1.1 Theory

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

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

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

$$Soft\max(x_i + c) = \frac{e^{x_i + c}}{\sum_j e^{x_j + c}}$$

This can be simplified as shown:

$$Soft\max(x_i + c) = \frac{e^{x_i} e^{c}}{e^{c} \sum_j e^{x_j}}$$

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

Therefore, Softmax(xi) = Softmax(xi + c) so softmax is invariant to translation.

When c = 0, the numerator values will be in the range (0,inf). 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:

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

Rewriting the new W as W':

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

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:

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

We can differentiate the function with respect to x:

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

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

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

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

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

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

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

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

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

1.5 Theory

We have:

$$y = Wx + b$$

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

Taking the partial derivatives:

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

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

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

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

Through chain rule:

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

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

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

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

```
############################## Q 2.2.1 ##############################
# x is a matrix
# a sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

    Keyword arguments:
    X -- input vector [Examples x D]
    params -- a dictionary containing parameters
    name -- name of the layer
    activation -- the activation function (default is sigmoid)
    """
    pre_act, post_act = None, None
    # get the layer parameters
    W = params['W' + name]
    b = params['b' + name]

    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\t',test.min(),test.max())
# implement forward
h1 = forward(x,params,'layer1')
print(h1.shape)
```

2.2.2 Code

```
############################## Q 2.2.2 ##############################
# 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

```
############################## Q 2.2.3 ##############################
# compute total loss and accuracy
# y is size [examples,classes]
# probs is size [examples,classes]
def compute_loss_and_acc(y, probs):
    loss = -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

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

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

    Keyword arguments:
    delta -- errors to backprop
    params -- a dictionary containing parameters
    name -- name of the layer
    activation_deriv -- the derivative of the activation_func
    """
    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

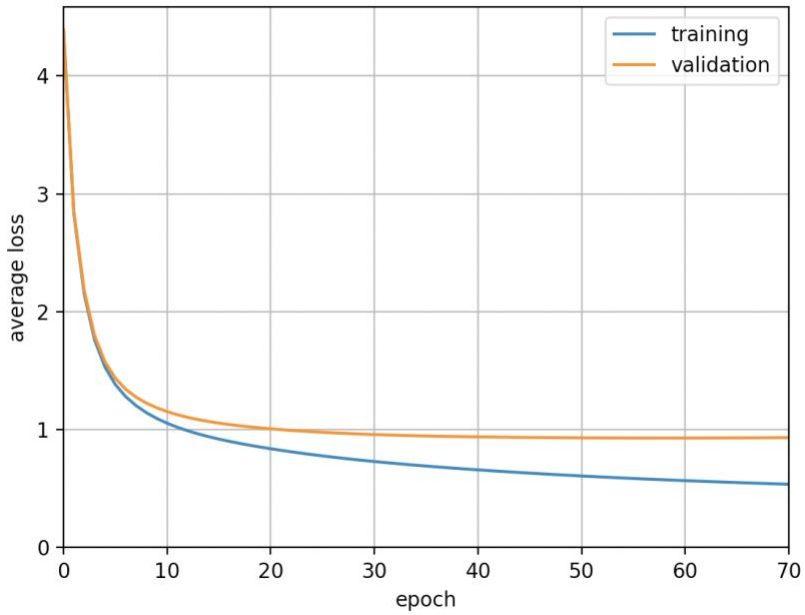
```python
############################## Q 2.4 ##############################
# split x and y into random batches
# return a list of [(batch1_x,batch1_y)...]
def get_random_batches(x,y,batch_size):
    batches = []
    examples = x.shape[0]
    num_batches = int(examples/batch_size)

    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
```

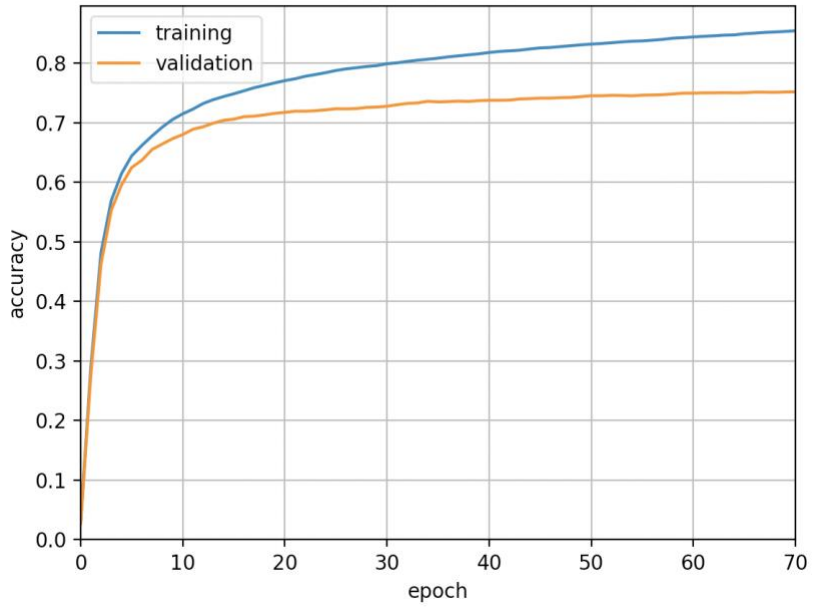
```python
# Q 2.4
batches = get_random_batches(x,y,5)
# print batch sizes
print([_[0].shape[0] for _ in batches])
batch_num = len(batches)

# 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

```python
# 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

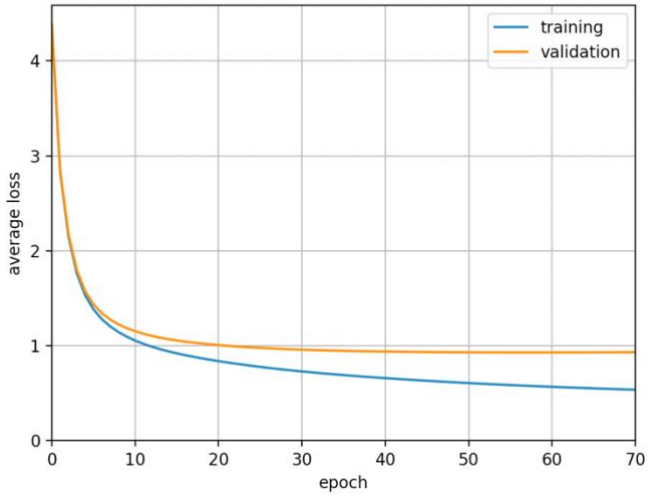


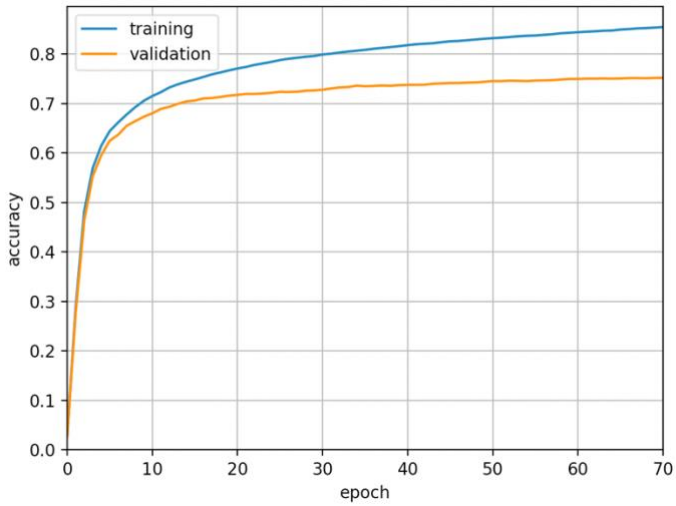


Epochs: 70
Batch Size: 32
Learning Rate: 0.003
Hidden Size: 64

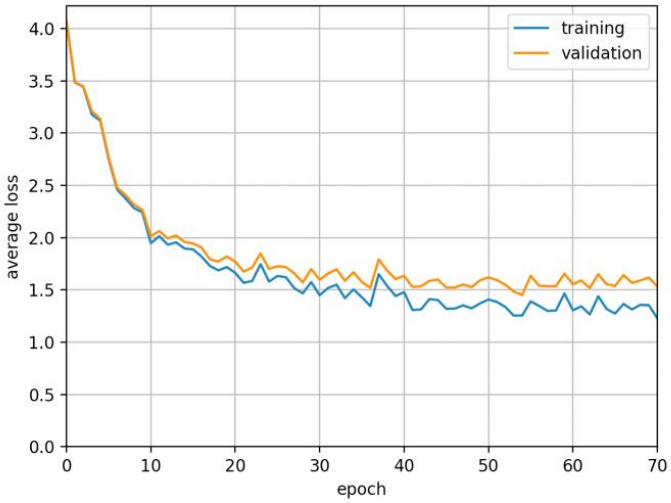
3.2 Writeup

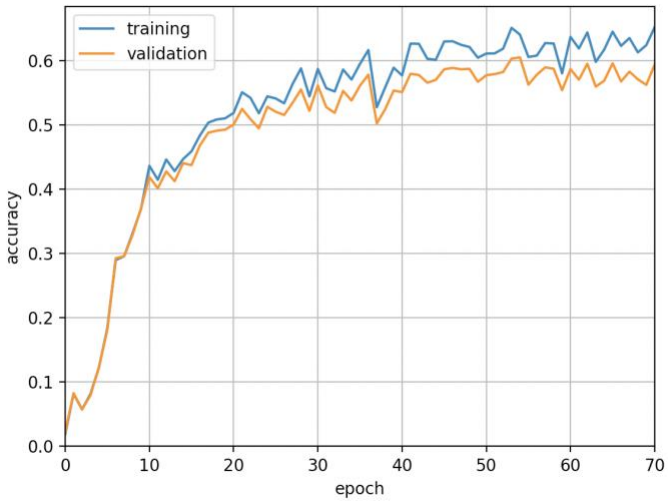
Tuned Learning Rate (from 3.2):



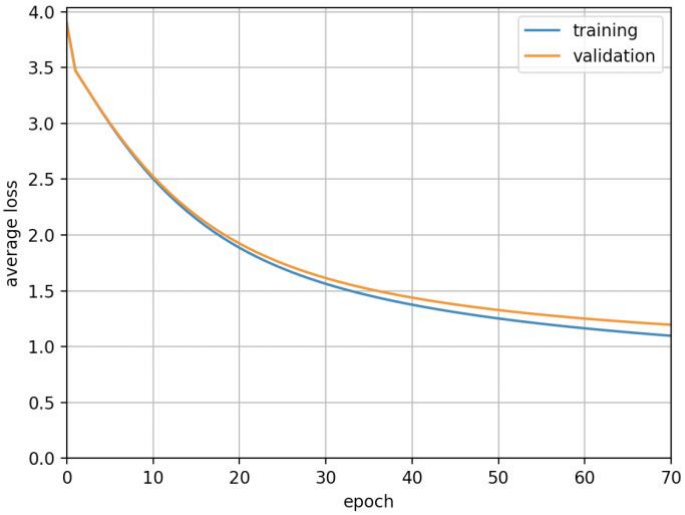


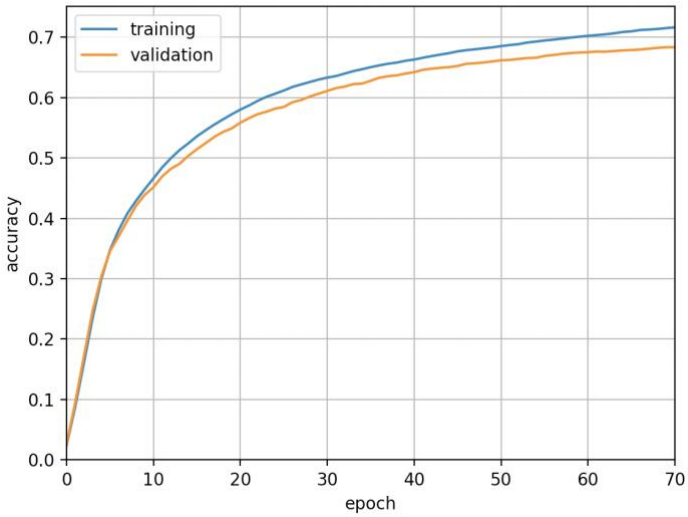
10x learning rate:





