, params, layer_names[0])
32 initialize_weights(hidden_size, hidden_size, params, layer_names[1])
33 initialize_weights(hidden_size, hidden_size, params, layer_names[2])
34 initialize_weights(hidden_size, train_x.shape[1], params, layer_names[3])
35
36 # Initialize momentum accumulators with zeros
37 layer_names = ['input', 'hlayer1', 'hlayer2', 'output']
38 for layer in layer_names:
39     params['m_W' + layer] = np.zeros_like(params['W' + layer])
40     params['m_b' + layer] = np.zeros_like(params['b' + layer])
41
42 # Initialize training loss list
43 list_of_train_loss_per_iter = []
44
45 # should look like your previous training loops
46 for itr in range(max_iters):

```

```

47     total_loss = 0
48     for xb, _ in batches:
49         # training loop can be exactly the same as q2!
50         # your loss is now squared error
51         # delta is the d/dx of (x-y)^2
52         # to implement momentum
53         # just use 'm_'+name variables
54         # to keep a saved value over timestamps
55         # params is a Counter(), which returns a 0 if an element is missing
56         # so you should be able to write your loop without any special
57         # conditions
58
59         #####
60         ##### your code here #####
61         #####
62         # Forward propagation
63         h1 = forward(xb, params, layer_names[0], relu)
64         h2 = forward(h1, params, layer_names[1], relu)
65         h3 = forward(h2, params, layer_names[2], relu)
66         probs = forward(h3, params, layer_names[3], sigmoid)
67
68         # Compute and Update loss
69         loss = np.sum((probs - xb) ** 2)
70         total_loss += loss
71
72         # Back propagation
73         # delta1 = derivative of (x-y)^2
74         delta1 = 2 * (probs - xb)
75         delta2 = backwards(delta1, params, layer_names[3], sigmoid_deriv)
76         delta3 = backwards(delta2, params, layer_names[2], relu_deriv)
77         delta4 = backwards(delta3, params, layer_names[1], relu_deriv)
78         backwards(delta4, params, layer_names[0], relu_deriv)
79
80         # Apply gradient descent with momentum
81         for layer in layer_names:
82             # Update weights
83             params['m_W' + layer] = 0.9 * params['m_W' + layer]
84                 - learning_rate * params['grad_W' + layer]
85             params['W' + layer] += params['m_W' + layer]
86             # Update biases
87             params['m_b' + layer] = 0.9 * params['m_b' + layer]
88                 - learning_rate * params['grad_b' + layer]
89             params['b' + layer] += params['m_b' + layer]
90
91         # Update training loss
92         list_of_train_loss_per_iter.append(total_loss)
93
94     if itr % 2 == 0:

```

```

95         print("itr: {:02d} \t loss: {:.2f}".format(itr, total_loss))
96     if itr % lr_rate == lr_rate-1:
97         learning_rate *= 0.9
98
99     # Q5.2
100     import matplotlib.pyplot as plt
101     # Plot the training process
102     plt.figure()
103     plt.plot(np.arange(1, max_iters+1), list_of_train_loss_per_iter, color = 'r')
104     plt.title(f"Training Process: loss v.s. epoch")
105     plt.xlabel('epoch')
106     plt.ylabel('loss')
107     plt.show()
108
109     # Q5.3.1
110     import matplotlib.pyplot as plt
111     # visualize some results
112     #####
113     ##### your code here #####
114     #####
115     # Forward propagation for the validation data
116     h1 = forward(valid_x, params, layer_names[0], relu)
117     h2 = forward(h1, params, layer_names[1], relu)
118     h3 = forward(h2, params, layer_names[2], relu)
119     valid_probs = forward(h3, params, layer_names[3], sigmoid)
120
121     # Extract 5 classes from the total 36 classes
122     class_index = [1, 5, 10, 20, 30]
123     valid_y = valid_data['valid_labels']
124     extracted_classes, single_class = [], []
125     for index in class_index:
126         for i in range(len(valid_y)):
127             if np.argmax(valid_y, axis = 1)[i] == index:
128                 single_class.append(i)
129             extracted_classes.append(single_class)
130             single_class = []
131
132     # Visualize 2 validation and reconstructed images for each class
133     for i in range(len(extracted_classes)):
134         for j in range(2):
135             index = extracted_classes[i][j]
136             # Plot the validation and reconstructed images
137             fig, [ax1, ax2] = plt.subplots(1, 2)
138             ax1.imshow(valid_x[index].reshape(32,32).T)
139             ax1.set_title("Validation Image")
140             ax2.imshow(valid_probs[index].reshape(32,32).T)
141             ax2.set_title("Reconstructed Image")
142             plt.show()

```

