

1.1 Theory

Softmax is defined as below, for each index i in a vector x:

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

If we add in a translation term c, softmax is defined as below:

$$\text{Softmax}(x_i + c) = \frac{e^{x_i+c}}{\sum_j e^{x_j+c}}$$

This can be simplified as shown:

$$\text{Softmax}(x_i + c) = \frac{e^{x_i} e^c}{e^c \sum_j e^{x_j}}$$

$$\text{Softmax}(x_i + c) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Therefore, $\text{Softmax}(x_i) = \text{Softmax}(x_i + c)$ so softmax is invariant to translation.

When $c = 0$, the numerator values will be in the range $(0, \infty)$. When $c = -\max x_i$, the numerator values will be in the range $(0, 1]$. Having all the values scaled between 0 and 1 avoids large exponent terms which helps avoid overflow.

1.2 Theory

- The range of each element is $(0,1)$ for $d>2$. The sum of all elements will be 1.
- “*Softmax takes an arbitrary real valued vector x and turns it into a probability distribution with the same number of elements in x .*”
- 1. Find exponential value for each element in x
2. Total sum of all elements
3. Probability for each element in x

1.3 Theory

The x values for a network without a nonlinear activation function will change according to the linear function:

$$x_{i+1} = W_i x_i + b_i$$

When applied to neural networks:

$$\begin{aligned}y &= W_n x_n + b_n \\y &= W_n (W_{n-1} x_{n-1} + b_{n-1}) + b_n \\y &= W_n W_{n-1} x_{n-1} + W_n b_{n-1} + b_n\end{aligned}$$

Rewriting the new W as W':

$$\begin{aligned}y &= W' x + b' \\y &= W' (W^{'}_{n-2} x_{n-2} + b_{n-2}) + b' \\\vdots \\y &= W x + b\end{aligned}$$

We end up with $y = Wx + b$ which is the same as solving a linear regression problem

1.4 Theory

Starting with the sigmoid activation function:

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

We can differentiate the function with respect to x:

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

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

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

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

If we add and subtract 1 in the numerator, we can rewrite the derivative in terms of sigma(x):

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

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

$$\frac{d}{dx} \sigma(x) = \frac{1}{(1 + e^{-x})} \left(1 - \frac{1}{1 + e^{-x}} \right)$$

$$\frac{d}{dx} \sigma(x) = \sigma(x) (1 - \sigma(x))$$

1.5 Theory

We have:

$$y = Wx + b$$

$$y_j = \sum_{i=1}^d x_i W_{ij} + b_j$$

Taking the partial derivatives:

$$\frac{\partial y_j}{\partial W_{ij}} = x_i \quad \frac{\partial y_j}{\partial x_i} = W_{ij} \quad \frac{\partial y_i}{\partial b_j} = 1$$

$$\frac{\partial y}{\partial W} = \begin{bmatrix} x_1 & \dots & x_1 \\ x_2 & \dots & x_2 \\ \vdots & & \vdots \\ x_d & \dots & x_d \end{bmatrix} \rightarrow \in \mathbb{R}^{d \times k}$$

$$\frac{\partial y_j}{\partial x} = W_j \in \mathbb{R}^{d \times 1}$$

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

Through chain rule:

$$\frac{\partial J}{\partial W} = \sum_{j=1}^k \frac{\partial J}{\partial y_i} \frac{\partial y_j}{\partial W} = x \delta^T \in \mathbb{R}^{dxk}$$

$$\frac{\partial J}{\partial x} = \sum_{j=1}^k \frac{\partial J}{\partial y_j} \frac{\partial y_j}{\partial x} = W \delta \in \mathbb{R}^{d \times 1}$$

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

1.6 Theory

1. The sigmoid activation function has a derivative that is small in the region where the function is close to 0 or 1. In other words, the slope of the function is close to 0 in these regions. When the derivative is small, the gradient of the activation function is also small. If the gradient is small, then the weights in the network will not be updated significantly during training, and the network may not learn effectively. This is known as the vanishing gradient problem.
2. The output range of the tanh function is between -1 and 1, while the output range of the sigmoid function is between 0 and 1. The tanh function is often preferred over the sigmoid function because the output of the tanh function is centered at 0, which makes it easier to model outputs that are centered around 0. This can make it easier to train certain types of neural network models. Additionally, the output of the tanh function is also more symmetrical around 0 than the sigmoid function which can make it more numerically stable and easier to optimize.
3. The vanishing gradient is a problem that occurs when the derivative of a function becomes very small, resulting in a slow learning process for neural networks. This is more likely to happen when the function has a steep slope, which is the case for the sigmoid function. The tanh function on the other hand has a slope that is less steep, which means it is less likely to suffer from the vanishing gradient problem.
4. The tanh function can be written in terms of the sigmoid function as follows:

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

This is a simple scaling and shifting of the sigmoid function. Sigmoid ranges from 0 to 1, so multiplying it by 2 and then subtracting 1 shifts its range to (-1,1) from (0,1).

2.1 Network Initialization

2.1.1 Theory

It's not a good idea to initialize a network with all zeros. If all the weights are initialized with 0, then the derivative with respect to the loss function will just be the same for all values. The weights won't change between iterations. The same outputs would be generated no matter what the inputs are, which we do not want to do during training. The biases have no effect when initialized to 0. Therefore, assigning random values is better than just assigning 0s.

2.1.2 Code

```
def initialize_weights(in_size,out_size,params,name=''):
    b = np.zeros(out_size)
    W = np.random.randn(in_size,out_size) * np.sqrt(6 / (in_size+out_size))
    params['W' + name] = W
    params['b' + name] = b
```

2.1.3 Theory

Random initialization increases the chances of reaching ideal parameters and minimizing loss. It helps find the optimal weight values. Randomization also helps in breaking symmetry and ensures that every neuron is performing different computations in order to learn faster.

We scale the weights to ensure that there are not vanishing or exploding gradient problems. This way the signals during forward and backwards propagation won't expand to very large values or diminish to very small values. If that happened, there could be slow training since the gradient descent may get stuck on flat regions or saturated regions.

2.2 Forward Propagation

2.2.1 Code

```
#####
# x is a matrix
# a sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

#####
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]

    pre_act = np.matmul(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 2.2.1
# implement sigmoid
test = sigmoid(np.array([-1000, 1000]))
print('should be zero and one\n\t' + str(test.min()) + '\n\t' + str(test.max()))
# implement forward
h1 = forward(x, params, 'layer1')
print(h1.shape)
```

2.2.2 Code

```
#####
# x is [examples,classes]
# softmax should be done for each row
def softmax(x):
    x_shift = x - np.expand_dims(np.amax(x, axis=1), axis=1)
    res = np.exp(x_shift) / np.expand_dims(np.sum(np.exp(x_shift), axis=1), axis=1)
    return res
```

```
# Q 2.2.2
# 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)
```

2.2.3 Code

```
# ##### 0 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 = -np.sum(y*np.log(probs))
    probs_idxs = np.argmax(probs, axis=1)
    y_idxs = np.argmax(y, axis=1)
    correct = np.where(probs_idxs == y_idxs)
    acc = correct[0].shape[0] / y.shape[0]
    return loss, acc
```

```
# 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] or higher, and 0.25
# if it is not, check softmax!
print("{} , {:.2f} ".format(loss, acc))
```

2.3 Backwards Propagation

```
#####
# 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
    """
    grad_X, grad_W, grad_b = None, None, None
    # everything you may need for this layer
    W = params['W' + name]
    b = params['b' + name]
    X, pre_act, post_act = params['cache_' + name]

    # do the derivative through activation first
    # (don't forget activation_deriv is a function of post_act)
    # then compute the derivative W, b, and X
    #####
    ##### your code here #####
    #####
    grad = delta * activation_deriv(post_act)
    grad_W = np.matmul(X.T, grad)
    grad_b = np.matmul(grad.T, np.ones(grad.shape[0]))
    grad_X = np.matmul(grad, W.T)

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

2.4 Training Loop

```
#####
# 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 = []
    examples = x.shape[0]
    num_batches = int(examples/batch_size)

    for i in range(num_batches):
        idx = np.random.choice(examples,batch_size,False)
        xbatch = x[idx,:]
        ybatch = y[idx,:]
        batches.append((xbatch,ybatch))

    return batches

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

# WRITE A TRAINING LOOP HERE
max_iters = 500
learning_rate = 1e-3
# with default settings, you should get loss < 35 and accuracy > 75%
for itr in range(max_iters):
    total_loss = 0
    total_acc = 0
    avg_acc = 0
    for xb,yb in batches:
        # forward
        h1 = forward(xb, params, 'layer1')
        probs = forward(h1, params, 'output', softmax)

        # loss
        loss, acc = compute_loss_and_acc(yb,probs)

        # be sure to add loss and accuracy to epoch totals
        total_loss += loss
        total_acc += acc

        # backward
        delta1 = probs - yb
        delta2 = backwards(delta1, params, 'output', linear_deriv)
        delta3 = backwards(delta2, params, 'layer1', sigmoid_deriv)

        # apply gradient
        # gradients should be summed over batch samples
        params['Woutput'] -= learning_rate * params['grad_Woutput']
        params['boutput'] -= learning_rate * params['grad_boutput']
        params['Wlayer1'] -= learning_rate*params['grad_Wlayer1']
        params['blayer1'] -= learning_rate*params['grad_blayer1']

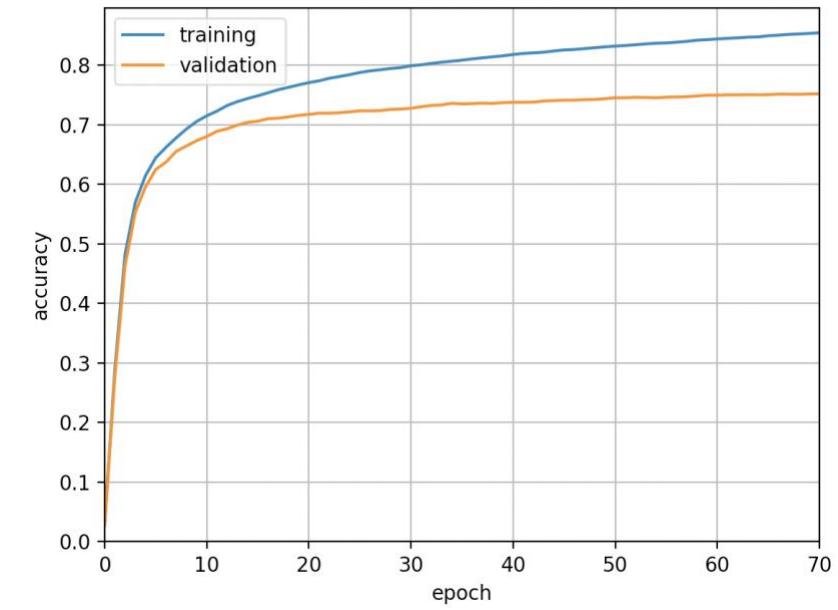
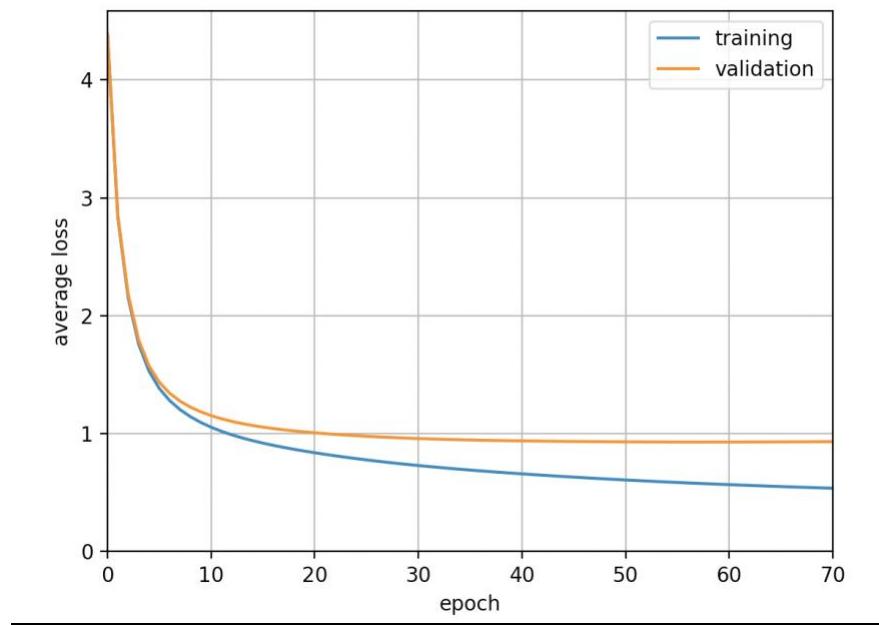
    avg_acc = total_acc/len(batches)
    if itr % 100 == 0:
        print("itr: {:02d} \t loss: {:.2f} \t acc : {:.2f}".format(itr,total_loss,avg_acc))
```

2.5 Numerical Gradient Checker

```
# Q 2.5 should be implemented in this file
# save the old params
import copy
params_orig = copy.deepcopy(params)
# compute gradients using finite difference
eps = 1e-6
for k,v in params.items():
    if '_' in k:
        continue
    if 'W' in k:
        for i in range(v.shape[0]):
            for j in range(v.shape[1]):
                v[i,j] += eps
                h1 = forward(x, params, 'layer1')
                probs = forward(h1, params, 'output', softmax)
                loss1, acc1 = compute_loss_and_acc(y, probs)
                v[i,j] -= 2*eps
                h1 = forward(x, params, 'layer1')
                probs = forward(h1, params, 'output', softmax)
                loss2, acc2 = compute_loss_and_acc(y, probs)
                params['grad_' + k][i,j] = (loss1-loss2)/(2*eps)
                v[i,j] += eps
    if 'b' in k:
        for i in range(v.shape[0]):
            v[i] += eps
            h1 = forward(x, params, 'layer1')
            probs = forward(h1, params, 'output', softmax)
            loss1, acc1 = compute_loss_and_acc(y, probs)
            v[i] -= 2*eps
            h1 = forward(x, params, 'layer1')
            probs = forward(h1, params, 'output', softmax)
            loss2, acc2 = compute_loss_and_acc(y, probs)
            params['grad_' + k][i] = (loss1-loss2)/(2*eps)
            v[i] += eps