1/10th learning rate:

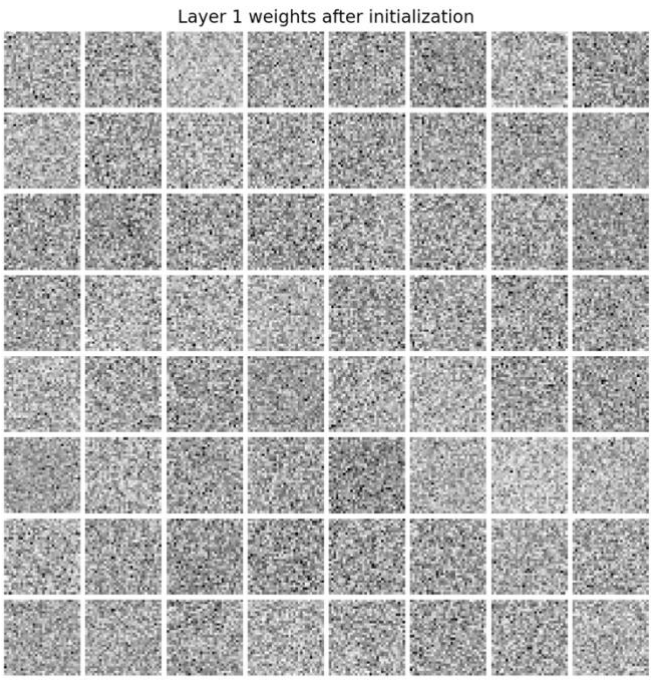




From the plots shown above, we can see that the learning rate has a direct impact on final accuracy and loss values. The images show that an optimal learning rate will gradually converge to the minimum to learn the ideal parameters in order to minimize loss and have the best accuracy. When the learning 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.

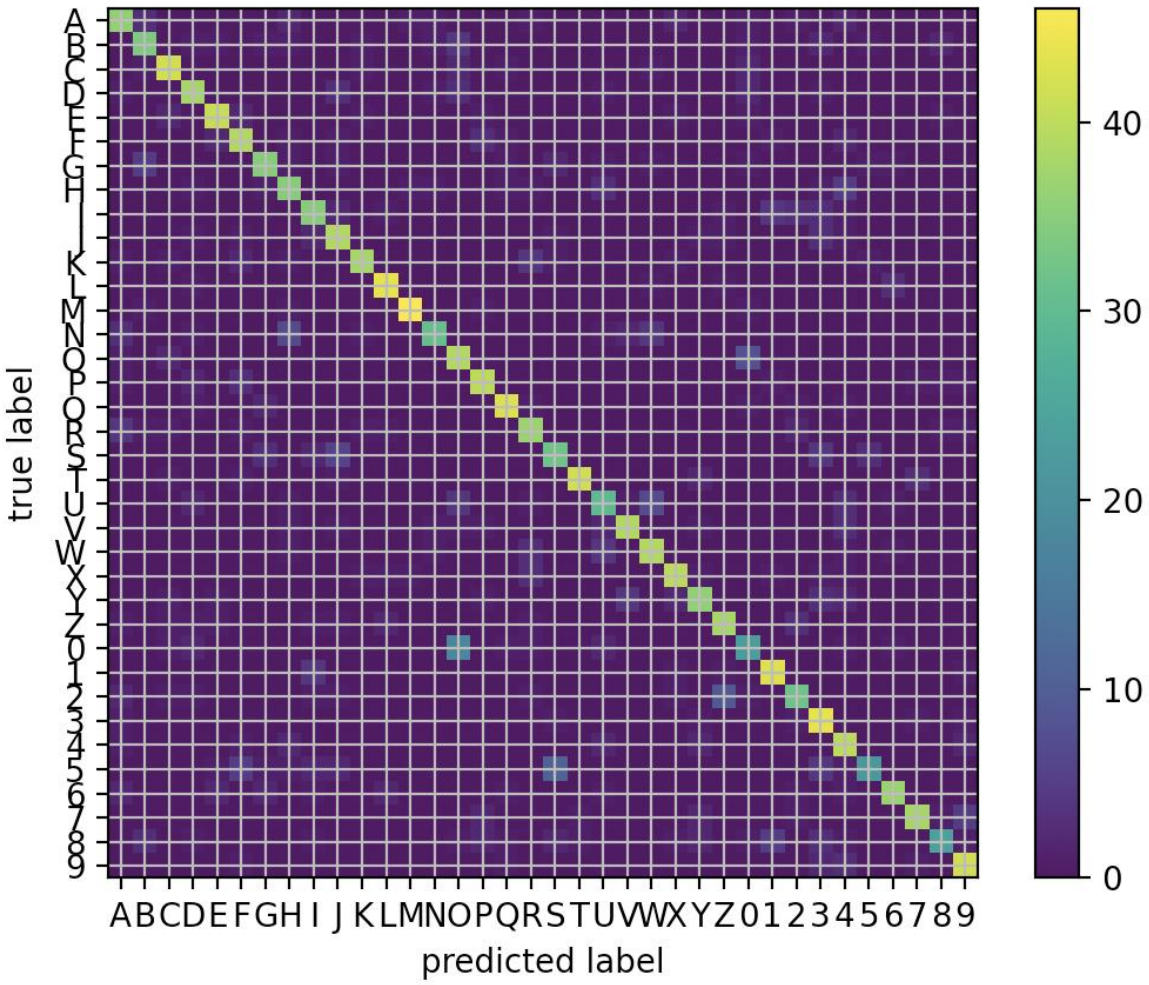
Best Final Training Accuracy = 75.4%
Best Final Test Accuracy = 76.3%

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.

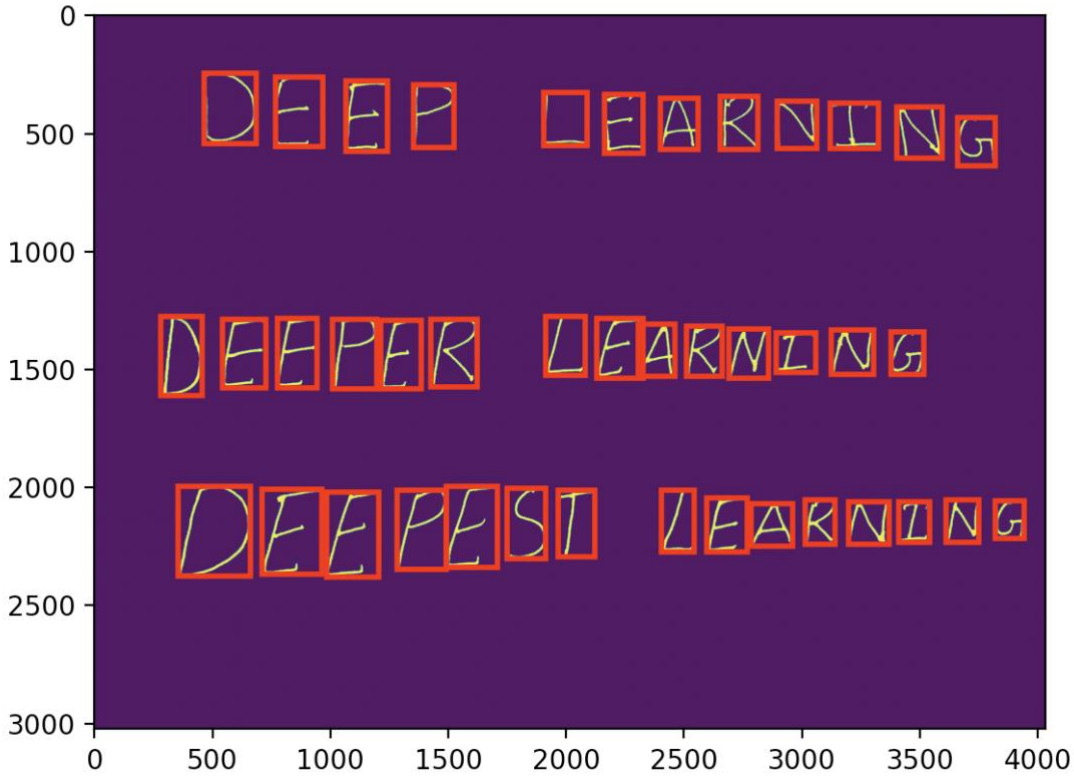
4.1 Theory

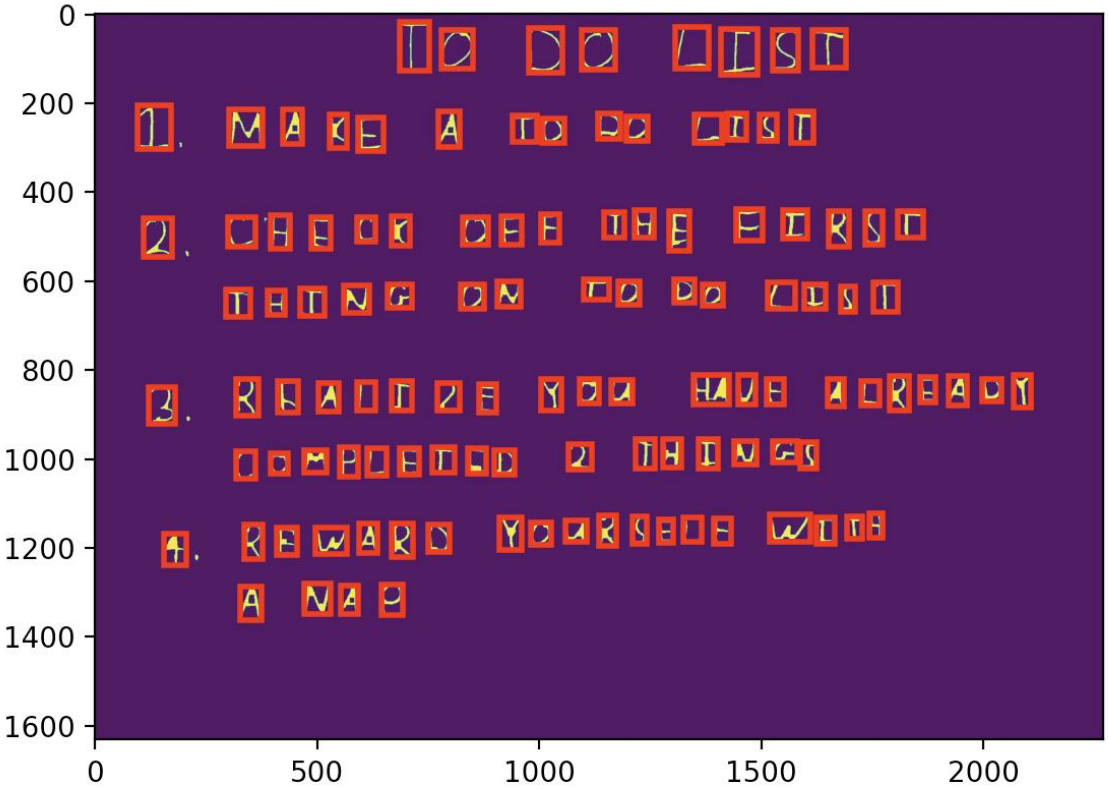
The method assumes that the letters are isolated from each other and that each letter forms a connected component. It also assumes that the letters are all similar sizes. Below are some examples of letters that may fail since they could be disconnected, connected to different letters, or of varying sizes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