```
143
144
145
146 # Q5.3.2
147 # skimage version == 0.18.1
148 from skimage.metrics import peak_signal_noise_ratio as psnr
149 # evaluate PSNR
150 #####
151 ##### your code here #####
152 #####
153 # Forward propagation for the validation data
154 h1 = forward(valid_x, params, layer_names[0], relu)
155 h2 = forward(h1, params, layer_names[1], relu)
156 h3 = forward(h2, params, layer_names[2], relu)
157 valid_probs = forward(h3, params, layer_names[3], sigmoid)
158
159 # Compute the average psnr for all validation images
160 total_psnr = 0
161 for i in range(len(valid_x)):
162     total_psnr += psnr(valid_x[i], valid_probs[i])
163 average_psnr = total_psnr / len(valid_x)
164 print(f"Average PSNR = {average_psnr}")
```

0.6 run_q6_1.py

```

1  # Import package
2  import numpy as np
3  import matplotlib.pyplot as plt
4  import scipy.io
5  import torch
6  import torch.nn as nn
7  import torch.nn.functional as func
8  import torchvision
9  import torchvision.transforms as transforms
10 import torch.optim as optim
11 from nn import get_random_batches
12
13 # Load the NIST36 dataset
14 train_data = scipy.io.loadmat('../data/nist36_train.mat')
15 valid_data = scipy.io.loadmat('../data/nist36_valid.mat')
16 test_data = scipy.io.loadmat('../data/nist36_test.mat')
17 train_x, train_y = train_data['train_data'], train_data['train_labels']
18 valid_x, valid_y = valid_data['valid_data'], valid_data['valid_labels']
19 test_x, test_y = test_data['test_data'], test_data['test_labels']
20
21 # Turn the dataset (numpy) into tensor
22 train_x, train_y = torch.from_numpy(train_x).float(), torch.from_numpy(train_y)
23 valid_x, valid_y = torch.from_numpy(valid_x).float(), torch.from_numpy(valid_y)
24 test_x, test_y = torch.from_numpy(test_x).float(), torch.from_numpy(test_y)
25
26 # Q6.1.1 Fully-connected Neural Network
27 # Set up the hyperparameters
28 max_iters = 100
29 batch_size = 72
30 learning_rate = 0.1
31 hidden_size = 64
32 batches = get_random_batches(train_x, train_y, batch_size)
33 batch_num = len(batches)
34
35 # Build the fully-connected network
36 class FullyConnectedNetwork(nn.Module):
37     def __init__(self):
38         super(FullyConnectedNetwork, self).__init__()
39         self.fc_layer_set = nn.Sequential(
40             nn.Linear(train_x.shape[1], hidden_size),
41             nn.Sigmoid(),
42             nn.Linear(hidden_size, train_y.shape[1]),
43         )
44
45     def forward(self, x):
46         # Fully-connected layer

```

```

47         x = self.fc_layer_set(x)
48         return x
49
50     # Train the fully-connected layer
51     # Load the model and Set up the criterion and optimizer
52     fcn_model = FullyConnectedNetwork()
53     criterion = nn.CrossEntropyLoss()
54     optimizer = optim.SGD(fcn_model.parameters(), lr = learning_rate)
55
56     # Initialize training/validation loss and accuracy list
57     list_of_train_loss_per_iter = []
58     list_of_train_acc_per_iter = []
59     list_of_valid_loss_per_iter = []
60     list_of_valid_acc_per_iter = []
61
62     # Train the model
63     for iter in range(max_iters):
64         total_train_loss = 0
65         total_train_acc = 0
66         for xb, yb in batches:
67             # Turn yb into labelb
68             labelb = torch.argmax(yb, dim = 1)
69
70             # Forward propagation
71             optimizer.zero_grad()
72             probs = fcn_model(xb)
73
74             # Back propagation + Optimize
75             loss = criterion(probs, labelb)
76             loss.backward()
77             optimizer.step()
78
79             # Compute loss and accuracy
80             total_train_loss += loss.item()
81             _, pred_label = torch.max(probs.data, 1)
82             acc = torch.eq(pred_label, labelb).float().sum()
83             total_train_acc += acc.item()
84
85         # Average and Update training loss and accuracy
86         average_train_loss = total_train_loss / batch_num
87         average_train_acc = total_train_acc / train_x.shape[0]
88         list_of_train_loss_per_iter.append(average_train_loss)
89         list_of_train_acc_per_iter.append(average_train_acc)
90
91     # Validation process
92     # Turn valid_y into valid_label
93     valid_label = torch.argmax(valid_y, dim = 1)
94

```