    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))
```

3.1 Code



Epochs: 70

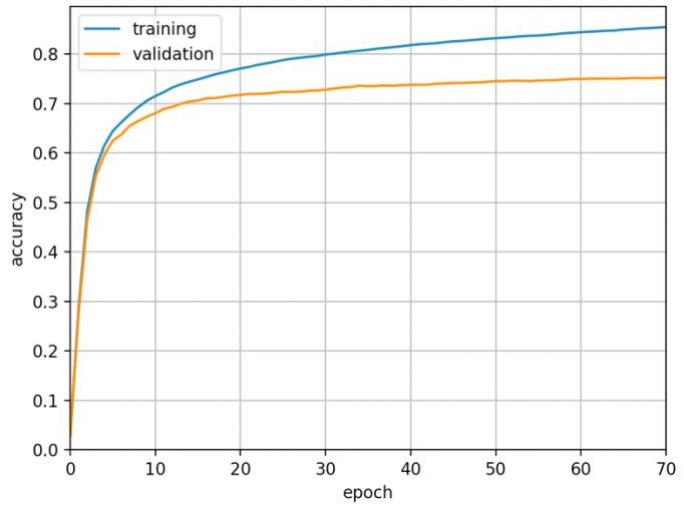
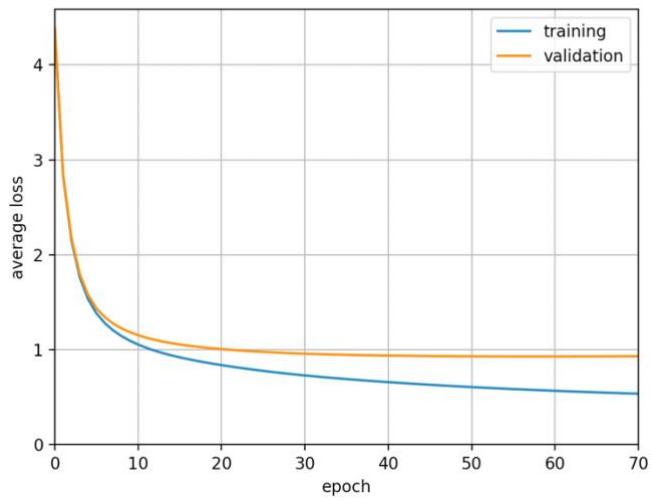
Batch Size: 32

Learning Rate: 0.003

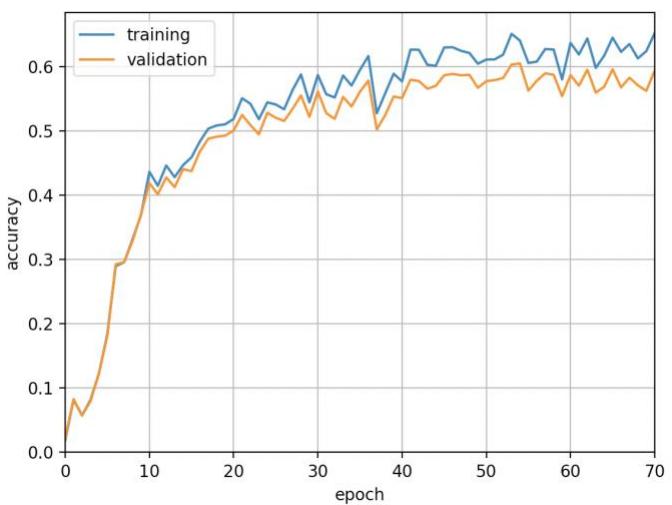
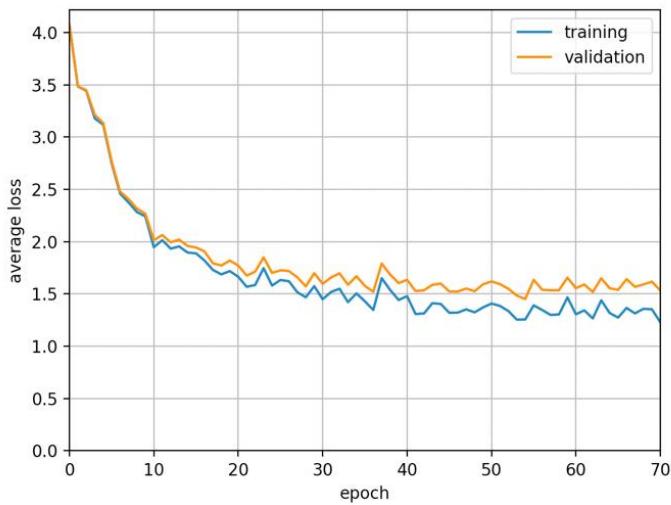
Hidden Size: 64

3.2 Writeup

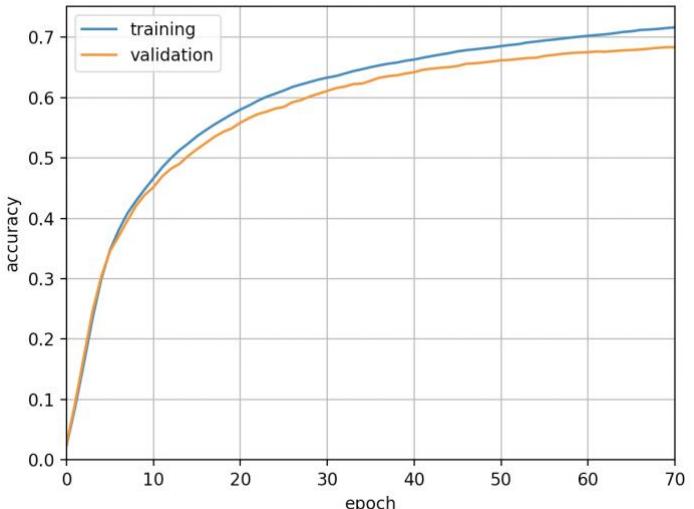
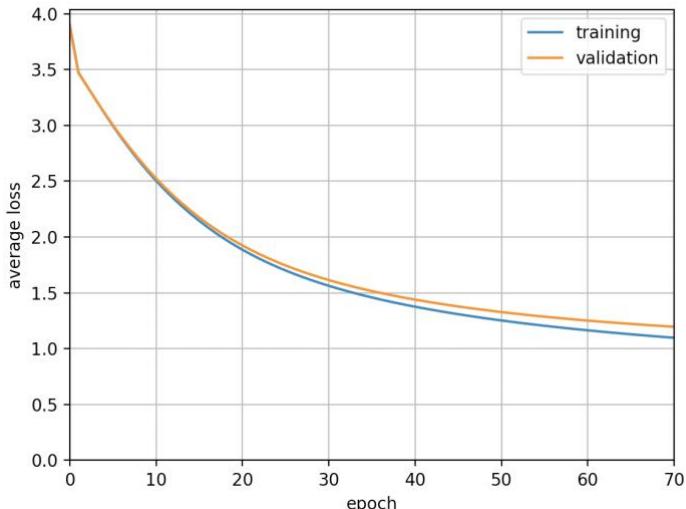
Tuned Learning Rate (from 3.2):



10x learning rate:



1/10th learning rate:

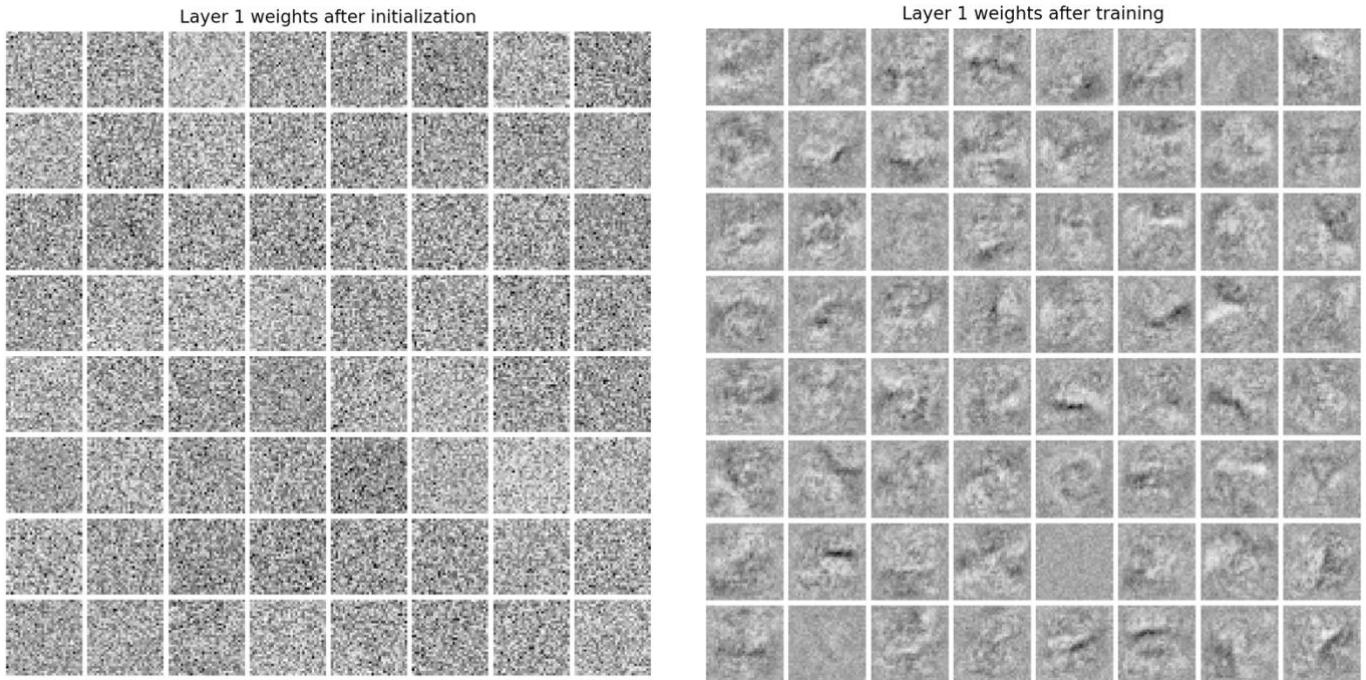


From the plots shown above, we can see that the learning rate has a direct impact on final accuracy and loss values. The images show that an optimal learning rate will gradually converge to the minimum to learn the ideal parameters in order to minimize loss and have the best accuracy. When the learning rate was too large, the plots were jagged because there was a large step size. The loss was higher and the accuracy was lower. When the learning rate is too large, it's possible that the model may not converge to the minimum value and the values will just keep bouncing around. When the learning rate was small, the plots were smoother but they still had lower accuracy and higher loss. With a very small learning rate, very small steps are made so the training is much slower and may not converge to the minimum value within the set number of epochs.

Best Final Training Accuracy = 75.4%

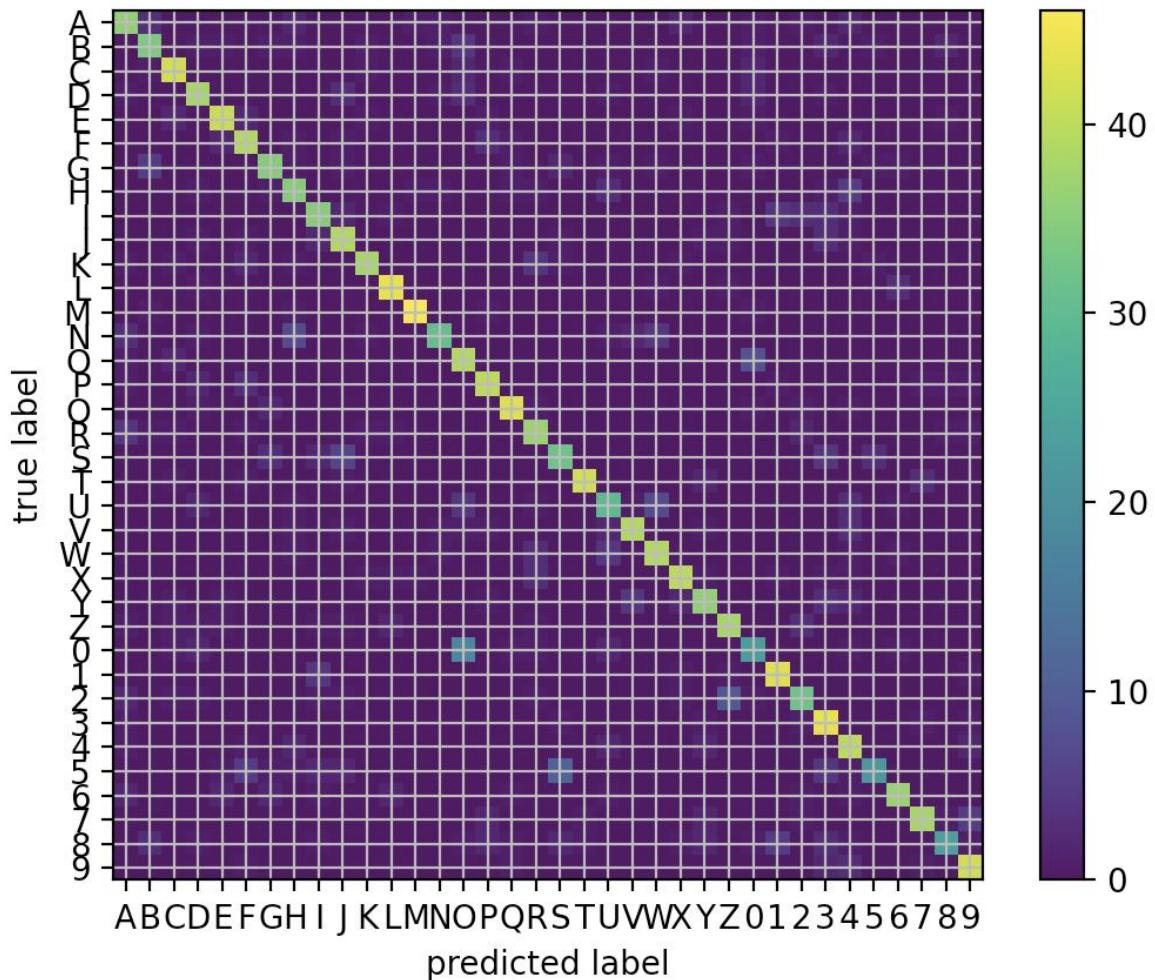
Best Final Test Accuracy = 76.3%

3.3 Writeup



Above are visualizations showing the layer 1 weights after initialization and the layer 1 weights after training. We can see that in the first image of the weights after initialization, the values random and do not show any clear pattern. The second image which shows the weights after training has distinct patterns which indicate that the weights have been updated and the network has been trained to use optimized parameters.

3.4 Writeup



Above is the visualization of the confusion matrix. We can see that the results are pretty good because of the clear diagonal line. However, there are still some predictions which were mistaken. For example, we can see that 'O' was commonly mistaken for zero, which makes sense since those characters look very similar. '5' and 'S' were also commonly mistaken which also makes sense since those are also similar characters and may be hard to distinguish based on handwriting. The model also struggled to accurately predict '8' and 'B.' These characters can often look like each other based on handwriting.

4.1 Theory

The method assumes that the letters are isolated from each other and that each letter forms a connected component. It also assumes that the letters are all similar sizes. Below are some examples of letters that may fail since they could be disconnected, connected to different letters, or of varying sizes and thicknesses.

H E L L O

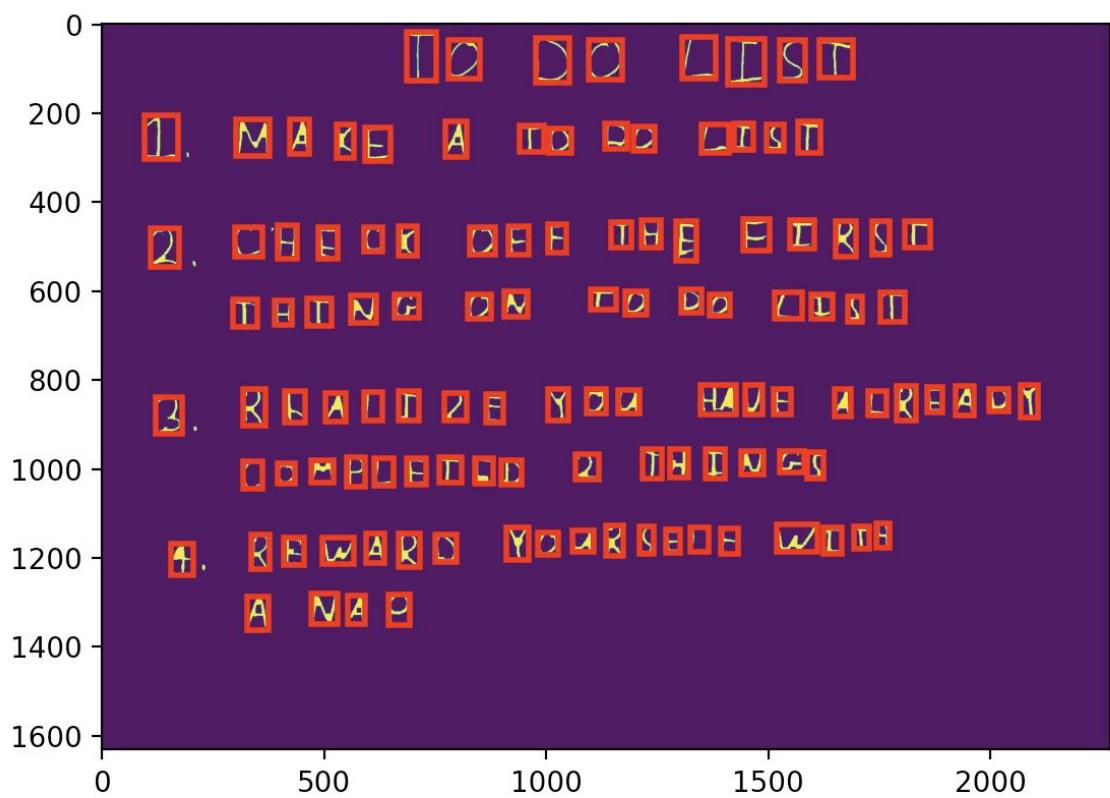
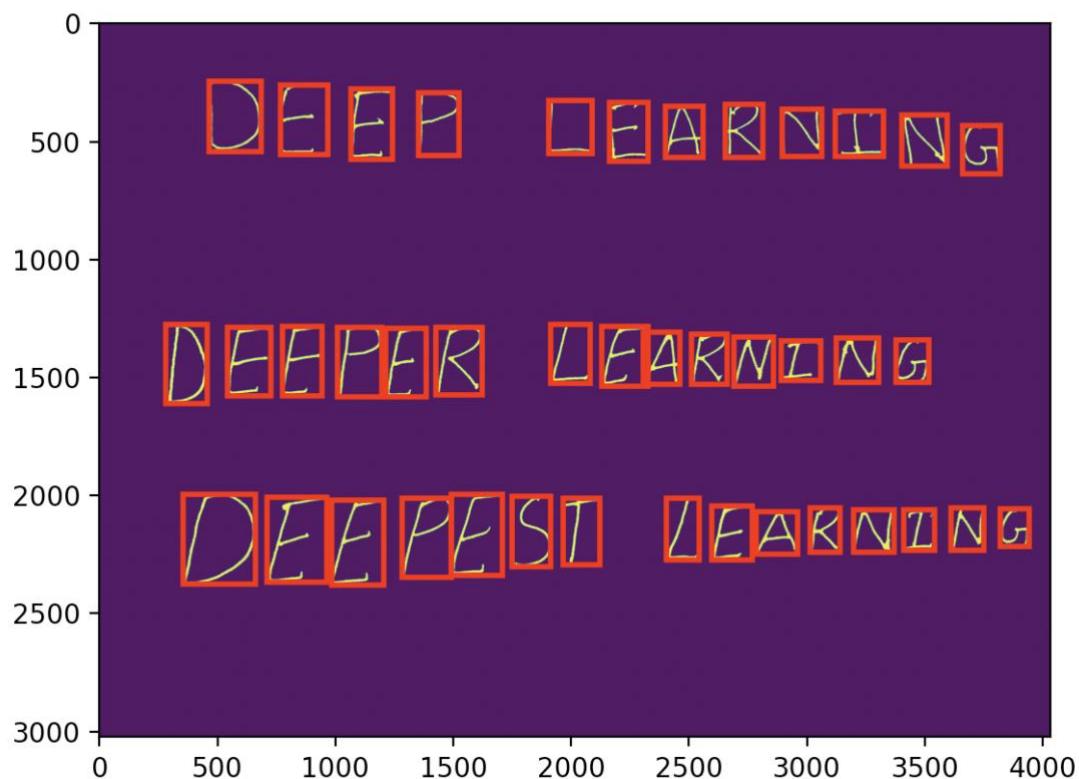
hello

He llo

4.2 Code

