Computer Vision (16-720 B): Homework 5

Neural Networks for Recognition

Name - Saharsh Agarwal
AndrewID - saharsh2

Question 1 : Theory

Q 1.1

Answer -

$$softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{1}$$

Replacing (x_i) by $(x_i + c)$ for every i in equation 1:

$$softmax(x_i + c) = \frac{e^{x_i + c}}{\sum_j e^{x_j + c}} \tag{2}$$

$$softmax(x_i + c) = \frac{e^{x_i} * e^c}{\sum_j (e^{x_j} * e^c)} \tag{3}$$

Taking e^c common from the numerator and denominator, and cancelling as it is not 0 :-

$$softmax(x_i + c) = \frac{e^{x_i}}{\sum_j e^{x_j}} = softmax(x_i) \tag{4}$$

The amount of translation **c** will get cancelled out both in numerator and denominator, thus, making softmax translation invariant (proved above).

The range of e^{x_i} is $(0, \infty)$. When put c = 0, the range of $e^{x_i + c}$, i.e., the numerator still remains $(0, \infty)$. However, when c = $-max(x_i)$, all the values of $(x_i + c)$ are either negative or 0 (0 only when x_i is maximum). Thus, the values of e^{x_i} is restricted to $(0, 1]$. Thus, c = $-max(x_i)$ is used to make the softmax function numerically stable. It prevents overflow of the values by restricting the output.

Q 1.2

Answer -

Each value of the softmax function is like a probability. Thus, the value of **softmax(x_i)** is [0,1]. Since these i(s) are the only possible outputs, the sum over all the elements with corresponding i(s) is 1. Sum of all the probabilities is 1.

Answer -

Softmax takes an arbitrary real valued vector x and turns it into a **vector of probabilities**.

Answer -

Step 1: $s_i = e^{x_i}$

x_i is interpreted as log probabilities. s_i is differentiable and gives non-negative results - necessary for cross-entropy calculation. s_i lies between 0 and infinity, i.e., $(0, \infty)$.

Step 2: $S = \sum_i s_i$

Sum of all the exponential terms so that each element of step 1 is nomalized.

Step 3: $softmax(x_i) = \frac{1}{S} s_i$

Normalization to convert x_i vector to vector of probabilities. It provides the probability of each class/element in vector x_i. Exponentiation enhances the difference between higher and lower values.

Q 1.3

The input is **X**, weights are **W** and biases are **b**. [y_0 = X]. Pre-activation for first layer:

$$y_{(t)}(x) = W_t^{(T)} y_{t-1} + b_t^{(T)} \tag{5}$$

where t is the layer number (1 to n) and y_{t-1} is the output from the previous layer. Since no non-linear activation is used, the result y for pre and post activation are the same. Forward calculation shows -

$$y_{(1)}(x) = W_1^{(T)} y_0 + b_1^{(T)} \tag{6}$$

$$y_{(2)}(x) = W_2^{(T)} y_1 + b_2^{(T)} = W_2^{(T)}(W_1^{(T)} y_0 + b_1^{(T)}) + b_2^{(T)} \tag{7}$$

$$= (W_2^{(T)}.W_1^{(T)}) y_0 + (W_2^{(T)}.b_1^{(T)} + b_2^{(T)}) = W_2'^{(T)} y_0 + b_2'^{(T)} \tag{8}$$

Thus $W_2'^{(T)} = W_2^{(T)}.W_1^{(T)}$ and $b_2'^{(T)} = W_2^{(T)}.b_1^{(T)} + b_2^{(T)}$. It can be seen that without non-linear activation moving forward in a multi-layer neural network is like linear regression that can be genralised as below -

$$y_{(n)}(x) = W_n''^{(T)} y_0 + b_n''^{(T)} \tag{9}$$

such that

$$W_n''^{(T)} = \prod_{i=1}^{n}(W_i^{(T)}), \qquad b_n''^{(T)} = \sum_{i=1}^{n}(\prod_{j=i+1}^{n}(W_j^{(T)})).b_i^{(T)} \tag{10}$$

This proves the equivalence to linear regression.

Q 1.4

$$\sigma(x) = \frac{1}{1+e^{-x}} = Sigmoid(x) \tag{11}$$

Derivative of sigmoid:

$$\frac{d}{dx}\sigma(x) = \frac{d}{dx}(1+e^{-x})^{-1} \tag{12}$$

Using the chain rule:

$$\frac{d}{dx}(1+e^{-x})^{-1} = -(1+e^{-x})^{-2}\frac{d}{dx}(1+e^{-x}) = -(1+e^{-x})^{-2}[\frac{d}{dx}(1) + \frac{d}{dx}(e^{-x})] \tag{13}$$

$$= -(1+e^{-x})^{-2}\frac{d}{dx}(e^{-x}) = (1+e^{-x})^{-2}(e^{-x}) = \frac{e^{-x}}{(1+e^{-x})^2} \tag{14}$$

$$= \frac{e^{-x}}{1+e^{-x}} \times \frac{1}{1+e^{-x}} = \frac{e^{-x}+1-1}{1+e^{-x}} \times \frac{1}{1+e^{-x}} \tag{15}$$

$$= (1 - \frac{1}{1+e^{-x}}) \times \frac{1}{1+e^{-x}} = (1-\sigma(x)).\sigma(x) \tag{16}$$

Hence, derivative of the sigmoid function can be written in terms of sigmoid function (without directly accessing x).

Q 1.5

Given y = Wx+b and $\frac{\partial J}{\partial y} = \delta \epsilon R^{k\times 1}$. $y_j = \sum_{i=1}^{d} x_i W_{ij} + b_j$.

Taking derivative:

$$\frac{\partial y_i}{\partial W_{ij}} = x_i, \qquad \frac{\partial y_i}{\partial b_j} = 1, \qquad \frac{\partial y_i}{\partial x_i} = W_{ij} \tag{17}$$

$$\frac{\partial J}{\partial W} = \begin{bmatrix} \frac{\partial J}{\partial W_{11}} & . & . & . & \frac{\partial J}{\partial W_{1k}} \\ . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ \frac{\partial J}{\partial W_{d1}} & . & . & . & \frac{\partial J}{\partial W_{dk}} \end{bmatrix}^T \tag{18}$$

Applying chain rule to each element in the matrix:

$$\frac{\partial J}{\partial W} = \begin{bmatrix} \frac{\partial J}{\partial y_1} x_1 & . & . & . & \frac{\partial J}{\partial y_k} x_1 \\ . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ \frac{\partial J}{\partial y_1} x_d & . & . & . & \frac{\partial J}{\partial y_k} x_d \end{bmatrix}^T \tag{19}$$

$$\frac{\partial J}{\partial W} = \delta X^T \tag{20}$$

$$\frac{\partial J}{\partial X} = \begin{bmatrix} \frac{\partial J}{\partial x_1} \\ . \\ . \\ . \\ \frac{\partial J}{\partial x_d} \end{bmatrix} \tag{21}$$

$$\frac{\partial J}{\partial X} = \begin{bmatrix} \sum_{i=1}^{n} \frac{\partial J}{\partial y_i} . \frac{\partial y_i}{\partial x_1} \\ . \\ . \\ . \\ \sum_{i=1}^{n} \frac{\partial J}{\partial y_i} . \frac{\partial y_i}{\partial x_d} \end{bmatrix} \tag{22}$$

$$\frac{\partial J}{\partial X} = W^T \delta \tag{23}$$

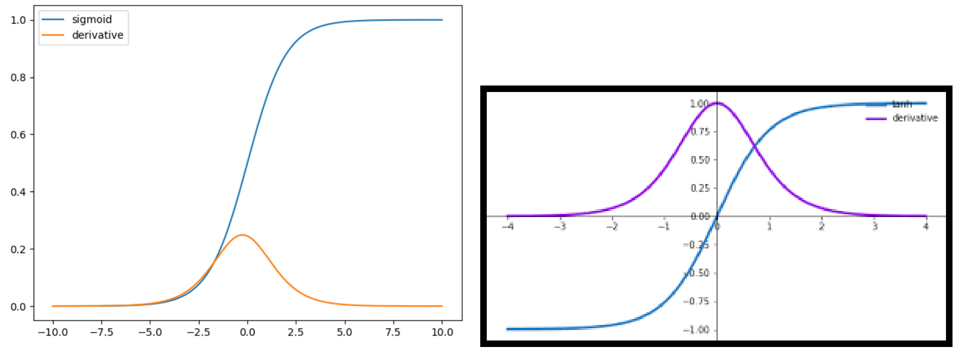
$$\frac{\partial J}{\partial b} = \begin{bmatrix} \frac{\partial J}{\partial b_1} \\ . \\ . \\ . \\ \frac{\partial J}{\partial b_k} \end{bmatrix} \tag{24}$$

Similar to differentiation with respect to weights, loss is differentiated with respect to bias.

Q 1.6

1. Value of the sigmoid's derivative lies between 0 and 0.25.

   Figure 1: Graph for Sigmoid and Tanh; and their Derivatives

   When traversing back in a deep neural network, the derivative of the activation function gets multiplied repeatedly at each layer it is present. As seen above, the maximum value of the sigmoid's derivative is 0.25 (at x=0). Thus, on repeated multiplication in deep networks, being less than 1, it reduces the gradients calculated significantly - leading to vanishing gradient problem.

2. Range of Sigmoid = 0 to 1. Range of Tanh = -1 to 1.

   Range of Sigmoid's Derivative = 0 to 0.25. Range of Tanh's Derivative = 0 to 1.

   The advantage of tanh over the sigmoid function is that its derivative is more steep and can avoid vanishing gradient problem for a deep network. It can be more efficient because it has a wider range for faster learning and grading. This can be seen from the ranges above. The symmetry about x=0 for (-1,1) also helps.

3. As seen in the figure above, the gradient for tanh goes to as high as 1 compared to sigmoid's 0.25. While backpropagating, this tanh derivative is multiplied to the other terms for the layers it is present in. Since the values tanh derivative is less than 1, it can have the vanishing gradients problem for networks. However, since it can go up to 1, the decrement upon multiplication may not be very huge and the problem is delayed. Hence, tanh has a less of vanishing gradient problem.

4.
$$\sigma(x) = \frac{1}{1+e^{-x}} = Sigmoid(x) \tag{25}$$

$$Tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{26}$$

   Owing to symmetry around x=0 for sigmoid function:

$$1 - \sigma(x) = \sigma(-x) \tag{27}$$

$$Tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^x + e^{-x} - 2e^{-x}}{e^x + e^{-x}} = 1 + \frac{-2e^{-x}}{e^x + e^{-x}} \tag{28}$$

$$= 1 - \frac{2}{e^{2x}+1} = 1 - 2\sigma(-2x) = 1 - 2(1-\sigma(2x)) = 2\sigma(2x) - 1 \tag{29}$$

   Hence, tanh is just the shifted and rescaled version of the sigmoid funtion.

Question 2.1 : Network Initialization

Q 2.1.1

Answer -

Zero initialization - all the parameters in the network, i.e., weights and biases are set to zero. This causes neurons to perform similar calculation in each iteration, thereby producing similar results. The output from each neuron will be the same in the first iteration and while back-propagating the derivatives are scaled by the same value for each neuron in a layer (irrespective of the input). This will give same update for all the parameters and will continue. Thus, neurons learn similar features in each iteration, unable to break symmetry. Neurons won't be learning new and different things easily. This makes linear layers less useful and makes it difficult to reach global optimum. Zero-initialized network can output a sub-optimal solution after training (stuck in local minima/maxima depending on the cost function).