```

95     # Forward propagation
96     valid_probs = fcn_model(valid_x)
97
98     # Compute loss and accuracy
99     valid_loss = criterion(valid_probs, valid_label)
100     _, pred_valid_label = torch.max(valid_probs.data, 1)
101     valid_acc = torch.eq(pred_valid_label, valid_label).float().mean()
102
103     # Update validation loss and accuracy
104     list_of_valid_loss_per_iter.append(valid_loss)
105     list_of_valid_acc_per_iter.append(valid_acc)
106
107     # Print the training and validation process
108     print("iter: {:02d}".format(iter+1))
109     print("train loss: {:.2f} \t train acc : {:.2f}".format(average_train_loss,
110                                                         average_train_acc))
111     print("validation loss : {:.2f} \t validation acc : {:.2f}"
112           .format(valid_loss, valid_acc))
113
114     # Test the model with the test dataset
115     # Turn test_y into test_label
116     test_label = torch.argmax(test_y, dim = 1)
117     # Forward propagation
118     test_probs = fcn_model(test_x)
119     # Compute loss and accuracy
120     test_loss = criterion(test_probs, test_label)
121     _, pred_test_label = torch.max(test_probs.data, 1)
122     test_acc = torch.eq(pred_test_label, test_label).float().mean()
123     print("Test\ntest loss : {:.2f} \t test accuracy : {:.2f}".format(test_loss,
124                                                                     test_acc))
125
126     # Plot the training and validation process (loss)
127     plt.figure()
128     plt.plot(np.arange(1, max_iters+1), list_of_train_loss_per_iter, color = 'r',
129                                                    label = 'training')
130     plt.plot(np.arange(1, max_iters+1), list_of_valid_loss_per_iter, color = 'b',
131                                                    label = 'validation')
132     plt.title(f"Training and Validation Process: loss v.s. epoch")
133     plt.xlabel('epoch')
134     plt.ylabel('loss')
135     plt.legend()
136     plt.show()
137
138     # Plot the training and validation process (accuracy)
139     plt.figure()
140     plt.plot(np.arange(1, max_iters+1), list_of_train_acc_per_iter, color = 'r',
141                                                    label = 'training')
142     plt.plot(np.arange(1, max_iters+1), list_of_valid_acc_per_iter, color = 'b',

```

```

143                                     label = 'validation')
144 plt.title(f"Training and Validation Process: accuracy v.s. epoch")
145 plt.xlabel('epoch')
146 plt.ylabel('accuracy')
147 plt.legend()
148 plt.show()
149
150 # Q6.1.2 Convolutional Neural Network
151 # Set up the hyperparameters
152 max_iters = 10
153 batch_size = 72
154 learning_rate = 4e-3
155 train_batches = get_random_batches(train_x, train_y, batch_size)
156 train_batch_num = len(train_batches)
157 valid_batches = get_random_batches(valid_x, valid_y, batch_size)
158 valid_batch_num = len(valid_batches)
159
160 # Build the CNN
161 class CNN(nn.Module):
162     def __init__(self):
163         super(CNN, self).__init__()
164         # Convolutional block
165         self.conv_layer_set = nn.Sequential(
166             nn.Conv2d(in_channels = 1, out_channels = 64, kernel_size = 3,
167                       padding = 1),
168             nn.BatchNorm2d(64),
169             nn.ReLU(inplace = True),
170             nn.MaxPool2d(kernel_size = 2, stride = 2),
171             nn.Dropout(p = 0.1),
172             nn.Conv2d(in_channels = 64, out_channels = 32, kernel_size = 3,
173                       padding = 1),
174             nn.BatchNorm2d(32),
175             nn.ReLU(inplace = True),
176             nn.MaxPool2d(kernel_size = 2, stride = 2),
177             nn.Dropout(p = 0.1)
178         )
179         # Fully-connected block
180         self.fc_layer_set = nn.Sequential(
181             nn.Linear(32 * 8 * 8, 1024),
182             nn.ReLU(inplace = True),
183             nn.Linear(1024, 36),
184         )
185
186     def forward(self, x):
187         # Convolution layer block
188         x = self.conv_layer_set(x)
189         x = x.view(-1, 32 * 8 * 8)
190         # Fully-connected layer block

```

```

191         x = self.fc_layer_set(x)
192         return x
193
194     # Train the CNN
195     # Choose the device to use
196     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
197
198     # Load the model and Set up the criterion and optimizer
199     cnn_model = CNN().to(device)
200     criterion = nn.CrossEntropyLoss()
201     optimizer = optim.Adam(cnn_model.parameters(), lr = learning_rate)
202
203     # Initialize training/validation loss and accuracy list
204     list_of_train_loss_per_iter = []
205     list_of_train_acc_per_iter = []
206     list_of_valid_loss_per_iter = []
207     list_of_valid_acc_per_iter = []
208
209     # Train the model
210     for iter in range(max_iters):
211         total_train_loss = 0
212         total_train_acc = 0
213         total_valid_loss = 0
214         total_valid_acc = 0
215         for xb, yb in train_batches:
216             # Reshape xb and Turn yb into labelb
217             xb = xb.reshape(batch_size, 1, 32, 32).to(device)
218             labelb = torch.argmax(yb, dim = 1).to(device)
219
220             # Forward propagation
221             optimizer.zero_grad()
222             probs = cnn_model(xb)
223
224             # Back propagation + Optimize
225             loss = criterion(probs, labelb)
226             loss.backward()
227             optimizer.step()
228
229             # Compute loss and accuracy
230             total_train_loss += loss.item()
231             _, pred_label = torch.max(probs.data, 1)
232             acc = torch.eq(pred_label, labelb).float().sum()
233             total_train_acc += acc.item()
234
235         # Average and Update training loss and accuracy
236         average_train_loss = total_train_loss / train_batch_num
237         average_train_acc = total_train_acc / train_x.shape[0]
238         list_of_train_loss_per_iter.append(average_train_loss)

```

[illegible]