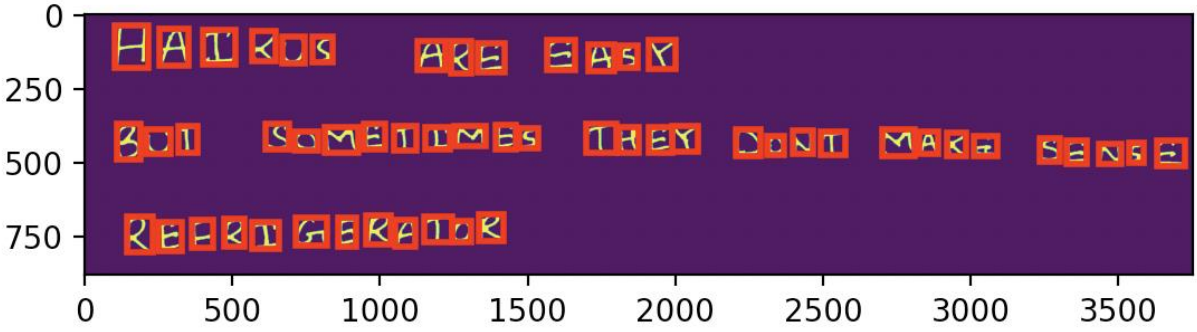
4.2 Code

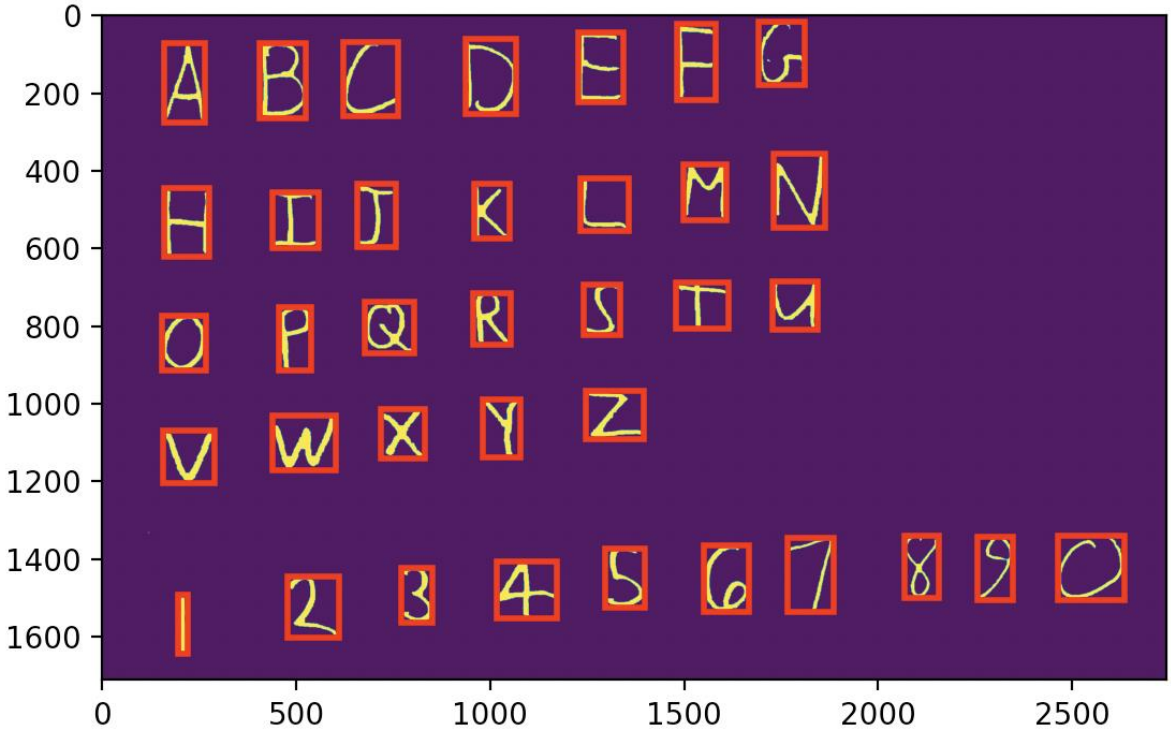
```
19  def findLetters(image):
20      image = skimage.color.rgb2gray(image)
21      thresh = threshold_otsu(image)
22      bw = closing(image < thresh, square(10))
23      cleared = clear_border(bw)
24      label_image = label(cleared)
25      bboxes = []
26      for region in regionprops(label_image):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

4.3 Writeup


DEEP LEARNING
DEEPER LEARNING
DEEPEST LEARNING

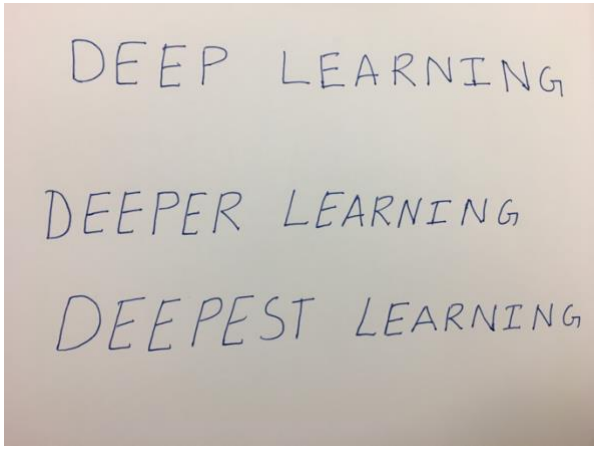

TO DO LIST
1. MAKE A TO DO LIST
2. CHECK OFF THE FIRST THING ON TO DO LIST
3. REALIZE YOU HAVE ACCOMPLISHED 2 THINGS
4. REWARD YOURSELF WITH A NAP


HAIKUS ARE EASY
BUT SOMETIMES THEY DONT MAKE SENSE
REERT GERATOR
0
250
500
750
0
500
1000
1500
2000
2500
3000
3500

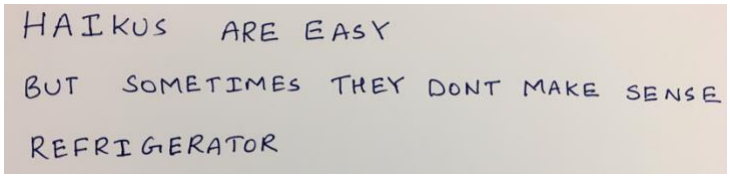

0
200
400
600
800
1000
1200
1400
1600
0
500
1000
1500
2000
2500

4.4 Code / Writeup

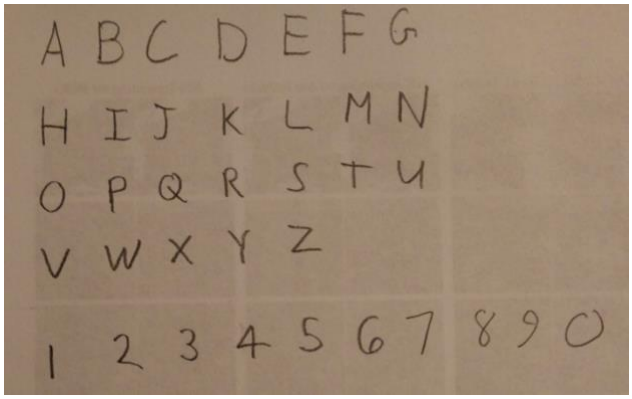
Below is the extracted text with each of the associated images



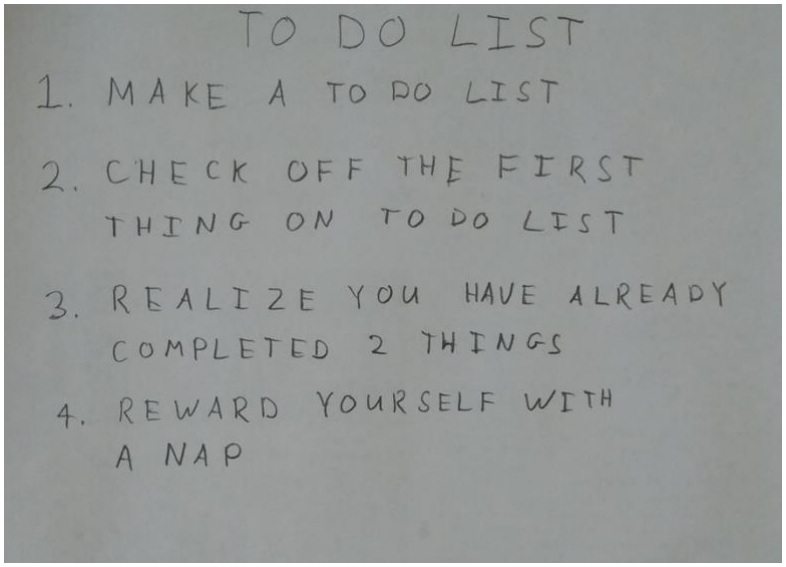

```
DEEPLEARMIN4
DEEPERLEARRIN6
DEEYESTLEARNING
```




```
HAIHUSAREEASX
ELTSCMETIM8STHEXDONTMAKRSEMSE
REFRI68RAT0R
```




```
XBCUET6
HIIKLMH
0PRRSYH
V4XYZ
1X34SG7XYO
```




```
T0D0LIST
IMAKEAY0Q06ZSY
2DH8RK0PBYH8P8RSY
YMZNA0MY08QLD8Y
DR8ALZ2RYQ8MRRALRRAMY
8QNPLQTKB2Y48NAS
4RRVARDY0RRQQLPH8YQ
ANAP
```

Code Snippets:

```python
import os
import matplotlib.pyplot as plt
import matplotlib.patches
import pickle
import string
import cv2
import skimage.io
from q4 import *
import numpy as np
from nn import *
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=UserWarning)

def get_crop(bbox, bw, image_it):
    y1, x1, y2, x2 = bbox
    image = bw[y1:y2, x1:x2]
    w = x2 - x1
    h = y2 - y1
    if h > w:
        padding = int(h / 2)
        image = np.pad(image, (padding, padding), "constant")
    elif w > h:
        padding = int(w / 2)
        image = np.pad(image, (padding, padding), "constant")
    idx_0 = np.where(image == 0)
    idx_1 = np.where(image == 1)
    image[idx_0] = 1
    image[idx_1] = 0
    if image_it == 0:
        erosion_amt = 30
    elif image_it == 1:
        erosion_amt = 17
    elif image_it == 2:
        erosion_amt = 17
    else:
        erosion_amt = 7
    image = cv2.erode(image.astype(float), np.ones((erosion_amt, erosion_amt)), iterations=1)
    image = cv2.resize(image.T, (32, 32)).flatten()
    return image
```