Q 2.1.2

Figure 2: Code for Xavier Initialization in **python/nn.py**

```
############################## Q 2.1 ##############################
# initialize b to 0 vector
# b should be a 1D array, not a 2D array with a singleton dimension
# we will do XW + b.
# X be [Examples, Dimensions]
def initialize_weights(in_size,out_size,params,name=''):

    limit = (6**0.5)/((in_size+out_size)**0.5)
    W = np.random.uniform(low = -limit,high = limit,size = (in_size,out_size))
    b = np.zeros(out_size)

    ##########################
    ##### your code here #####
    ##########################

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

Q 2.1.3

Random initialization breaks the symmetry among neurons in the same layer and allows them to learn new and different features. This gives better accuracy and learning than zero-initialization.

We scale the initialization depending on layer size so that the activations and gradients (while back-propagating) do not explode or vanish, i.e., to to keep them stable even in deep networks.

Having gradients of very different magnitudes at different layers may yield to ill-conditioning and slower training - a problem that might be encountered in standard initialisation without scaling.

Question 2.2 : Forward Propagation

Q 2.2.1

Figure 3: Code for Forward with sigmoid activation in **python/nn.py**

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

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

    return res

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

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


    ##########################
    ##### your code here #####
    ##########################

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

    return post_act
```

Figure 4: Code for Forward with sigmoid activation in **python/nn.py**

```
PS C:\Users\sahar\Desktop\Acads\CVB-Spring22\hw5-1\hw5\python> python .\run_q2.py
0.0, 0.08
0.0, 0.07
C:\Users\sahar\Desktop\Acads\CVB-Spring22\hw5-1\hw5\python\nn.py:33: RuntimeWarning: overflow encountered in exp
  res = 1/(1+np.exp(-x))
should be zero and one   0.0 1.0
(40, 2) (2, 25) (25,)
(40, 25)
```

Number of examples = 40
Data Columns = 2 (input neurons)
Output Neurons = 25

Q 2.2.2

Figure 5: Code for Softmax Function in **python/nn.py**

```
################################ Q 2.2.2  ################################
# x is [examples,classes]
# softmax should be done for each row
def softmax(x):
    ###########################
    ##### your code here #####
    ###########################
    print(x.shape)
    mx = np.max(x,axis=1)
    mx = mx.reshape((mx.shape[0],1))
    top = np.exp(x-mx)
    sm = np.sum(top, axis = 1)
    sm = sm.reshape((sm.shape[0],1))
    res = top/sm

    return res
```

Q 2.2.3

Figure 6: Code for Loss and Accuracy in **python/nn.py**

```
################################ 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):
    ############################
    ##### your code here #####
    ############################
    ypred = np.argmax(probs, axis = 1)
    yans = np.argmax(y, axis = 1)
    t = np.count_nonzero(ypred == yans)
    acc = t/y.shape[0]

    loss = -(y*np.log(probs)).sum()

    return loss, acc
```

Figure 7: Result for Loss and Accuracy in **python/nn.py**

```
PS C:\Users\sahar\Desktop\Acads\CVB-Spring22\hw5-1\hw5\python> python .\run_q2.py
0.0, 0.07
0.0, 0.07
C:\Users\sahar\Desktop\Acads\CVB-Spring22\hw5-1\hw5\python\nn.py:33: RuntimeWarning: overflow encountered in exp
  res = 1/(1+np.exp(-x))
should be zero and one   0.0 1.0
(40, 2) (2, 25) (25,)
(40, 25)
(40, 25) (25, 4) (4,)
(40, 4)
0.07465753033143795 0.9999999999999998 1.0000000000000002 (40, 4)
59.45613569839223, 0.25
```

Question 2.3: Backwards Propagation

Figure 8: Code and Result for Back-propagation in **python/nn.py**

```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_W = X.T @ (delta*activation_deriv(post_act))
    grad_b = np.sum(delta*activation_deriv(post_act), axis = 0)
    grad_X = (delta*activation_deriv(post_act)) @ W.T

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

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
69.88312578699376, 0.25
Wlayer1 (2, 25) (2, 25)
Woutput (25, 4) (25, 4)
blayer1 (25,) (25,)
boutput (4,) (4,)
```

Question 2.4: Training Loop

Figure 9: Code for batch creation in **python/nn.py**

```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 = []
    ##########################
    ##### your code here #####
    ##########################
    #print(y,x.shape,y.shape) # y is one-hot encoded
    l = y.shape[0]
    nob = int(np.ceil(l/batch_size))
    #print(l,nob,batch_size) # 40,8,5
    for b in range(nob):
        idx = np.random.randint(0,l,batch_size)
        #print(idx)
        tup = (x[(idx),:], y[(idx),:])
        #print(tup)
        batches.append(tup)

    #print(batches)
    return batches
```

Figure 10: Code for training in **python/run_q2.py**

```
# 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
    avg_acc = 0
    for xb,yb in batches:
        ##########################
        ##### your code here #####
        ##########################
        # forward
        h1 = forward(xb,params,'layer1',sigmoid)
        prob = forward(h1,params,'output',softmax)
        #print(prob.shape)
        #print(1, xb.shape, yb.shape, prob.shape)
        loss, acc = compute_loss_and_acc(yb, prob)
        # loss
        # be sure to add loss and accuracy to epoch totals
        total_loss += loss
        avg_acc += acc

        # backward
        delta1 = prob-yb
        # apply gradient
        delta2 = backwards(delta1, params,'output',linear_deriv)
        delta3 = backwards(delta2, params, 'layer1', sigmoid_deriv)
        # gradients should be summed over batch samples

        params['blayer1'] = params['blayer1']-(learning_rate*params['grad_blayer1'])
        params['boutput'] = params['boutput']-(learning_rate*params['grad_boutput'])
        params['Wlayer1'] = params['Wlayer1']-(learning_rate*params['grad_Wlayer1'])
        params['Woutput'] = params['Woutput']-(learning_rate*params['grad_Woutput'])

    avg_acc /= len(batches)


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

Figure 11: Result for training in **python/run_q2.py**

```
Wlayer1 (2, 25) (2, 25)
Woutput (25, 4) (25, 4)
blayer1 (25,) (25,)
boutput (4,) (4,)
[5, 5, 5, 5, 5, 5, 5, 5]
itr: 00          loss: 59.77     acc : 0.35
itr: 100         loss: 35.27     acc : 0.72
itr: 200         loss: 29.11     acc : 0.72
itr: 300         loss: 25.53     acc : 0.75
itr: 400         loss: 23.09     acc : 0.90
```

Question 2.5: Numerical Gradient Checker

Figure 12: Code for Gradient Checker in **python/run_q2.py**

```
# compute gradients using finite difference
eps = 1e-6
#print(params.items())
for k,v in params.items():
    if '_' in k:
        continue
    # we have a real parameter!
    # print(k) #wlayer, blayer, wout,bout
    #print(v.shape) # 2,25 ... 25 ...25,4 ... 4
    if len(v.shape)==2:
        for i in range(v.shape[0]):
            for j in range(v.shape[1]):
                copy_params_minus = copy.deepcopy(params)
                copy_params_minus[k][i,j] = v[i,j]-eps
                copy_params_plus = copy.deepcopy(params)
                copy_params_plus[k][i,j] = v[i,j]+eps

                h1_minus = forward(x,copy_params_minus,'layer1')
                probs_minus = forward(h1_minus,copy_params_minus,'output',softmax)
                loss_minus, acc_minus = compute_loss_and_acc(y,probs_minus)

                h1_plus = forward(x,copy_params_plus,'layer1')
                probs_plus = forward(h1_plus,copy_params_plus,'output',softmax)
                loss_plus, acc_plus = compute_loss_and_acc(y,probs_plus)

                params['grad_'+k][i,j] = (loss_plus-loss_minus)/(2*eps)

    else:
        for i in range(v.shape[0]):
            copy_params_minus = copy.deepcopy(params)
            copy_params_minus[k][i] = v[i]-eps
            copy_params_plus = copy.deepcopy(params)
            copy_params_plus[k][i] = v[i]+eps

            h1_minus = forward(x,copy_params_minus,'layer1')
            probs_minus = forward(h1_minus,copy_params_minus,'output',softmax)
            loss_minus, acc_minus = compute_loss_and_acc(y,probs_minus)

            h1_plus = forward(x,copy_params_plus,'layer1')
            probs_plus = forward(h1_plus,copy_params_plus,'output',softmax)
            loss_plus, acc_plus = compute_loss_and_acc(y,probs_plus)

            params['grad_'+k][i] = (loss_plus-loss_minus)/(2*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))
```

Figure 13: Result for Gradient Checker in **python/run_q2.py**

```
itr: 400          loss: 30.69     acc : 0.80
grad_Woutput 9.75e-07
grad_boutput 9.04e-09
grad_Wlayer1 1.60e-06
grad_blayer1 9.21e-07
total 3.51e-06
```

The result is within the permissible limits.

Question 3: Training Models

Q 3.1

Random seed fixed to 1. Iterations/epochs = 200, batch size = 128, learning rate = 0.001. Mini-batch approach selected for training.

Figure 14: Result for Loss

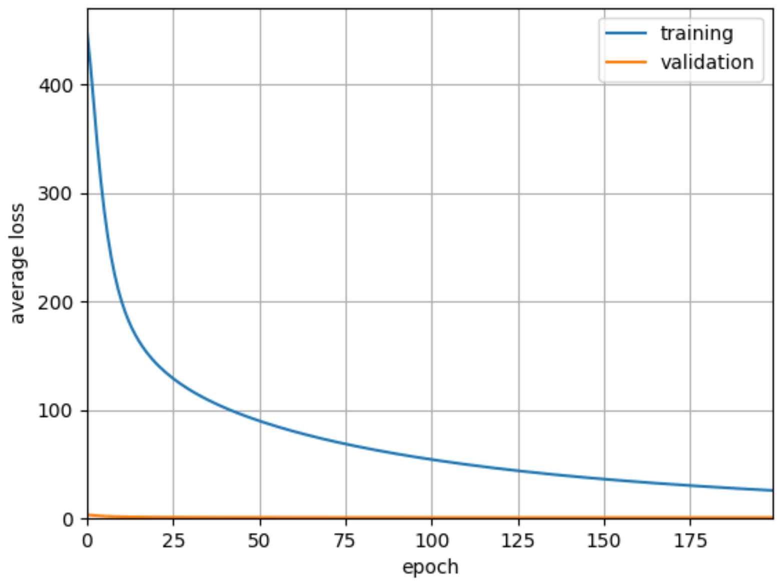


Figure 15: Result for Accuracy

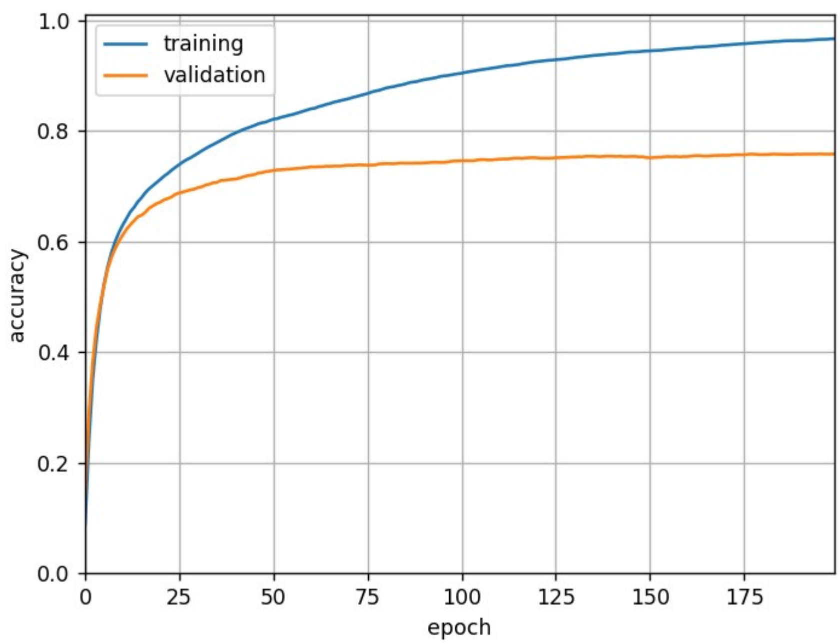


Figure 16: Training Logs

