16720 – hw5
mrohitth

Problem 1.1

Consider the vector $x + c$, when we apply $softmax(x + c)$, we get,

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

$$= \frac{e^{x_i} \,.\, e^{c}}{\sum_j e^{x_j} .\, e^{c}}$$

$$= \frac{e^{x_i} \,.\, e^{c}}{e^{c}\,(\sum_j e^{x_j})}$$

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

The above proof holds for all $x_i \in x$. Thus, $softmax(x + c) = softmax(x)$ $\forall c \in R$

When we use $c = -\max x_i$ or subtract the maximum value exponent term, we get $e^0 = 1$ and all the other terms will be scaled between 0 and 1, thus avoiding large exponent terms which improves numerical stability and avoids overflows.

Problem 1.2

- The range of each element is between (0,1]. The sum over all elements of $softmax(x)$ is 1.
- Softmax takes a real-valued vector x and converts it into a probability distribution - where the value of softmax(x_i) specifies the probability of choosing that value among all other values.
- The first step takes the values and maps it in the positive space between [0, ∞]. The second step acts as a normalization step, where we divide the result of the first step with a constant and the third step outputs the probability of the occurrence of the value x_i given all other values.

Problem 1.3

A network without non-linear activation functions will have the x value change according to linear function: $x_{i+1} = W_i x_i + b_i$
When applying to multi-layer neural networks, we have:

$$\begin{aligned}
y &= W_n x_n + b_n \\
&= W_n(W_{n-1}x_{n-1} + b_{n-1}) + b_n \\
&= WnWn{-}1xn{-}1 + Wnbn{-}1 + bn \\
&= W' x_{n-} + b' \\
&= W' (W_{n-2}x_{n-2} + b_{n-2}) + b' \\
&\ldots\ldots\ldots\ldots \\
&= Wx + b
\end{aligned}$$

which is the same as solving a linear regression problem

16720 – hw5
mrohitth

Problem 1.4

Consider the sigmoid activation function, we get,

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

Taking the derivative of the function, we get,

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

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

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

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

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

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

$$= \sigma(x)\,(1 - \sigma(x))$$

Problem 1.5

$$\frac{dJ}{dy} = \delta$$

$$\frac{dJ}{dy_j} = \delta \quad for\ j\ in\ 1\ to\ k$$

$$\frac{dJ}{dW_{ij}} = \frac{dJ}{dy_j}\frac{dy_j}{dW_{ij}} = \delta_j x_i$$

$$\frac{dJ}{dx_i} = \frac{dJ}{dy_j}\frac{dy_j}{dx_i} = \sum W_{ij} * \delta_j$$

$$\frac{dJ}{db_j} = \frac{dJ}{dy_j}\frac{dy_j}{db_j} = \delta_j$$

Thus, forming the matrices, we get,

$$\frac{dJ}{dW} = x\delta^T$$

$$\frac{dJ}{dx} = W\delta$$

$$\frac{dJ}{db} = \delta$$

Problem 1.6

- When we take the sigmoid function, it maps the input space which is in R to [0,1]. As a result of this, there are large regions of the input space which are mapped to an extremely small range. In these regions of the input space, even a large change in the input will produce a small change in the output - hence the gradient is small. Consider the gradient of the sigmoid between [0,1], it is constrained between [0,0.25]

- When the gradient is propagated through the network, the gradients become smaller and smaller, thus after several layers, the gradients almost "vanish", that is they become zero quickly - resulting in almost no change in the parameters of the network, making it difficult to train the network.

- The output range of the sigmoid function is [0,1] and the output range of the tanh function is [-1,1]. Choosing the tanh function is more preferable as it maps the input space which is in R to a large range of values and also, it maps positivevalues to positive values and negative values to negative values.

- The derivative values of tanh(x) are constrained between [0,1] for values between [-1,1], thus having stronger and largerderivatives for the same inputs compared to the sigmoid function. Thus, tanh(x) would have lesser of a vanishing gradient problem compared to sigmoid(x).

- To make tanh(x) have the range of sigmoid(x),

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

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

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

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

$$= 2\sigma(x)$$

So, tanh(x) + 1 = 2σ(2x)

And tanh(x) = 2σ(2x) - 1

Problem 2.1.1

Initializing all the weights to zero does not prove to be useful. If we do so, the weights we learn will be the same for the same values of inputs across iterations and thus, the network will not be able to effectively map the input space to the output space.

In the event we take a network with a few linear activations (especially in the last few layers) and also take the weights and biases to be zero, then the network will essentially not learn anything and the output of the network would always be zero.