```

287
288 # Plot the training and validation process (loss)
289 plt.figure()
290 plt.plot(np.arange(1, max_iters+1), list_of_train_loss_per_iter, color = 'r',
291          label = 'training')
292 plt.plot(np.arange(1, max_iters+1), list_of_valid_loss_per_iter, color = 'b',
293          label = 'validation')
294 plt.title(f"Training and Validation Process: loss v.s. epoch")
295 plt.xlabel('epoch')
296 plt.ylabel('loss')
297 plt.legend()
298 plt.show()
299
300 # Plot the training and validation process (accuracy)
301 plt.figure()
302 plt.plot(np.arange(1, max_iters+1), list_of_train_acc_per_iter, color = 'r',
303          label = 'training')
304 plt.plot(np.arange(1, max_iters+1), list_of_valid_acc_per_iter, color = 'b',
305          label = 'validation')
306 plt.title(f"Training and Validation Process: accuracy v.s. epoch")
307 plt.xlabel('epoch')
308 plt.ylabel('accuracy')
309 plt.legend()
310 plt.show()
311
312 # Q6.1.3 Convolutional Neural Network on CIFAR-10 dataset
313 # Set up the hyperparameters
314 max_iters = 30
315 batch_size = 200
316 learning_rate = 0.001
317
318 # Load the training and testing dataset
319 # Set up the transform for training and testing dataset
320 transform = transforms.Compose([transforms.ToTensor(),
321                                transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))])
322 # Load the dataset
323 train_set = torchvision.datasets.CIFAR10(root = './data', train = True,
324                                          download = True, transform = transform)
325 train_loader = torch.utils.data.DataLoader(train_set, batch_size = batch_size,
326                                           shuffle = True)
327 test_set = torchvision.datasets.CIFAR10(root = './data', train = False,
328                                          download = True, transform = transform)
329 test_loader = torch.utils.data.DataLoader(test_set, batch_size = batch_size,
330                                           shuffle = False)
331
332 # Build the CNN
333 class CNN_CIFAR10(nn.Module):
334     def __init__(self):

```

```

335     super(CNN_CIFAR10, self).__init__()
336     # Convolutional block
337     self.conv_layer_set = nn.Sequential(
338         nn.Conv2d(in_channels = 3, out_channels = 128, kernel_size = 3,
339                                     padding = 1),
340         nn.BatchNorm2d(128),
341         nn.ReLU(inplace = True),
342         nn.MaxPool2d(kernel_size = 2, stride = 2),
343         nn.Conv2d(in_channels = 128, out_channels = 256, kernel_size = 3,
344                                     padding = 1),
345         nn.BatchNorm2d(256),
346         nn.ReLU(inplace = True),
347         nn.MaxPool2d(kernel_size = 2, stride = 2),
348         nn.Dropout(p = 0.1)
349     )
350     # Fully-connected block
351     self.fc_layer_set = nn.Sequential(
352         nn.Linear(256 * 8 * 8, 1024),
353         nn.ReLU(inplace = True),
354         nn.Linear(1024, 512),
355         nn.ReLU(inplace = True),
356         nn.Linear(512, 10),
357     )
358
359     def forward(self, x):
360         # Convolution layer block
361         x = self.conv_layer_set(x)
362         x = x.view(-1, 256 * 8 * 8)
363         # Fully-connected layer block
364         x = self.fc_layer_set(x)
365         return x
366
367     # Train the CNN
368     # Choose the device to use
369     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
370
371     # Load the model and Set up the criterion and optimizer
372     cnn_cifar10_model = CNN_CIFAR10().to(device)
373     criterion = nn.CrossEntropyLoss()
374     optimizer = optim.Adam(cnn_cifar10_model.parameters(), lr = learning_rate)
375
376     # Initialize training/validation loss and accuracy list
377     list_of_train_loss_per_iter = []
378     list_of_train_acc_per_iter = []
379     list_of_valid_loss_per_iter = []
380     list_of_valid_acc_per_iter = []
381
382     # Train the model

```