```
itr: 186        loss: 28.18     acc : 0.96
itr: 188        loss: 27.81     acc : 0.96
itr: 190        loss: 27.44     acc : 0.96
itr: 192        loss: 27.08     acc : 0.96
itr: 194        loss: 26.73     acc : 0.96
itr: 196        loss: 26.38     acc : 0.96
itr: 198        loss: 26.04     acc : 0.97
Validation accuracy:  0.7577777777777778
Test accuracy:  0.7516666666666667
```

Figure 17: Code - My implementation

```python
train_loss = []
valid_loss = []
train_acc = []
valid_acc = []
for itr in range(max_iters):
    # # record training and validation loss and accuracy for plotting
    # h1 = forward(train_x,params,'layer1')
    # probs = forward(h1,params,'output',softmax)
    # loss, acc = compute_loss_and_acc(train_y, probs)
    # train_loss.append(loss/train_x.shape[0])
    # train_acc.append(acc)
    # h1 = forward(valid_x,params,'layer1')
    # probs = forward(h1,params,'output',softmax)
    # loss, acc = compute_loss_and_acc(valid_y, probs)
    # valid_loss.append(loss/valid_x.shape[0])
    # valid_acc.append(acc)

    total_loss = 0
    total_acc = 0
    for xb,yb in batches:
        # training loop can be exactly the same as q2!
        ##########################
        ##### your code here #####
        ##########################
        h1 = forward(xb,params,'layer1',sigmoid)
        prob = forward(h1,params,'output',softmax)
        #print(prob.shape)
        #print(1, xb.shape, yb.shape, prob.shape)
        loss, acc = compute_loss_and_acc(yb, prob)
        # loss
        # be sure to add loss and accuracy to epoch totals
        total_loss += loss
        total_acc += acc

        # backward
        delta1 = prob-yb
        # apply gradient
        delta2 = backwards(delta1, params,'output',linear_deriv)
        delta3 = backwards(delta2, params, 'layer1', sigmoid_deriv)
```

Figure 18: Code - continued

```
        params['blayer1'] = params['blayer1']-(learning_rate*params['grad_blayer1'])
        params['boutput'] = params['boutput']-(learning_rate*params['grad_boutput'])
        params['Wlayer1'] = params['Wlayer1']-(learning_rate*params['grad_Wlayer1'])
        params['Woutput'] = params['Woutput']-(learning_rate*params['grad_Woutput'])

    total_acc = total_acc/len(batches)
    total_loss = total_loss/len(batches)
    train_loss.append(total_loss)
    train_acc.append(total_acc)

    #validation
    h1 = forward(valid_x,params,'layer1')
    probs = forward(h1,params,'output',softmax)
    loss, acc = compute_loss_and_acc(valid_y, probs)
    valid_loss.append(loss/valid_x.shape[0])
    valid_acc.append(acc)

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

# record final training and validation accuracy and loss
# h1 = forward(train_x,params,'layer1')
# probs = forward(h1,params,'output',softmax)
# loss, acc = compute_loss_and_acc(train_y, probs)
# train_loss.append(loss/train_x.shape[0])
# train_acc.append(acc)
# h1 = forward(valid_x,params,'layer1')
# probs = forward(h1,params,'output',softmax)
# loss, acc = compute_loss_and_acc(valid_y, probs)
# valid_loss.append(loss/valid_x.shape[0])
# valid_acc.append(acc)