Problem 2.1.2

```
############################## 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=''):

    bound = np.sqrt(6) / np.sqrt(in_size + out_size)
    W, b = np.random.uniform(-bound, bound, (in_size, out_size)), np.zeros(out_size)

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

Problem 2.1.3

Random weights help as they allow for the network to compute different updates for different units and this ensures that no input patterns are lost in null space of the forward propagation and no gradient patterns are lost in the null space of the backward propagation.

Scaling the initialization based on the layer size helps to keep the information flowing in the network, by allowing the variance of the activations in subsequent layers be the same during forward propagation and the variance of the weights in subsequent layers be same during backward propagation.

Problem 2.2.1

```python
############################### Q 2.2.1 ###############################
# x is a matrix
# a sigmoid activation function
def sigmoid(x):

    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]


    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
```

Problem 2.2.2

```python
############################### Q 2.2.2  ###############################
# x is [examples,classes]
# softmax should be done for each row
def softmax(x):

    ss = np.max(x, axis=1)
    ss = ss[:, np.newaxis]
    ex = np.exp(x-ss)
    div = np.sum(ex, axis=1)
    div = div[:, np.newaxis]
    res = ex/div

    return res
```

Problem 2.2.3

```python
############################## 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, 0

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

    for i in range(y.shape[0]):
        if(np.argmax(y[i, :]) == np.argmax(probs[i, :])):
            acc+=1
    acc = acc/y.shape[0]


    return loss, acc
```

Problem 2.3

```python
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

    grad_op = activation_deriv(post_act)*delta

    grad_W = np.matmul(X.T, grad_op)
    grad_b = np.sum(grad_op, axis=0)
    grad_X = np.matmul(W, grad_op.T).T


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

Problem 2.4

```
############################### 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 = []

    random_batches = np.array_split(np.random.permutation(x.shape[0]), x.shape[0]//batch_size)

    for i in random_batches:
        batches.append((x[i,:],y[i,:]))

    return 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_acc=0
    total_loss = 0
    avg_acc = 0
    for xb,yb in batches:
        ##########################
        ##### your code here #####
        ##########################
        pass
        # forward
        h1 = forward(xb, params, 'layer1', sigmoid)
        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
        delta = probs-yb
        delta = backwards(delta, params, 'output', linear_deriv)
        backwards(delta, params, 'layer1', sigmoid_deriv)

        # apply gradient
        # gradients should be summed over batch samples
        for layer in ['output', 'layer1']:
            params['W' + layer] -= learning_rate * params['grad_W' + layer]
            params['b' + layer] -= learning_rate * params['grad_b' + layer]

    avg_acc = total_acc/batch_num

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

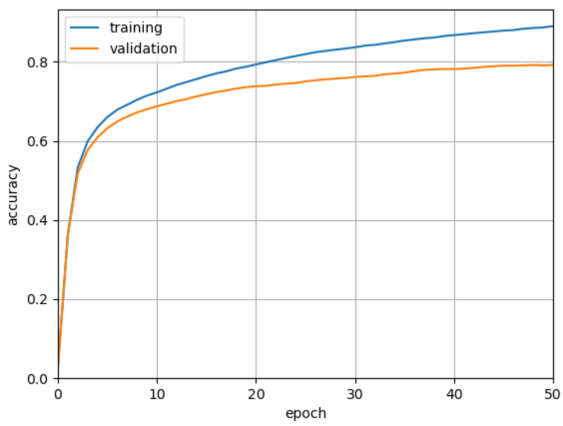
Problem 2.5

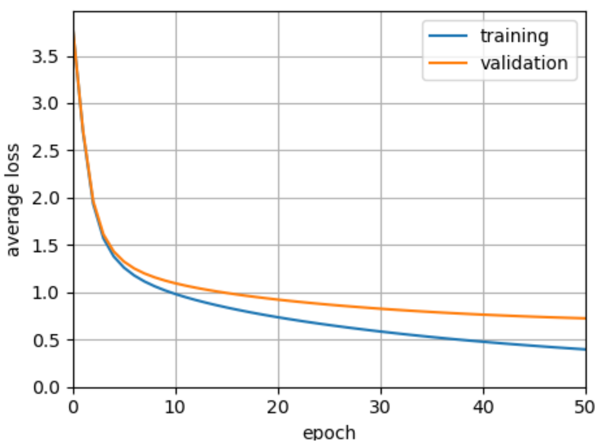
```
# compute gradients using finite difference
eps = 1e-6
for k,v in params.items():
    if '_' in k:
        continue

    if 'W' in k:
        for r in range(v.shape[0]):
            for c in range(v.shape[1]):
                d = np.zeros(v.shape)
                d[r, c] = 1

                params1 = copy.deepcopy(params)
                params1[k] = v + eps*d
                h1 = forward(xb, params1, 'layer1', sigmoid)
                probs = forward(h1, params1, 'output', softmax)
                f1, _ = compute_loss_and_acc(yb, probs)

                params2 = copy.deepcopy(params)
                params2[k] = v - eps * d
                h1 = forward(xb, params2, 'layer1', sigmoid)
                probs = forward(h1, params2, 'output', softmax)
                f2, _ = compute_loss_and_acc(yb, probs)

                params['grad_' + k][r, c] = (f1 - f2) / (2*eps)

    # Bias
    else:
        for i in range(v.shape[0]):
            d = np.zeros(v.shape)
            d[i] = 1
            params1 = copy.deepcopy(params)
            params1[k] = v + eps*d
            h1 = forward(xb, params1, 'layer1', sigmoid)
            probs = forward(h1, params1, 'output', softmax)
            f1, _ = compute_loss_and_acc(yb, probs)

            params2 = copy.deepcopy(params)
            params2[k] = v - eps*d
            h1 = forward(xb, params2, 'layer1', sigmoid)
            probs = forward(h1, params2, 'output', softmax)
            f2, _ = compute_loss_and_acc(yb, probs)

            params['grad_' + k][i] =  (f1 - f2) / (2*eps)

