Q1.1

$$softmax(x) = softmax(x + c)$$

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

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

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

Q1.2

- Range (0,1] and sum over all the elements is equal to 1
- "…a probability distribution of the input x-vector"
- 1 - $s = e^{x_i}$, calculates the exponential results of x_i and preserves the order
  - 2 - $s = \sum s_i$, calculates the sum of s_i (vector normalization value)
  - 3 - $\frac{1}{s} s_i$, Normalizes vector (between 0 and 1) and outputs the probability

Q1.3

Forward propagation of a multi-layer network, in solution below a 3-layer network will be used to show calculation and resulting linear regression problem equation:

Without activation: $y_i = W_i x_i + b_i$ biases

weights inputs

$$\text{i = 1: } y_1 = W_1 x_1 + b_1 \quad \textbf{(1)}$$

$$\text{i = 2: } y_2 = W_2 y_1 + b_2 \quad \textbf{(2)}$$

$$\text{i = 3: } y_3 = W_3 y_2 + b_3 \quad \textbf{(3)}$$

Substitute equations (1) and (2) into equation 3 to get:

$$y_3 = W_3(W_2(W_1 x_1 + b_1) + b_3)$$

$$y_3 = W_3 W_2 W_1 x_1 + W_3 W_2 b_1 + W_3 b_2 + b_3$$

This is the same as solving a linear regression problem, as it is just a linear combination of the weights, biases, and inputs ($y = Wx + b$).

Q1.4

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

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

$$= \frac{e^{-x}}{(1 + e^{-x})^2} = \underbrace{\frac{1}{1 + e^{-x}}}_{\boxed{\sigma(x)}} * \frac{e^{-x}}{(1 + e^{-x})}$$

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

Q1.5

Find $\frac{\partial J}{\partial W}, \frac{\partial J}{\partial x}, \frac{\partial J}{\partial b}$:

$$y = Wx + b$$

Scalar:

$$\text{Using: } y_i = \sum_{j=1}^{d} \left(x_j W_{ij} + b_i\right)$$

$$\frac{\partial J}{\partial W} = \sum_j \frac{\partial J}{\partial y} * \frac{\partial y}{\partial W} = \delta x^T \textbf{ (1)}$$

$$\frac{\partial J}{\partial x} = \sum_j \frac{\partial J}{\partial y} * \frac{\partial y}{\partial x} = W^T \delta \textbf{ (2)}$$

$$\frac{\partial J}{\partial b} = \sum_j \frac{\partial J}{\partial y} * \frac{\partial y}{\partial b} = \delta \textbf{ (3)}$$

Matrix:

$$\begin{bmatrix} \delta_1 x_1 & \delta_1 x_2 & \dots & \delta_1 x_d \\ \delta_2 x_q & \delta_2 x_2 & \dots & \delta_1 x_d \\ \vdots & \ddots & \ddots & \vdots \\ \delta_k x_1 & \delta_k x_2 & \dots & \delta_k x_d \end{bmatrix} \text{ (Matrix for equation 1, } \epsilon R^{k \times d}\text{)}$$

$$\begin{bmatrix} W_1 \delta_1 \\ W_2 \delta_2 \\ \vdots \\ W_d \delta_d \end{bmatrix} \text{(Matrix for equation 2, } \epsilon R^{d \times 1}\text{)}$$

$$\begin{bmatrix} \delta_1 \\ \delta_2 \\ \vdots \\ \delta_k \end{bmatrix} \text{(Matrix for equation 3, } \epsilon R^{k \times 1}\text{)}$$

Q1.6.1

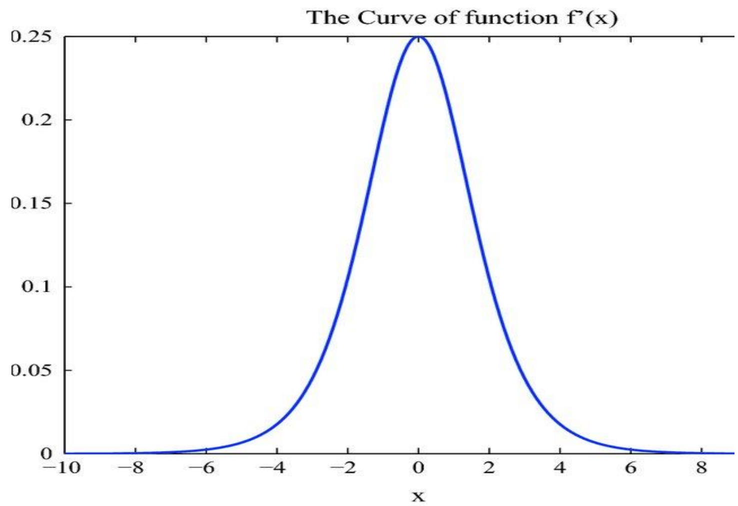


Figure 1: [Link 1]

In the plot shown above, the derivative of sigmoid (σ') is shown and the values range from (0,0.25] whereas the values for sigmoid will range from (0,1]. After applying the activation function (this is down more than once) to multiple layers the values in range of 0.25 will decrease quickly. This might cause the "vanishing gradient" problem, as these values will not affect the training implementation.

Q1.6.2

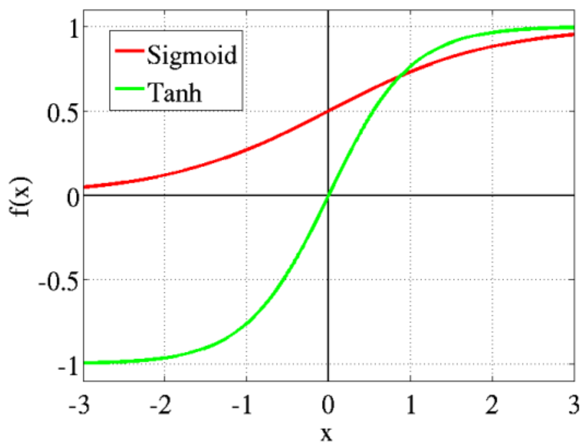


Figure 2: Link 2

As seen in the figure above the output range of sigmoid is (0,1] and tanh is (-1,1); tanh is better because the output will have an average value near zero. Also, tanh will correspond to x "sign" (i.e. when x-negative, tanh-negative and x-positive, tanh-positive).