```
19     def findLetters(image):
20         image = skimage.color.rgb2gray(image)
21         thresh = threshold_otsu(image)
22         bw = closing(image < thresh, square(10))
23         cleared = clear_border(bw)
24         label_image = label(cleared)
25         bboxes = []
26         for region in regionprops(label_image):
27             if region.area >= 300:
28                 bbox = region.bbox
29                 bboxes.append(bbox)
30     return bboxes, bw
```

4.3 Writeup



HATKOS ARE SAEY

BUT SOMETIMES THEY DONT MAKE SENSE

REJECT GENERATOR

A B C D E F G

H I J K L M N

O P Q R S T U

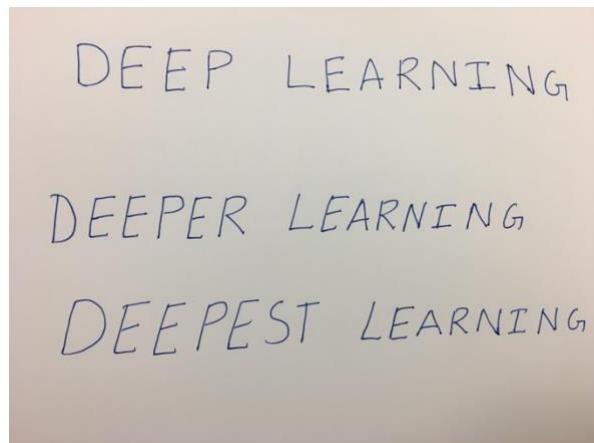
V W X Y Z

I L B F S G T

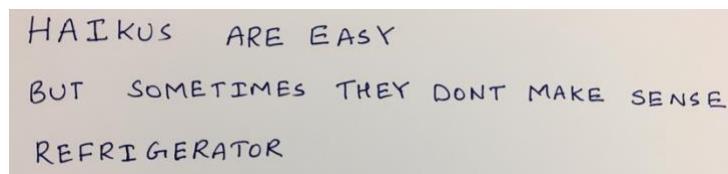
Y H O

4.4 Code / Writeup

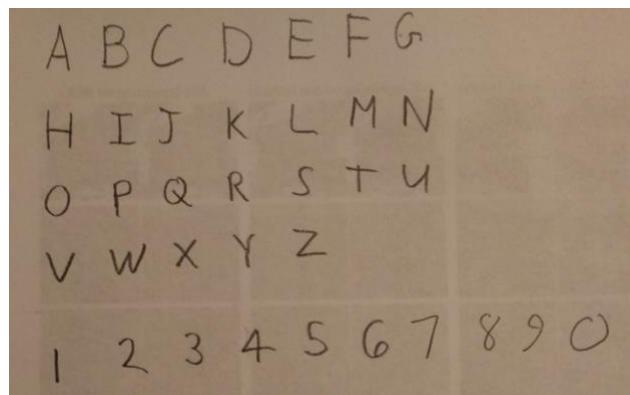
Below is the extracted text with each of the associated images



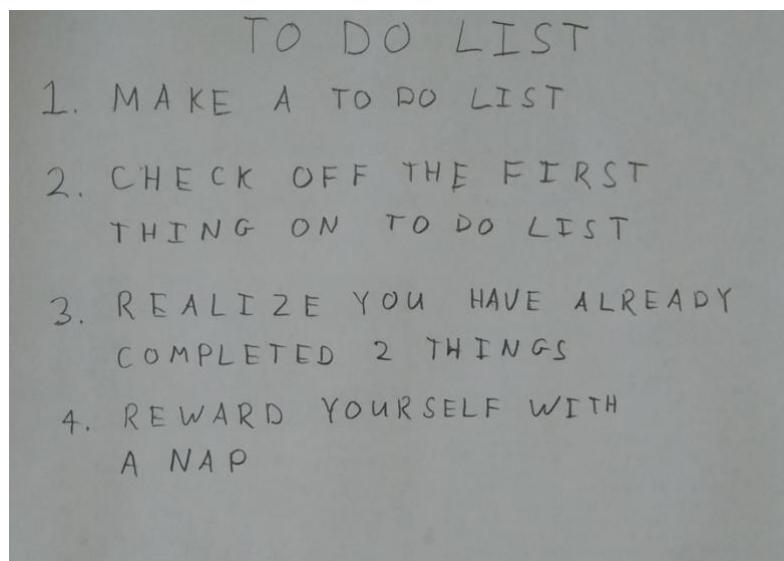
DEEPELEARMIN4
DEEPERLEARRIN6
DEEYESTLEARNING



HAIHUSAREEASX
ELTSCMETIM8STHEXDONTMAKRSEMSE
REFRI68RATOR



XBCUET6
HIIKLMH
OPRRSYH
V4XYZ
1X34SG7XYO



TODOLIST
IMAKEAY0Q06ZSY
2DH8RK0PBYH8P8RSY
YMZNA0MY08QLD8Y
DR8ALZ2RYQ8MRRALRRAMY
8QNPLQTKB2Y48NAS
4RRVARDY0RRQQLPH8YQ
ANAP

Code Snippets:

```
1      import os
2      import matplotlib.pyplot as plt
3      import matplotlib.patches
4      import pickle
5      import string
6      import cv2
7      import skimage.io
8      from q4 import *
9      import numpy as np
10     from nn import *
11
12     import warnings
13     warnings.simplefilter(action='ignore', category=FutureWarning)
14     warnings.simplefilter(action='ignore', category=UserWarning)
15
16     def get_crop(bbox, bw, image_it):
17         y1, x1, y2, x2 = bbox
18         image = bw[y1:y2, x1:x2]
19         w = x2 - x1
20         h = y2 - y1
21         if h > w:
22             padding = int(h / 2)
23             image = np.pad(image, (padding, padding), "constant")
24         elif w > h:
25             padding = int(w / 2)
26             image = np.pad(image, (padding, padding), "constant")
27             idx_0 = np.where(image == 0)
28             idx_1 = np.where(image == 1)
29             image[idx_0] = 1
30             image[idx_1] = 0
31             if image_it == 0:
32                 erosion_amt = 30
33             elif image_it == 1:
34                 erosion_amt = 17
35             elif image_it == 2:
36                 erosion_amt = 17
37             else:
38                 erosion_amt = 7
39             image = cv2.erode(image.astype(float), np.ones((erosion_amt, erosion_amt)), iterations=1)
40             image = cv2.resize(image.T, (32, 32)).flatten()
41
42     return image
```

```
41     def sort_boxes(box_arr):
42         x_arr = []
43         for i in box_arr:
44             x_arr.append(i[1])
45         x_arr.sort()
46         res = []
47         for i in x_arr:
48             for j in box_arr:
49                 if i == j[1]:
50                     res.append(j)
51
52     return res
53
54     img_count = 0
55     for img in os.listdir('../images'):
56         im1 = skimage.img_as_float(skimage.io.imread(os.path.join('../images', img)))
57         bboxes, bw = findLetters(im1)
58         plt.imshow(bw)
59         for bbox in bboxes:
60             minr, minc, maxr, maxc = bbox
61             rect = matplotlib.patches.Rectangle((minc, minr), maxc - minc, maxr - minr, fill=False, edgecolor='red', linewidth=2)
62             plt.gca().add_patch(rect)
63
64         plt.show()
65
66     # find the rows using..RANSAC, counting, clustering, etc.
67     rows = []
68     bboxes_in_row = [bboxes[0]]
69     for i in range(1, len(bboxes)):
70         bbox = bboxes[i]
71         y1, x1, y2, x2 = bbox
72         yc = (y1 + y2) / 2
73         bbox_first = bboxes_in_row[-1]
74         if yc < bbox_first[0] or yc > bbox_first[2]:
75             rows.append(bboxes_in_row)
76             bboxes_in_row = []
77         bboxes_in_row.append(bbox)
78     else:
79         bboxes_in_row.append(bbox)
80
81     rows.append(bboxes_in_row)
```