total_error = 0
for k in params.keys():
```

Problem 3.1

```
PS C:\Users\mathe\OneDrive\Desktop\CV\hw5\hw5\python> python .\run_q3.py
itr: 00         loss: 34852.33          acc : 0.18
itr: 02         loss: 18811.02          acc : 0.57
itr: 04         loss: 14243.28          acc : 0.65
itr: 06         loss: 12370.75          acc : 0.69
itr: 08         loss: 11235.84          acc : 0.72
itr: 10         loss: 10406.66          acc : 0.74
itr: 12         loss: 9737.54   acc : 0.75
itr: 14         loss: 9165.39   acc : 0.77
itr: 16         loss: 8658.59   acc : 0.78
itr: 18         loss: 8199.68   acc : 0.79
itr: 20         loss: 7778.33   acc : 0.80
itr: 22         loss: 7388.05   acc : 0.81
itr: 24         loss: 7024.56   acc : 0.82
itr: 26         loss: 6684.85   acc : 0.83
itr: 28         loss: 6366.69   acc : 0.84
itr: 30         loss: 6068.27   acc : 0.85
itr: 32         loss: 5787.98   acc : 0.86
itr: 34         loss: 5524.35   acc : 0.87
itr: 36         loss: 5276.06   acc : 0.87
itr: 38         loss: 5041.87   acc : 0.88
itr: 40         loss: 4820.70   acc : 0.88
itr: 42         loss: 4611.56   acc : 0.89
itr: 44         loss: 4413.60   acc : 0.90
itr: 46         loss: 4226.04   acc : 0.90
itr: 48         loss: 4048.20   acc : 0.91
Validation accuracy:  0.7916666666666666
Test accuracy:  0.7905555555555556
```

The validation accuracy of 79% was obtained with **learning rate = 0.003 or 3e-3** and **batch size=5**. The plots of loss and accuracy of the model trained for 50 epochs are shown below.

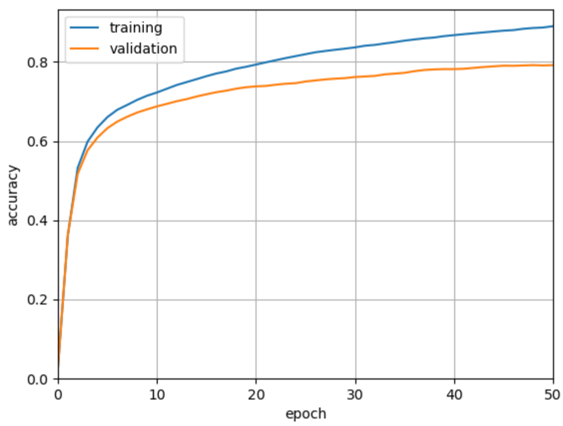


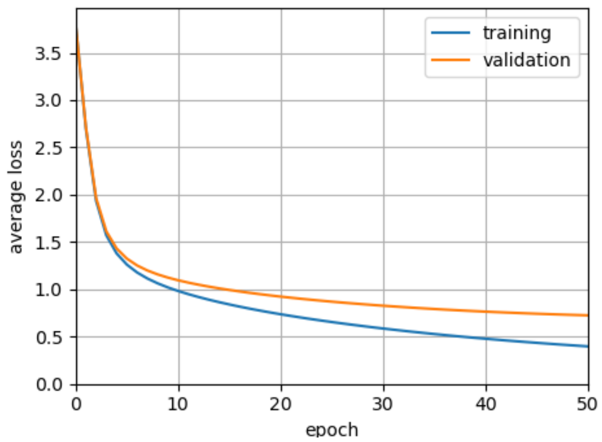


Problem 3.2

1) Highest Accuracy

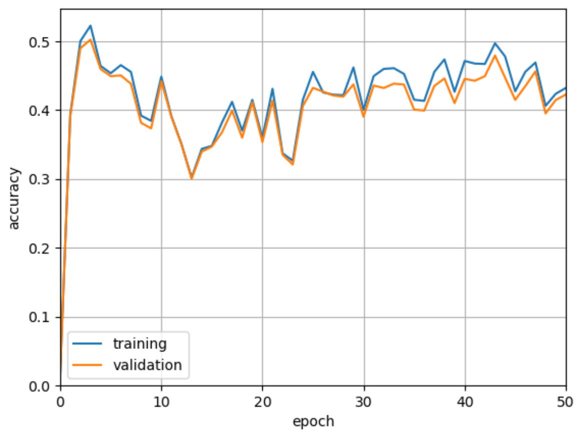
The validation accuracy of 79% was obtained with **learning rate = 0.003 or 3e-3** and **batch size=5**. The plots of loss and accuracy of the model trained for 50 epochs are shown below.