Q1.6.3

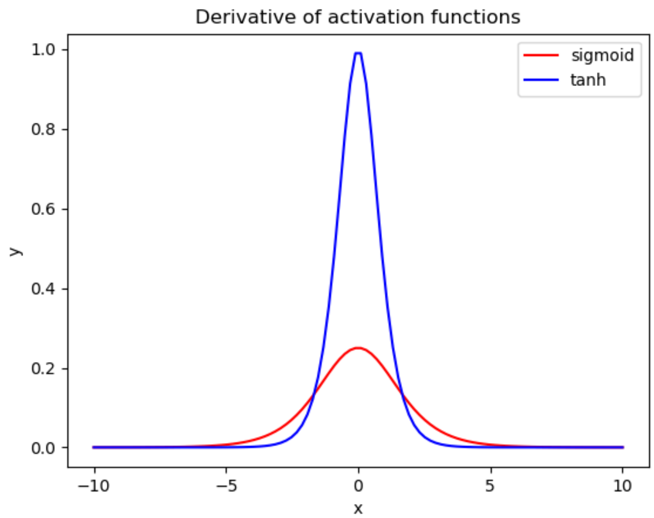


Figure 3: Link 3

The derivative of tanh will range from (0,1) so the gradient will have less of a vanishing gradient as it decreases slower because of the larger value (this is not seen in the derivative of the sigmoid value which ranges from (0,0.25) and vanishes quickly).

Q1.6.4

$$\sigma(x) = \frac{1}{1+e^{-x}} \text{ ; } \tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$$

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

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

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

Q2.1.1

Initializing a network with zeros will cause all nodes to have the same weights (and biases) so all the outputs will be the same for every input. This means that no meaningful information will be gathered. During backward propagation, the gradients will be gathered. During backward propagation, the gradients will be the same value for each node in the layer and the model won't learn. Instead, it will learn only one function versus learning many.

Q2.1.2

```python
############################## Q 2.1 ##############################
# initialize b to 0 vector
# b should be a 1D array, not a 2D array with a singleton dimension
# we will do XW + b.
# X be [Examples, Dimensions]
def initialize_weights(in_size,out_size,params,name=''):
    W, b = None, None
    b = np.sqrt(6/(in_size+out_size))
    W = np.random.uniform(-b,b,(in_size,out_size))

    b = np.zeros((out_size))
    params['W' + name] = W
    params['b' + name] = b

############################## Q 2.2.1 ##############################
# x is a matrix
# a sigmoid activation function

def sigmoid(x):
    res = None
    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
```

```python
    # get the layer parameters
    W = params['W' + name]
    b = params['b' + name]

    #Activation..
    pre_act = np.dot(X,W)
    pre_act = pre_act+b

    #Every value in pre_act
    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
```

Q2.1.3

In order to prevent symmetry during training and eliminate issues of initializing with zeros. This alsp helps to avoid having large values in forward propagation, so the network will converge faster, and exploding during backward propagations can be avoided. Scaling the initialization by layer size, will allow for similar values to be computed in each layer (variance in the weight gradients will be the same). Again, this helps get values that aren't very large and helps avoid the vanishing gradient problem.

Q2.2.1

```python
def sigmoid(x):
    res = None
    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]

    #Activation..
    pre_act = np.dot(X,W)
    pre_act = pre_act+b

    #Every value in pre_act
    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
```

Q2.2.2

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

    #get max x [examples,classes]
    x_max = np.max(x,axis=1,keepdims=True) #Keepdims will return a 2D array, when not using this parameter cannot perform subtraction of matrix
    shifted_x = x-x_max
    x_exp = np.exp(shifted_x)

    x_sum = np.exp(shifted_x).sum(axis=1, keepdims=True)
    res = x_exp/x_sum

    return res
```

Q2.2.3

```
############################## Q 2.2.3 ##############################
# compute total loss and accuracy
# y is size [examples,classes]
# probs is size [examples,classes]

def compute_loss_and_acc(y, probs):
    loss, acc = None, None
    acc = []
    loss = 0


    for i in range (0,y.shape[0]):
        #Compute loss
        loss = loss-np.sum(y[i]*np.log(probs[i]))
        probability = np.argmax(probs[i])
        y_2= np.argmax(y[i])

        #Find where probs and y are equal
        if probability == y_2:
            acc.append(True)
        else:
         acc.append(False)
    acc = np.mean(acc)

    return loss, acc

```

Q2.3

```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]
    # det = np.matmul(delta,activation_deriv(post_act))
    det = delta*activation_deriv(post_act)
    grad_W = np.dot(np.transpose(X),det)
    grad_b = np.sum(det, axis=0)
    grad_X = np.dot(det,np.transpose(W))

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

Q2.4

```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 = []
    #Takes x and y as inputs and splits it into random batches
    batches_num = x.shape[0]/batch_size

    #Get random batches
    random_batches_x_y = np.random.choice(x.shape[0],y.shape[0],False)

    i = 0
    while i < batches_num:
        if len(random_batches_x_y) > batch_size:
            indexes = random_batches_x_y[0:batch_size]
            random_batches_x_y = random_batches_x_y[batch_size:]
        else:
            indexes = random_batches_x_y

        batches.append((x[indexes],y[indexes]))
        i+=1

    return batches