```python

def sort_boxes(box_arr):
    x_arr = []
    for i in box_arr:
        x_arr.append(i[1])
    x_arr.sort()
    res = []
    for i in x_arr:
        for j in box_arr:
            if i == j[1]:
                res.append(j)
    return res

img_count = 0
for img in os.listdir('../images'):
    im1 = skimage.img_as_float(skimage.io.imread(os.path.join('../images',img)))
    bboxes, bw = findLetters(im1)
    plt.imshow(bw)
    for bbox in bboxes:
        minr, minc, maxr, maxc = bbox
        rect = matplotlib.patches.Rectangle((minc, minr), maxc - minc, maxr - minr, fill=False, edgecolor='red', linewidth=2)
        plt.gca().add_patch(rect)
    plt.show()

    # find the rows using..RANSAC, counting, clustering, etc.
    rows = []
    bboxes_in_row = [bboxes[0]]
    for i in range(1,len(bboxes)):
        bbox = bboxes[i]
        y1, x1, y2, x2 = bbox
        yc = (y1 + y2) / 2
        bbox_first = bboxes_in_row[-1]
        if yc < bbox_first[0] or yc > bbox_first[2]:
            rows.append(bboxes_in_row)
            bboxes_in_row = []
            bboxes_in_row.append(bbox)
        else:
            bboxes_in_row.append(bbox)
    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     for r in rows:
101         r = np.vstack(r)
102         h1 = forward(r, params, 'layer1')
103         probs = forward(h1, params, 'output', softmax)
104         letters_in_row = ""
105         for i in range(probs.shape[0]):
106             pred = np.argmax(probs[i, :])
107             letters_in_row += map_char[pred]
108         print(letters_in_row)
109     print("\n")
110     img_count+=1
111
```

6.1 Train a Neural Network in Pytorch

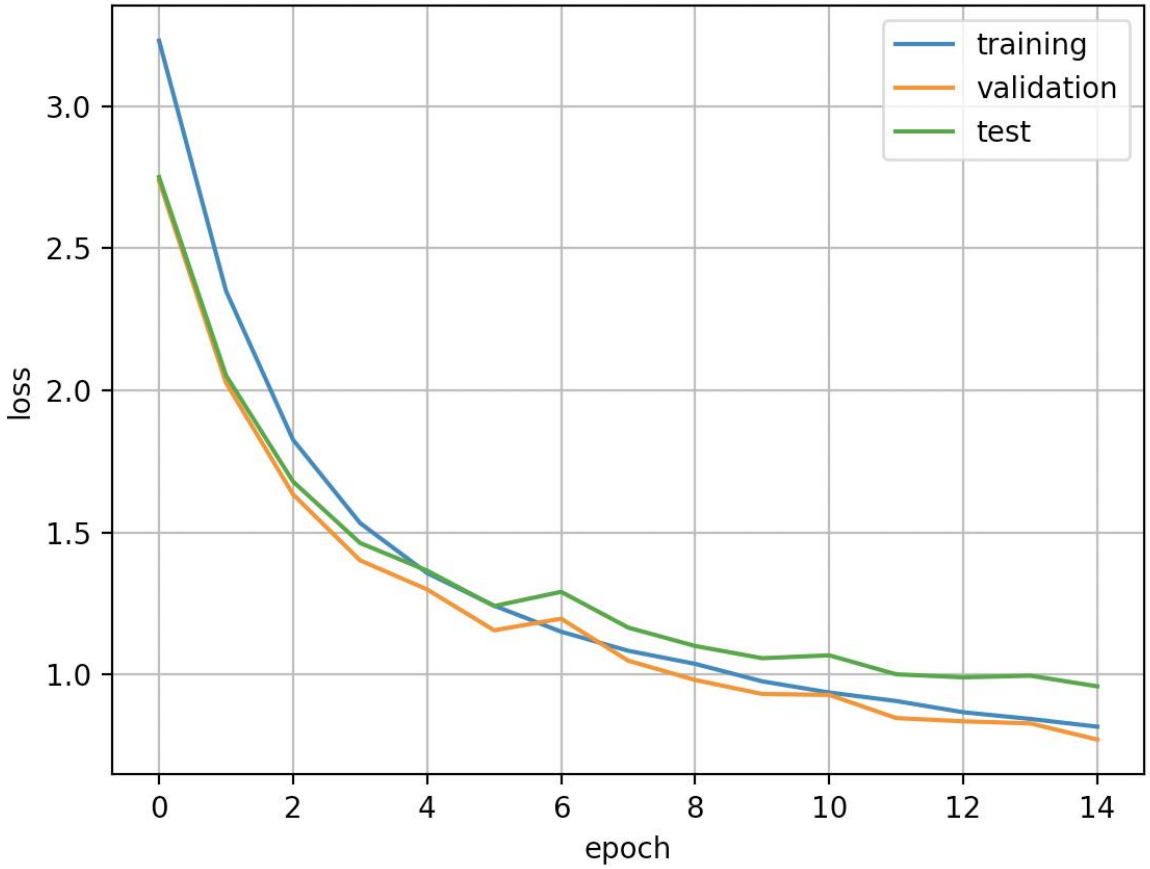
6.1.1 Code/Writeup

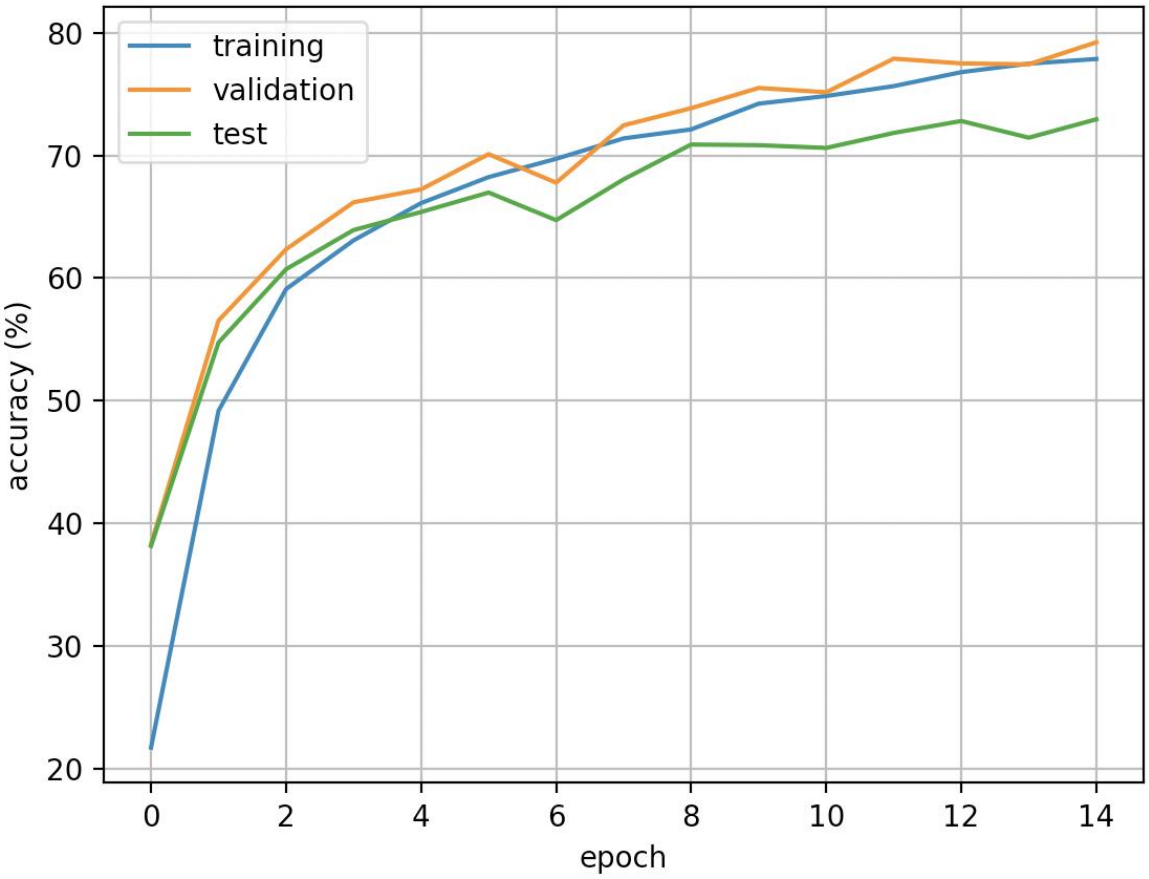
```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.io

import torch
from torch import nn
from torch.utils.data import DataLoader, TensorDataset

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

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


# convert to Tensor
train_x = torch.from_numpy(train_x).to(torch.float32)
train_y = np.argmax(train_y, axis=1)
train_y = torch.from_numpy(train_y)
train_data = TensorDataset(train_x,train_y)

# convert to Tensor
test_x = torch.from_numpy(test_x).to(torch.float32)
test_y = np.argmax(test_y, axis=1)
test_y = torch.from_numpy(test_y)
test_data = TensorDataset(train_x,train_y)

# convert to Tensor
valid_x = torch.from_numpy(valid_x).to(torch.float32)
valid_y = np.argmax(valid_y, axis=1)
valid_y = torch.from_numpy(valid_y)
valid_data = TensorDataset(valid_x,valid_y)

batch_size = 32

# Create data loaders
train_dataloader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
valid_dataloader = DataLoader(valid_data, batch_size=batch_size, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=batch_size, shuffle=True)

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

```python
# Define model
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(1024, 64),
            nn.Sigmoid(),
            nn.Linear(64, 36))

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

model = NeuralNetwork().to(device)
print(model)

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

def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.train()
    train_loss = 0
    correct = 0
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)
        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)
        train_loss = train_loss + loss.item()
        correct += (pred.argmax(1) == y).type(torch.float).sum().item()
        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
    correct /= size
    train_loss /= num_batches
    print(f"Train Accuracy: {(100 * correct):>0.1f}%")
    return 100*correct, train_loss
```

```python


def test(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)
            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)
            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")

```

```python
epochs = 15
train_acc = []
train_loss = []
validate_acc = []
validate_loss = []
test_acc = []
test_loss = []
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    tr_acc, tr_loss = train(train_dataloader, model, loss_fn, optimizer)
    train_acc.append(tr_acc)
    train_loss.append(tr_loss)
    val_acc, val_loss = validate(test_dataloader, model, loss_fn)
    validate_acc.append(val_acc)
    validate_loss.append(val_loss)
    te_acc, te_loss = test(valid_dataloader, model, loss_fn)
    test_acc.append(te_acc)
    test_loss.append(te_loss)
print("Done!")

# plot loss
plt.plot(range(len(train_loss)), train_loss, label="training")
plt.plot(range(len(validate_loss)), validate_loss, label="validation")
plt.plot(range(len(test_loss)), test_loss, label="test")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.legend()
plt.grid()
plt.show()

# plot training accuracy
plt.plot(range(len(train_acc)), train_acc, label="training")
plt.plot(range(len(validate_acc)), validate_acc, label="validation")
plt.plot(range(len(test_acc)), test_acc, label="test")
plt.xlabel("epoch")
plt.ylabel("accuracy (%)")
plt.legend()
plt.grid()
plt.show()
```


training
validation
test
loss
epoch
3.0
2.5
2.0
1.5
1.0
0
2
4
6
8
10
12
14


training
validation
test
accuracy (%)
epoch
80
70
60
50
40
30
20
0
2
4
6
8
10
12
14