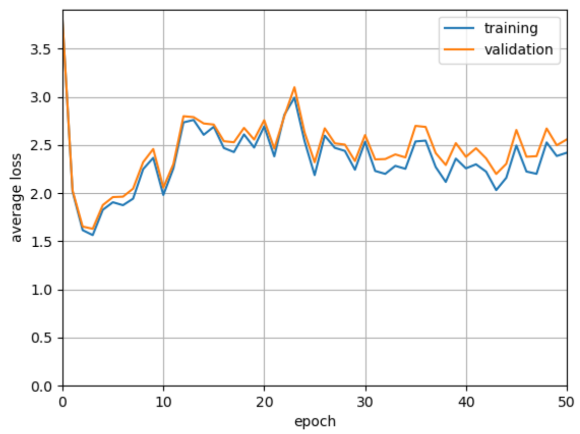




2) Higher Learning Rate

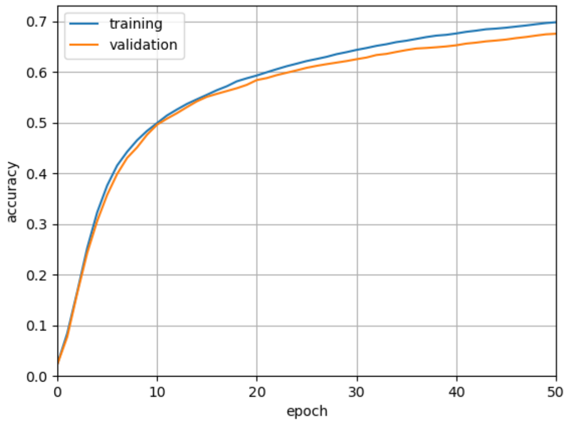
When the learning rate was 0.03 (10 times higher), the losses were higher than the previous case while the accuracy was lower. The curves were also bumpier because the step size was too large.

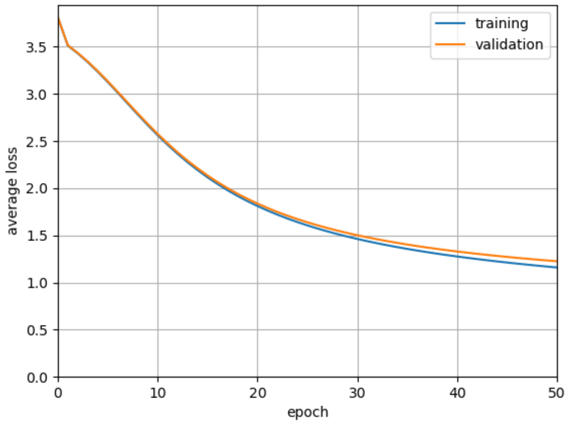




3) Lower Learning Rate

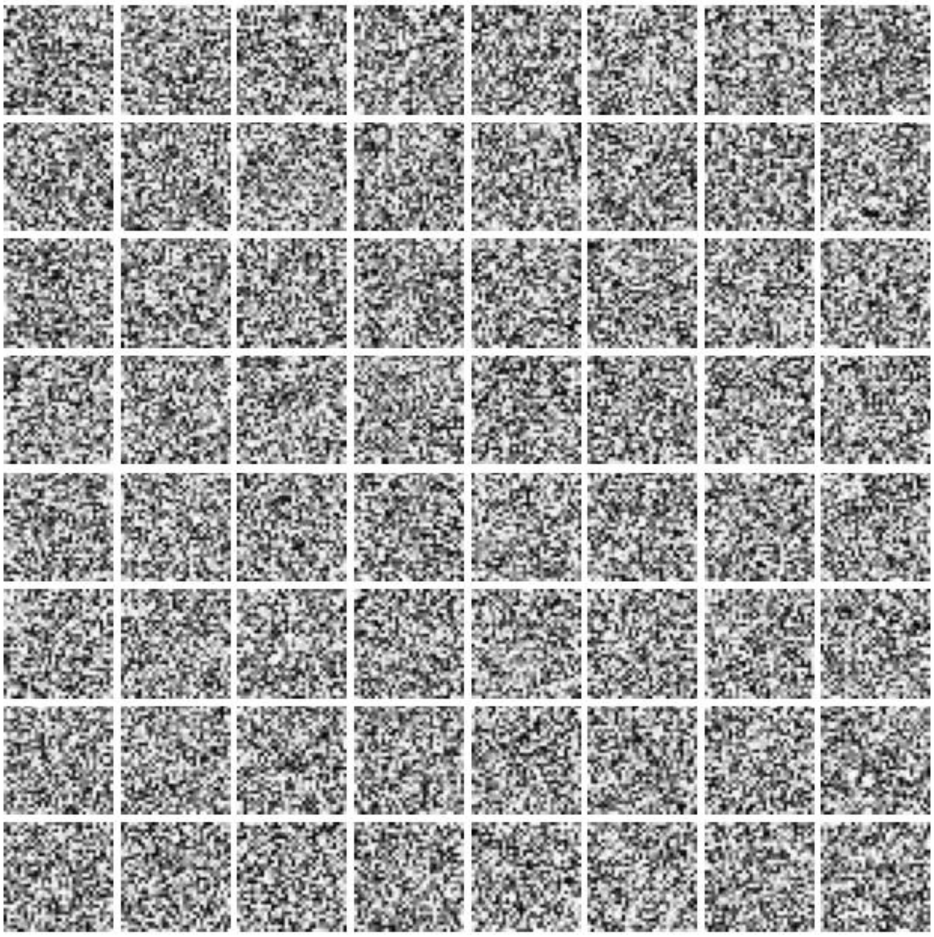
When the learning rate was 0.0003 (1/10 times), the curves are smooth as those shown, but because the step size was too small, it did not converge to the optimum within the maximum number of iterations (50) that we set.

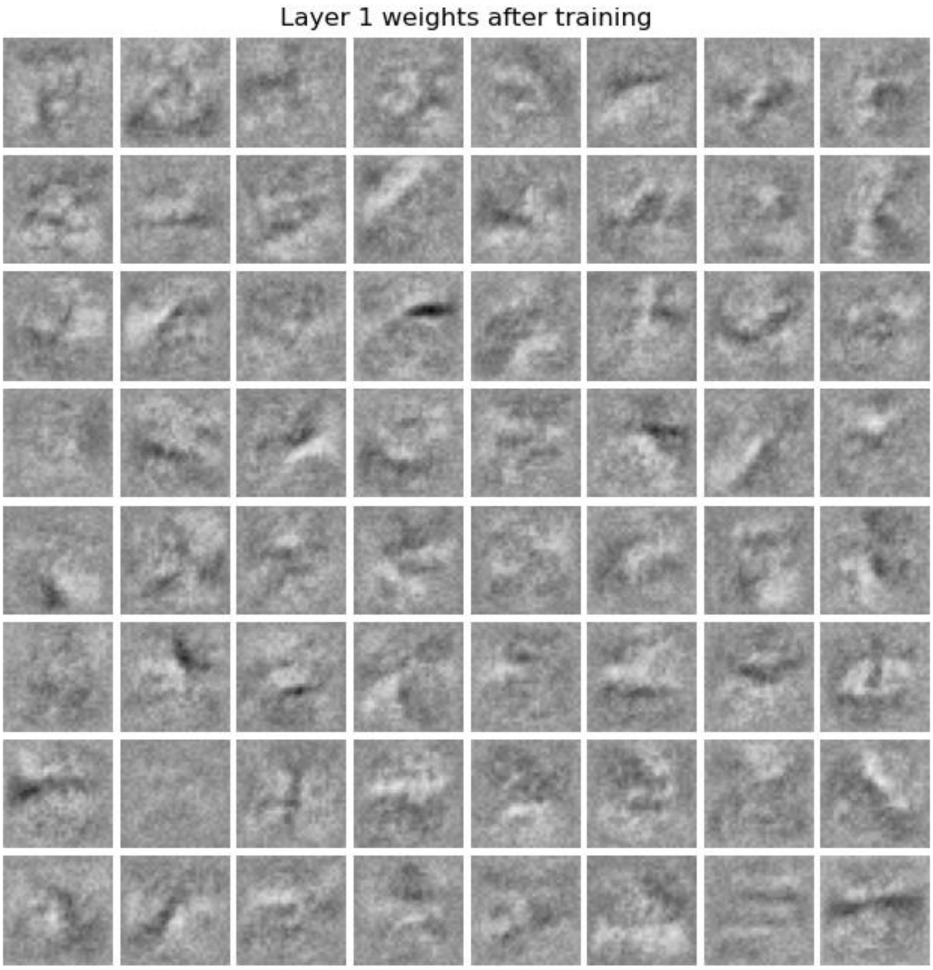




Problem 3.3

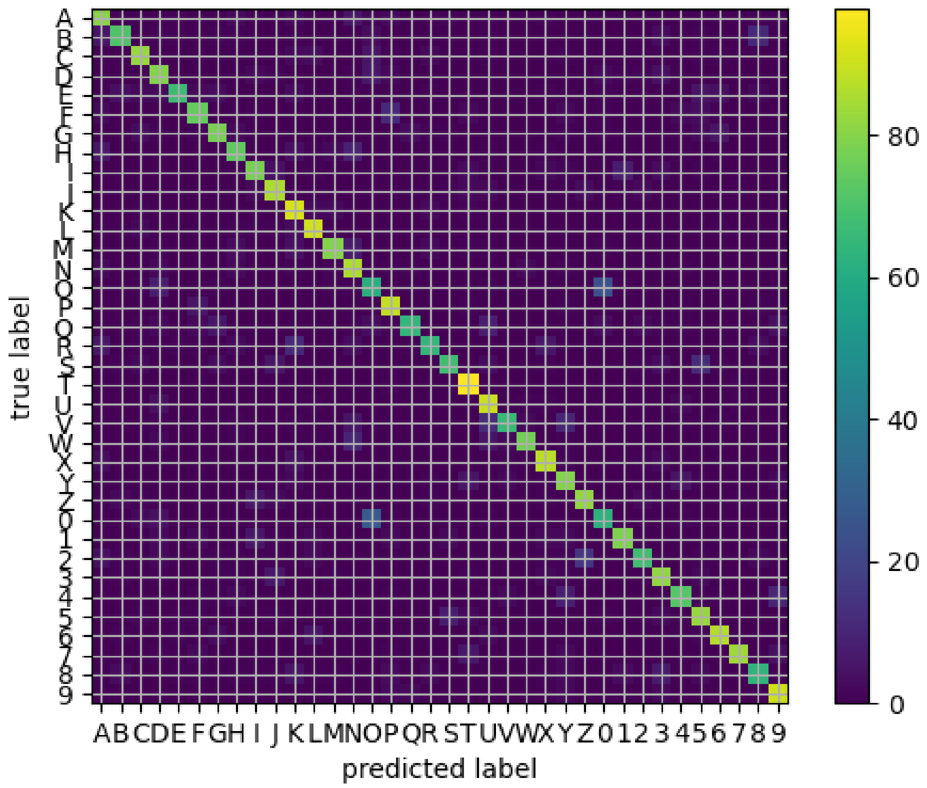
Layer 1 weights after initialization