```

```python
# 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:
        # forward
        yl = forward(xb,params,'layer1',sigmoid)
        probas = forward(yl,params,'output',softmax)
        # loss
        loss,acc = compute_loss_and_acc(yb,probas)

        # be sure to add loss and accuracy to epoch totals
        total_loss+=loss
        # avg_acc+=1
        avg_acc +=acc
        # backward
        delta = probas - yb
        delta_2 = backwards(delta, params, 'output', linear_deriv)
        backwards(delta_2, params, 'layer1', sigmoid_deriv)
        # apply gradient

        #W Layer
        params['Wlayer1'] = params['Wlayer1']-learning_rate*params['grad_Wlayer1']
        params["Woutput"] = params["Woutput"] - learning_rate*params["grad_Woutput"]

        #B layer
        params["boutput"] = params["boutput"] - learning_rate*params["grad_boutput"]
        params["blayer1"] = params["blayer1"] - learning_rate*params["grad_blayer1"]

        # gradients should be summed over batch samples
    avg_acc = avg_acc/len(batches)
```

Q2.5

```python
eps = 1e-6
N = v.shape[0]

for k,v in params.items():      # for each value inside the parameter
    if '_' in k:
        continue
    grad_ = params['grad_'+k]
    if 'b' in k: #Bias
        for i in range(0,N):
            v_val = v[i]
            v[i] = v[i] - eps
            #run the network
            h_ = forward(x,params,'layer1')
            prob = forward(h_,params,'output',softmax)

            #Don't reassign loss,acc; get loss
            loss_new,acc_new = compute_loss_and_acc(y,prob)
            v[i] = v_val

            #add epsilon
            v[i] = v[i] + eps
            h_2 = forward(x,params,'layer1')
            prob_2 = forward(h_2,params,'output',softmax)

            loss3,acc3 = compute_loss_and_acc(y,prob_2)
            #   compute derivative with central diffs
            grad_[i] = (loss3-loss_new)/(2*eps)
            #   restore the original parameter value
            v[i] = v[i]-eps
    elif 'W' in k:
        for i in range(v.shape[0]):
            for j in range(v.shape[1]):
                v_val = v[i,j]
                v[i][j] = v[i][j]-eps

                #run the network
                h_ = forward(x,params,'layer1')
```

```python
                prob = forward(h_,params,'output',softmax)
                loss_new, acc_new = compute_loss_and_acc(y, prob)


                v[i][j] = v_val

                #add epsilon
                v[i][j] += eps
                h_2 = forward(x,params,'layer1')
                prob_2 = forward(h_2,params,'output',softmax)

                loss3,acc3 = compute_loss_and_acc(y,prob_2)

                #   compute derivative with central diffs
                grad_[i] = (loss3-loss_new)/(2*eps)
                #   restore the original parameter value
                v[i][j] = v_val
```

Q3.1

Batch size = 32; Learning rate = 3e-3

```
Validation accuracy:  0.7747222222222222
Test accuracy:  0.7916666666666666
```

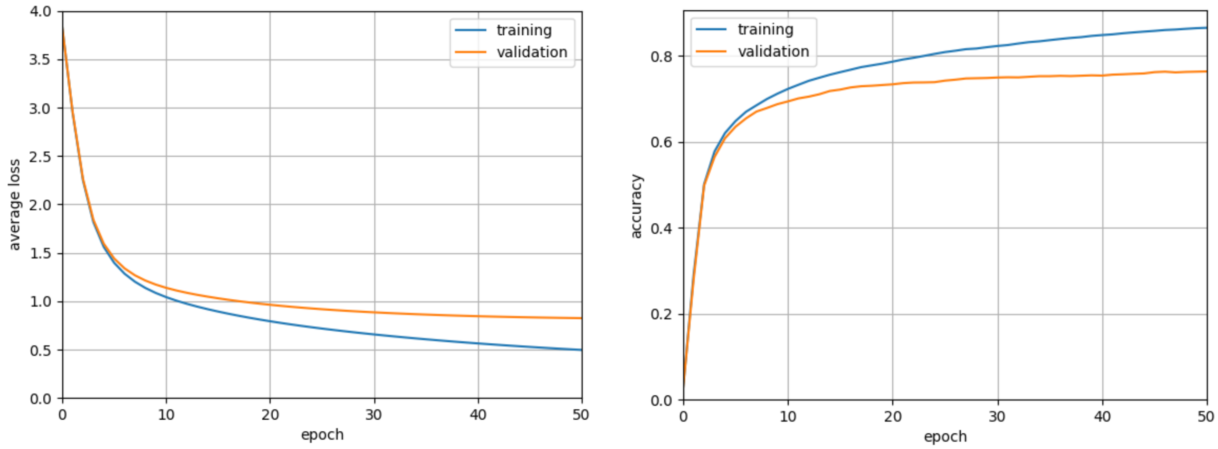
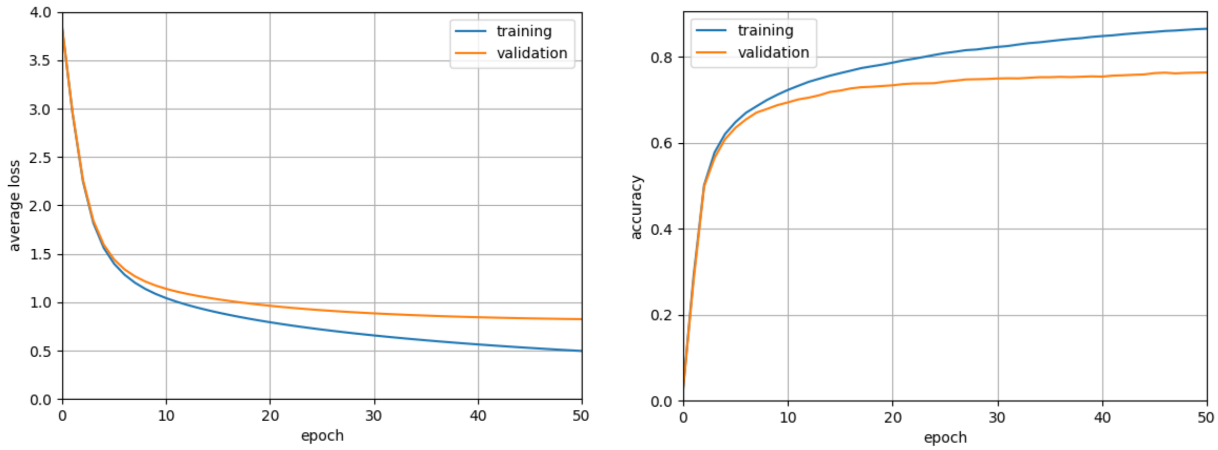