6.1.2 Code / Writeup

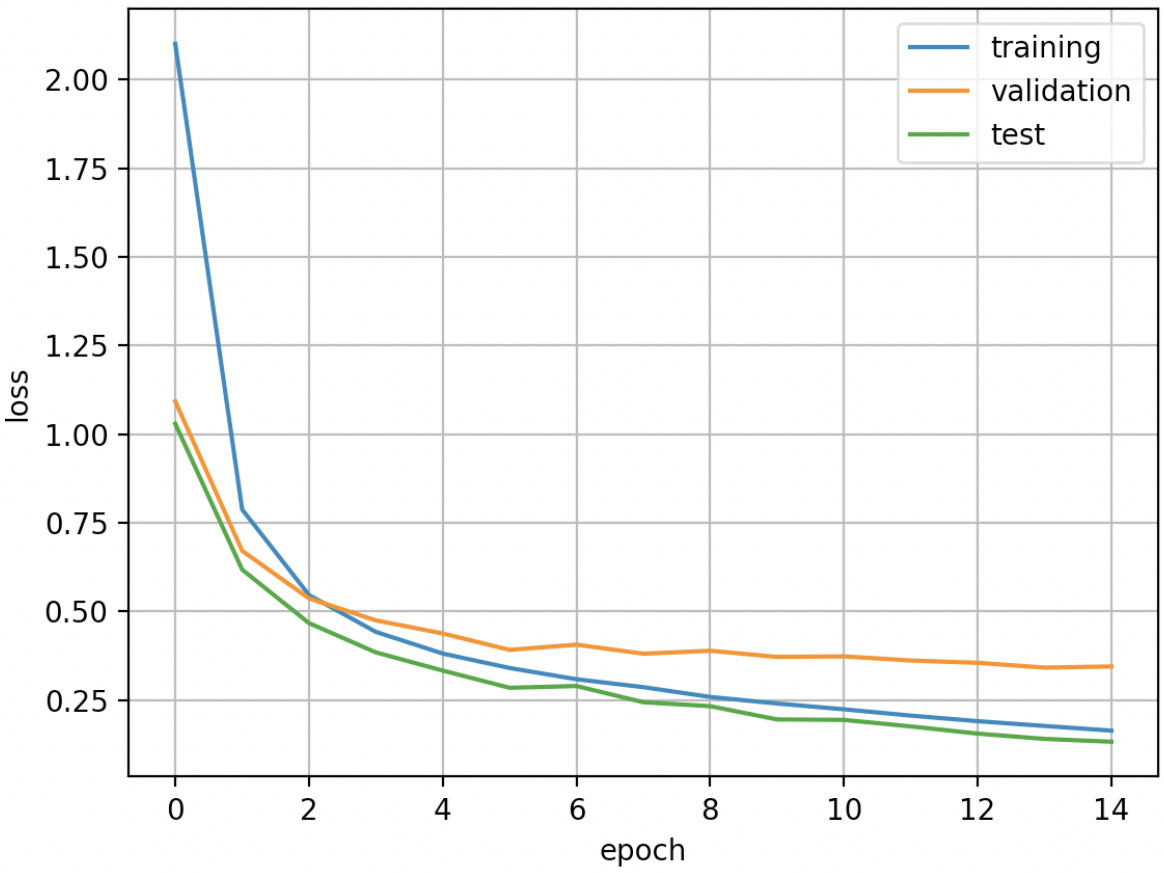
```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.io
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, TensorDataset

# load data
train_data = scipy.io.loadmat('../data/nist36_train.mat')
valid_data = scipy.io.loadmat('../data/nist36_valid.mat')
test_data = scipy.io.loadmat('../data/nist36_test.mat')
train_x, train_y = train_data['train_data'], train_data['train_labels']
valid_x, valid_y = valid_data['valid_data'], valid_data['valid_labels']
test_x, test_y = test_data['test_data'], test_data['test_labels']

# convert to Tensor
train_x = torch.from_numpy(train_x).to(torch.float32)
train_y = np.argmax(train_y, axis=1)
train_y = torch.from_numpy(train_y)
train_data = TensorDataset(train_x,train_y)
# convert to Tensor
test_x = torch.from_numpy(test_x).to(torch.float32)
test_y = np.argmax(test_y, axis=1)
test_y = torch.from_numpy(test_y)
test_data = TensorDataset(train_x,train_y)
# convert to Tensor
valid_x = torch.from_numpy(valid_x).to(torch.float32)
valid_y = np.argmax(valid_y, axis=1)
valid_y = torch.from_numpy(valid_y)
valid_data = TensorDataset(valid_x,valid_y)

batch_size = 32

# Create data loaders
train_dataloader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
valid_dataloader = DataLoader(valid_data, batch_size=batch_size, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=batch_size, shuffle=True)

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

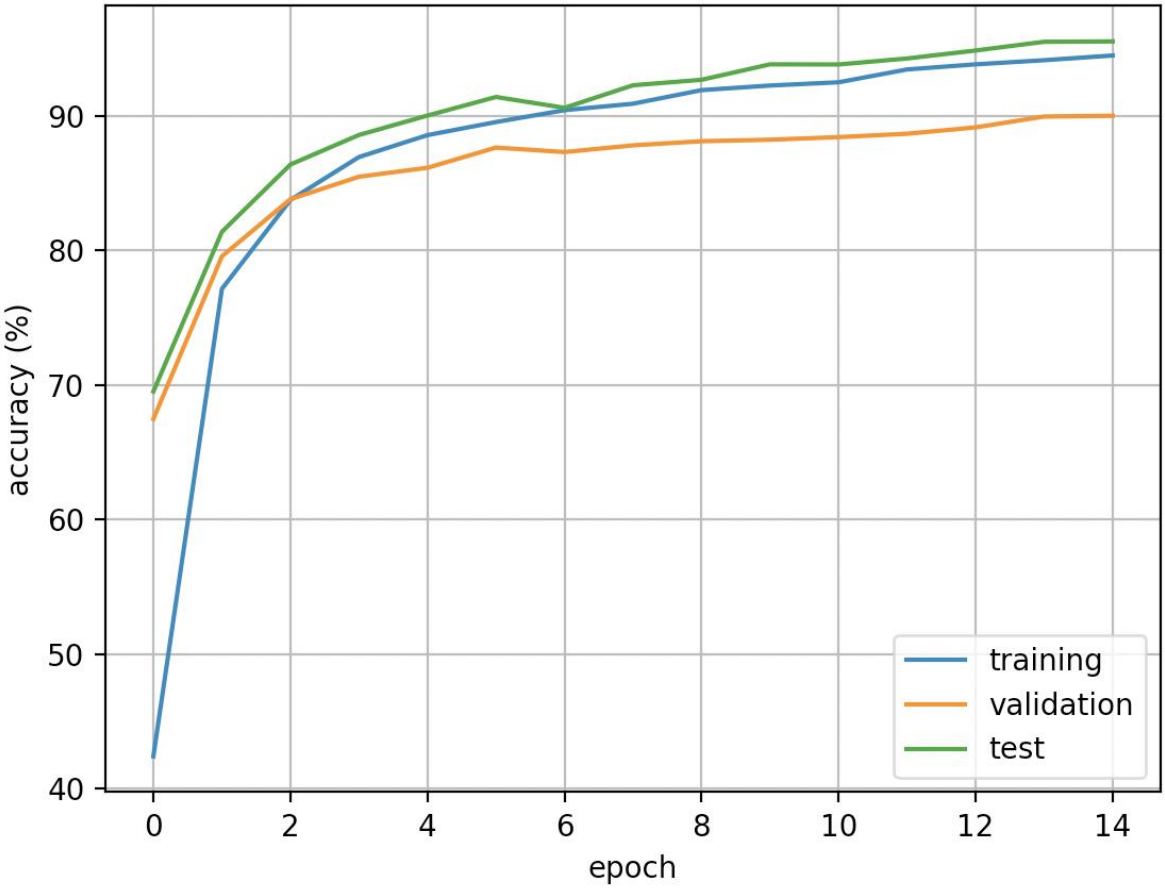
```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 7, 3)
        self.conv2 = nn.Conv2d(7, 13, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(325, 84)  # 5*5 from image dimension
        self.fc2 = nn.Linear(84, 36)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square, you can specify with a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), (2, 2))
        x = torch.flatten(x, 1) # flatten all dimensions except the batch dimension
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = Net().to(device)
print(model)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

```python
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.train()
    train_loss = 0
    correct = 0
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)
        X = X.view(-1, 1, 32, 32)
        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)
        train_loss = train_loss + loss.item()
        correct += (pred.argmax(1) == y).type(torch.float).sum().item()
        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
    correct /= size
    train_loss /= num_batches
    print(f"Train Accuracy: {(100 * correct):>0.1f}%")
    return 100*correct, train_loss
```

```python
def test(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"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")

```

```python

epochs = 15
train_acc = []
train_loss = []
validate_acc = []
validate_loss = []
test_acc = []
test_loss = []
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    tr_acc, tr_loss = train(train_dataloader, model, loss_fn, optimizer)
    train_acc.append(tr_acc)
    train_loss.append(tr_loss)
    val_acc, val_loss = validate(valid_dataloader, model, loss_fn)
    validate_acc.append(val_acc)
    validate_loss.append(val_loss)
    te_acc, te_loss = test(test_dataloader, model, loss_fn)
    test_acc.append(te_acc)
    test_loss.append(te_loss)
print("Done!")

# plot loss
plt.plot(range(len(train_loss)), train_loss, label="training")
plt.plot(range(len(validate_loss)), validate_loss, label="validation")
plt.plot(range(len(test_loss)), test_loss, label="test")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.legend()
plt.grid()
plt.show()

# plot training accuracy
plt.plot(range(len(train_acc)), train_acc, label="training")
plt.plot(range(len(validate_acc)), validate_acc, label="validation")
plt.plot(range(len(test_acc)), test_acc, label="test")
plt.xlabel("epoch")
plt.ylabel("accuracy (%)")
plt.legend()
plt.grid()
plt.show()
```


training
validation
test
loss
epoch
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0
2
4
6
8
10
12
14