On comparing the two figures, we can see that the initial weights have no patterns in them as we initialized the layers with random uniform distribution. The weights after 50 epoch training have some more clear patterns, visible strokes of the letters and numbers which look somewhat averaged over the dataset

Problem 3.4



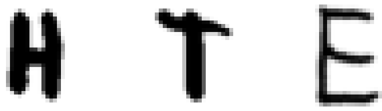
From the above confusion matrix, it can be seen that the brighter the grid, the greater the number of correctly predicted labels in the grid. Since the main diagonal is the heaviest, we can conclude that the results are pretty good.

After training 50 epochs the top few pairs that are commonly confused are number ’0’ and letter ’O’, ’2’ and ’Z’, ’5’ and ’S’, ’4’ and ’Y’, ’0’ and ’D’, which are often misjudged manuallytoo.

Problem 4.1

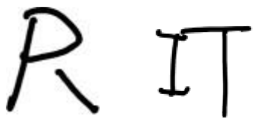
The assumptions the sample method makes are as follows:

1) The stroke width, style of writing of the characters extracted are assumed to be similar to the NIST36 dataset that were trained. Also, the pixel values of the extracted characters are assumed to be similar to the ones in the dataset, however the dataset does not have perfect binary black-and-white images upon inspection.



From the above image, the stroke widths are different than in the dataset and theseimages might fail to be detected correctly

2) The cropped image is assumed to not have any noises, any overlap between two lettersand parts of letters to be fully connected.

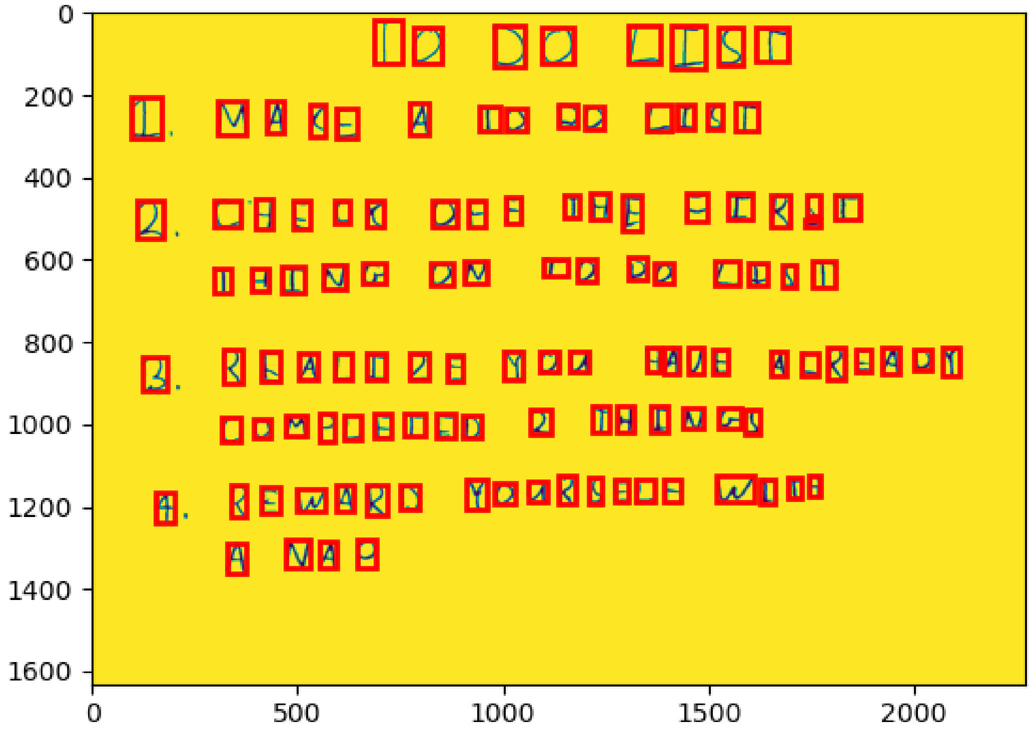


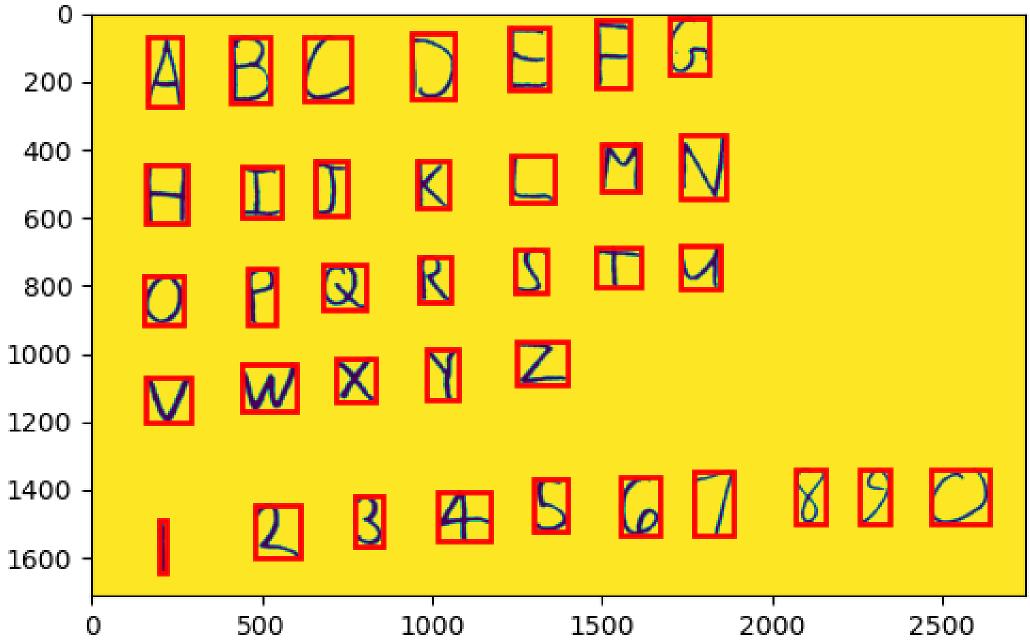
In the above image, the first letter has incomplete parts, and second letter and thirdletters are overlapping. This image might fail to be detected correctly

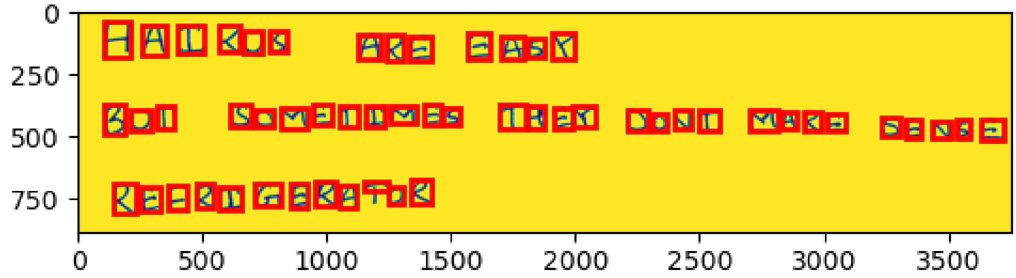
Problem 4.2

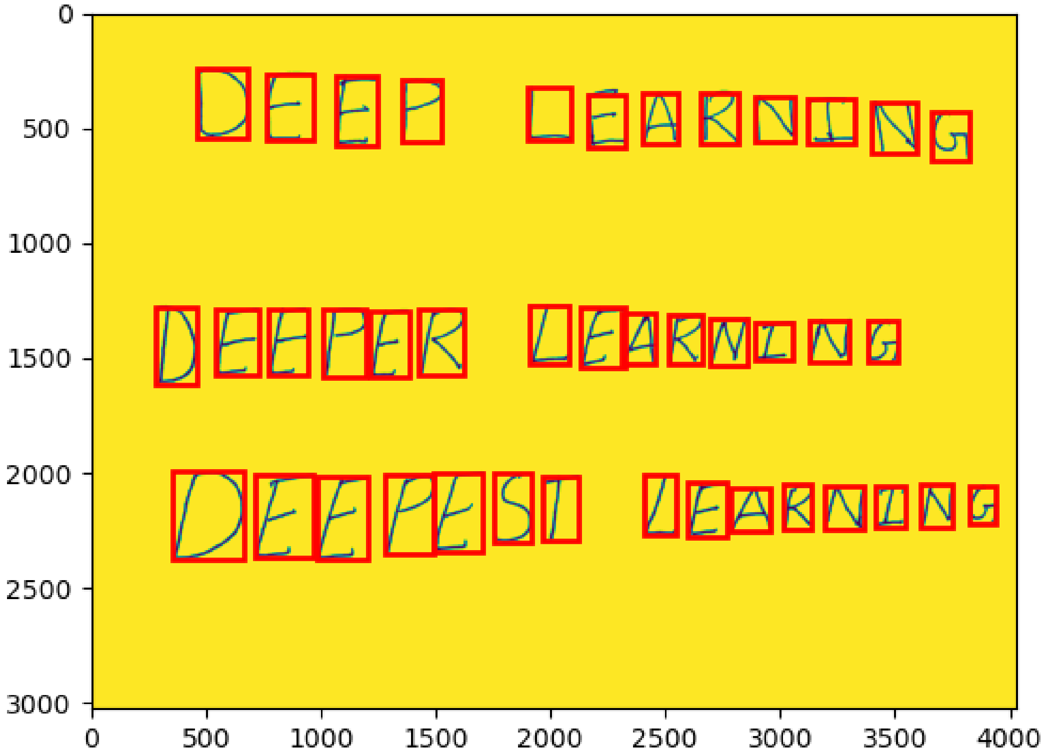
```
# find the rows using..RANSAC, counting, clustering, etc.
heights = [bbox[2]-bbox[0] for bbox in bboxes]
mean_height = sum(heights)/len(heights)