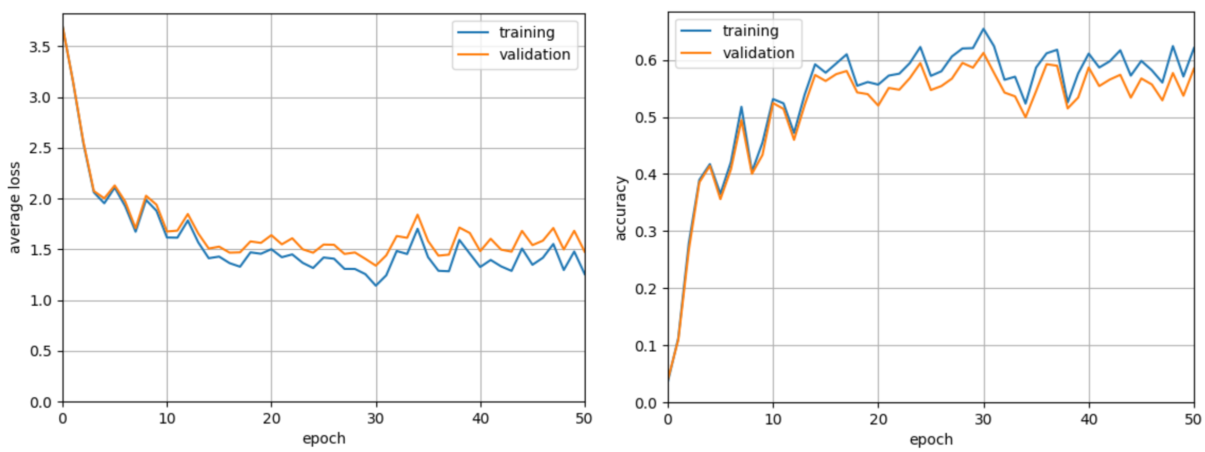
Figure 4: Learning rate of 3e-3

Q3.2



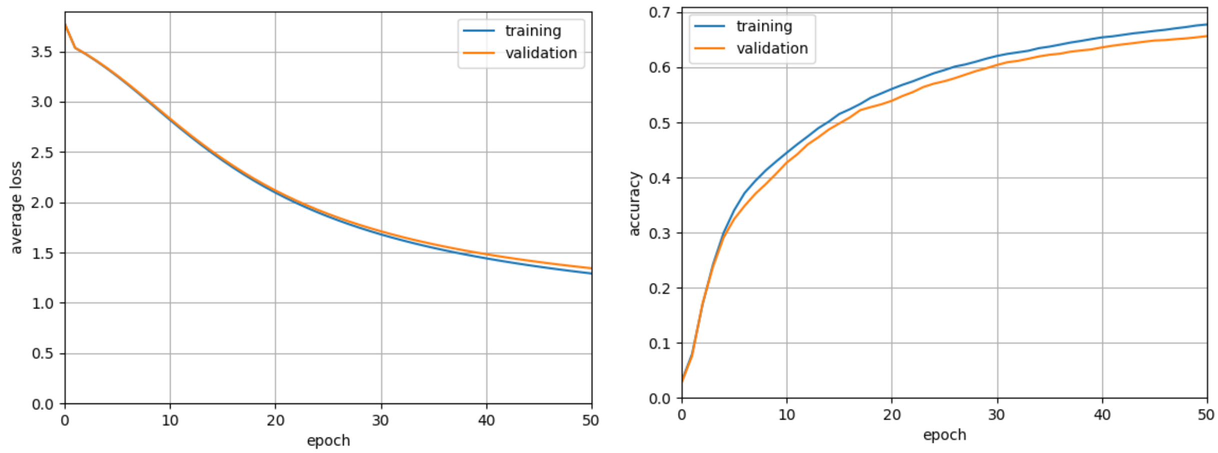

```
Validation accuracy:  0.7747222222222222
Test accuracy:  0.7916666666666666
```

Figure 5: Best Learning rate of 3e-3




```
Validation accuracy:  0.5847222222222223
Test accuracy:  0.5833333333333334
```

Figure 6: Learning rate of 3e-3*10




```
Validation accuracy:  0.6566666666666666
Test accuracy:  0.6605555555555556
```

Figure 7: Learning rate of 3e-3*1/10

The **best learning rate was the tuned learning rate of 3e-3** with a validation accuracy of 77.47% and test accuracy of 79.17%. When the learning rate is larger (see figure 6) there was more oscillations in the loss and the accuracy as the system adjusts, the accuracy was lower compared to the best learning rate and there was also more loss. Then with the decreased learning rate (see figure 7) the accuracy decreased but the system graphs overalls are smooth (smoothest graphs out of all learning rates).