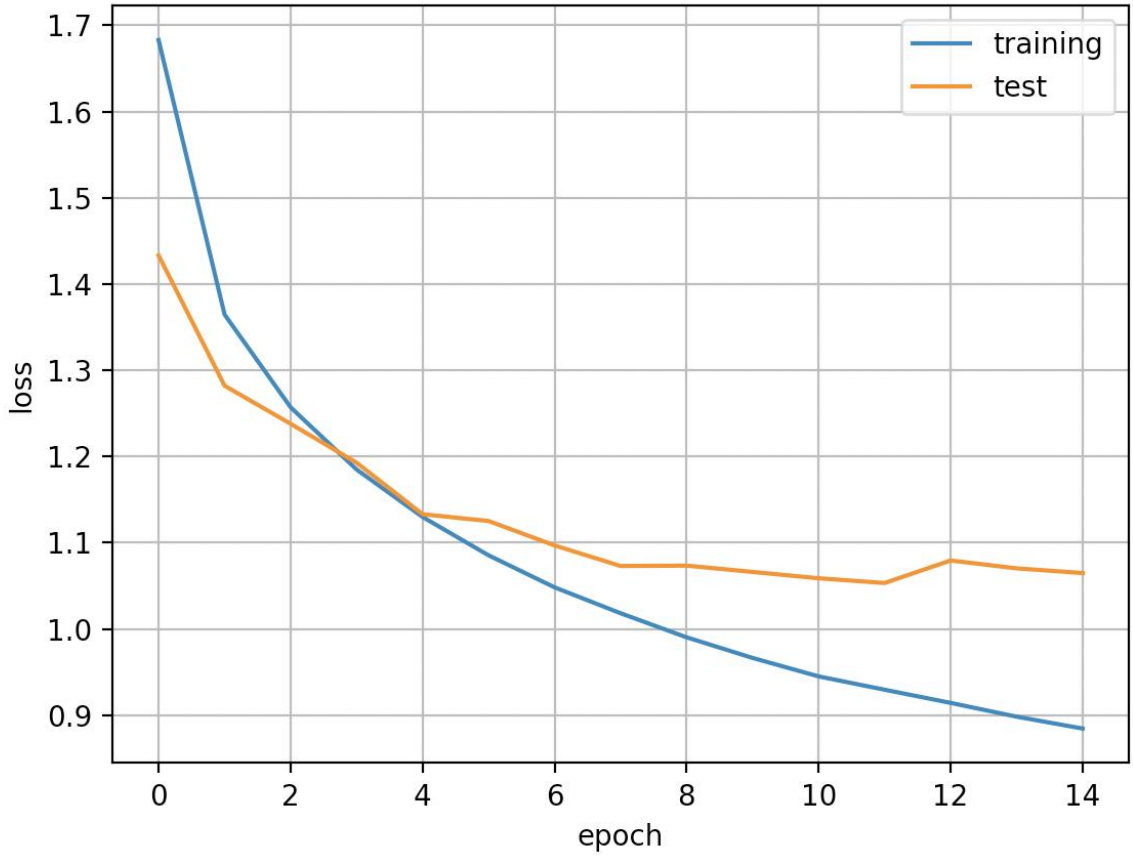

training
validation
test
accuracy (%)
epoch
90
80
70
60
50
40
0
2
4
6
8
10
12
14

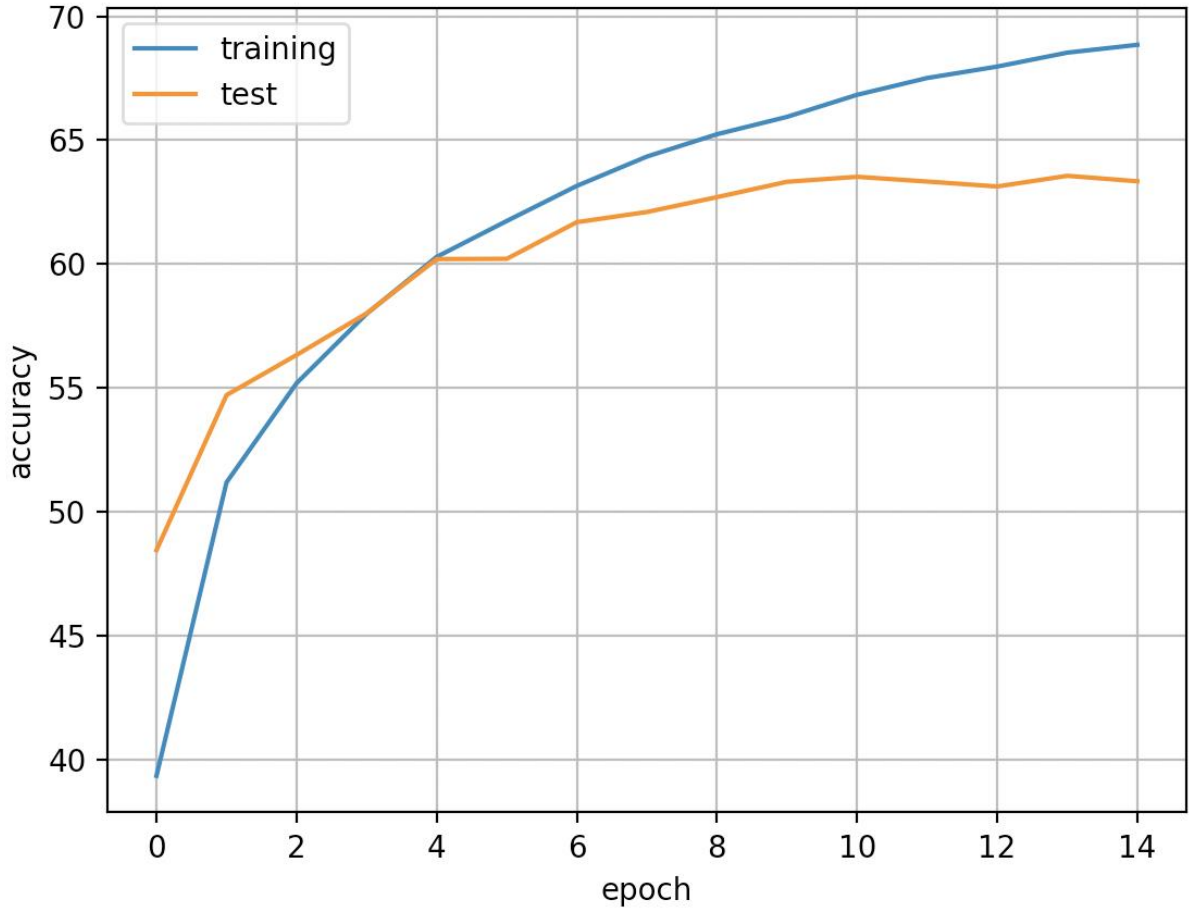
6.1.3 Code / Writeup

```python
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
import torchvision
import torchvision.transforms as transforms

transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

batch_size = 36

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
train_dataloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True)
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
test_dataloader = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=False)
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

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


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 7, 3)
        self.conv2 = nn.Conv2d(7, 13, 5)
        self.fc1 = nn.Linear(325, 84)
        self.fc2 = nn.Linear(84, 36)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square, you can specify with a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), (2, 2))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = Net().to(device)
print(model)
```

```python
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.train()
    train_loss = 0
    correct = 0
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)
        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)
        train_loss = train_loss + loss.item()
        correct += (pred.argmax(1) == y).type(torch.float).sum().item()
        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

    correct /= size
    train_loss /= num_batches
    print(f"Train Accuracy: {(100 * correct):>0.1f}%")

    return 100*correct, train_loss


def test(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)
            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")

```

```python

epochs = 15
train_acc = []
train_loss = []
test_acc = []
test_loss = []
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    tr_acc, tr_loss = train(train_dataloader, model, loss_fn, optimizer)
    train_acc.append(tr_acc)
    train_loss.append(tr_loss)
    te_acc, te_loss = test(test_dataloader, model, loss_fn)
    test_acc.append(te_acc)
    test_loss.append(te_loss)
print("Done!")

# plot loss
plt.plot(range(len(train_loss)), train_loss, label="training")
plt.plot(range(len(test_loss)), test_loss, label="test")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.legend()
plt.grid()
plt.show()

# plot training accuracy
plt.plot(range(len(train_acc)), train_acc, label="training")
plt.plot(range(len(test_acc)), test_acc, label="test")
plt.xlabel("epoch")
plt.ylabel("accuracy")
plt.legend()
plt.grid()
plt.show()
```


training
test
loss
epoch
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1.0
0.9
0
2
4
6
8
10
12
14