# report validation accuracy; aim for 75%
print('Validation accuracy: ', valid_acc[-1])
```

Q 3.2

Figure 19: Result for Loss - Learning Rate (My - 0.001)

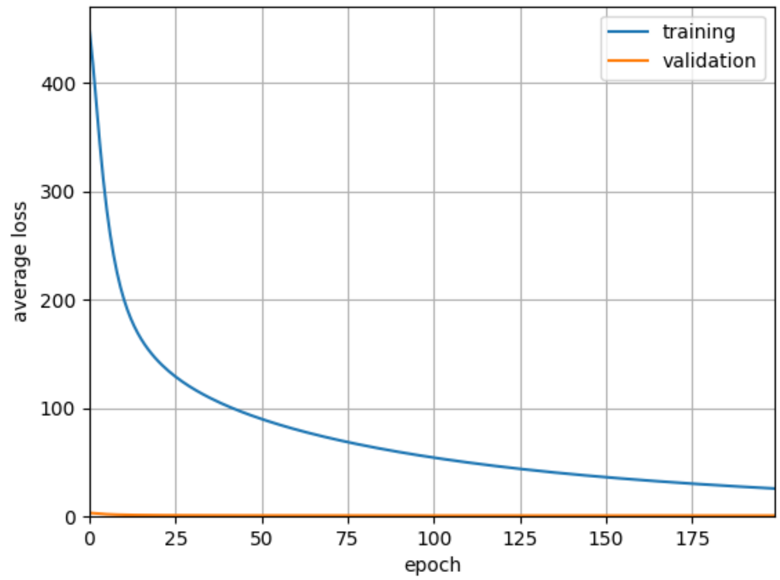
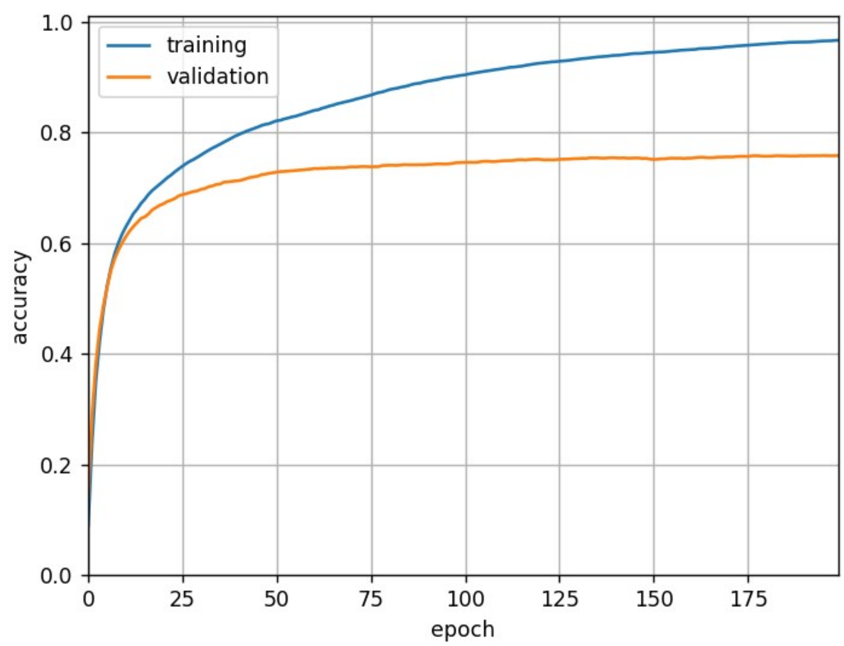


Figure 20: Result for Accuracy - Learning Rate (My - 0.001)



Best results I got. 75%+ on test.

Figure 21: Result for Loss - Learning Rate 10X (= 0.01)

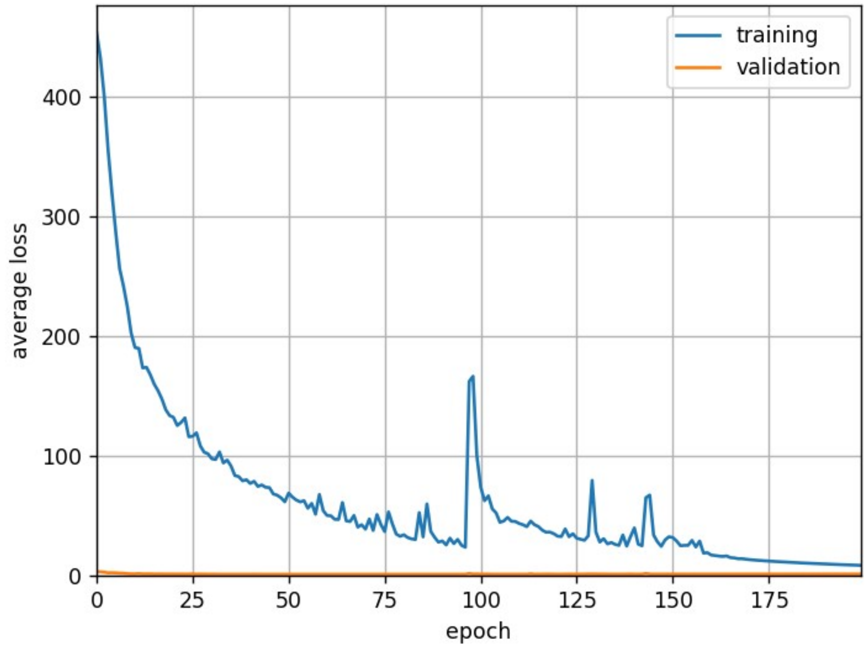
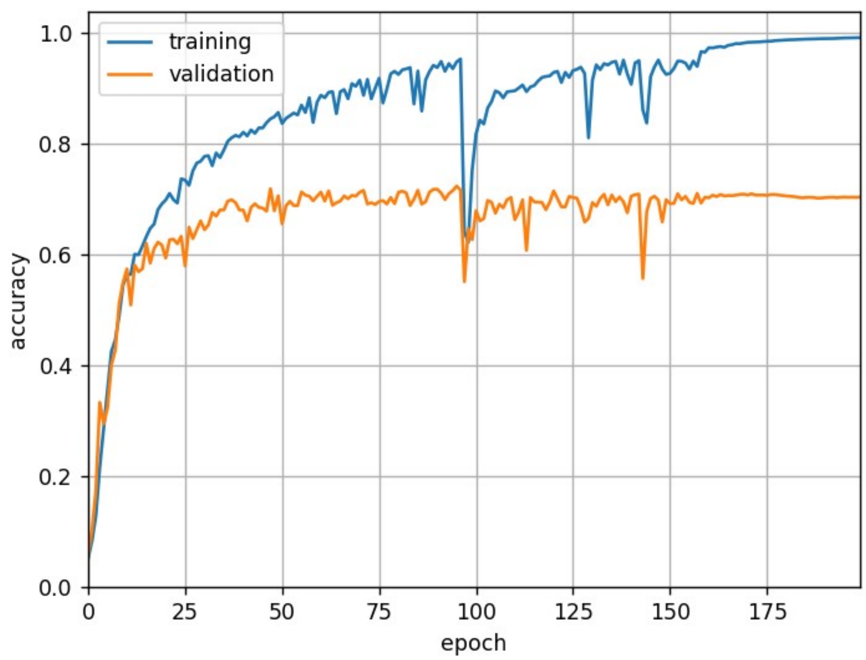


Figure 22: Result for Accuracy - Learning Rate 10X (= 0.01)



Validation accuracy: 0.7038888888888889, Test accuracy: 0.7177777777777777

Increasing the learning rate way beyond the tuned value (cet. par.) increases the chances of divergence. As seen the graphs, it shows that learning rate is very high and keeps missing the minima. Thus, the training is not smooth and accuracy is lower.

Figure 23: Result for Loss - Learning Rate 1/10X (= 0.0001)

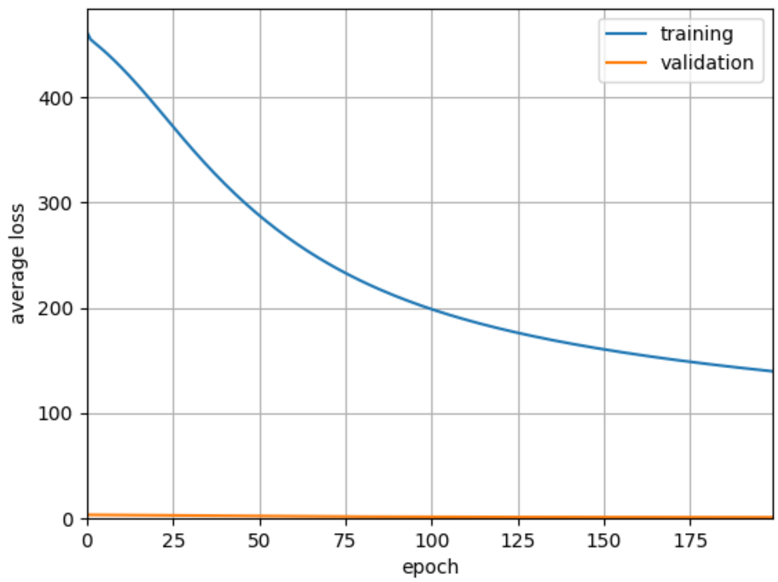
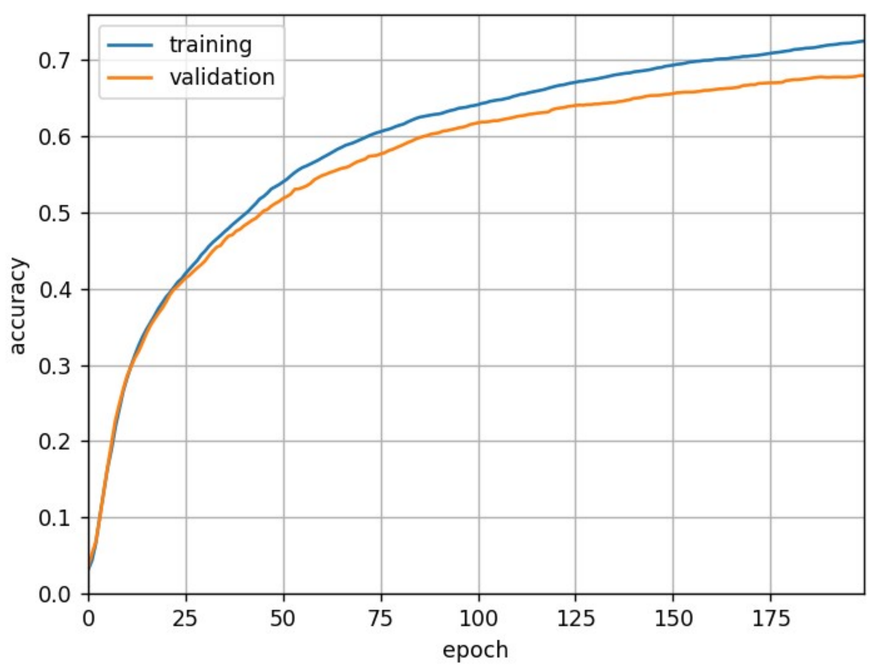


Figure 24: Result for Accuracy - Learning Rate 1/10X (= 0.0001)

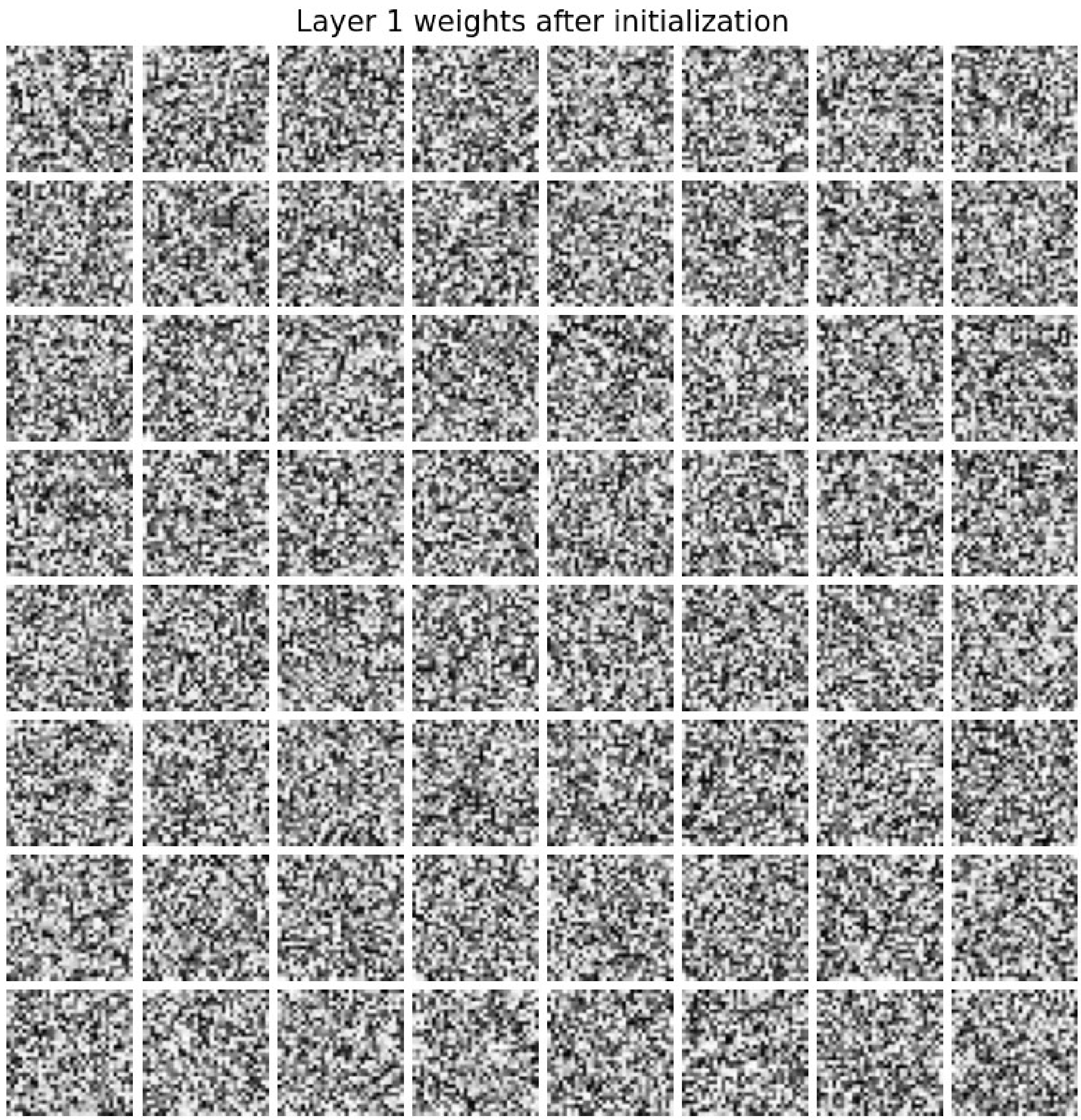


Validation accuracy: 0.6794444444444444, Test accuracy: 0.6788888888888889

Decreasing the learning rate so much gives underfitting. The network is learning the features but is very slow in converging. A smooth graph is obtained but there is huge scope to learn further by increasing the epochs. Hence, for same epoch - the accuracy here is much lesser than ours (tuned).

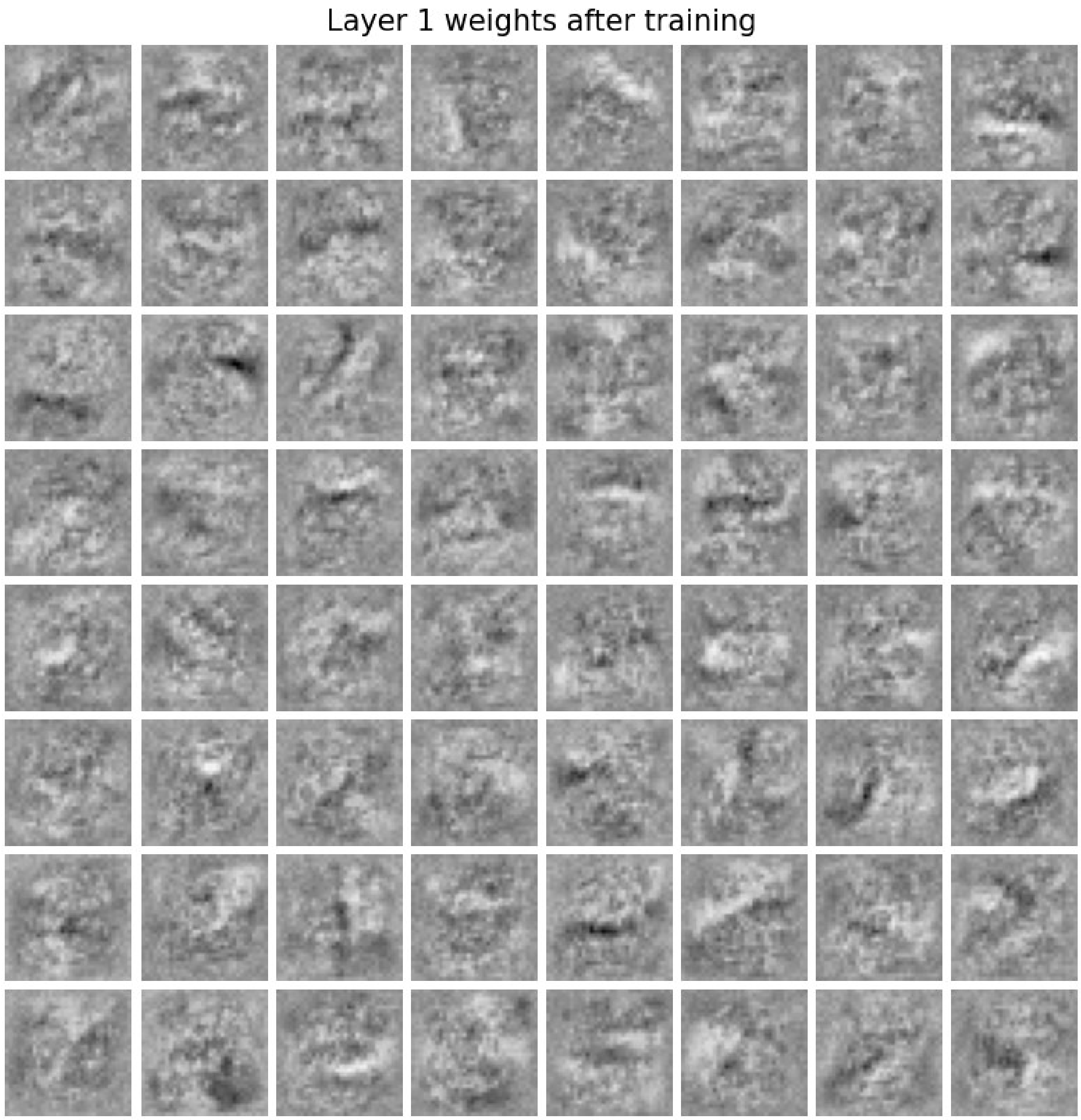
Q 3.3

Figure 25: Weights before training



The weights, before training, look like a random scatter. They have no structure - signifying that it is random and doesn't have pre-learnt any features.

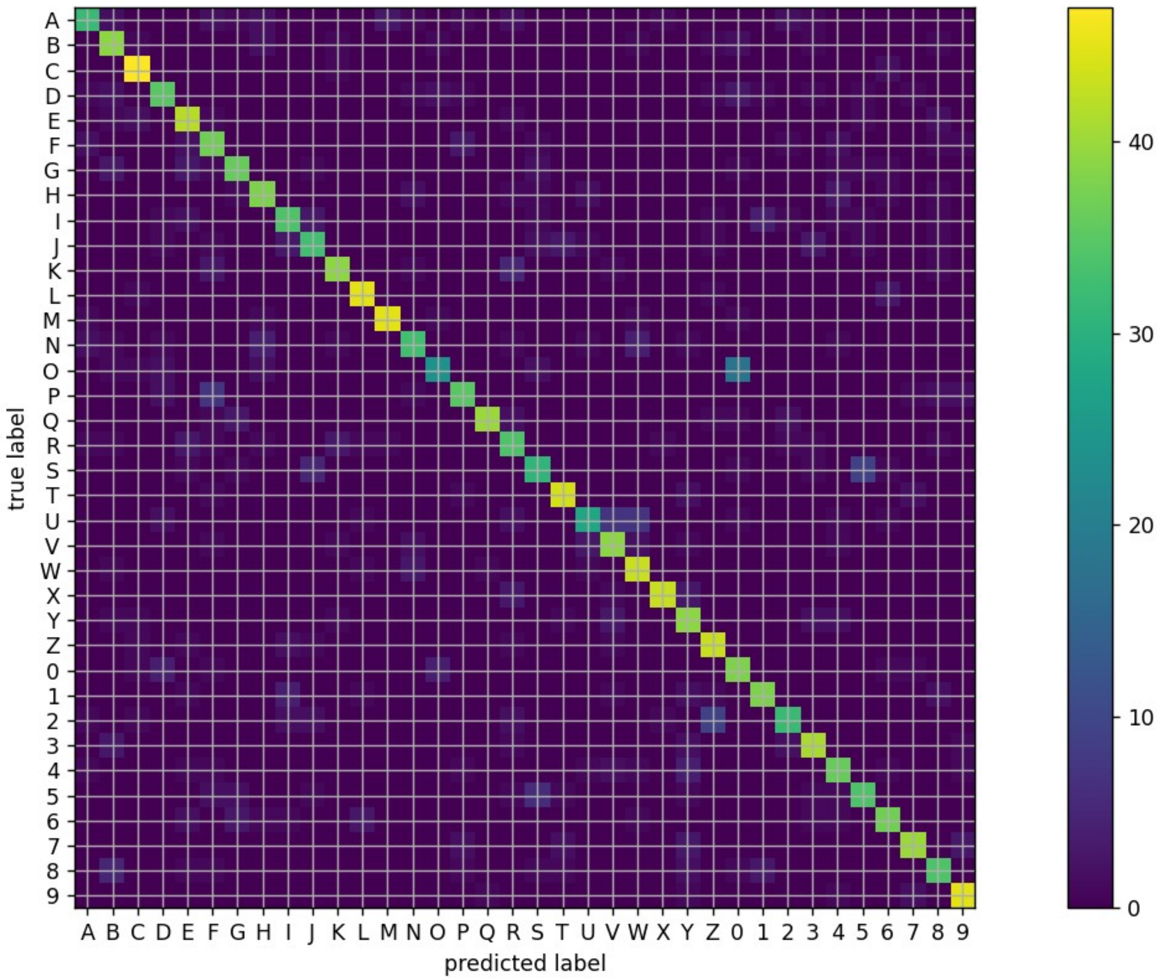
Figure 26: Weights After training



The weights, when visualized after training, start to show some kind of pattern/structure. This shows that the hidden layers start responding to certain image features and this helps in correct prediction.

Q 3.4

Figure 27: Confusion Matrix



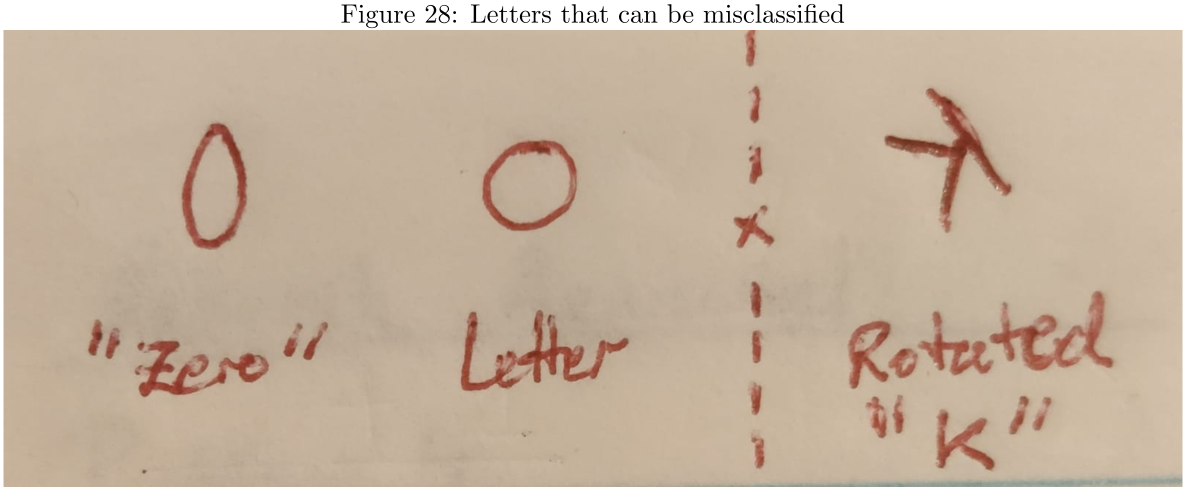
A few pairs that are commonly confused -
0 and **o**
S and **5**
P and **F**
2 and **Z**
K and **R**

Question 4: Extract Text from Images

Q 4.1

Assumptions are :-
1) Characters must be isolated clearly from each other. They should not overlap.
2) The characters should not be rotated because the training data has only one orientation.
3) All characters within an image should be of similar to training data. Cursive Writing or non-clear & non-standard characters are difficult to decipher.
4) Characters with similar features can be misclassified. Pairs that are generally misclassified are mentioned in page before - eg. : 0 and o.

Figure 28: Letters that can be misclassified



Examples -
1. Letter o and digit 0 have very similar features and easily misqualified.
2. Letter "K" when rotated

Q 4.2

Figure 29: Code for findLetters