Q3.3

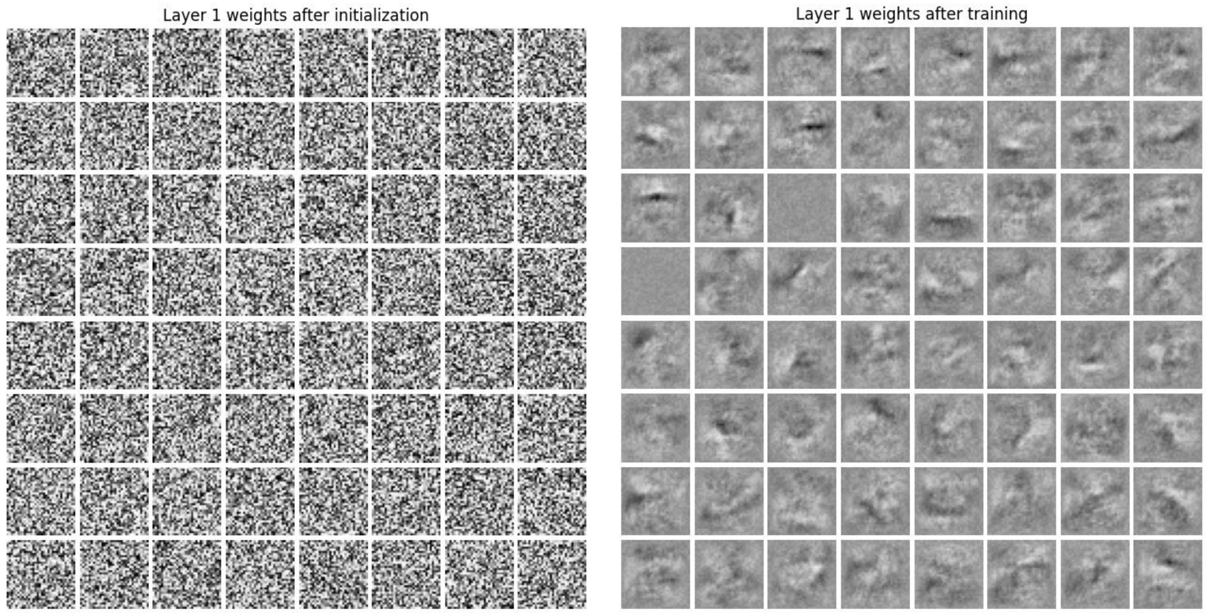


Figure 8: Network of weight after initialization (left) and after training (right)

The weights after initialization show random distribution, while the first layer after training there is more clarity to the weights (can start to see what we are looking for).

Q3.4

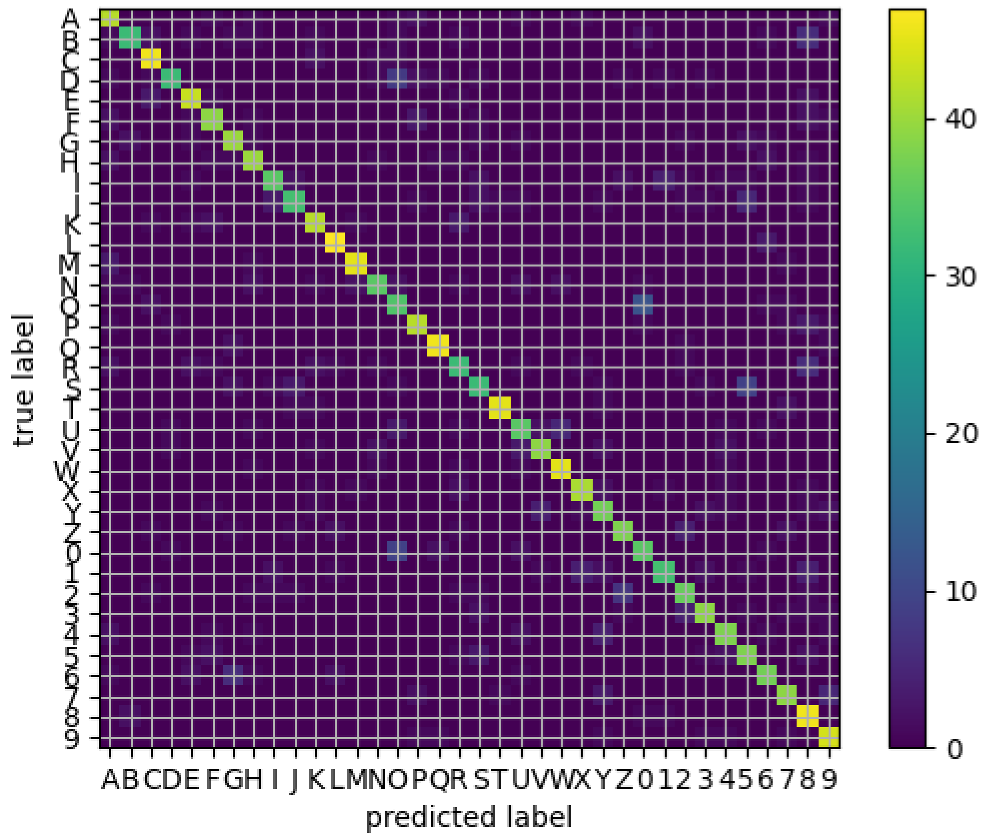


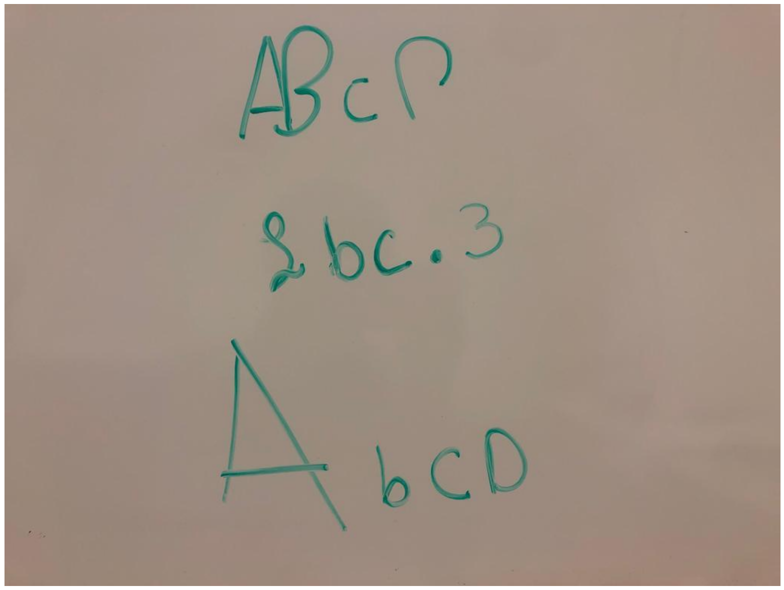
Figure 9: Confusion Matrix

The most commonly confused sets are (from looking at the image above): [‘G’,6], [‘O’,0], [‘S’,5],[‘I’,1] and [‘D’,0]. This is because the shapes look very similar, and the trained network (done by the computer) will have a difficult time distinguishing between the two labels.