```

383 for iter in range(max_iters):
384     total_train_loss = 0
385     total_train_acc = 0
386     total_valid_loss = 0
387     total_valid_acc = 0
388     for train_data in train_loader:
389         # Extract xb and labelb
390         xb, labelb = train_data
391         xb, labelb = xb.to(device), labelb.to(device)
392
393         # Forward propagation
394         optimizer.zero_grad()
395         probs = cnn_cifar10_model(xb)
396
397         # Back propagation + Optimize
398         loss = criterion(probs, labelb)
399         loss.backward()
400         optimizer.step()
401
402         # Compute loss and accuracy
403         total_train_loss += loss.item()
404         _, pred_label = torch.max(probs.data, 1)
405         acc = torch.eq(pred_label, labelb).float().sum()
406         total_train_acc += acc.item()
407
408         # Average and Update training loss and accuracy
409         average_train_loss = total_train_loss / len(train_loader)
410         average_train_acc = total_train_acc / len(train_set)
411         list_of_train_loss_per_iter.append(average_train_loss)
412         list_of_train_acc_per_iter.append(average_train_acc)
413
414         # Print the training process
415         print("iter: {}".format(iter+1))
416         print("train loss: {:.2f} \t train acc : {:.2f}".format(average_train_loss,
417                                                                 average_train_acc))
418
419 # Test the model with the test dataset
420 test_loss = 0
421 test_acc = 0
422 for test_data in test_loader:
423     # Extract test_xb and test_labelb
424     test_xb, test_labelb = test_data
425     test_xb, test_labelb = test_xb.to(device), test_labelb.to(device)
426
427     # Forward propagation
428     optimizer.zero_grad()
429     test_probs = cnn_cifar10_model(test_xb)
430

```

[illegible]


```

479 # Build the CNN
480 class CNN_SUN(nn.Module):
481     def __init__(self):
482         super(CNN_SUN, self).__init__()
483         # Convolutional block
484         self.conv_layer_set = nn.Sequential(
485             nn.Conv2d(in_channels = 3, out_channels = 64, kernel_size = 3,
486                       padding = 1),
487             nn.BatchNorm2d(64),
488             nn.ReLU(inplace = True),
489             nn.MaxPool2d(kernel_size = 2, stride = 2),
490             nn.Conv2d(in_channels = 64, out_channels = 128, kernel_size = 3,
491                       padding = 1),
492             nn.BatchNorm2d(128),
493             nn.ReLU(inplace = True),
494             nn.MaxPool2d(kernel_size = 2, stride = 2),
495             nn.Dropout(p = 0.1)
496         )
497         # Fully-connected block
498         self.fc_layer_set = nn.Sequential(
499             nn.Linear(128 * 8 * 8, 1024),
500             nn.ReLU(inplace = True),
501             nn.Linear(1024, 512),
502             nn.ReLU(inplace = True),
503             nn.Linear(512, 10),
504         )
505
506     def forward(self, x):
507         # Convolution layer block
508         x = self.conv_layer_set(x)
509         x = x.view(-1, 128 * 8 * 8)
510         # Fully-connected layer block
511         x = self.fc_layer_set(x)
512         return x
513
514 # Train the CNN
515 # Choose the device to use
516 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
517
518 # Load the model and Set up the criterion and optimizer
519 cnn_sun_model = CNN_SUN().to(device)
520 criterion = nn.CrossEntropyLoss()
521 optimizer = optim.Adam(cnn_sun_model.parameters(), lr = learning_rate)
522
523 # Initialize training/validation loss and accuracy list
524 list_of_train_loss_per_iter = []
525 list_of_train_acc_per_iter = []
526 list_of_valid_loss_per_iter = []

```

```

527 list_of_valid_acc_per_iter = []
528
529 # Train the model
530 for iter in range(max_iters):
531     total_train_loss = 0
532     total_train_acc = 0
533     total_valid_loss = 0
534     total_valid_acc = 0
535     for train_data in train_loader:
536         # Extract xb and labelb
537         xb, labelb = train_data
538         xb, labelb = xb.to(device), labelb.to(device)
539
540         # Forward propagation
541         optimizer.zero_grad()
542         probs = cnn_sun_model(xb)
543
544         # Back propagation + Optimize
545         loss = criterion(probs, labelb)
546         loss.backward()
547         optimizer.step()
548
549         # Compute loss and accuracy
550         total_train_loss += loss.item()
551         _, pred_label = torch.max(probs.data, 1)
552         acc = torch.eq(pred_label, labelb).float().sum()
553         total_train_acc += acc.item()
554
555         # Average and Update training loss and accuracy
556         average_train_loss = total_train_loss / len(train_loader)
557         average_train_acc = total_train_acc / len(train_set)
558         list_of_train_loss_per_iter.append(average_train_loss)
559         list_of_train_acc_per_iter.append(average_train_acc)
560
561         # Print the training process
562         print("iter: {:02d}".format(iter+1))
563         print("train loss: {:.2f} \t train acc : {:.2f}".format(average_train_loss,
564                                                                 average_train_acc))
565
566 # Test the model with the test dataset
567 test_loss = 0
568 test_acc = 0
569 for test_data in test_loader:
570     # Extract test_xb and test_labelb
571     test_xb, test_labelb = test_data
572     test_xb, test_labelb = test_xb.to(device), test_labelb.to(device)
573
574     # Forward propagation

```