```
80
81     # crop the bounding boxes
82     # note: before you flatten, transpose the image (that's how the dataset is!)
83     # consider doing a square crop, and even using np.pad() to get your images looking more like the dataset
84     for r in range(len(rows)):
85         rows[r] = sort_boxes(rows[r])
86         for character in range(len(rows[r])):
87             rows[r][character] = get_crop(rows[r][character], bw, img_count)
88
89     # load the weights
90     # run the crops through your neural network and print them out
91     letters = np.array([
92         [_, for _ in string.ascii_uppercase[:26]] + [str(_) for _ in range(10)]])
93     params = pickle.load(open('q3_weights.pickle', 'rb'))
94     map_char = {0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F',
95                 6: 'G', 7: 'H', 8: 'I', 9: 'J', 10: 'K', 11: 'L',
96                 12: 'M', 13: 'N', 14: 'O', 15: 'P', 16: 'Q', 17: 'R',
97                 18: 'S', 19: 'T', 20: 'U', 21: 'V', 22: 'W', 23: 'X',
98                 24: 'Y', 25: 'Z', 26: '0', 27: '1', 28: '2', 29: '3',
99                 30: '4', 31: '5', 32: '6', 33: '7', 34: '8', 35: '9'}
100
101    for r in rows:
102        r = np.vstack(r)
103        h1 = forward(r, params, 'layer1')
104        probs = forward(h1, params, 'output', softmax)
105        letters_in_row = ""
106        for i in range(probs.shape[0]):
107            pred = np.argmax(probs[i, :])
108            letters_in_row += map_char[pred]
109            print(letters_in_row)
110        print("\n")
111        img_count+=1
```

6.1 Train a Neural Network in Pytorch

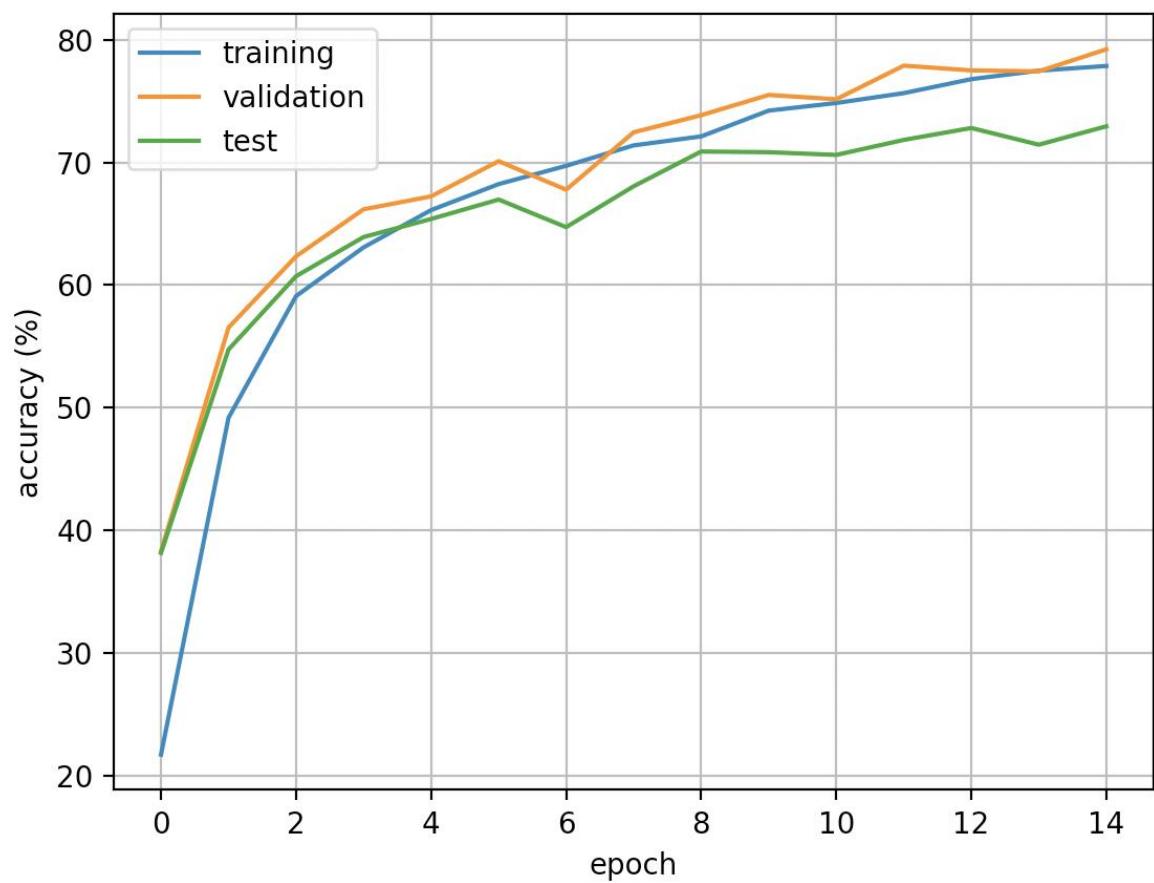
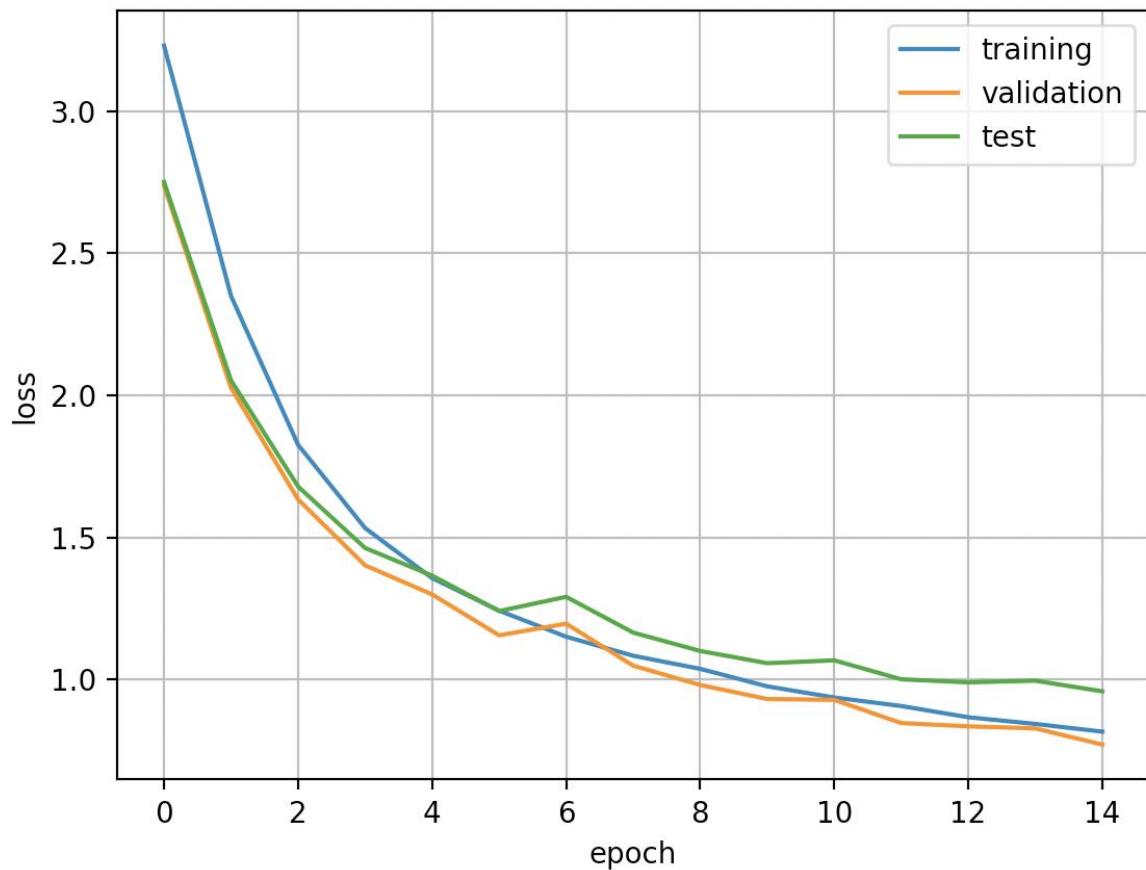
6.1.1 Code/Writeup

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import scipy.io
4
5 import torch
6 from torch import nn
7 from torch.utils.data import DataLoader, TensorDataset
8
9 train_data = scipy.io.loadmat('../data/nist36_train.mat')
10 valid_data = scipy.io.loadmat('../data/nist36_valid.mat')
11 test_data = scipy.io.loadmat('../data/nist36_test.mat')
12
13 train_x, train_y = train_data['train_data'], train_data['train_labels']
14 valid_x, valid_y = valid_data['valid_data'], valid_data['valid_labels']
15 test_x, test_y = test_data['test_data'], test_data['test_labels']
16
17
18 # convert to Tensor
19 train_x = torch.from_numpy(train_x).to(torch.float32)
20 train_y = np.argmax(train_y, axis=1)
21 train_y = torch.from_numpy(train_y)
22 train_data = TensorDataset(train_x, train_y)
23
24
25 # convert to Tensor
26 test_x = torch.from_numpy(test_x).to(torch.float32)
27 test_y = np.argmax(test_y, axis=1)
28 test_y = torch.from_numpy(test_y)
29 test_data = TensorDataset(test_x, test_y)
30
31
32 # convert to Tensor
33 valid_x = torch.from_numpy(valid_x).to(torch.float32)
34 valid_y = np.argmax(valid_y, axis=1)
35 valid_y = torch.from_numpy(valid_y)
36 valid_data = TensorDataset(valid_x, valid_y)
37
38 batch_size = 32
39
40
41 # Create data loaders
42 train_dataloader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
43 valid_dataloader = DataLoader(valid_data, batch_size=batch_size, shuffle=True)
44 test_dataloader = DataLoader(test_data, batch_size=batch_size, shuffle=True)
45
46
47 # Get cpu or gpu device for training.
48 device = "cuda" if torch.cuda.is_available() else "cpu"
49 print(f"Using {device} device")
```

```
47     # Define model
48     class NeuralNetwork(nn.Module):
49         def __init__(self):
50             super().__init__()
51             self.flatten = nn.Flatten()
52             self.linear_relu_stack = nn.Sequential(
53                 nn.Linear(1024, 64),
54                 nn.Sigmoid(),
55                 nn.Linear(64, 36))
56
57         def forward(self, x):
58             x = self.flatten(x)
59             logits = self.linear_relu_stack(x)
60             return logits
61
62     model = NeuralNetwork().to(device)
63     print(model)
64
65     loss_fn = nn.CrossEntropyLoss()
66     optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
67
68     def train(dataloader, model, loss_fn, optimizer):
69         size = len(dataloader.dataset)
70         num_batches = len(dataloader)
71         model.train()
72         train_loss = 0
73         correct = 0
74         for batch, (X, y) in enumerate(dataloader):
75             X, y = X.to(device), y.to(device)
76             # Compute prediction error
77             pred = model(X)
78             loss = loss_fn(pred, y)
79             train_loss += loss.item()
80             correct += (pred.argmax(1) == y).type(torch.float).sum().item()
81             # Backpropagation
82             optimizer.zero_grad()
83             loss.backward()
84             optimizer.step()
85
86             if batch % 100 == 0:
87                 loss, current = loss.item(), batch * len(X)
88                 print(f"loss: {loss:.7f} [{current:.5d}/{size:.5d}]")
89             correct /= size
90             train_loss /= num_batches
91             print(f"Train Accuracy: {((100 * correct):>0.1f}%)"
92         return 100*correct, train_loss
```

```
93
94
95 ► def test(dataloader, model, loss_fn):
96     size = len(dataloader.dataset)
97     num_batches = len(dataloader)
98     model.eval()
99     test_loss, correct = 0, 0
100    with torch.no_grad():
101        for X, y in dataloader:
102            X, y = X.to(device), y.to(device)
103            pred = model(X)
104            test_loss += loss_fn(pred, y).item()
105            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
106        test_loss /= num_batches
107        correct /= size
108        acc = 100*correct
109        return acc, test_loss
110    print(f"Test Error: \n Accuracy: {(acc):>0.1f}%, Avg loss: {test_loss:>8f} \n")
111
112    def validate(dataloader, model, loss_fn):
113        size = len(dataloader.dataset)
114        num_batches = len(dataloader)
115        model.eval()
116        test_loss, correct = 0, 0
117        with torch.no_grad():
118            for X, y in dataloader:
119                X, y = X.to(device), y.to(device)
120                pred = model(X)
121                test_loss += loss_fn(pred, y).item()
122                correct += (pred.argmax(1) == y).type(torch.float).sum().item()
123            test_loss /= num_batches
124            correct /= size
125            acc = 100*correct
126            return acc, test_loss
127    print(f"Validation Error: \n Accuracy: {(acc):>0.1f}%, Avg loss: {test_loss:>8f} \n")
128
```

```
130     epochs = 15
131     train_acc = []
132     train_loss = []
133     validate_acc = []
134     validate_loss = []
135     test_acc = []
136     test_loss = []
137     for t in range(epochs):
138         print(f"Epoch {t+1}\n-----")
139         tr_acc, tr_loss = train(train_dataloader, model, loss_fn, optimizer)
140         train_acc.append(tr_acc)
141         train_loss.append(tr_loss)
142         val_acc, val_loss = validate(test_dataloader, model, loss_fn)
143         validate_acc.append(val_acc)
144         validate_loss.append(val_loss)
145         te_acc, te_loss = test(valid_dataloader, model, loss_fn)
146         test_acc.append(te_acc)
147         test_loss.append(te_loss)
148     print("Done!")
149
150     # plot loss
151     plt.plot(range(len(train_loss)), train_loss, label="training")
152     plt.plot(range(len(validate_loss)), validate_loss, label="validation")
153     plt.plot(range(len(test_loss)), test_loss, label="test")
154     plt.xlabel("epoch")
155     plt.ylabel("loss")
156     plt.legend()
157     plt.grid()
158     plt.show()
159
160     # plot training accuracy
161     plt.plot(range(len(train_acc)), train_acc, label="training")
162     plt.plot(range(len(validate_acc)), validate_acc, label="validation")
163     plt.plot(range(len(test_acc)), test_acc, label="test")
164     plt.xlabel("epoch")
165     plt.ylabel("accuracy (%)")
166     plt.legend()
167     plt.grid()
168     plt.show()
```



6.1.2 Code / Writeup

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import scipy.io
4 import torch
5 import torch.nn as nn
6 import torch.nn.functional as F
7 from torch.utils.data import DataLoader, TensorDataset
8
9 # load data
10 train_data = scipy.io.loadmat('../data/nist36_train.mat')
11 valid_data = scipy.io.loadmat('../data/nist36_valid.mat')
12 test_data = scipy.io.loadmat('../data/nist36_test.mat')
13 train_x, train_y = train_data['train_data'], train_data['train_labels']
14 valid_x, valid_y = valid_data['valid_data'], valid_data['valid_labels']
15 test_x, test_y = test_data['test_data'], test_data['test_labels']
16
17 # convert to Tensor
18 train_x = torch.from_numpy(train_x).to(torch.float32)
19 train_y = np.argmax(train_y, axis=1)
20 train_y = torch.from_numpy(train_y)
21 train_data = TensorDataset(train_x, train_y)
22 # convert to Tensor
23 test_x = torch.from_numpy(test_x).to(torch.float32)
24 test_y = np.argmax(test_y, axis=1)
25 test_y = torch.from_numpy(test_y)
26 test_data = TensorDataset(test_x, test_y)
27 # convert to Tensor
28 valid_x = torch.from_numpy(valid_x).to(torch.float32)
29 valid_y = np.argmax(valid_y, axis=1)
30 valid_y = torch.from_numpy(valid_y)
31 valid_data = TensorDataset(valid_x, valid_y)
32 |
33 batch_size = 32
34
35 # Create data loaders
36 train_dataloader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
37 valid_dataloader = DataLoader(valid_data, batch_size=batch_size, shuffle=True)
38 test_dataloader = DataLoader(test_data, batch_size=batch_size, shuffle=True)
39
40 # Get cpu or gpu device for training.
41 device = "cuda" if torch.cuda.is_available() else "cpu"
42 print(f"Using {device} device")
```