Q4.1

Assumptions made:

- Letters are of similar sizes and all text is either numbers or letters, that we have training data available for
- Letters are separate from each other, so no overlapping of characters and each character is fully connected



In the image above I have drawn out some examples of the overlapping characters and incomplete characters (first row). The second row of the image shows two symbols instead of characters, and the last row shows different sizes of characters.

Q4.2

```python
def findLetters(image):
    #Draw rectangles on the image to show the bounding boxes
    # insert processing in here
    # one idea estimate noise -> denoise -> greyscale -> threshold -> morphology -> label -> skip small boxes
    # this can be 10 to 15 lines of code using skimage functions
    bboxes = []
    bw_img = None
    area_box = 0
    area_box_list = []

    #denoise image
    img_noise = skimage.restoration.denoise_bilateral(image, multichannel=True)
    # convert image to grayscale
    img_gray = skimage.color.rgb2gray(img_noise)
    # get a binary image first
    threshold = skimage.filters.threshold_otsu(img_gray)

    #Process the binary image
    mask = img_gray>threshold

    #Use morphology: # Thicken letters; skimage.morphology dilation step
    mask = skimage.morphology.erosion(mask)
    mask = skimage.morphology.erosion(mask)
    mask = skimage.morphology.dilation(mask)

    bw_img = mask
    # mask = skimage.morphology.remove_small_objects(mask, 50)
    # mask = skimage.morphology.remove_small_holes(mask, 50)

    # skimage to find boundaries of the letters; skimage.measure.regionprops
    labels = skimage.measure.label(bw_img, background=1, connectivity=2)
    bb = skimage.measure.regionprops(labels,bw_img)

    # check if boundary box is larger than average box size
    # skip the small boxes
    # determine threshold (100-200) for comparing the top edge of the bounding boxes to the next bbox
    # compare against the letter next to it
```

```python

    for box in bb:
        area_box = area_box+ box.area
        area_box_list.append(area_box)
    avg_box_size = area_box/len(bb)
    for i in bb:
        if i.area >avg_box_size/4:
            bboxes.append(i.bbox)
            # print("here")
    # bw_img = (~bw_img).astype(np.float)
    return bboxes, bw_img
```