```
575         optimizer.zero_grad()
576         test_probs = cnn_sun_model(test_xb)
577
578         # Compute and Update loss and accuracy
579         loss = criterion(test_probs, test_labelb)
580         test_loss += loss.item()
581         _, pred_test_label = torch.max(test_probs.data, 1)
582         acc = torch.eq(pred_test_label, test_labelb).float().sum()
583         test_acc += acc.item()
584
585         # Print the test loss and accuracy
586         test_loss = test_loss / len(test_loader)
587         test_acc = test_acc / len(test_set)
588         print("Test\ntest loss : {:.2f} \t test accuracy : {:.2f}".format(test_loss,
589                                                                           test_acc))
590
591         # Plot the training process (loss)
592         plt.figure()
593         plt.plot(np.arange(1, max_iters+1), list_of_train_loss_per_iter, color = 'r')
594         plt.title(f"Training Process: loss v.s. epoch")
595         plt.xlabel('epoch')
596         plt.ylabel('loss')
597         plt.show()
598
599         # Plot the training and validation process (accuracy)
600         plt.figure()
601         plt.plot(np.arange(1, max_iters+1), list_of_train_acc_per_iter, color = 'r')
602         plt.title(f"Training Process: accuracy v.s. epoch")
603         plt.xlabel('epoch')
604         plt.ylabel('accuracy')
605         plt.show()
```

0.7 run_q6_2.py

```

1  # Import package
2  import numpy as np
3  import matplotlib.pyplot as plt
4  import scipy.io
5  import torch
6  import torch.nn as nn
7  import torch.nn.functional as func
8  import torchvision
9  import torchvision.transforms as transforms
10 import torch.optim as optim
11
12
13 # Fine-tune a single layer classifier with SqueezeNet
14 # Set up the hyperparameters
15 batch_size = 32
16 max_iters1 = 20
17 max_iters2 = 20
18 learning_rate1 = 0.001
19 learning_rate2 = 0.0001
20
21 # Load flowers17 dataset
22 # For fine tuning
23 finetune_transform = transforms.Compose([transforms.Resize(256),
24     transforms.CenterCrop(224),
25     transforms.ToTensor(),
26     transforms.Normalize((0.425, 0.425, 0.425), (0.225, 0.225, 0.225))])
27 finetune_train_set = torchvision.datasets.ImageFolder(root =
28     '../data/oxford-flowers17/train', transform = finetune_transform)
29 finetune_test_set = torchvision.datasets.ImageFolder(root =
30     '../data/oxford-flowers17/test', transform = finetune_transform)
31 # Load the dataset into DataLoader
32 finetune_train_loader = torch.utils.data.DataLoader(finetune_train_set,
33     batch_size = batch_size, shuffle = True)
34 finetune_test_loader = torch.utils.data.DataLoader(finetune_test_set,
35     batch_size = batch_size, shuffle = False)
36
37 # Load the SqueezeNet
38 squeeze_model = torchvision.models.squeezenet1_1(pretrained = True)
39 # Replace the classifier layer into 102 classes
40 squeeze_model.classifier[1] = nn.Conv2d(in_channels = 512, out_channels = 102,
41     kernel_size = 1)
42
43 # Train the SqueezeNet : 2 Steps
44 # Choose the device to use
45 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
46

```

```

47 # Load the SqueezeNet
48 squeeze_model = torchvision.models.squeezenet1_1(pretrained = True).to(device)
49 # Replace the classifier layer into 102 classes
50 squeeze_model.classifier[1] = nn.Conv2d(in_channels = 512, out_channels = 102,
51                                         kernel_size = 1).to(device)
52
53 # 1st Step : train the single classifier layer
54 for param in squeeze_model.parameters():
55     param.requires_grad = False
56 for param in squeeze_model.classifier.parameters():
57     param.requires_grad = True
58
59 # Set up the criterion and optimizer
60 criterion = nn.CrossEntropyLoss()
61 optimizer = optim.Adam(squeeze_model.classifier.parameters(), lr =
62                         learning_rate1)
63
64 # Initialize training loss and accuracy list
65 list_of_train_loss_per_iter = []
66 list_of_train_acc_per_iter = []
67
68 # Train the classifier layer
69 print("Train the single layer classifier")
70 for iter in range(max_iters1):
71     total_train_loss = 0
72     total_train_acc = 0
73     for train_data in finetune_train_loader:
74         # Extract xb and labelb
75         xb, labelb = train_data
76         xb, labelb = xb.to(device), labelb.to(device)
77
78         # Forward propagation
79         optimizer.zero_grad()
80         probs = squeeze_model(xb)
81
82         # Back propagation + Optimize
83         loss = criterion(probs, labelb)
84         loss.backward()
85         optimizer.step()
86
87         # Compute loss and accuracy
88         total_train_loss += loss.item()
89         _, pred_label = torch.max(probs.data, 1)
90         acc = torch.eq(pred_label, labelb).float().sum()
91         total_train_acc += acc.item()
92
93 # Average and Update training loss and accuracy
94 average_train_loss = total_train_loss / len(finetune_train_loader)

```