```
44     class Net(nn.Module):
45         def __init__(self):
46             super(Net, self).__init__()
47             # 1 input image channel, 6 output channels, 5x5 square convolution
48             # kernel
49             self.conv1 = nn.Conv2d(1, 7, 3)
50             self.conv2 = nn.Conv2d(7, 13, 5)
51             # an affine operation: y = Wx + b
52             self.fc1 = nn.Linear(325, 84) # 5*5 from image dimension
53             self.fc2 = nn.Linear(84, 36)
54
55         def forward(self, x):
56             # Max pooling over a (2, 2) window
57             x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
58             # If the size is a square, you can specify with a single number
59             x = F.max_pool2d(F.relu(self.conv2(x)), (2, 2))
60             x = torch.flatten(x, 1) # flatten all dimensions except the batch dimension
61             x = F.relu(self.fc1(x))
62             x = self.fc2(x)
63             return x
64
65     model = Net().to(device)
66     print(model)
67     loss_fn = nn.CrossEntropyLoss()
68     optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

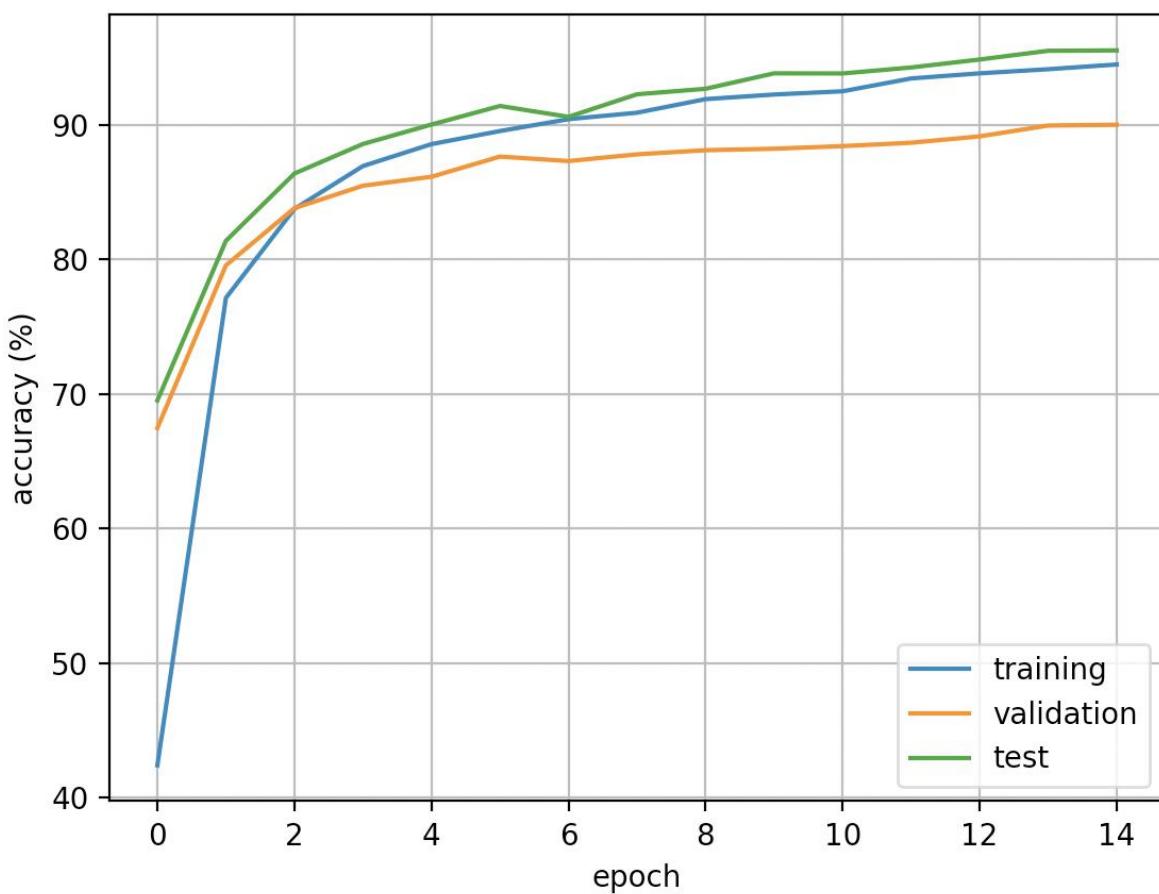
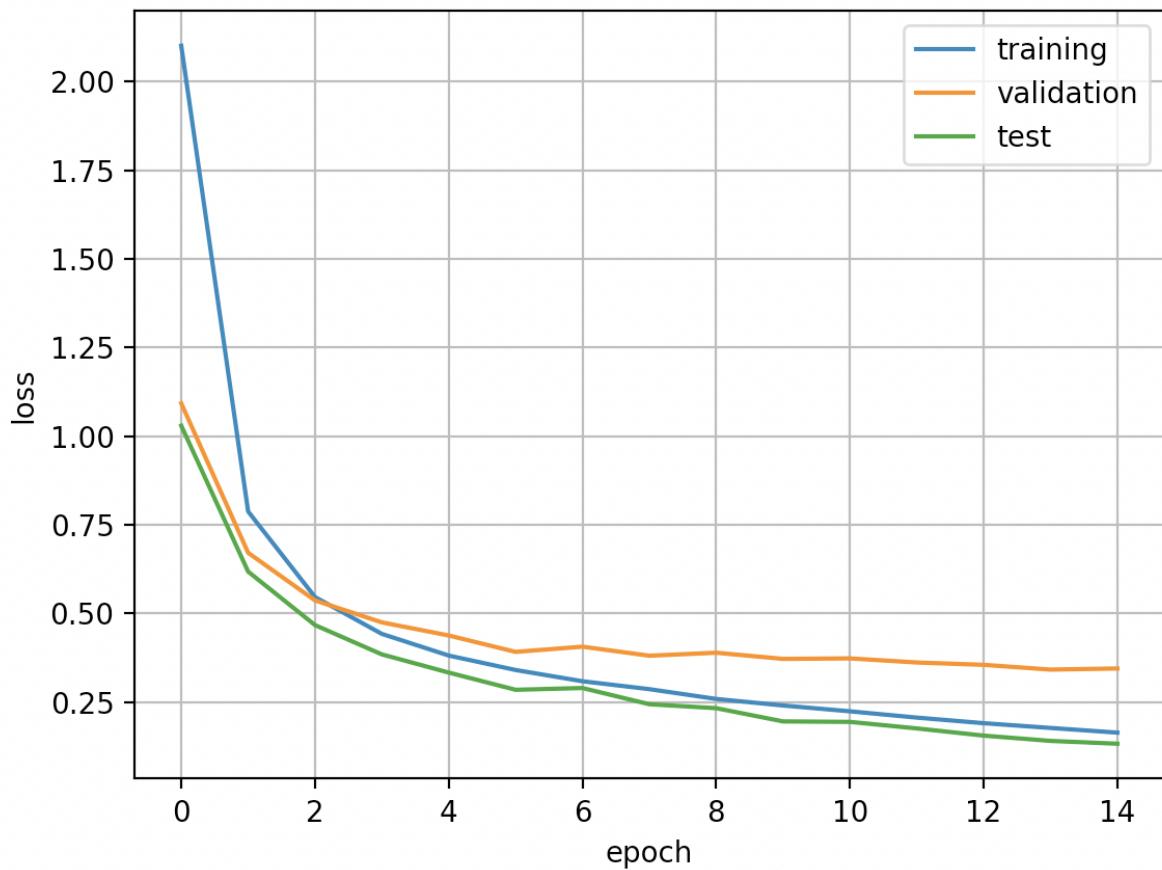
```
70     def train(dataloader, model, loss_fn, optimizer):
71         size = len(dataloader.dataset)
72         num_batches = len(dataloader)
73         model.train()
74         train_loss = 0
75         correct = 0
76         for batch, (X, y) in enumerate(dataloader):
77             X, y = X.to(device), y.to(device)
78             X = X.view(-1, 1, 32, 32)
79             # Compute prediction error
80             pred = model(X)
81             loss = loss_fn(pred, y)
82             train_loss = train_loss + loss.item()
83             correct += (pred.argmax(1) == y).type(torch.float).sum().item()
84             # Backpropagation
85             optimizer.zero_grad()
86             loss.backward()
87             optimizer.step()
88             if batch % 100 == 0:
89                 loss, current = loss.item(), batch * len(X)
90                 print(f"loss: {loss:>7f} [{current:>5d}/{size:>5d}]")
91             correct /= size
92             train_loss /= num_batches
93             print(f"Train Accuracy: {(100 * correct):>0.1f}%")
94
95     return 100*correct, train_loss
```

```
97  ►  def test(dataloader, model, loss_fn):
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
```

```
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            X, y = X.to(device), y.to(device)
            X = X.view(-1, 1, 32, 32)
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    acc = 100*correct
    return acc, test_loss
print(f"Test Error: \n Accuracy: {(acc):>0.1f}%, Avg loss: {test_loss:>8f} \n")

def validate(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            X, y = X.to(device), y.to(device)
            X = X.view(-1, 1, 32, 32)
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    acc = 100*correct
    return acc, test_loss
print(f"Validation Error: \n Accuracy: {(acc):>0.1f}%, Avg loss: {test_loss:>8f} \n")
```

```
133
134     epochs = 15
135     train_acc = []
136     train_loss = []
137     validate_acc = []
138     validate_loss = []
139     test_acc = []
140     test_loss = []
141     for t in range(epochs):
142         print(f"Epoch {t+1}\n-----")
143         tr_acc, tr_loss = train(train_dataloader, model, loss_fn, optimizer)
144         train_acc.append(tr_acc)
145         train_loss.append(tr_loss)
146         val_acc, val_loss = validate(valid_dataloader, model, loss_fn)
147         validate_acc.append(val_acc)
148         validate_loss.append(val_loss)
149         te_acc, te_loss = test(test_dataloader, model, loss_fn)
150         test_acc.append(te_acc)
151         test_loss.append(te_loss)
152     print("Done!")
153
154     # plot loss
155     plt.plot(range(len(train_loss)), train_loss, label="training")
156     plt.plot(range(len(validate_loss)), validate_loss, label="validation")
157     plt.plot(range(len(test_loss)), test_loss, label="test")
158     plt.xlabel("epoch")
159     plt.ylabel("loss")
160     plt.legend()
161     plt.grid()
162     plt.show()
163
164     # plot training accuracy
165     plt.plot(range(len(train_acc)), train_acc, label="training")
166     plt.plot(range(len(validate_acc)), validate_acc, label="validation")
167     plt.plot(range(len(test_acc)), test_acc, label="test")
168     plt.xlabel("epoch")
169     plt.ylabel("accuracy (%)")
170     plt.legend()
171     plt.grid()
172     plt.show()
```

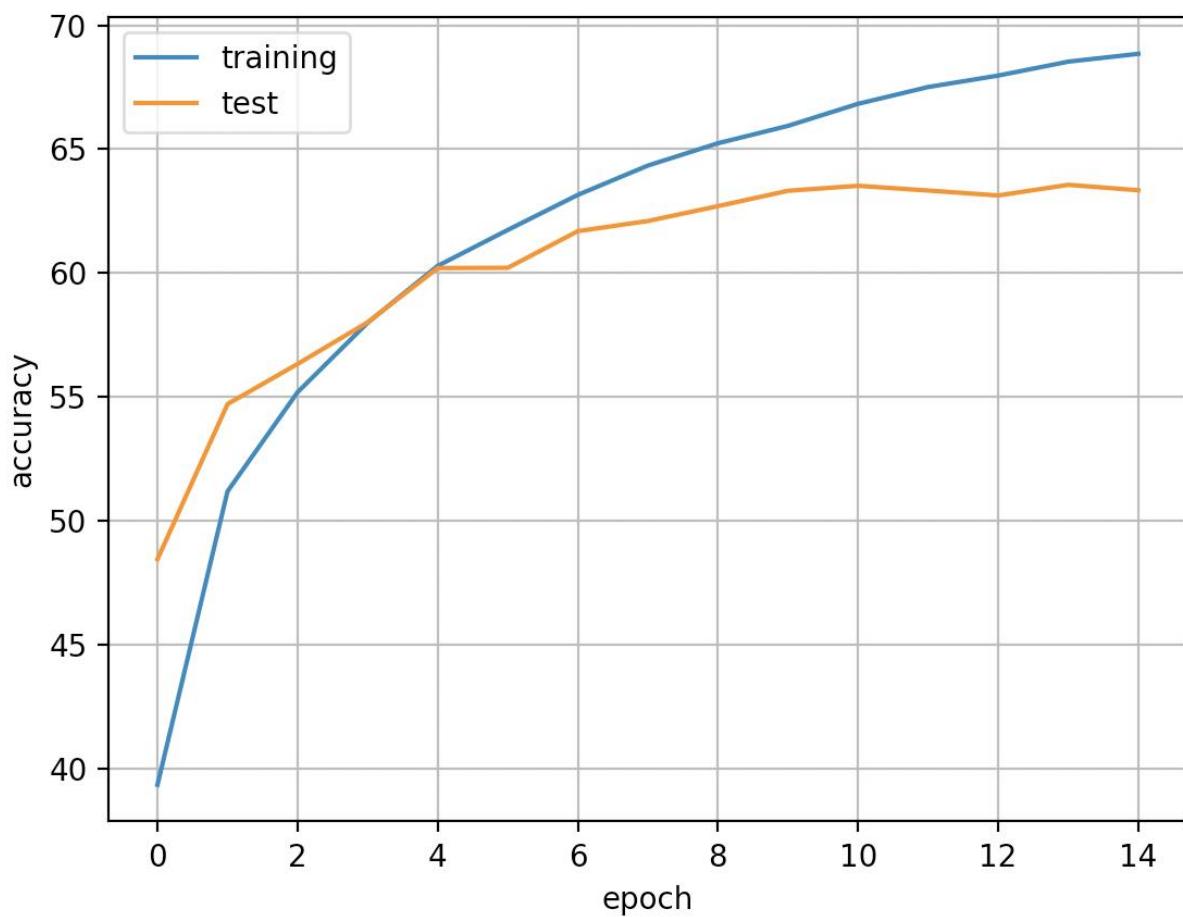
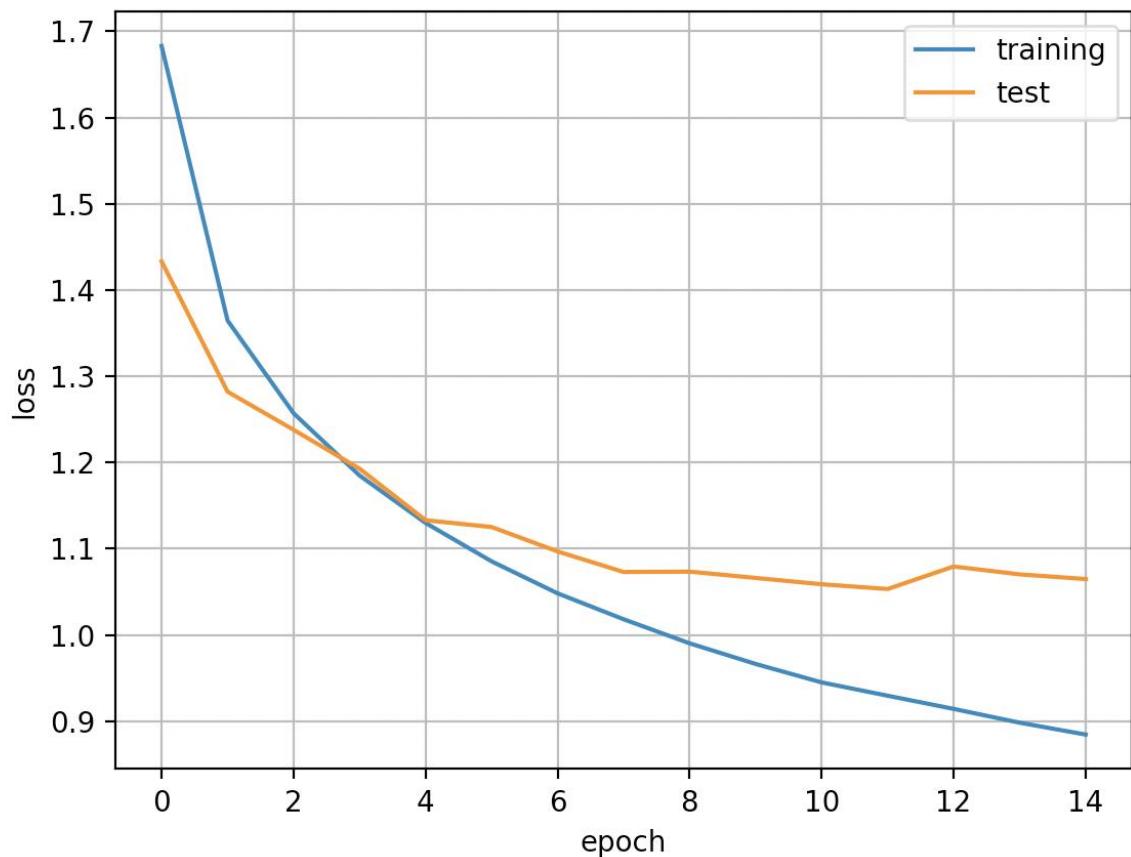