Q4.3

Figure 10: Accuracy of 1st image

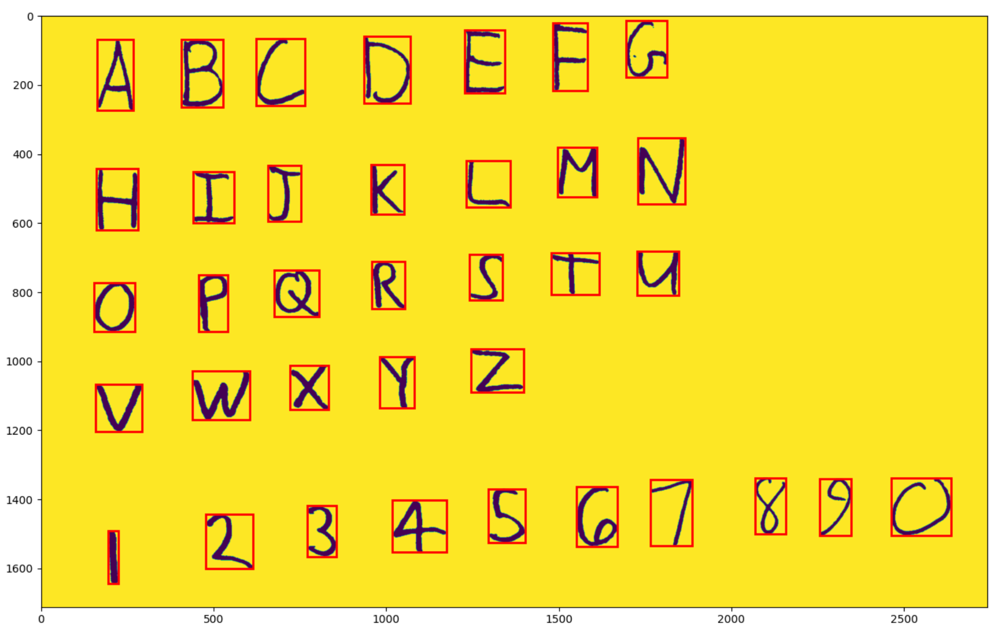


Figure 11: Accuracy of 2nd image

Figure 12: Accuracy of 3rd image

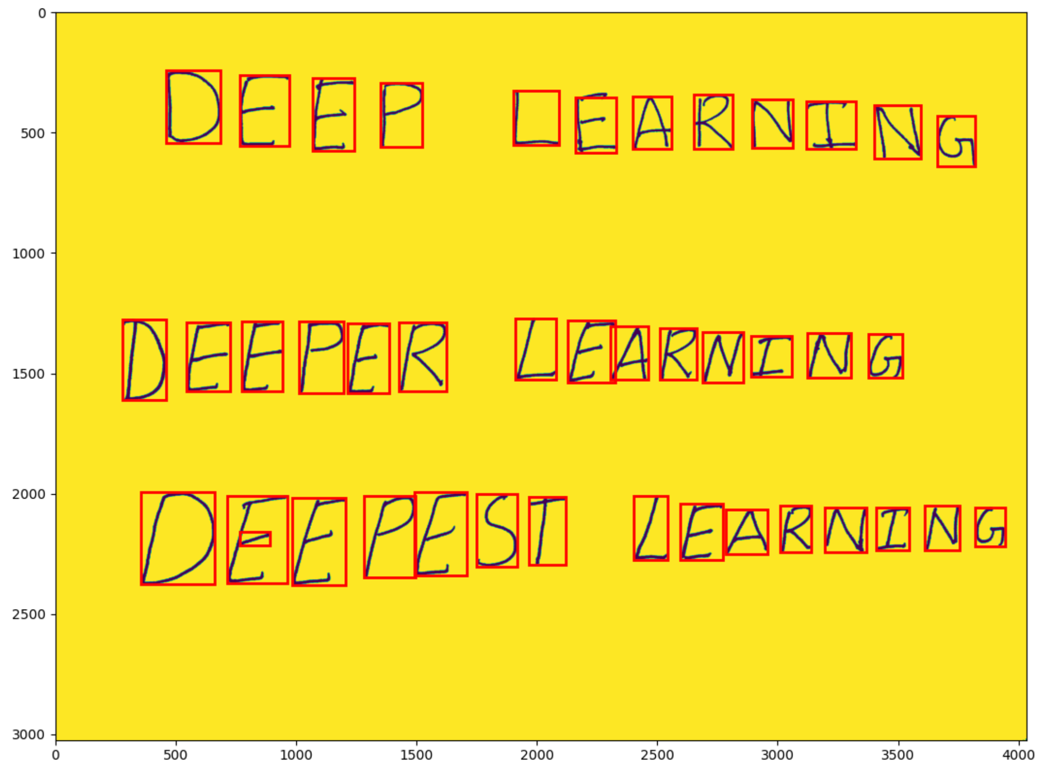


Figure 13: Accuracy of 4th image

Q4.4

01_List:

T0 D0 LIST

I MAXE A TD D0LIST

2 CH8CK OFE 7HE FIR9CT

7HING 0N T0 D0 LIST

3 R2ALIZE YOU HAVE ALREAUY

COMPLET2D ZYHINGS

4 RE4ARD Y0URSELF WITH

A NAP

02_letters:

A B C U E F G

H I J K L M U

Q P Q R J T U

V W X Y Z

8 Z 3 F S G 7 8 7 O

03_haiku:

HAIKUS ARE EMASY

BUT SQMETIMES THEY DON'T MAKG SENGG

M

REFRIGERAT90R

04_deep:

D5EP LC2RMING

DFFYFX L8AKNIXG

DFECPFST LFARNING

This was the best accuracy/classification of the images that I could get, even after performing dilation to make the characters thicker such that they would match (or look similar) to the thicker characters that the network was trained on in 3.1. I suspect there might not be enough dilation but there was not a noticeable difference in the characters even after multiple "rounds" of dilation.

Q5.1.1

```python
# Q5.1 & Q5.2
# initialize layers here
initialize_weights(1024, hidden_size, params, 'layer1') #ReLU
initialize_weights(hidden_size, hidden_size, params, 'hidden1') #ReLU
initialize_weights(hidden_size, hidden_size, params, 'hidden2') #ReLU
initialize_weights(hidden_size, 1024, params, 'output') #Sigmoid

# should look like your previous training loops
losses = []
for itr in range(max_iters):
    total_loss = 0
    for xb,_ in batches:

        #forward
        h = forward(xb,params,'layer1',relu)
        h_1 = forward(h,params,'hidden',relu)
        h_2 = forward(h_1,params,'hidden2',relu)
        output_ = forward(h_2,params,'output',sigmoid)

        # loss
        p_X = np.square(output_-xb)
        loss = p_X.sum()

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

        # backward
        delta = output_ - xb
        delta_2 = backwards(2*delta, params, 'output', sigmoid_deriv)
        delta_3 = backwards(delta_2, params, 'hidden2', relu_deriv)
        delta_4 = backwards(delta_3, params, 'hidden', relu_deriv)
        backwards(delta_4,params,'layer1',relu_deriv)
```

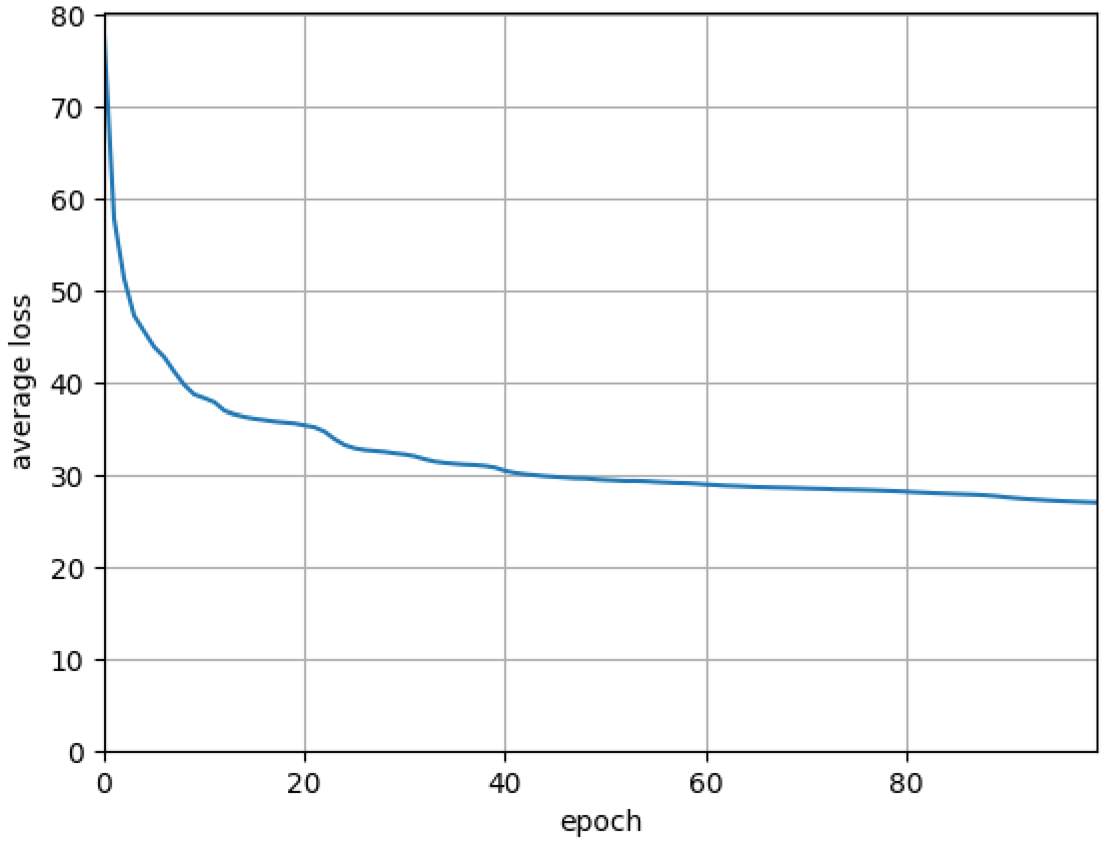
```python

        #print(params.keys())
        params_w_list = ['Wlayer1','Wlayer2','Wlayer3','Woutput','Whidden1','Whidden2']
        params_b_list = ['blayer1','blayer2','blayer3','boutput','bhidden1','bhidden2']

        # apply gradient  (5.1.1)
        #W Layer
        for i in params_w_list:
            params[i] -=learning_rate*params['grad_'+ i]
        #B layer
        for j in params_b_list:
            params[j] -=learning_rate*params['grad_'+ j]



    losses.append(total_loss/train_x.shape[0])
    if itr % 2 == 0:
        print("itr: {:02d} \t loss: {:.2f}".format(itr,total_loss))
    if itr % lr_rate == lr_rate-1:
        learning_rate *= 0.9
```