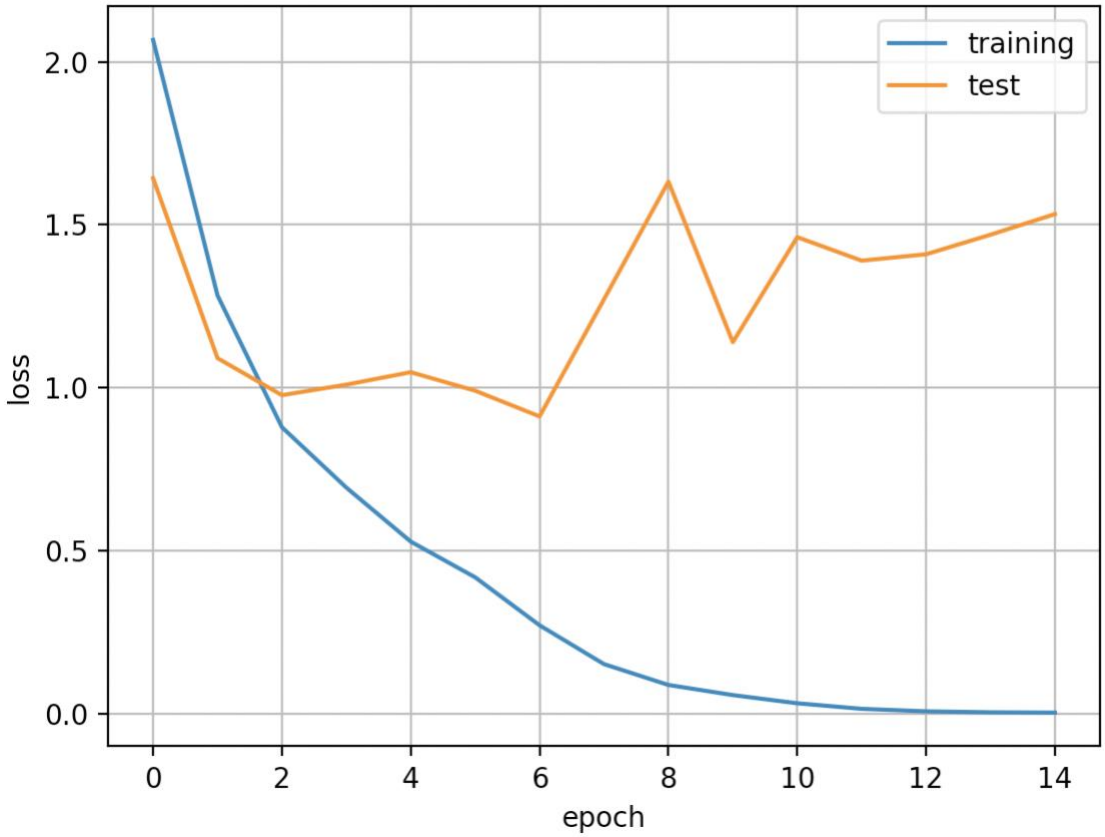

training
test
accuracy
epoch
70
65
60
55
50
45
40
0
2
4
6
8
10
12
14

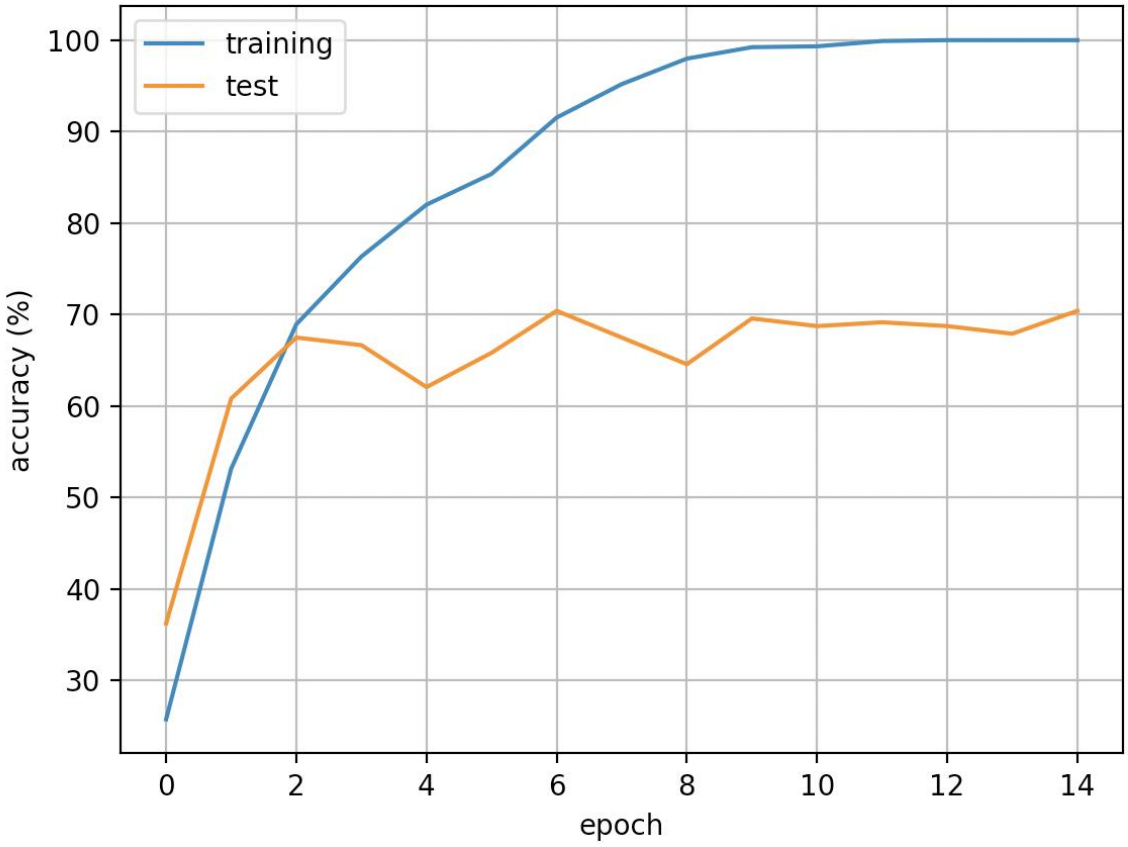
6.1.4 Code / Writeup

```python
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
import torchvision
import torchvision.transforms as transforms

transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
     transforms.Resize((256,256))])

batch_size = 36

trainset = torchvision.datasets.ImageFolder(root='../Train', transform=transform)
train_dataloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,shuffle=True)
testset = torchvision.datasets.ImageFolder(root='../Test', transform=transform)
test_dataloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,shuffle=False)

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

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 7, 3)
        self.conv2 = nn.Conv2d(7, 13, 5)
        self.fc1 = nn.Linear(48373, 84)
        self.fc2 = nn.Linear(84, 8)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square, you can specify with a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), (2, 2))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = Net().to(device)
print(model)
```

```python
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.train()
    train_loss = 0
    correct = 0
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)
        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)
        train_loss = train_loss + loss.item()
        correct += (pred.argmax(1) == y).type(torch.float).sum().item()
        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
    correct /= size
    train_loss /= num_batches
    print(f"Train Accuracy: {(100 * correct):>0.1f}%")
    return 100*correct, train_loss

def test(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)
            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")
```

```python
epochs = 15
train_acc = []
train_loss = []
test_acc = []
test_loss = []
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    tr_acc, tr_loss = train(train_dataloader, model, loss_fn, optimizer)
    train_acc.append(tr_acc)
    train_loss.append(tr_loss)
    te_acc, te_loss = test(test_dataloader, model, loss_fn)
    test_acc.append(te_acc)
    test_loss.append(te_loss)
print("Done!")

# plot loss
plt.plot(range(len(train_loss)), train_loss, label="training")
plt.plot(range(len(test_loss)), test_loss, label="test")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.legend()
plt.grid()
plt.show()

# plot training accuracy
plt.plot(range(len(train_acc)), train_acc, label="training")
plt.plot(range(len(test_acc)), test_acc, label="test")
plt.xlabel("epoch")
plt.ylabel("accuracy (%)")
plt.legend()
plt.grid()
plt.show()
```


training
test
loss
epoch
2.0
1.5
1.0
0.5
0.0
0
2
4
6
8
10
12
14


training
test
accuracy (%)
epoch
100
90
80
70
60
50
40
30
0
2
4
6
8
10
12
14

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

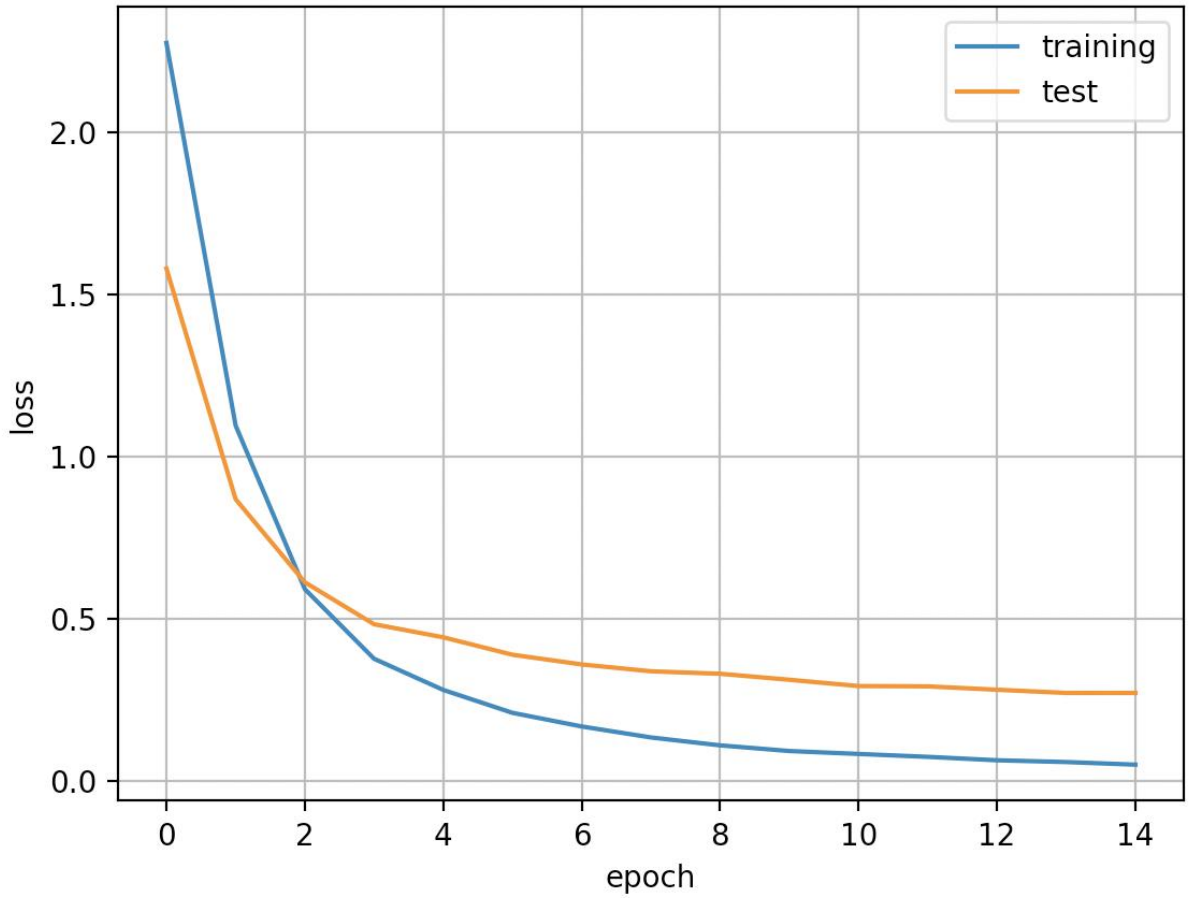
6.2 Fine Tuning Code / Writeup

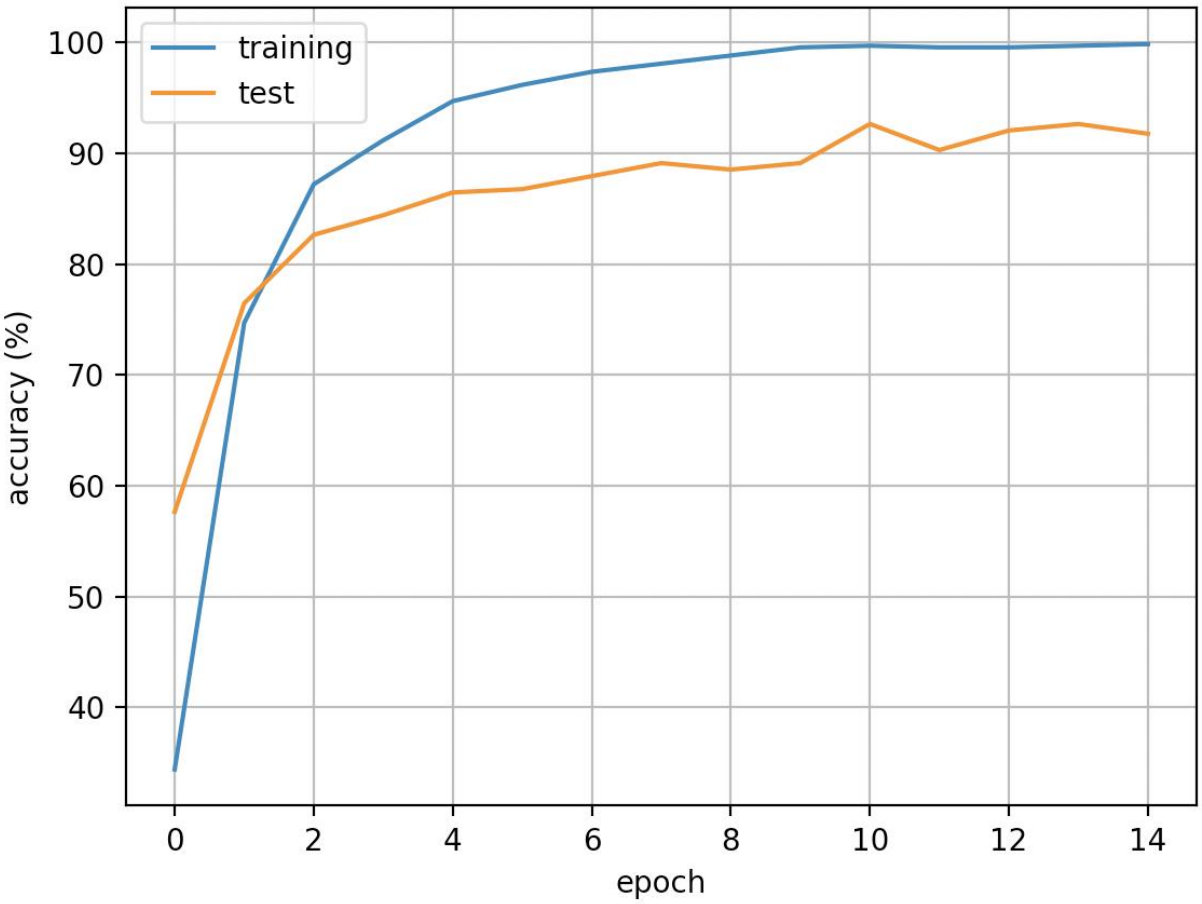
Squeezenet Code:

```python
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
import torchvision
import torchvision.transforms as transforms
from torchvision.models import squeezenet1_1

transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
     transforms.Resize((224,224))])
batch_size = 36

trainset = torchvision.datasets.ImageFolder(root='../data/oxford-flowers17/train', transform=transform)
train_dataloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,shuffle=True)
testset = torchvision.datasets.ImageFolder(root='../data/oxford-flowers17/test', transform=transform)
test_dataloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,shuffle=False)

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

model = squeezenet1_1(pretrained=True).to(device)

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

final_conv = nn.Conv2d(512,17,kernel_size=1)
model.classifier = nn.Sequential(nn.Dropout(p=0.5), final_conv, nn.ReLU(inplace=True), nn.AdaptiveAvgPool2d((1,1)))

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

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

```python
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.train()
    train_loss = 0
    correct = 0
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)
        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)
        train_loss = train_loss + loss.item()
        correct += (pred.argmax(1) == y).type(torch.float).sum().item()
        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

    correct /= size
    train_loss /= num_batches
    print(f"Train Accuracy: {(100 * correct):>0.1f}%")

    return 100*correct, train_loss


def test(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)
            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")
```

```python
epochs = 15
train_acc = []
train_loss = []
test_acc = []
test_loss = []
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    tr_acc, tr_loss = train(train_dataloader, model, loss_fn, optimizer)
    train_acc.append(tr_acc)
    train_loss.append(tr_loss)
    te_acc, te_loss = test(test_dataloader, model, loss_fn)
    test_acc.append(te_acc)
    test_loss.append(te_loss)
print("Done!")