6.1.3 Code / Writeup

```
1 import matplotlib.pyplot as plt
2 import torch
3 import torch.nn as nn
4 import torch.nn.functional as F
5 from torch.utils.data import DataLoader
6 import torchvision
7 import torchvision.transforms as transforms
8
9 transform = transforms.Compose(
10     [transforms.ToTensor(),
11      transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
12
13 batch_size = 36
14
15 trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
16 train_dataloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True)
17 testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
18 test_dataloader = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=False)
19 classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
20
21 # Get cpu or gpu device for training.
22 device = "cuda" if torch.cuda.is_available() else "cpu"
23 print(f"Using {device} device")
24
25
26 class Net(nn.Module):
27     def __init__(self):
28         super(Net, self).__init__()
29         self.conv1 = nn.Conv2d(3, 7, 3)
30         self.conv2 = nn.Conv2d(7, 13, 5)
31         self.fc1 = nn.Linear(325, 84)
32         self.fc2 = nn.Linear(84, 36)
33
34     def forward(self, x):
35         # Max pooling over a (2, 2) window
36         x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
37         # If the size is a square, you can specify with a single number
38         x = F.max_pool2d(F.relu(self.conv2(x)), (2, 2))
39         x = torch.flatten(x, 1)
40         x = F.relu(self.fc1(x))
41         x = self.fc2(x)
42         return x
43
44 model = Net().to(device)
45 print(model)
```

```
45
46
47     loss_fn = nn.CrossEntropyLoss()
48     optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
49
50     def train(dataloader, model, loss_fn, optimizer):
51         size = len(dataloader.dataset)
52         num_batches = len(dataloader)
53         model.train()
54         train_loss = 0
55         correct = 0
56
57         for batch, (X, y) in enumerate(dataloader):
58             X, y = X.to(device), y.to(device)
59             # Compute prediction error
60             pred = model(X)
61             loss = loss_fn(pred, y)
62             train_loss = train_loss + loss.item()
63             correct += (pred.argmax(1) == y).type(torch.float).sum().item()
64             # Backpropagation
65             optimizer.zero_grad()
66             loss.backward()
67             optimizer.step()
68             if batch % 100 == 0:
69                 loss, current = loss.item(), batch * len(X)
70                 print(f"loss: {loss:.7f} [{current:.5d}/{size:.5d}]")
71
72         correct /= size
73         train_loss /= num_batches
74         print(f"Train Accuracy: {(100 * correct):.1f}%")
75
76
77     def test(dataloader, model, loss_fn):
78         size = len(dataloader.dataset)
79         num_batches = len(dataloader)
80         model.eval()
81         test_loss, correct = 0, 0
82         with torch.no_grad():
83             for X, y in dataloader:
84                 X, y = X.to(device), y.to(device)
85                 pred = model(X)
86                 test_loss += loss_fn(pred, y).item()
87                 correct += (pred.argmax(1) == y).type(torch.float).sum().item()
88         test_loss /= num_batches
89         correct /= size
90         acc = 100*correct
91         return acc, test_loss
92
93         print(f"Test Error: \n Accuracy: {acc:.1f}%, Avg loss: {test_loss:.8f} \n")
94
```

```
94
95     epochs = 15
96     train_acc = []
97     train_loss = []
98     test_acc = []
99     test_loss = []
100    for t in range(epochs):
101        print(f"Epoch {t+1}\n-----")
102        tr_acc, tr_loss = train(train_dataloader, model, loss_fn, optimizer)
103        train_acc.append(tr_acc)
104        train_loss.append(tr_loss)
105        te_acc, te_loss = test(test_dataloader, model, loss_fn)
106        test_acc.append(te_acc)
107        test_loss.append(te_loss)
108    print("Done!")
109
110    # plot loss
111    plt.plot(range(len(train_loss)), train_loss, label="training")
112    plt.plot(range(len(test_loss)), test_loss, label="test")
113    plt.xlabel("epoch")
114    plt.ylabel("loss")
115    plt.legend()
116    plt.grid()
117    plt.show()
118
119    # plot training accuracy
120    plt.plot(range(len(train_acc)), train_acc, label="training")
121    plt.plot(range(len(test_acc)), test_acc, label="test")
122    plt.xlabel("epoch")
123    plt.ylabel("accuracy")
124    plt.legend()
125    plt.grid()
126    plt.show()
```

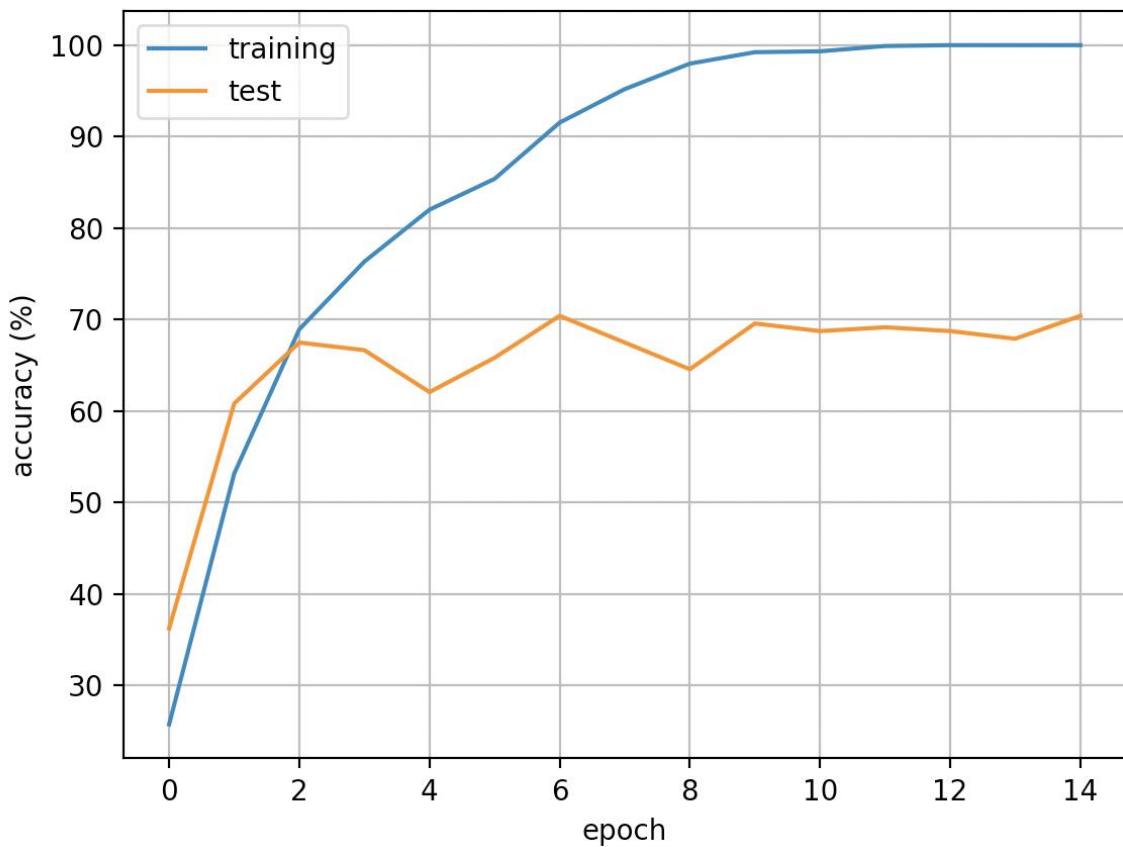
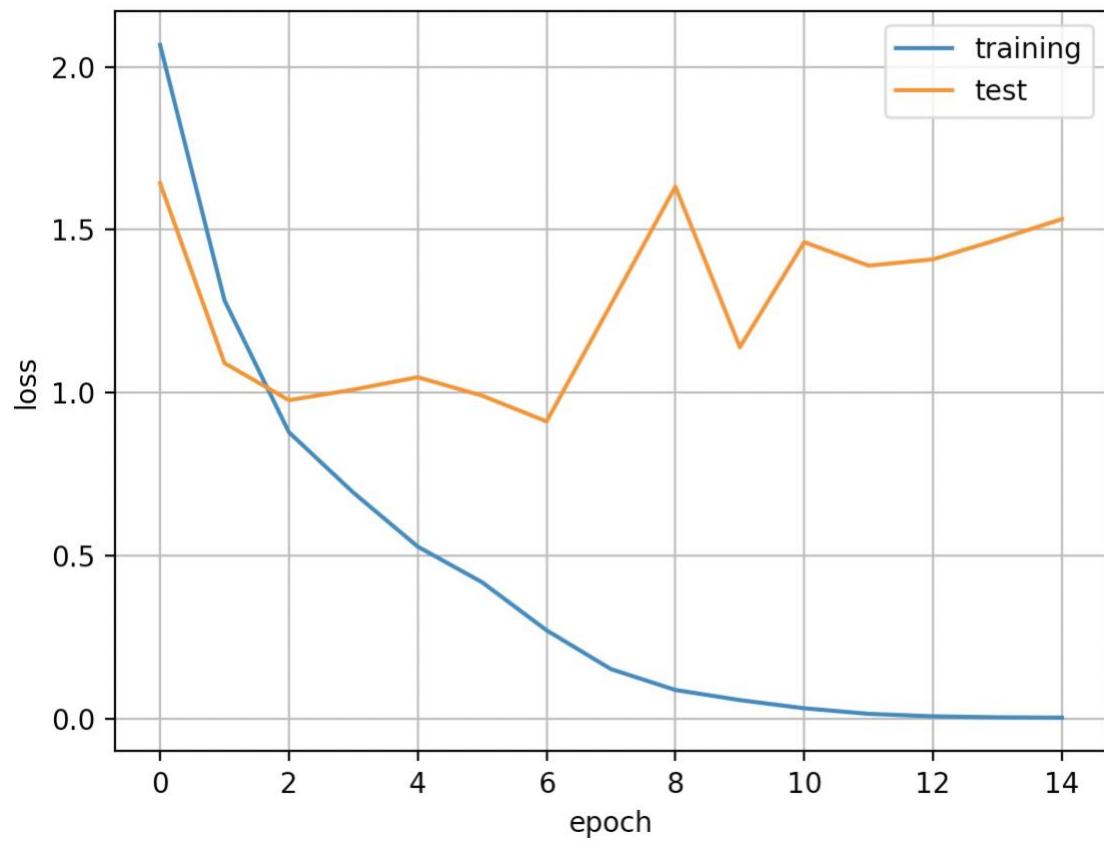