```
def findLetters(img):
    bb = []
# Estimate the average noise standard deviation across color channels.
    sigma_est = skimage.restoration.estimate_sigma(img, average_sigmas=True, multichannel= True, channel_axis=-1) #single sigma as avg.
    print(sigma_est)
#plt.imshow(img)
    denoised_img = skimage.restoration.denoise_bilateral(img, win_size=5, sigma_color=sigma_est, multichannel=True, channel_axis=-1)
#denoise 4 other options
#plt.imshow(denoised_img)
    grey_img = skimage.color.rgb2gray(denoised_img)
#plt.imshow(grey_img)
    thresh = skimage.filters.threshold_otsu(grey_img) #try_all_thrshold
#print(thresh)
    binary = grey_img<thresh
#plt.imshow(binary)
    im = skimage.morphology.closing(binary, skimage.morphology.square(5))
#plt.imshow(im)
#plt.show()
    im1 = skimage.morphology.dilation(im, skimage.morphology.square(9)) #mota karte hue
    #plt.imshow(im1, cmap="gray")
    #plt.show()
    label_img = skimage.measure.label(im1,connectivity=2, background=0)
    reg = skimage.measure.regionprops(label_image=label_img)
#print(len(reg))

    a = 0
    for r in reg:
        a += r.area
    mean_a = a/len(reg)
    print("Avg Area =",mean_a)

    for r in reg:
        if r.area >= mean_a/5:
            t,l,b,r = r.bbox
            bb.append([t,l,b,r]) #row,col,row,col

    im1 = 1-im1 #letters and background interchanged
    return bb, im1
```

Q 4.3

Figure 30: Image 1

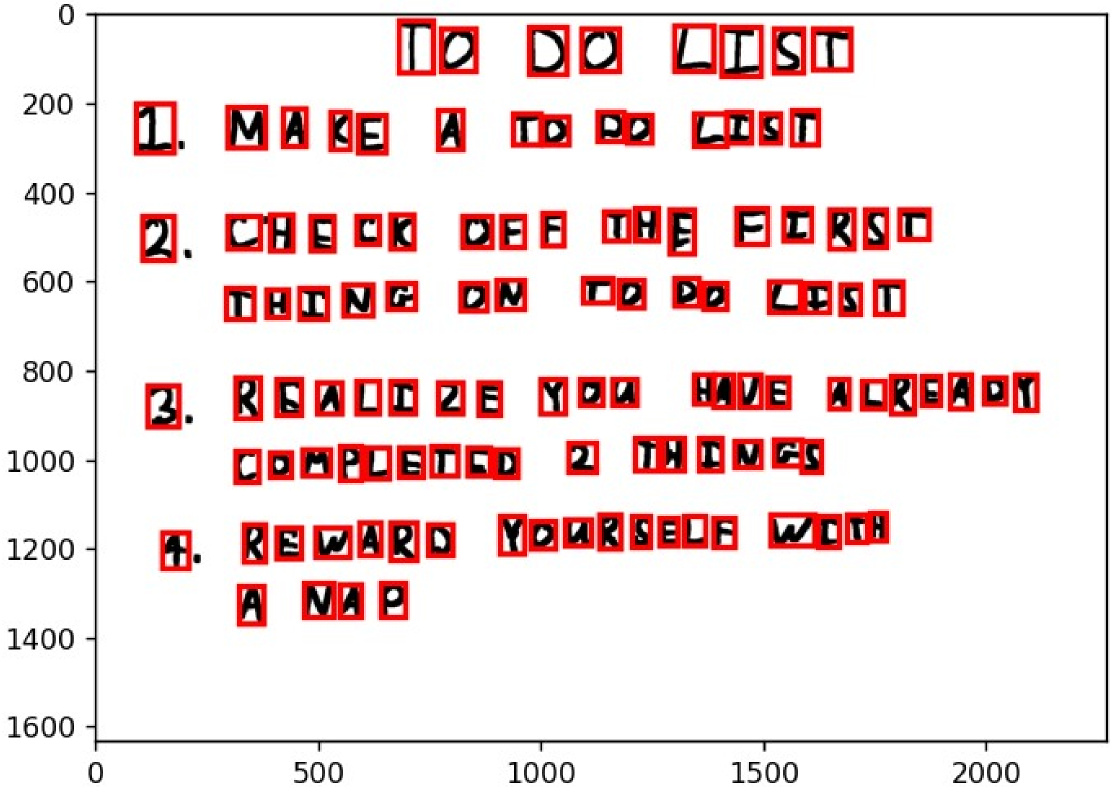


Figure 31: Image 2

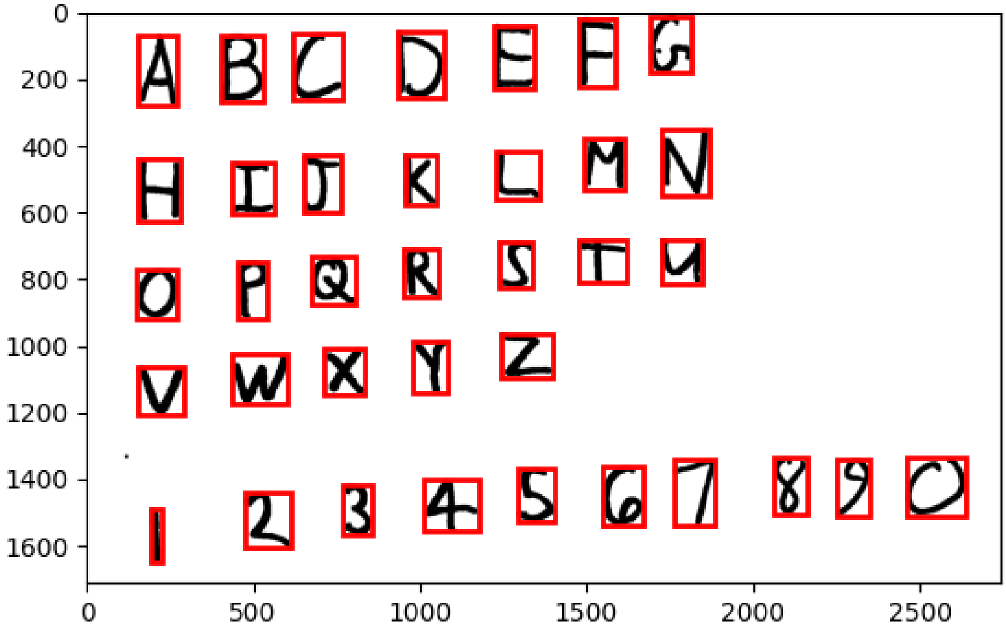


Figure 32: Image 3

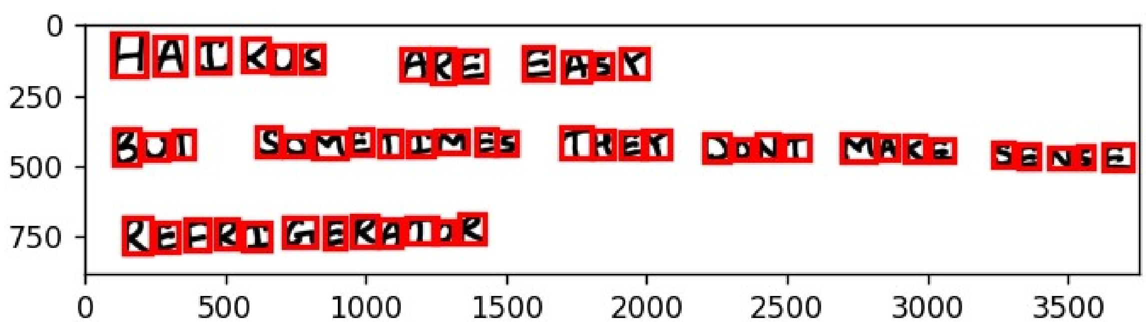
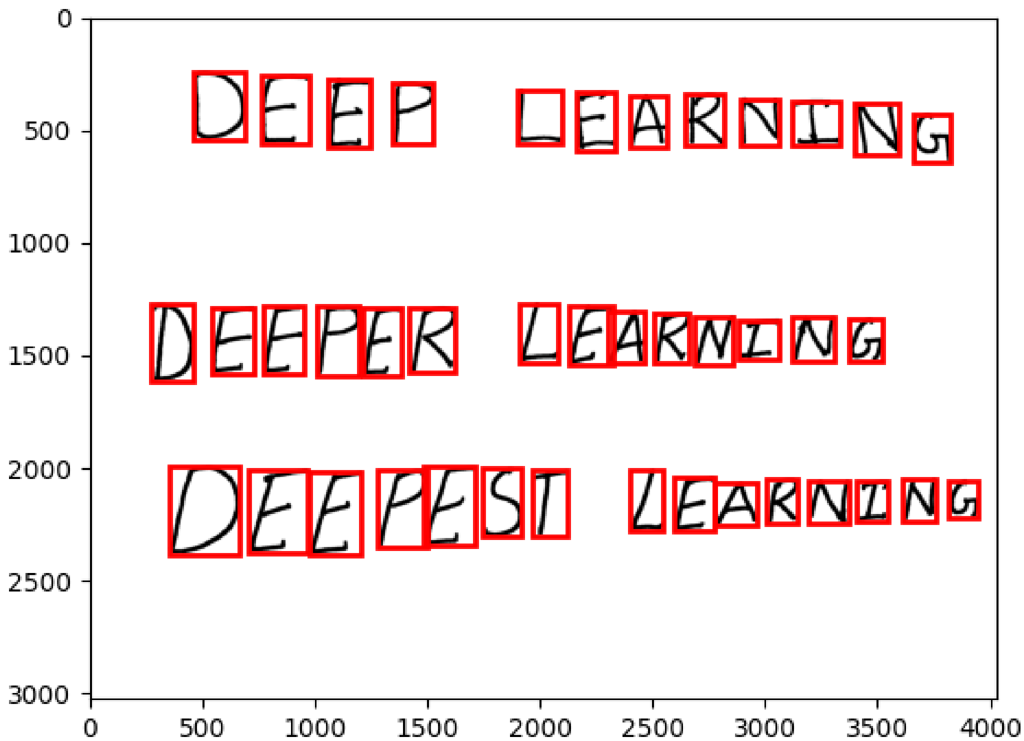


Figure 33: Image 4



Q 4.4

Question 5: Autoencoder

Q 5.1.1

Parts 5.1 and 5.2 are interlinked and parts of the same program section.

Figure 34: Code for setting up the layers for autoencoder