```

95     average_train_acc = total_train_acc / len(finetune_train_set)
96     list_of_train_loss_per_iter.append(average_train_loss)
97     list_of_train_acc_per_iter.append(average_train_acc)
98
99     # Print the training process
100    print("iter: {:02d}".format(iter+1))
101    print("train loss: {:.2f} \t train acc : {:.2f}".format(average_train_loss,
102                                                         average_train_acc))
103
104    # 2nd Step : finetune the SqueezeNet
105    for param in squeeze_model.parameters():
106        param.requires_grad = True
107
108    # Update the optimizer
109    optimizer = optim.Adam(squeeze_model.parameters(), lr = learning_rate2)
110
111    # Train the SqueezeNet
112    print("Train the SqueezeNet")
113    for iter in range(max_iters2):
114        total_train_loss = 0
115        total_train_acc = 0
116        for train_data in finetune_train_loader:
117            # Extract xb and labelb
118            xb, labelb = train_data
119            xb, labelb = xb.to(device), labelb.to(device)
120
121            # Forward propagation
122            optimizer.zero_grad()
123            probs = squeeze_model(xb)
124
125            # Back propagation + Optimize
126            loss = criterion(probs, labelb)
127            loss.backward()
128            optimizer.step()
129
130            # Compute loss and accuracy
131            total_train_loss += loss.item()
132            _, pred_label = torch.max(probs.data, 1)
133            acc = torch.eq(pred_label, labelb).float().sum()
134            total_train_acc += acc.item()
135
136    # Average and Update training loss and accuracy
137    average_train_loss = total_train_loss / len(finetune_train_loader)
138    average_train_acc = total_train_acc / len(finetune_train_set)
139    list_of_train_loss_per_iter.append(average_train_loss)
140    list_of_train_acc_per_iter.append(average_train_acc)
141
142    # Print the training process

```

```

143     print("iter: {:02d}".format(iter+1))
144     print("train loss: {:.2f} \t train acc : {:.2f}".format(average_train_loss,
145                                                             average_train_acc))
146
147     # Test the model with the test dataset
148     test_loss = 0
149     test_acc = 0
150     for test_data in finetune_test_loader:
151         # Extract test_xb and test_labelb
152         test_xb, test_labelb = test_data
153         test_xb, test_labelb = test_xb.to(device), test_labelb.to(device)
154
155         # Forward propagation
156         optimizer.zero_grad()
157         test_probs = squeeze_model(test_xb)
158
159         # Compute and Update loss and accuracy
160         loss = criterion(test_probs, test_labelb)
161         test_loss += loss.item()
162         _, pred_test_label = torch.max(test_probs.data, 1)
163         acc = torch.eq(pred_test_label, test_labelb).float().sum()
164         test_acc += acc.item()
165
166     # Print the test loss and accuracy
167     test_loss = test_loss / len(finetune_test_loader)
168     test_acc = test_acc / len(finetune_test_set)
169     print("Test\ntest loss : {:.2f} \t test accuracy : {:.2f}".format(test_loss,
170                                                                       test_acc))
171
172     # Plot the training process (loss)
173     plt.figure()
174     plt.plot(np.arange(1, max_iters1+max_iters2+1), list_of_train_loss_per_iter,
175                                                     color = 'r')
176     plt.title(f"Training Process: loss v.s. epoch")
177     plt.xlabel('epoch')
178     plt.ylabel('loss')
179     plt.show()
180
181     # Plot the training and validation process (accuracy)
182     plt.figure()
183     plt.plot(np.arange(1, max_iters1+max_iters2+1), list_of_train_acc_per_iter,
184                                                     color = 'r')
185     plt.title(f"Training Process: accuracy v.s. epoch")
186     plt.xlabel('epoch')
187     plt.ylabel('accuracy')
188     plt.show()
189
190     # Self-defined CNN

```

```

191 # Set up the hyperparameters
192 batch_size = 32
193 max_iters = 40
194 learning_rate = 0.001
195
196 # Load flowers17 dataset
197 # For self-defined CNN
198 transform = transforms.Compose([transforms.Resize((32, 32)),
199                                transforms.ToTensor(),
200                                transforms.Normalize((0.425, 0.425, 0.425), (0.225, 0.225, 0.225))])
201 train_set = torchvision.datasets.ImageFolder(root =
202                                               '../data/oxford-flowers17/train', transform = transform)
203 test_set = torchvision.datasets.ImageFolder(root =
204                                              '../data/oxford-flowers17/test', transform = transform)
205 # Load the dataset into DataLoader
206 train_loader = torch.utils.data.DataLoader(train_set, batch_size = batch_size,
207                                             shuffle = True)
208 test_loader = torch.utils.data.DataLoader(test_set, batch_size = batch_size,
209                                           shuffle = False)
210
211 # Build the self-defined CNN (LeNet)
212 class CNN(nn.Module):
213     def __init__(self):
214         super(CNN, self).__init__()
215         # Convolutional block : 3 Convolutional Layers
216         self.conv_layer_set = nn.Sequential(
217             nn.Conv2d(in_channels = 3, out_channels = 6, kernel_size = 5,
218                      stride = 1),
219             nn.ReLU(inplace = True),
220             nn.MaxPool2d(kernel_size = 2, stride = 2),
221             nn.Conv2d(in_channels = 6, out_channels = 16, kernel_size = 5,
222                      stride = 1),
223             nn.ReLU(inplace = True),
224             nn.MaxPool2d(kernel_size = 2, stride = 2),
225             nn.Conv2d(in_channels = 16, out_channels = 120, kernel_size = 5,
226                      stride = 1),
227             nn.ReLU(inplace = True),
228         )
229         # Fully-connected block : 2 FC Layers
230         self.fc_layer_set = nn.Sequential(
231             nn.Linear(120, 84),
232             nn.ReLU(inplace = True),
233             nn.Linear(84, 102),
234         )
235
236     def forward(self, x):
237         # Convolution layer block
238         x = self.conv_layer_set(x)

```