6.1.4 Code / Writeup

```
1  import matplotlib.pyplot as plt
2  import torch
3  import torch.nn as nn
4  import torch.nn.functional as F
5  from torch.utils.data import DataLoader
6  import torchvision
7  import torchvision.transforms as transforms
8
9  transform = transforms.Compose(
10     [transforms.ToTensor(),
11      transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
12      transforms.Resize((256, 256))])
13
14 batch_size = 36
15
16 trainset = torchvision.datasets.ImageFolder(root='../Train', transform=transform)
17 train_dataloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True)
18 testset = torchvision.datasets.ImageFolder(root='../Test', transform=transform)
19 test_dataloader = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=False)
20
21 # Get cpu or gpu device for training.
22 device = "cuda" if torch.cuda.is_available() else "cpu"
23 print(f"Using {device} device")
24
25 class Net(nn.Module):
26     def __init__(self):
27         super(Net, self).__init__()
28         self.conv1 = nn.Conv2d(3, 7, 3)
29         self.conv2 = nn.Conv2d(7, 13, 5)
30         self.fc1 = nn.Linear(48373, 84)
31         self.fc2 = nn.Linear(84, 8)
32
33     def forward(self, x):
34         # Max pooling over a (2, 2) window
35         x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
36         # If the size is a square, you can specify with a single number
37         x = F.max_pool2d(F.relu(self.conv2(x)), (2, 2))
38         x = torch.flatten(x, 1)
39         x = F.relu(self.fc1(x))
40         x = self.fc2(x)
41
42         return x
43
44 model = Net().to(device)
45 print(model)
```

```
46     loss_fn = nn.CrossEntropyLoss()
47     optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
48
49     def train(dataloader, model, loss_fn, optimizer):
50         size = len(dataloader.dataset)
51         num_batches = len(dataloader)
52         model.train()
53         train_loss = 0
54         correct = 0
55         for batch, (X, y) in enumerate(dataloader):
56             X, y = X.to(device), y.to(device)
57             # Compute prediction error
58             pred = model(X)
59             loss = loss_fn(pred, y)
60             train_loss = train_loss + loss.item()
61             correct += (pred.argmax(1) == y).type(torch.float).sum().item()
62             # Backpropagation
63             optimizer.zero_grad()
64             loss.backward()
65             optimizer.step()
66             if batch % 100 == 0:
67                 loss, current = loss.item(), batch * len(X)
68                 print(f"loss: {loss:.7f} [{current:.5d}/{size:.5d}]")
69             correct /= size
70             train_loss /= num_batches
71             print(f"Train Accuracy: {(100 * correct):>0.1f}%")
72         return 100*correct, train_loss
73
74     def test(dataloader, model, loss_fn):
75         size = len(dataloader.dataset)
76         num_batches = len(dataloader)
77         model.eval()
78         test_loss, correct = 0, 0
79         with torch.no_grad():
80             for X, y in dataloader:
81                 X, y = X.to(device), y.to(device)
82                 pred = model(X)
83                 test_loss += loss_fn(pred, y).item()
84                 correct += (pred.argmax(1) == y).type(torch.float).sum().item()
85         test_loss /= num_batches
86         correct /= size
87         acc = 100*correct
88         return acc, test_loss
89         print(f"Test Error: \n Accuracy: {(acc):>0.1f}%, Avg loss: {test_loss:>8f} \n")
```

```
91     epochs = 15
92     train_acc = []
93     train_loss = []
94     test_acc = []
95     test_loss = []
96     for t in range(epochs):
97         print(f"Epoch {t+1}\n-----")
98         tr_acc, tr_loss = train(train_dataloader, model, loss_fn, optimizer)
99         train_acc.append(tr_acc)
100        train_loss.append(tr_loss)
101        te_acc, te_loss = test(test_dataloader, model, loss_fn)
102        test_acc.append(te_acc)
103        test_loss.append(te_loss)
104    print("Done!")
105
106    # plot loss
107    plt.plot(range(len(train_loss)), train_loss, label="training")
108    plt.plot(range(len(test_loss)), test_loss, label="test")
109    plt.xlabel("epoch")
110    plt.ylabel("loss")
111    plt.legend()
112    plt.grid()
113    plt.show()
114
115    # plot training accuracy
116    plt.plot(range(len(train_acc)), train_acc, label="training")
117    plt.plot(range(len(test_acc)), test_acc, label="test")
118    plt.xlabel("epoch")
119    plt.ylabel("accuracy (%)")
120    plt.legend()
121    plt.grid()
122    plt.show()
```



Below is the table of ablation study I conducted in HW1 when we were using Bag of Words. As you can see from the table, the highest accuracy value I found was only 59%. From the accuracy shown in the plot above using my neural network, the highest accuracy I was able to achieve is around 70% for the testing. Therefore, the neural network was able to outperform what we did in homework 1.

	Changing K	filter scales	L	alpha	K	accuracy
DEFAULT	K = 10	1,2		1	25	10 0.504310345
	K = 30	1,2		1	25	30 0.551724138
HIGHEST	K = 50	1,2		1	25	50 0.592672414
	K = 60	1,2		1	25	60 0.543103448
	K = 100	1,2		1	25	100 0.547413793
	Changing alpha	filter scales	L	alpha	K	accuracy
	alpha = 25	1,2		1	25	50 0.592672414
	alpha = 50	1,2		1	50	50 0.584051724
	alpha = 100	1,2		1	100	50 0.549568966
	Changing L	filter scales	L	alpha	K	accuracy
	L = 1	1,2		1	25	50 0.592672414
	L = 2	1,2		2	25	50 0.577586207
	L = 1	1,2		1	25	100 0.547413793
	L = 2	1,2		2	25	100 0.55387931
	L = 2	1,2,4,8,16		2	150	150 0.581896552
	L = 3	1,2,4,8,16		3	150	150 0.584051724
	L = 3	1,2,4,8,16		3	100	100 0.590517241
	L = 4	1,2,4,8,16		4	100	100 0.568965517
	Changing filter scales	filter scales	L	alpha	K	accuracy
	scales = 1,2	1,2		1	25	50 0.592672414
	scales = 1,2,4	1,2,4		1	25	50 0.558189655
	scales = 1,2,4,8	1,2,4,8		1	25	50 0.560344828
	Testing Higher Input Values	filter scales	L	alpha	K	accuracy
		1,2,4,8		2	400	400 0.586206897
		1,2,4,8,8*sqrt(2)		2	200	300 0.581896552
		1,2,4,8,16		3	100	200 0.584051724
		1,2,4,8,16		2	100	100 0.5625
		1,2,4,8,16		2	50	50 0.55387931
		1,2,4,8		2	50	100 0.588362069

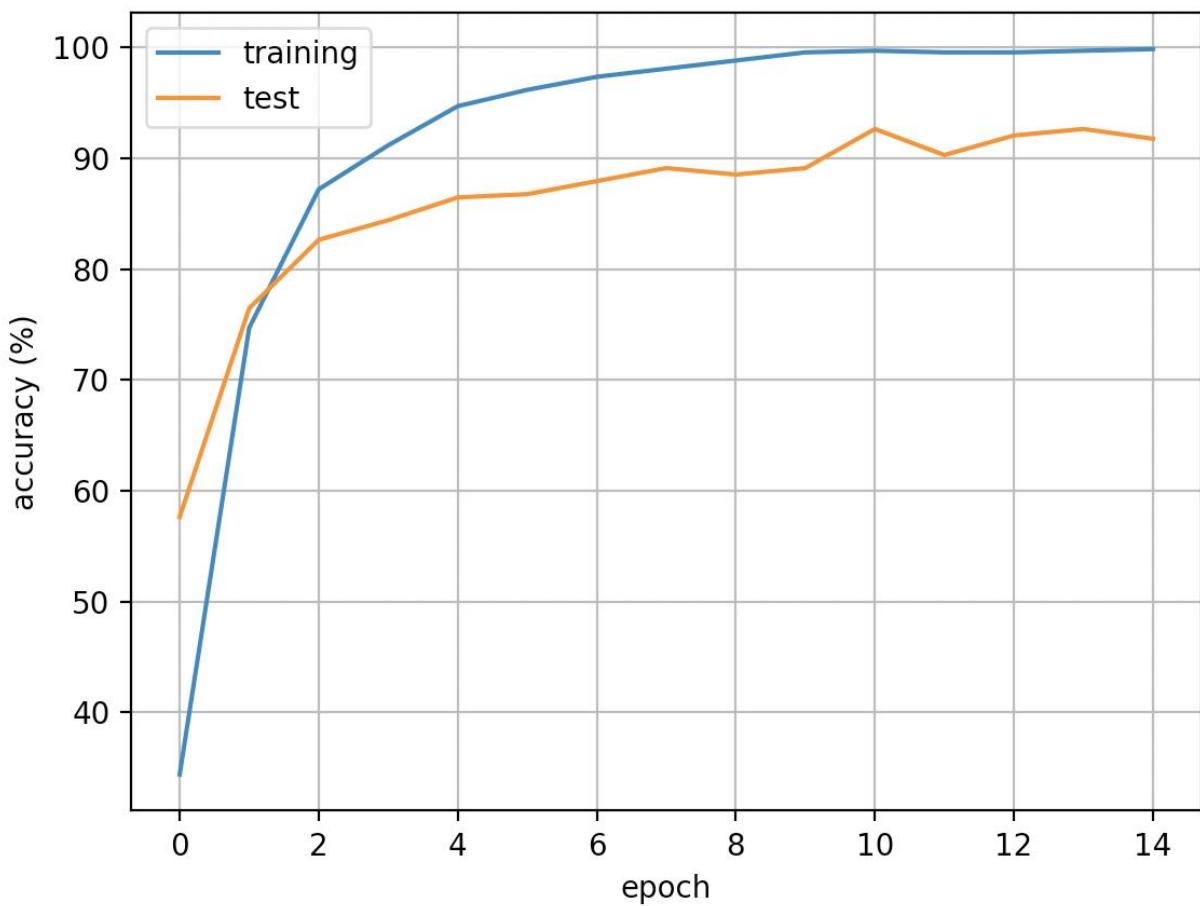
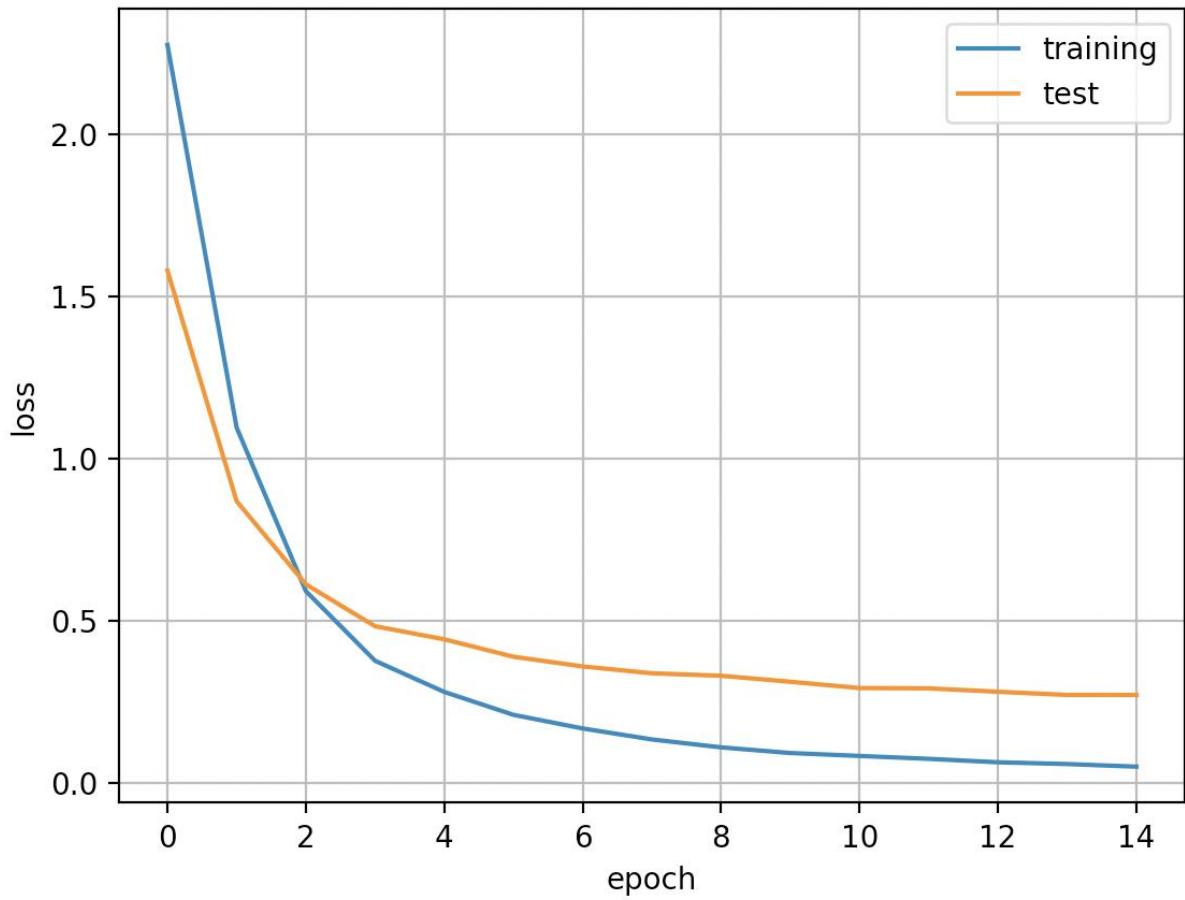
6.2 Fine Tuning Code / Writeup

SqueezeNet Code:

```
 1  import matplotlib.pyplot as plt
 2  import torch
 3  import torch.nn as nn
 4  from torch.utils.data import DataLoader
 5  import torchvision
 6  import torchvision.transforms as transforms
 7  from torchvision.models import squeezenet1_1
 8
 9  transform = transforms.Compose(
10      [transforms.ToTensor(),
11       transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
12       transforms.Resize((224,224))])
13  batch_size = 36
14
15  trainset = torchvision.datasets.ImageFolder(root='../data/oxford-flowers17/train', transform=transform)
16  train_dataloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True)
17  testset = torchvision.datasets.ImageFolder(root='../data/oxford-flowers17/test', transform=transform)
18  test_dataloader = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=False)
19
20  # Get cpu or gpu device for training.
21  device = "cuda" if torch.cuda.is_available() else "cpu"
22  print(f"Using {device} device")
23
24  model = squeezenet1_1(pretrained=True).to(device)
25
26  for param in model.parameters():
27      param.requires_grad = False
28
29  final_conv = nn.Conv2d(512,17,kernel_size=1)
30  model.classifier = nn.Sequential(nn.Dropout(p=0.5), final_conv, nn.ReLU(inplace=True), nn.AdaptiveAvgPool2d((1,1)))
31
32  for param in model.classifier.parameters():
33      param.requires_grad = True
34
35  loss_fn = nn.CrossEntropyLoss()
36  optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