```
# Q5.1 & Q5.2
# initialize layers here
##########################
##### your code here #####
##########################
## print(train_x.shape) # 10800, 1024
initialize_weights(train_x.shape[1], hidden_size, params, 'inp')
initialize_weights(hidden_size, hidden_size, params, 'hidden1')
initialize_weights(hidden_size, hidden_size, params, 'hidden2')
initialize_weights(hidden_size, train_x.shape[1], params, 'output')

#print(params.elements())
#print(params.keys())

wnb_names = [i for i in params.keys()]
for k in wnb_names:
    params['init_'+k] = np.zeros(params[k].shape) #updates of weights as 0 const
```

The loss function has been coded in the following page. Weight initialization has been shown above.

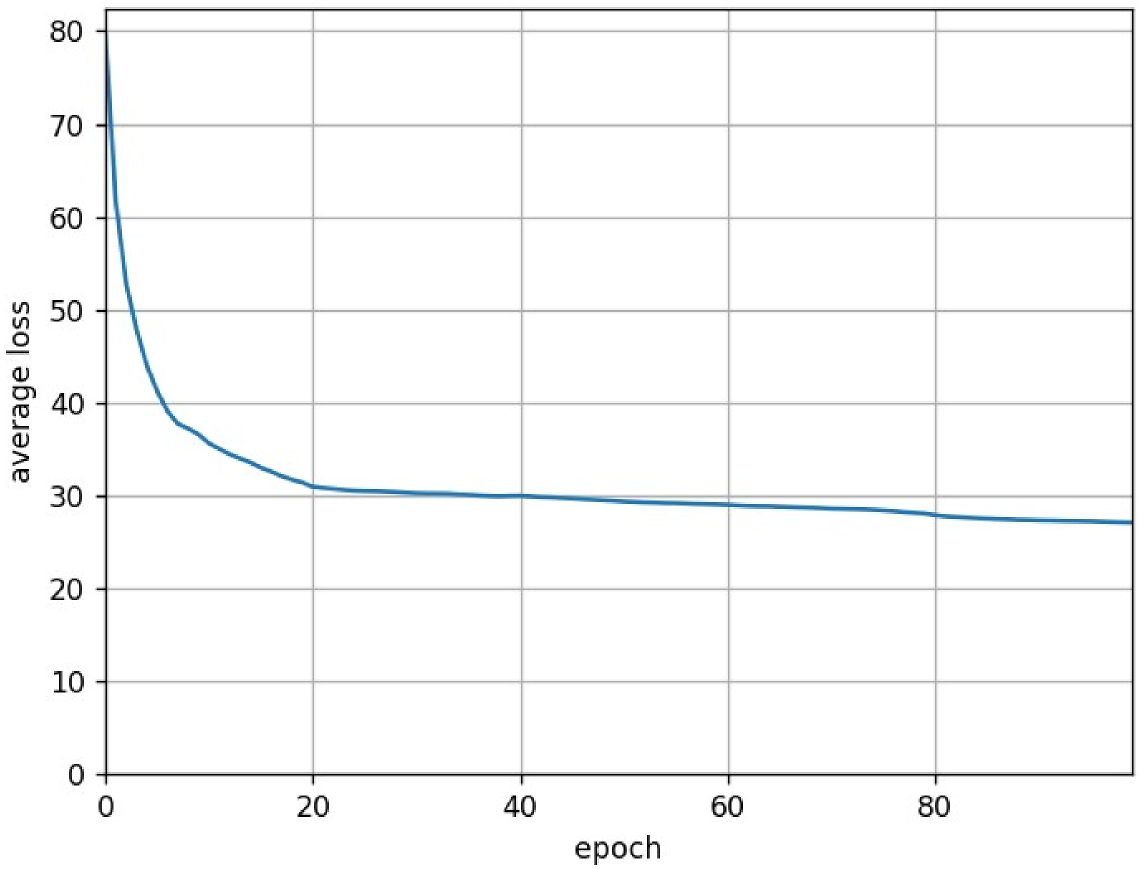
Q 5.1.2

Figure 35: Code for training

```
# should look like your previous training loops
losses = []
#prob_prev = None
for itr in range(max_iters):
    total_loss = 0
    loss = 0
    for xb,_ in batches:
        # delta is the d/dx of (x-y)^2
        # to implement momentum
        #   just use 'm_'+name variables
        #   to keep a saved value over timestamps
        #   params is a Counter(), which returns a 0 if an element is missing

        ##########################
        ##### your code here #####
        ##########################
         # forward
        h1 = forward(xb,params,'inp',relu)
        h2 = forward(h1,params,'hidden1',relu)
        h3 = forward(h2,params,'hidden2',relu)
        prob = forward(h3,params,'output',sigmoid)
        #print(prob.shape)
        #if prob_prev==None or prob.all()!=prob_prev.all():
        #    prob_prev=prob
        #    print("yes")

        loss = ((xb-prob)**2).sum()
        total_loss += loss

        # backward
        delta1 = -2*(xb-prob)
        # apply gradient
        delta2 = backwards(delta1, params,'output',sigmoid_deriv)
        delta3 = backwards(delta2, params, 'hidden2', relu_deriv)
        delta4 = backwards(delta3, params, 'hidden1', relu_deriv)
        backwards(delta4, params, 'inp', relu_deriv)
        # gradients should be summed over batch samples