```
239         x = x.view(-1, 120)
240         # Fully-connected layer block
241         x = self.fc_layer_set(x)
242         return x
243
244     # Train the self-defined CNN
245     # Choose the device to use
246     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
247
248     # Load the model and Set up the criterion and optimizer
249     cnn_model = CNN().to(device)
250     criterion = nn.CrossEntropyLoss()
251     optimizer = optim.Adam(cnn_model.parameters(), lr = learning_rate)
252
253     # Initialize training loss and accuracy list
254     list_of_train_loss_per_iter = []
255     list_of_train_acc_per_iter = []
256
257     # Train the model
258     for iter in range(max_iters):
259         total_train_loss = 0
260         total_train_acc = 0
261         for train_data in train_loader:
262             # Extract xb and labelb
263             xb, labelb = train_data
264             xb, labelb = xb.to(device), labelb.to(device)
265
266             # Forward propagation
267             optimizer.zero_grad()
268             probs = cnn_model(xb)
269
270             # Back propagation + Optimize
271             loss = criterion(probs, labelb)
272             loss.backward()
273             optimizer.step()
274
275             # Compute loss and accuaracy
276             total_train_loss += loss.item()
277             _, pred_label = torch.max(probs.data, 1)
278             acc = torch.eq(pred_label, labelb).float().sum()
279             total_train_acc += acc.item()
280
281         # Average and Update training loss and accuracy
282         average_train_loss = total_train_loss / len(train_loader)
283         average_train_acc = total_train_acc / len(train_set)
284         list_of_train_loss_per_iter.append(average_train_loss)
285         list_of_train_acc_per_iter.append(average_train_acc)
286
```

```

287     # Print the training process
288     print("iter: {:02d}".format(iter+1))
289     print("train loss: {:.2f} \t train acc : {:.2f}".format(average_train_loss,
290                                                         average_train_acc))
291
292 # Test the model with the test dataset
293 test_loss = 0
294 test_acc = 0
295 for test_data in test_loader:
296     # Extract test_xb and test_labelb
297     test_xb, test_labelb = test_data
298     test_xb, test_labelb = test_xb.to(device), test_labelb.to(device)
299
300     # Forward propagation
301     optimizer.zero_grad()
302     test_probs = cnn_model(test_xb)
303
304     # Compute and Update loss and accuracy
305     loss = criterion(test_probs, test_labelb)
306     test_loss += loss.item()
307     _, pred_test_label = torch.max(test_probs.data, 1)
308     acc = torch.eq(pred_test_label, test_labelb).float().sum()
309     test_acc += acc.item()
310
311 # Print the test loss and accuracy
312 test_loss = test_loss / len(test_loader)
313 test_acc = test_acc / len(test_set)
314 print("Test\ntest loss : {:.2f} \t test accuracy : {:.2f}".format(test_loss,
315                                                                    test_acc))
316
317 # Plot the training process (loss)
318 plt.figure()
319 plt.plot(np.arange(1, max_iters+1), list_of_train_loss_per_iter, color = 'r')
320 plt.title(f"Training Process: loss v.s. epoch")
321 plt.xlabel('epoch')
322 plt.ylabel('loss')
323 plt.show()
324
325 # Plot the training and validation process (accuracy)
326 plt.figure()
327 plt.plot(np.arange(1, max_iters+1), list_of_train_acc_per_iter, color = 'r')
328 plt.title(f"Training Process: accuracy v.s. epoch")
329 plt.xlabel('epoch')
330 plt.ylabel('accuracy')
331 plt.show()

```

0.8 util.py

```
1 import numpy as np
2
3 # use for a "no activation" layer
4 def linear(x):
5     return x
6
7 def linear_deriv(post_act):
8     return np.ones_like(post_act)
9
10 def tanh(x):
11     return np.tanh(x)
12
13 def tanh_deriv(post_act):
14     return 1-post_act**2
15
16 def relu(x):
17     return np.maximum(x,0)
18
19 def relu_deriv(x):
20     return (x > 0).astype(np.float)
```