```
38     def train(dataloader, model, loss_fn, optimizer):
39         size = len(dataloader.dataset)
40         num_batches = len(dataloader)
41         model.train()
42         train_loss = 0
43         correct = 0
44         for batch, (X, y) in enumerate(dataloader):
45             X, y = X.to(device), y.to(device)
46             # Compute prediction error
47             pred = model(X)
48             loss = loss_fn(pred, y)
49             train_loss = train_loss + loss.item()
50             correct += (pred.argmax(1) == y).type(torch.float).sum().item()
51             # Backpropagation
52             optimizer.zero_grad()
53             loss.backward()
54             optimizer.step()
55             if batch % 100 == 0:
56                 loss, current = loss.item(), batch * len(X)
57                 print(f"loss: {loss:.3f} [{current:.3f}/{size:.3f}]")
58
59         correct /= size
60         train_loss /= num_batches
61         print(f"Train Accuracy: {(100 * correct):.1f}%")
62
63     return 100*correct, train_loss
64
65
66     def test(dataloader, model, loss_fn):
67         size = len(dataloader.dataset)
68         num_batches = len(dataloader)
69         model.eval()
70         test_loss, correct = 0, 0
71         with torch.no_grad():
72             for X, y in dataloader:
73                 X, y = X.to(device), y.to(device)
74                 pred = model(X)
75                 test_loss += loss_fn(pred, y).item()
76                 correct += (pred.argmax(1) == y).type(torch.float).sum().item()
77         test_loss /= num_batches
78         correct /= size
79         acc = 100*correct
80         return acc, test_loss
81
82         print(f"Test Error: \n Accuracy: {(acc):.1f}%, Avg Loss: {test_loss:.8f} \n")
```

```
83     epochs = 15
84     train_acc = []
85     train_loss = []
86     test_acc = []
87     test_loss = []
88     for t in range(epochs):
89         print(f"Epoch {t+1}\n-----")
90         tr_acc, tr_loss = train(train_dataloader, model, loss_fn, optimizer)
91         train_acc.append(tr_acc)
92         train_loss.append(tr_loss)
93         te_acc, te_loss = test(test_dataloader, model, loss_fn)
94         test_acc.append(te_acc)
95         test_loss.append(te_loss)
96     print("Done!")
97
98     # plot loss
99     plt.plot(range(len(train_loss)), train_loss, label="training")
100    plt.plot(range(len(test_loss)), test_loss, label="test")
101    plt.xlabel("epoch")
102    plt.ylabel("loss")
103    plt.legend()
104    plt.grid()
105    plt.show()
106
107    # plot training accuracy
108    plt.plot(range(len(train_acc)), train_acc, label="training")
109    plt.plot(range(len(test_acc)), test_acc, label="test")
110    plt.xlabel("epoch")
111    plt.ylabel("accuracy (%)")
112    plt.legend()
113    plt.grid()
114    plt.show()
```



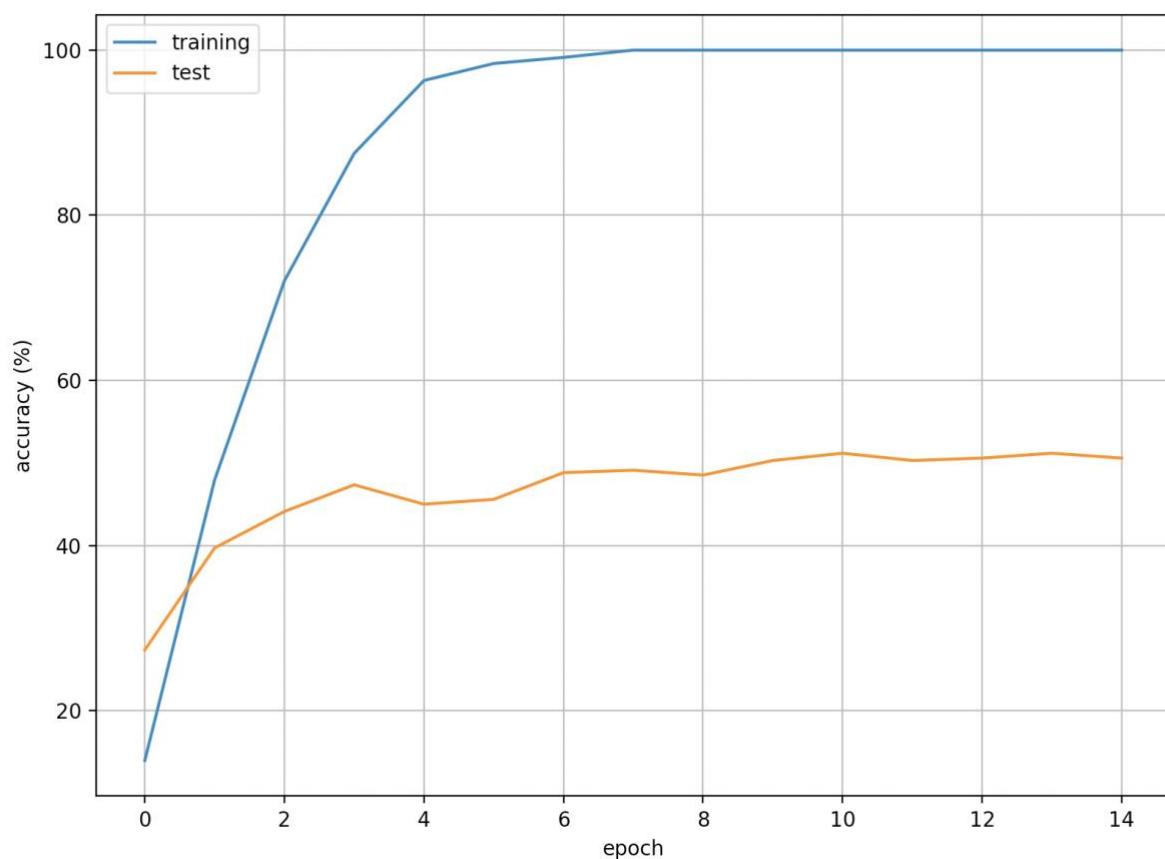
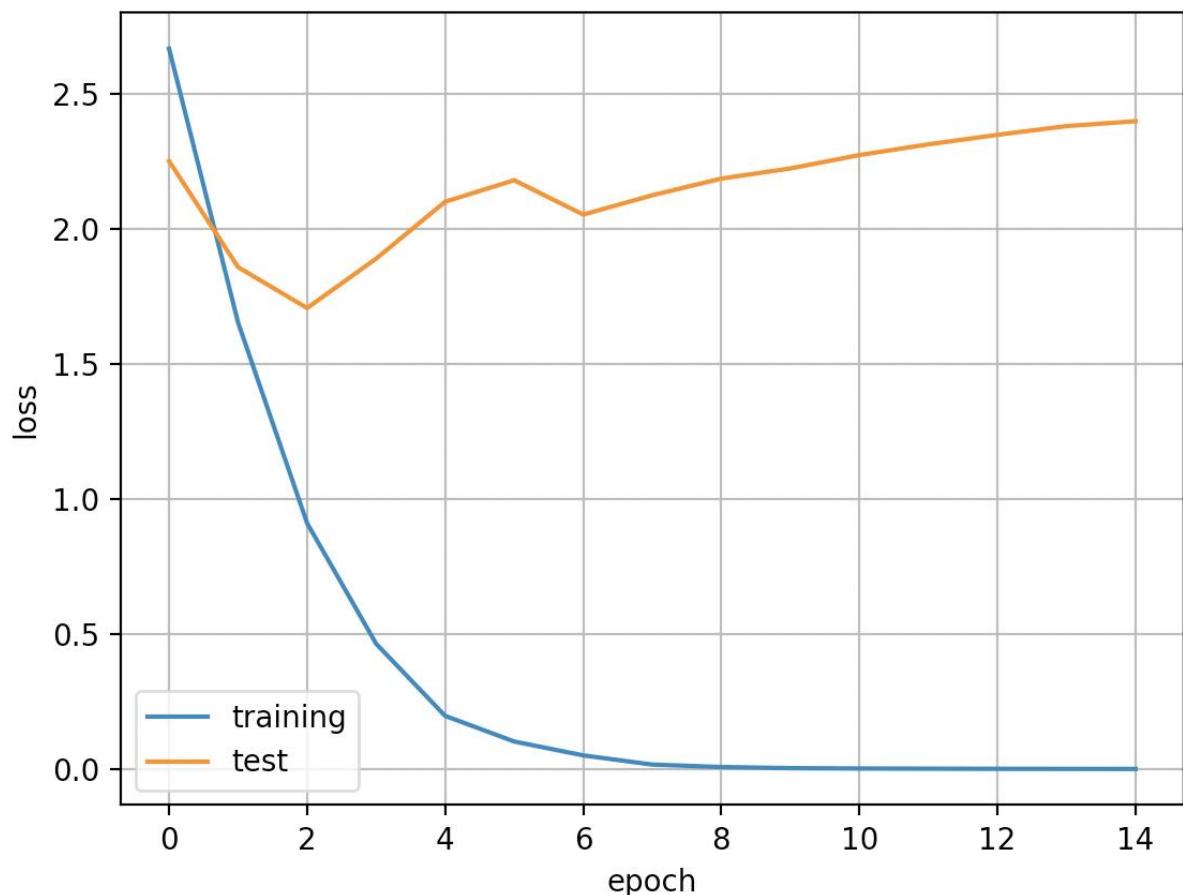
Custom Architecture:

```
1  import matplotlib.pyplot as plt
2  import torch
3  import torch.nn as nn
4  import torch.nn.functional as F
5  from torch.utils.data import DataLoader
6  import torchvision
7  import torchvision.transforms as transforms
8
9  transform = transforms.Compose(
10     [transforms.ToTensor(),
11      transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
12      transforms.Resize((224, 224))])
13
14 batch_size = 32
15
16 trainset = torchvision.datasets.ImageFolder(root='./data/oxford-flowers17/train', transform=transform)
17 train_dataloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True)
18 # validset = torchvision.datasets.ImageFolder(root='./data/oxford-flowers17/val', transform=transform)
19 # valid_dataloader = torch.utils.data.DataLoader(validset, batch_size=batch_size, shuffle=True)
20 testset = torchvision.datasets.ImageFolder(root='./data/oxford-flowers17/test', transform=transform)
21 test_dataloader = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=False)
22
23
24 # Get cpu or gpu device for training.
25 device = "cuda" if torch.cuda.is_available() else "cpu"
26 print(f"Using {device} device")
27
28 class Net(nn.Module):
29     def __init__(self):
30         super(Net, self).__init__()
31         self.conv1 = nn.Conv2d(3, 7, 3)
32         self.conv2 = nn.Conv2d(7, 13, 5)
33         self.fc1 = nn.Linear(36517, 84)
34         self.fc2 = nn.Linear(84, 17)
35
36     def forward(self, x):
37         # Max pooling over a (2, 2) window
38         x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
39         # If the size is a square, you can specify with a single number
40         x = F.max_pool2d(F.relu(self.conv2(x)), (2, 2))
41         x = torch.flatten(x, 1)
42         x = F.relu(self.fc1(x))
43         x = self.fc2(x)
44
45         return x
```

```
44         return x
45
46     model = Net().to(device)
47     print(model)
48
49     loss_fn = nn.CrossEntropyLoss()
50     optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
51
52     def train(dataloader, model, loss_fn, optimizer):
53         size = len(dataloader.dataset)
54         num_batches = len(dataloader)
55         model.train()
56         train_loss = 0
57         correct = 0
58
59         for batch, (X, y) in enumerate(dataloader):
60             X, y = X.to(device), y.to(device)
61             # Compute prediction error
62             pred = model(X)
63             loss = loss_fn(pred, y)
64             train_loss = train_loss + loss.item()
65             correct += (pred.argmax(1) == y).type(torch.float).sum().item()
66             # Backpropagation
67             optimizer.zero_grad()
68             loss.backward()
69             optimizer.step()
70             if batch % 100 == 0:
71                 loss, current = loss.item(), batch * len(X)
72                 print(f"loss: {loss:.7f} [{current:.5d}/{size:.5d}]")
73
74         correct /= size
75         train_loss /= num_batches
76         print(f"Train Accuracy: {(100 * correct):.1f}%")
77
78     return 100*correct, train_loss
```

```
79
80  ► def test(dataloader, model, loss_fn):
81      size = len(dataloader.dataset)
82      num_batches = len(dataloader)
83      model.eval()
84      test_loss, correct = 0, 0
85      with torch.no_grad():
86          for X, y in dataloader:
87              X, y = X.to(device), y.to(device)
88              pred = model(X)
89              test_loss += loss_fn(pred, y).item()
90              correct += (pred.argmax(1) == y).type(torch.float).sum().item()
91      test_loss /= num_batches
92      correct /= size
93      acc = 100*correct
94      return acc, test_loss
95      print(f"Test Error: \n Accuracy: {(acc):>0.1f}%, Avg loss: {test_loss:>8f}\n")
96
97      epochs = 15
98      train_acc = []
99      train_loss = []
100     # validate_acc = []
101     # validate_loss = []
102     test_acc = []
103     test_loss = []
104     for t in range(epochs):
105         print(f"Epoch {t+1}\n-----")
106         tr_acc, tr_loss = train(train_dataloader, model, loss_fn, optimizer)
107         train_acc.append(tr_acc)
108         train_loss.append(tr_loss)
109         # val_acc, val_loss = validate(valid_dataloader, model, loss_fn)
110         # validate_acc.append(val_acc)
111         # validate_loss.append(val_loss)
112         te_acc, te_loss = test(test_dataloader, model, loss_fn)
113         test_acc.append(te_acc)
114         test_loss.append(te_loss)
115     print("Done!")
```

```
96
97     epochs = 15
98     train_acc = []
99     train_loss = []
100    # validate_acc = []
101    # validate_loss = []
102    test_acc = []
103    test_loss = []
104    for t in range(epochs):
105        print(f"Epoch {t+1}\n-----")
106        tr_acc, tr_loss = train(train_dataloader, model, loss_fn, optimizer)
107        train_acc.append(tr_acc)
108        train_loss.append(tr_loss)
109    # val_acc, val_loss = validate(valid_dataloader, model, loss_fn)
110    # validate_acc.append(val_acc)
111    # validate_loss.append(val_loss)
112    te_acc, te_loss = test(test_dataloader, model, loss_fn)
113    test_acc.append(te_acc)
114    test_loss.append(te_loss)
115    print("Done!")
116
117    # plot loss
118    plt.plot(range(len(train_loss)), train_loss, label="training")
119    plt.plot(range(len(test_loss)), test_loss, label="test")
120    plt.xlabel("epoch")
121    plt.ylabel("loss")
122    plt.legend()
123    plt.grid()
124    plt.show()
125
126    # plot training accuracy
127    plt.plot(range(len(train_acc)), train_acc, label="training")
128    plt.plot(range(len(test_acc)), test_acc, label="test")
129    plt.xlabel("epoch")
130    plt.ylabel("accuracy (%)")
131    plt.legend()
132    plt.grid()
133    plt.show()
```



Above we have the results of the custom architecture using the oxford flower image set. In the part above, we had plots from the squeezenet model for the same image set. From those sets of plots, it's clear that squeezenet outperforms the architecture which I made. The squeezenet results had an accuracy above 90% while the custom architecture is at about 50% final accuracy after 15 epochs. Also, the squeezenet results have much lower loss than the custom results.