Q5.1.2

```
# should look like your previous training loops
losses = []
for itr in range(max_iters):
    total_loss = 0
    for xb,_ in batches:

        #forward
        h = forward(xb,params,'layer1',relu)
        h_1 = forward(h,params,'hidden',relu)
        h_2 = forward(h_1,params,'hidden2',relu)
        output_ = forward(h_2,params,'output',sigmoid)

        # loss
        p_X = np.square(output_-xb)
        loss = p_X.sum()

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

        # backward
        delta = output_ - xb
        delta_2 = backwards(2*delta, params, 'output', sigmoid_deriv)
        delta_3 = backwards(delta_2, params, 'hidden2', relu_deriv)
        delta_4 = backwards(delta_3, params, 'hidden', relu_deriv)
        backwards(delta_4,params,'layer1',relu_deriv)

        params_w_list = ['Wlayer1','Wlayer2','Wlayer3','Woutput']
        params_b_list = ['blayer1','blayer2','blayer3','boutput']

        # apply gradient  (5.1.1)
        #W Layer
        # for i in params_w_list:
        #     params[i] +=learning_rate*params['grad_'+ i]
        # # #B layer
```

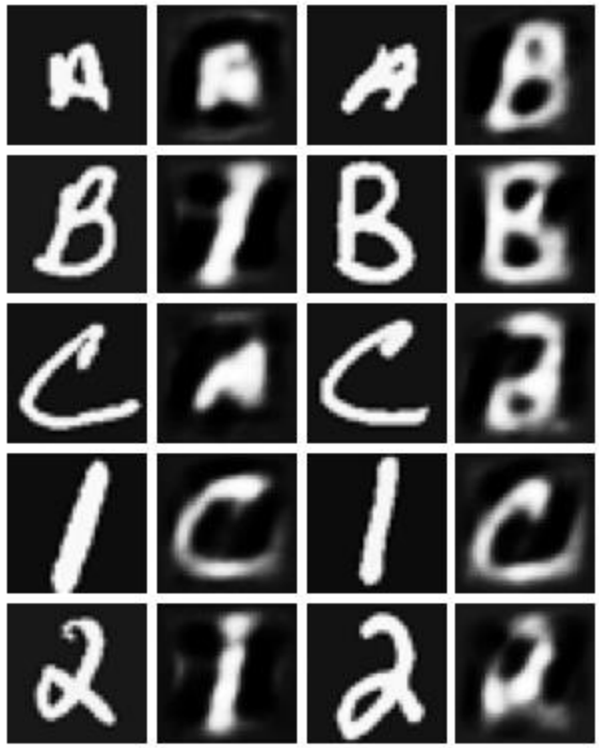
```
        #print(params.keys())
        params_w_list = ['Wlayer1','Wlayer2','Wlayer3','Woutput','Whidden1','Whidden2']
        params_b_list = ['blayer1','blayer2','blayer3','boutput','bhidden1','bhidden2']

        # apply gradient  (5.1.1)
        # #W Layer
        # for i in params_w_list:
        #     params[i] -=learning_rate*params['grad_'+ i]
        # #B layer
        # for j in params_b_list:
        #     params[j] -=learning_rate*params['grad_'+ j]

        #Momentum
        #just use 'm_'+name variables
        #W layer

        for i in params_w_list:
            params['m_'+i] = 0.9*params['m_'+i] - learning_rate*params['grad_'+i]
            params[i] += params['m_'+ i]
        #B layer
        for j in params_b_list:
            params['m_'+j] = 0.9*params['m_'+j] - learning_rate*params['grad_'+j]
            params[j] += params['m_'+ j]

    losses.append(total_loss/train_x.shape[0])
```

Q5.2



For the first 0-20 epochs the average loss shows a sharp decrease before it starts to slow down (near the 20^{th} epoch), then decrease in loss remains steady (i.e., the rate of change begins to plateau and lull).

Q5.3.1



In the reconstructed images, the characters are less visible and more blurred and while there are matches between some of the characters this is not the case for all of them. The network is essentially taking the most notable features of the original character and attempting to reconstruct this in the reconstructed image.

Q5.3.2

```
15.57999246829877
```

The PSNR value achieved was around 15.58%.

Q6

- For this homework assignment I opted to do the extra credit question, as I did not have time to finish implementing question 6. I will be implementing after the deadline, for my own learning purposes as it is important for me to know but I unfortunately do not have the time needed to complete this question.