        for l in ['output','hidden2','hidden1','inp']:
            params['init_W' + l] = 0.9*params['init_W' + l] - learning_rate * params['grad_W' + l]
            params['W' + l] += params['init_W' + l]
            params['init_b' + l] = 0.9*params['init_b' + l]  - learning_rate * params['grad_b' + l]
            params['b' + l]+= params['init_b' + l]
```

Q 5.2

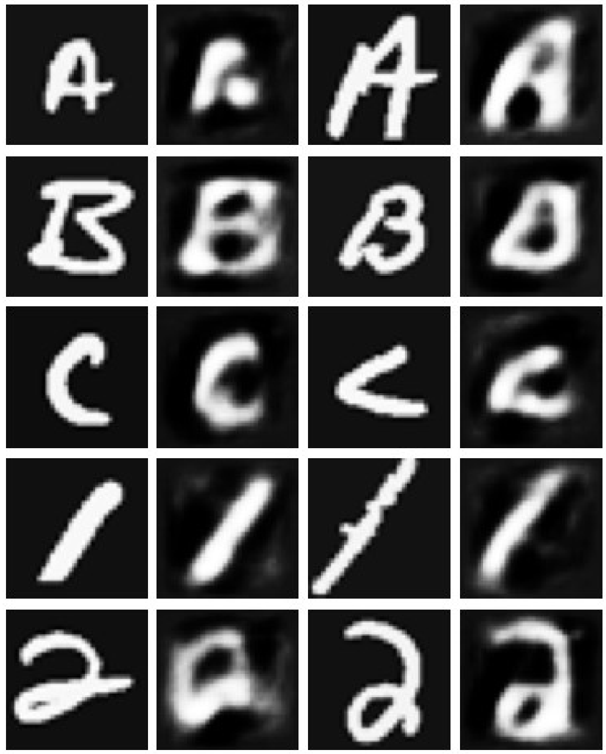
Figure 36: Graph after 100 epochs



Training over more epochs reduces the error/loss. Initially, the the losses reduce drastically and at later stages it tends to drop slowly until it plateaus. However, it is seen to not become 0.

Q 5.3.1

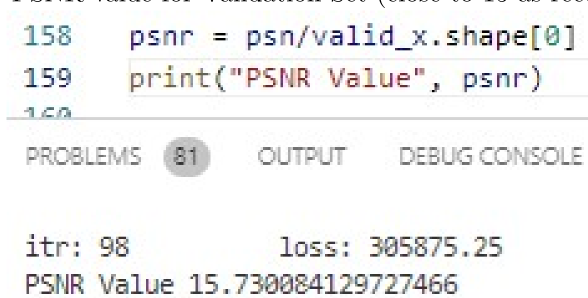
Figure 37: Visualization for autoencoder



The images generated in the 5 classes are close to the original visualized input. In all the cases, they look like the blurred version. Thus, the general appearance of the characters is the same as original but reconstruction looks noisy.

Q 5.3.2

Figure 38: PSNR value for Validation Set (close to 15 as recommended)



Q 6.1.1 - run_q6.py

Figure 39: Accuracy Plot

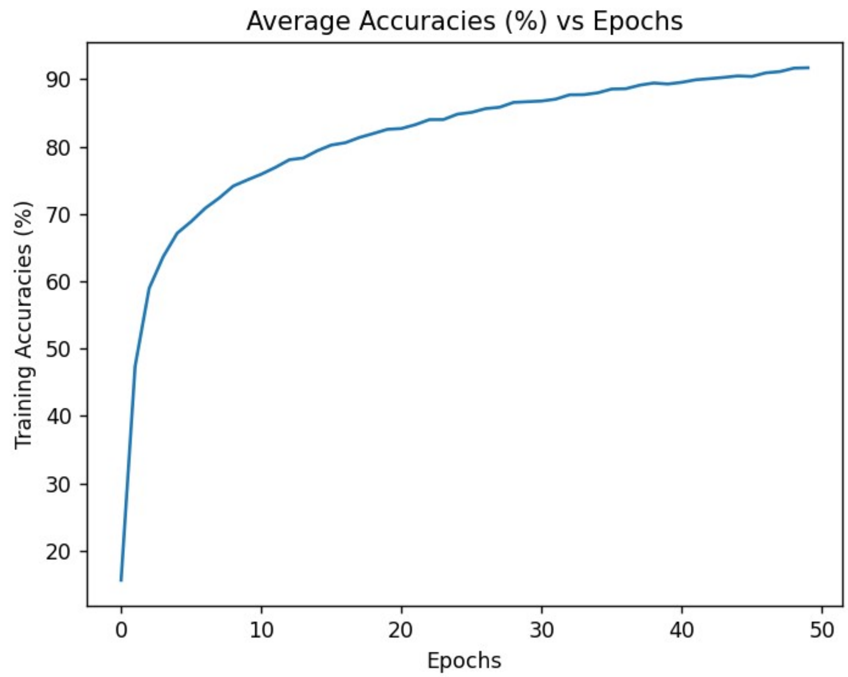
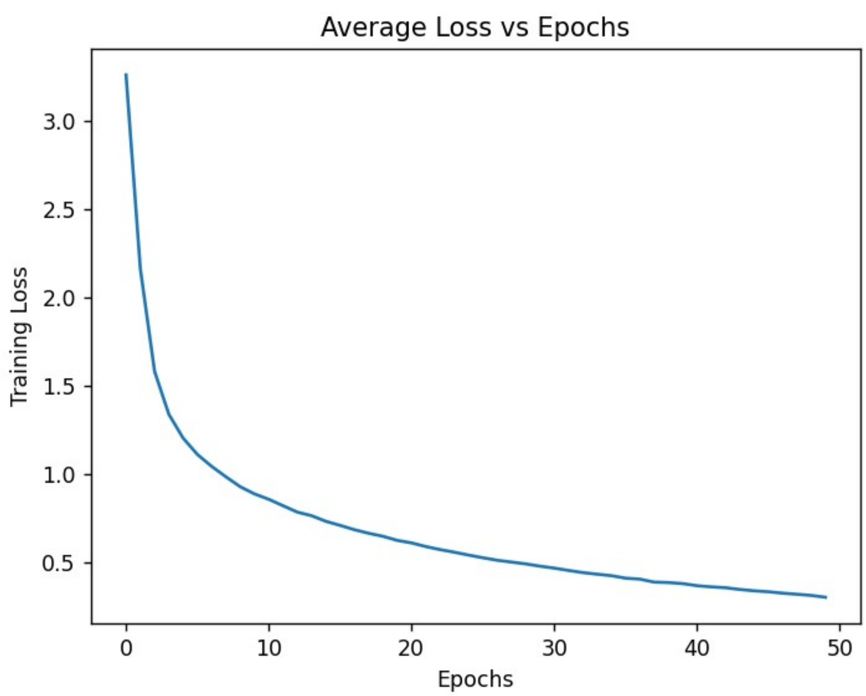


Figure 40: Loss Plot



Accuracy above 92% for a fully connected network.

Figure 41: Code MLP - run_q6.py

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.fc1 = nn.Linear(1024, 64)
        self.s1 = nn.Sigmoid()
        self.fc2 = nn.Linear(64, 36)

    def forward(self, x):
        x = self.fc1(x)
        x = self.s1(x)
        x = self.fc2(x)
        return x

if __name__ == '__main__':
    np.random.seed(1)
    device = torch.device("cpu") #torch not installed with cuda enabled
    print("device =", device)

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

    train_x = train_data['train_data'].astype(np.float32)
    train_y = train_data['train_labels'].astype(np.int32)
    valid_x = valid_data['valid_data'].astype(np.float32)
    valid_y = valid_data['valid_labels'].astype(np.int32)

    epochs = 50 # same as q2
    bs = 5 # same as q2
    model = Net().to(device)
    ### print(model)
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=0.003, momentum=0.9)

    train_data_tensor = TensorDataset(torch.from_numpy(train_x), torch.from_numpy(train_y))
    test_data_tensor = TensorDataset(torch.from_numpy(valid_x), torch.from_numpy(valid_y))
    train_loader = DataLoader(train_data_tensor, batch_size=bs, shuffle=True, num_workers=1)
    valid_loader = DataLoader(test_data_tensor, batch_size=bs, shuffle=True, num_workers=1)
```

Figure 42: Code MLP - run_q6.py

```
model.train()
acc = []
losses = []
for itr in range(epochs):
    correct = 0
    total_loss = 0
    count = 0
    for idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        output = model(data)
        target = torch.max(target, 1)[1]
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        total_loss += loss.item()

        pred = torch.max(output, 1)[1]
        correct += pred.eq(target).sum().item()
        count = count + 1

    acc.append(100. * correct / (count*bs))
    losses.append(total_loss/count)
    if itr % 2 == 0:
        print("itr: {:02d} \t loss: {:.2f} \t acc : {:.2f}".format(itr,(total_loss/count),(100. * correct / (count*bs))))

plt.figure()
a = [i for i in acc]
plt.plot(np.arange(epochs), a, label = "Training Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Training Accuracies (%)")
plt.title("Average Accuracies (%) vs Epochs")
plt.show()

plt.figure()
l = [i for i in losses]
plt.plot(np.arange(epochs), l, label = "Training Loss")
plt.xlabel("Epochs")
plt.ylabel("Training Loss")
plt.title("Average Loss vs Epochs")
plt.show()
```

Q 6.1.2 - run_q6_cnn.py

Figure 43: Accuracy Plot

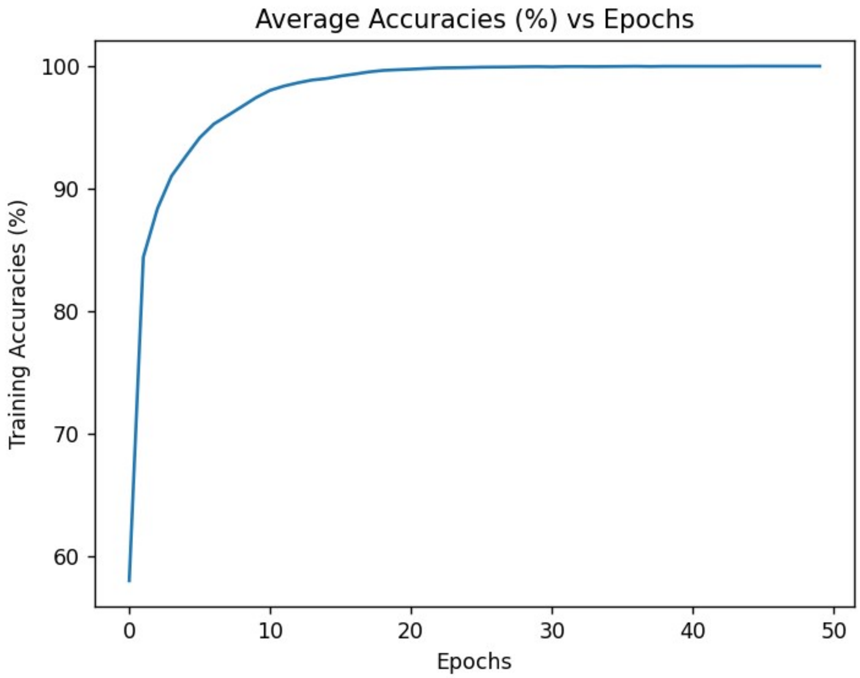
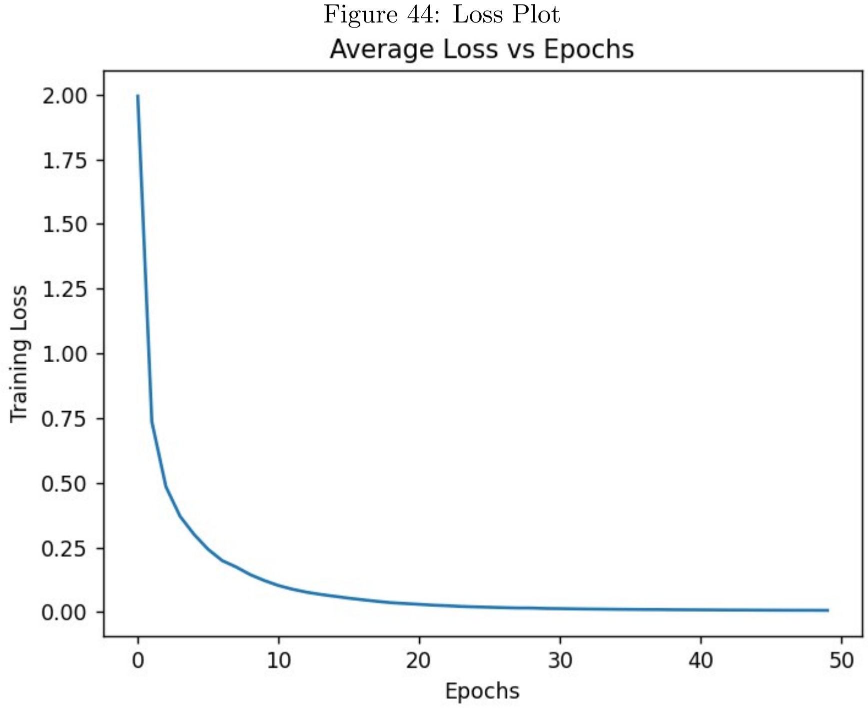


Figure 44: Loss Plot



It can be easily seen that CNN gives a higher accuracy much faster. For the same number of epochs (50 in our case) Conv Net has provided a significantly higher training accuracy of close to 99.7% compared to around 92% for MLP. Graphs also show that CNN converge much faster and hence, should be trained for lesser epochs or with additional transforms - to avoid over-fitting.