# plot loss
plt.plot(range(len(train_loss)), train_loss, label="training")
plt.plot(range(len(test_loss)), test_loss, label="test")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.legend()
plt.grid()
plt.show()

# plot training accuracy
plt.plot(range(len(train_acc)), train_acc, label="training")
plt.plot(range(len(test_acc)), test_acc, label="test")
plt.xlabel("epoch")
plt.ylabel("accuracy (%)")
plt.legend()
plt.grid()
plt.show()
```


training
test
loss
epoch
0.0
0.5
1.0
1.5
2.0
0
2
4
6
8
10
12
14


training
test
accuracy (%)
epoch
40
50
60
70
80
90
100
0
2
4
6
8
10
12
14

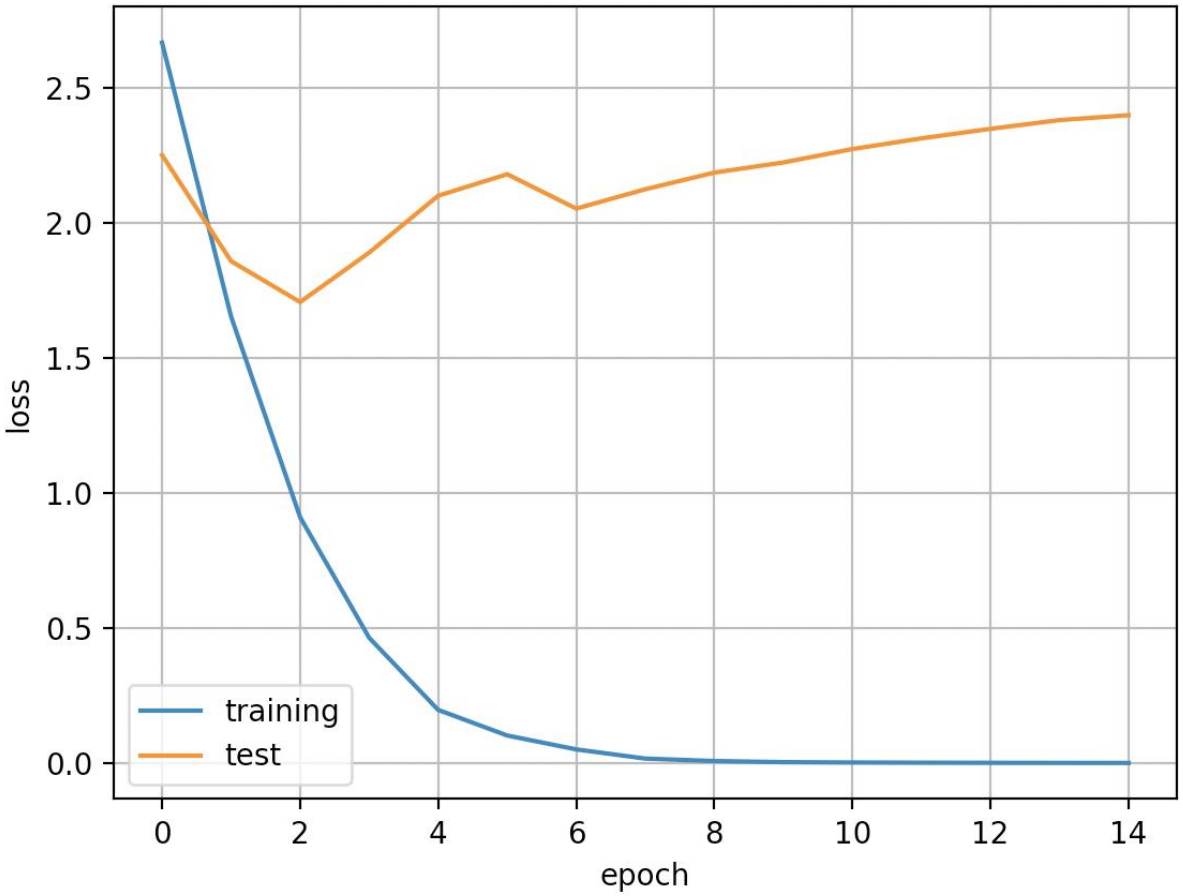
Custom Architecture:

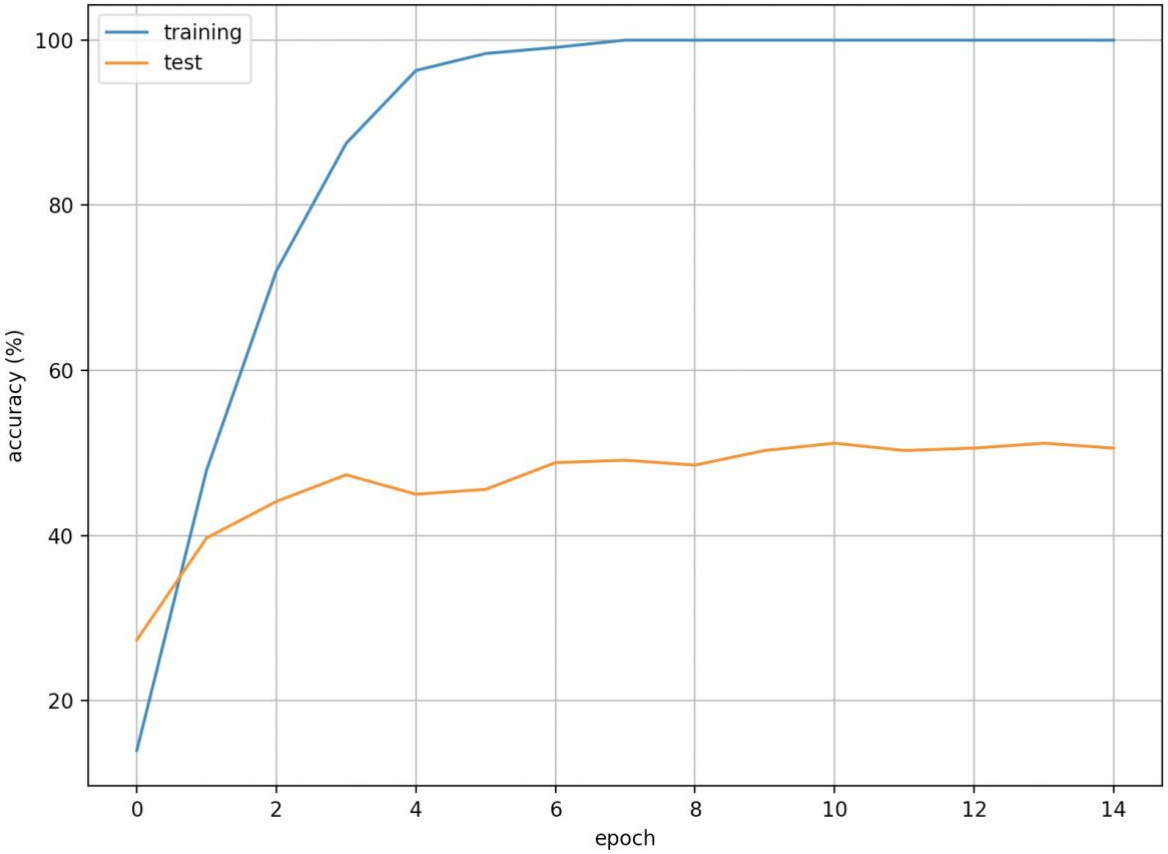
```python
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
import torchvision
import torchvision.transforms as transforms

transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
     transforms.Resize((224,224))])

batch_size = 36

trainset = torchvision.datasets.ImageFolder(root='../data/oxford-flowers17/train', transform=transform)
train_dataloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,shuffle=True)
# validset = torchvision.datasets.ImageFolder(root='../data/oxford-flowers17/val', transform=transform)
# valid_dataloader = torch.utils.data.DataLoader(validset, batch_size=batch_size,shuffle=True)
testset = torchvision.datasets.ImageFolder(root='../data/oxford-flowers17/test', transform=transform)
test_dataloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,shuffle=False)


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

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 7, 3)
        self.conv2 = nn.Conv2d(7, 13, 5)
        self.fc1 = nn.Linear(36517, 84)
        self.fc2 = nn.Linear(84, 17)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square, you can specify with a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), (2, 2))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

```
        return x

model = Net().to(device)
print(model)

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

def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.train()
    train_loss = 0
    correct = 0
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)
        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)
        train_loss = train_loss + loss.item()
        correct += (pred.argmax(1) == y).type(torch.float).sum().item()
        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

    correct /= size
    train_loss /= num_batches
    print(f"Train Accuracy: {(100 * correct):>0.1f}%")

    return 100*correct, train_loss
```

```python

def test(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)
            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")

epochs = 15
train_acc = []
train_loss = []
# validate_acc = []
# validate_loss = []
test_acc = []
test_loss = []
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    tr_acc, tr_loss = train(train_dataloader, model, loss_fn, optimizer)
    train_acc.append(tr_acc)
    train_loss.append(tr_loss)
    # val_acc, val_loss = validate(valid_dataloader, model, loss_fn)
    # validate_acc.append(val_acc)
    # validate_loss.append(val_loss)
    te_acc, te_loss = test(test_dataloader, model, loss_fn)
    test_acc.append(te_acc)
    test_loss.append(te_loss)
print("Done!")
```

```python

epochs = 15
train_acc = []
train_loss = []
# validate_acc = []
# validate_loss = []
test_acc = []
test_loss = []
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    tr_acc, tr_loss = train(train_dataloader, model, loss_fn, optimizer)
    train_acc.append(tr_acc)
    train_loss.append(tr_loss)
    # val_acc, val_loss = validate(valid_dataloader, model, loss_fn)
    # validate_acc.append(val_acc)
    # validate_loss.append(val_loss)
    te_acc, te_loss = test(test_dataloader, model, loss_fn)
    test_acc.append(te_acc)
    test_loss.append(te_loss)
print("Done!")

# plot loss
plt.plot(range(len(train_loss)), train_loss, label="training")
plt.plot(range(len(test_loss)), test_loss, label="test")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.legend()
plt.grid()
plt.show()

# plot training accuracy
plt.plot(range(len(train_acc)), train_acc, label="training")
plt.plot(range(len(test_acc)), test_acc, label="test")
plt.xlabel("epoch")
plt.ylabel("accuracy (%)")
plt.legend()
plt.grid()
plt.show()
```


loss
epoch
training
test
0.0
0.5
1.0
1.5
2.0
2.5
0
2
4
6
8
10
12
14


accuracy (%)
epoch
training
test
20
40
60
80
100
0
2
4
6
8
10
12
14

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.