Figure 45: Code CNN

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(1,8,kernel_size=3,stride=1), #30
            nn.BatchNorm2d(8),
            nn.ReLU(),
            nn.MaxPool2d(stride=2, kernel_size=2), #15
            nn.Conv2d(8,16,kernel_size=3,stride=1), #13
            nn.BatchNorm2d(16),
            nn.ReLU(),

            nn.Flatten())

        self.fc1 = nn.Linear(13*13*16, 48)
        self.s1 = nn.Sigmoid()
        self.fc2 = nn.Linear(48, 36)

    def forward(self, x):
        x = self.conv(x)
        x = self.fc1(x)
        x = self.s1(x)
        x = self.fc2(x)
        return x

if __name__ == '__main__':
    np.random.seed(1)
    device = torch.device("cpu") #torch not installed with cuda enable
    print("device =", device)

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

    train_x = train_data['train_data'].astype(np.float32)
    #print(train_x.shape)
    train_x = np.reshape(train_x,(train_x.shape[0],1,32,32))
```

Q 6.1.3 - CIFAR - run_q6_cifar.py

Figure 46: Accuracy Plot

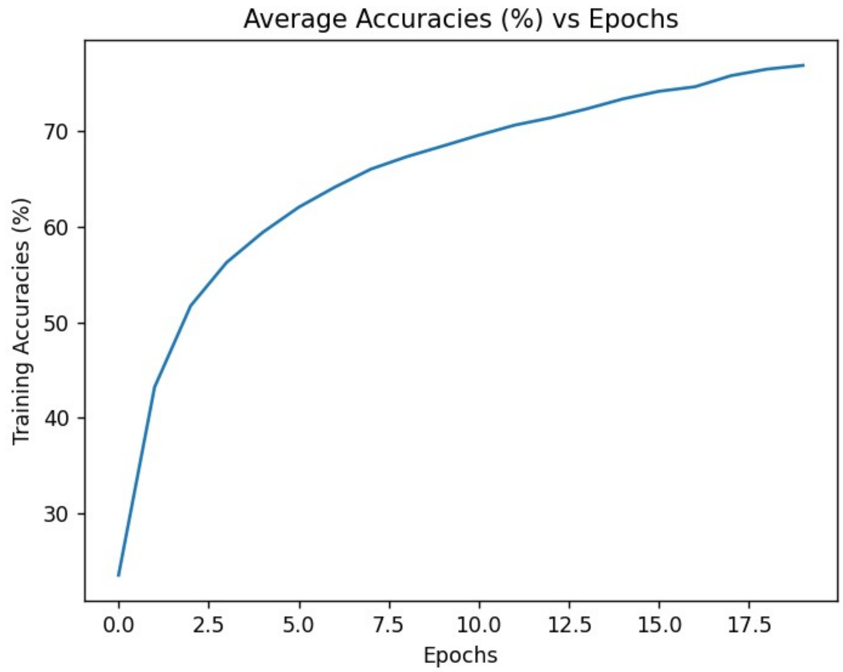


Figure 47: Loss Plot

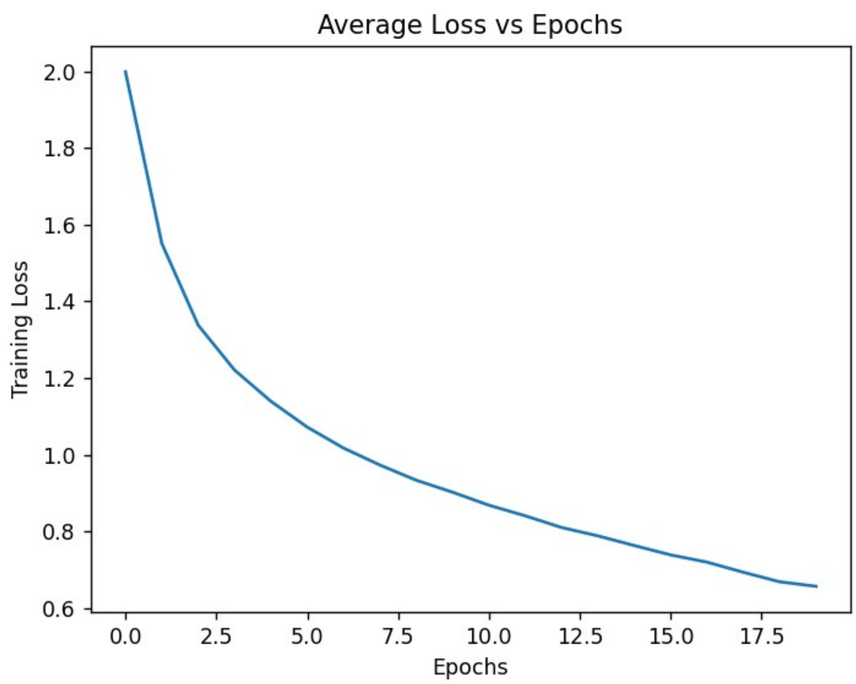


Figure 48: CNN Network for CIFAR Dataset

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(3,16,kernel_size=5,stride=1), #28
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(stride=2, kernel_size=2), #14
            nn.Conv2d(16,16,kernel_size=5,stride=1), #10
            nn.BatchNorm2d(16),
            nn.ReLU(),

            nn.Flatten())

        self.fc1 = nn.Linear(10*10*16, 64)
        self.s1 = nn.Sigmoid()
        self.fc2 = nn.Linear(64, 32)
        self.s2 = nn.Sigmoid()
        self.fc3 = nn.Linear(32,10)

    def forward(self, x):
        x = self.conv(x)
        x = self.fc1(x)
        x = self.s1(x)
        x = self.fc2(x)
        x = self.s2(x)
        x = self.fc3(x)
        return x
```

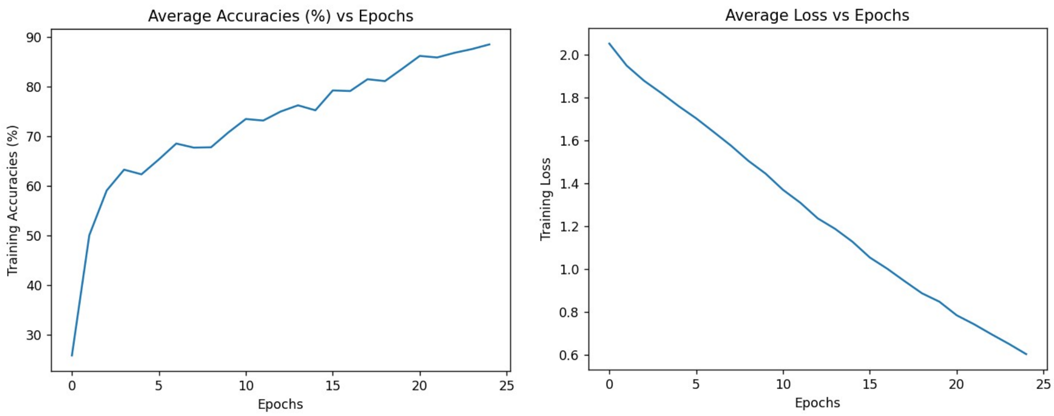
Figure 49: Training for the network

```
transform = transforms.Compose([transforms.ToTensor(),transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
batch_size = 4
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,shuffle=True, num_workers=2)
testset = torchvision.datasets.CIFAR10(root='./data', train=False,download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat','deer', 'dog', 'frog', 'horse', 'ship', 'truck')

epochs = 20
bs = batch_size
model = Net().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

model.train()
acc = []
losses = []
for itr in range(epochs):
    print(itr)
    correct = 0
    total_loss = 0
    count = 0
    for idx, (data, target) in enumerate(trainloader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        #target = torch.max(target, 1)[1]
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

        pred = torch.max(output, 1)[1]
        correct += pred.eq(target).sum().item()
        count = count + 1

    acc.append(100. * correct / (count*bs))
    losses.append(total_loss/count)
```

Q 6.1.4 - Scene Classification - run_q6_scene.py

Best results for scene classification -
* achieved in HW1 was close to 65% after high parameter tuning.
* achieved in this HW using CNN - 88..44% in 25 epochs only.

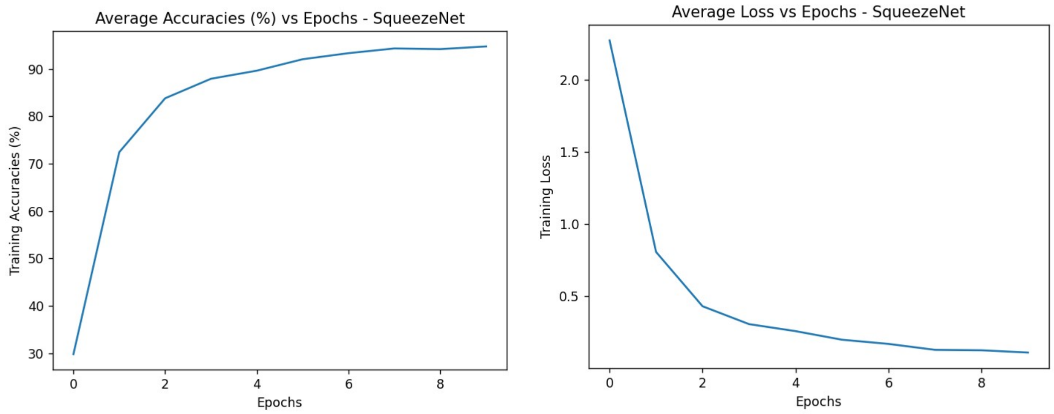
Figure 50: Accuracy and Loss on the SUN Dataset for HW1



There seems no over-fitting with the current epochs and parameters (based on the graphs above). The accuracy with neural networks is much higher compared to the Vision based approach because it is able to learn more features about a particular scene that just 4 filters at different scales as in HW1. First layer of CNN also looks at regions of the image for extracting information from there, just like the spatial pyramid beginning. However, CNN look even at overlapping regions and hence, gathers more information which might have been helpful in better performance.

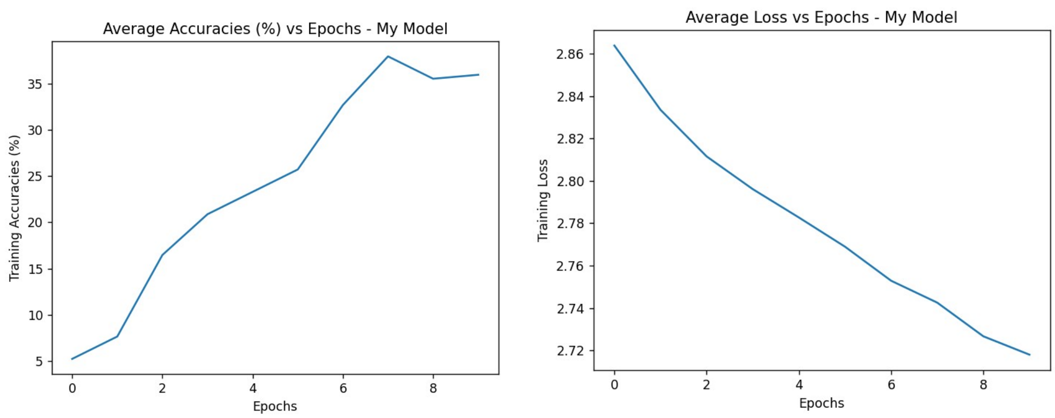
Q 6.2 - Fine Tuning - run_q6_2_fine_tune.py

Figure 51: Accuracy and Loss - SqueezeNet 1.1



After 10 epochs only, the SqueezeNet (pre-trained with only last layer changed) Model reached accuracy over 94%.

Figure 52: Accuracy and Loss - My Model



However, our trained network in 10 epochs performed worse compared to a bigger and pretrained network (as expected). In 10 epochs, it barely gave a training accuracy of over 35%. Seeing the graph it is clear that the network can perform better with more epochs. There is no guarantee that such a small network will perform as good as SqueezeNet with any amount of training. The architecture is in the next page.

Figure 53: Network - My Model: 3 conv. layers and 3 dense layers

```
class Net(nn.Module):
    def __init__(self, num_classes):
        super(Net, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(3,32,kernel_size=5,stride=1), #224 in 220 out
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(stride=2, kernel_size=2), #110
            nn.Conv2d(32,64,kernel_size=3,stride=1), #108
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(stride=2, kernel_size=2), #54
            nn.Conv2d(64,64,kernel_size=3,stride=1), #52
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Flatten()
            )

        self.fc1 = nn.Linear(52*52*64, 1024)
        self.s1 = nn.Sigmoid()
        self.fc2 = nn.Linear(1024, 512)
        self.s2 = nn.Sigmoid()
        self.fc3 = nn.Linear(512,num_classes)

    def forward(self, x):
        x = self.conv(x)
        x = self.fc1(x)
        x = self.s1(x)
        x = self.fc2(x)
        x = self.s2(x)
        x = self.fc3(x)
        return x
```