centers = [((bbox[2]+bbox[0])//2, (bbox[3]+bbox[1])//2, bbox[2]-bbox[0], bbox[3]-bbox[1]) for bbox in bboxes]
centers = sorted(centers, key=lambda p: p[0])
rows = []
pre_h = centers[0][0]

row = []
for c in centers:
    if c[0] > pre_h + mean_height:
        row = sorted(row, key = lambda p:p[1])
        rows.append(row)
        row = [c]
        pre_h = c[0]
    else:
        row.append(c)
row = sorted(row, key = lambda p:p[1])
rows.append(row)


# crop the bounding boxes
# note.. before you flatten, transpose the image (that's how the dataset is!)
# consider doing a square crop, and even using np.pad() to get your images looking more like the dataset
kernel = np.array([[0, 1, 0], [1, 1, 1], [0, 1, 0]])
data = []
for row in rows:
    row_data = []
    for y, x, h, w in row:
        # crop out the character
        crop = bw[y-h//2:y+h//2, x-w//2:x+w//2]
        # pad it to square
        h_pad, w_pad = 0, 0
        if h > w:
            h_pad = h//20
            w_pad = (h-w)//2+h_pad
        elif h < w:
            w_pad = w//20
            h_pad = (w-h)//2+w_pad
        crop = np.pad(crop, ((h_pad, h_pad), (w_pad, w_pad)), 'constant', constant_values=(1, 1))
        # resize to 32*32
        crop = skimage.transform.resize(crop, (32, 32))
        crop = skimage.morphology.erosion(crop, kernel)
        crop = np.transpose(crop)
        row_data.append(crop.flatten())
    data.append(np.array(row_data))
```

Problem 4.3


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


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





From the above images it can be seen that the algorithm was able to detect all the lettersin the given images with 100% accuracy.

Problem 4.4

The extracted text and its accuracy from the detection is as below:

```
TQDQLIST
IMAKEATDQDLIST
2LHFCKDFF7HEFIRFWT
THINGQNTQDQLIST
3RIALIZEY0UHAUEALR6ADT
CQMPLFT5DITHINGS
9RFWARDYDU8GELFWITH
ANAP
```

```
ABCDEFG
HIJKLMN
QPQKSTW
VWXYZ
1Z3GS6789J
```

```
HAIKUSARHHAGY
BLTSDMETIMESTHEYDDWTMAKGBHNGE
RBGRIGERAMQR
```

```
JEEPLKAKMING
DEPPEKLEARNING
DEBPE5TLEARNING
```

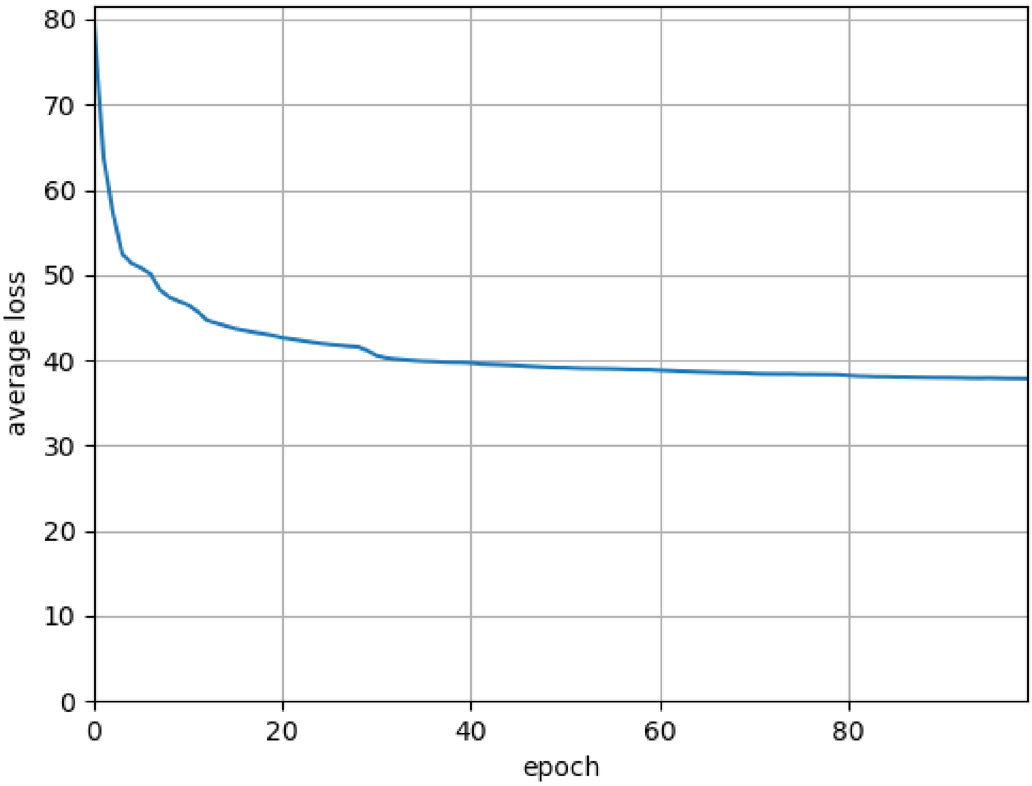
We can see that, overall, the results were pretty good and the accuracy is pretty decent as well.

16720 – hw5
mrohitth

Problem 5.1.1 and 5.1.2

```
# Q5.1 & Q5.2
# initialize layers here
initialize_weights(train_x.shape[1],hidden_size,params,'layer1')
initialize_weights(hidden_size,hidden_size,params,'layer2')
initialize_weights(hidden_size,hidden_size,params,'layer3')
initialize_weights(hidden_size,train_x.shape[1],params,'output')

keys = [key for key in params.keys()]
for k in keys:
    params['m_'+k] = np.zeros(params[k].shape)
train_loss=[]


# should look like your previous training loops
losses = []
for itr in range(max_iters):
    total_loss = 0
    for xb,_ in batches:
        # training loop can be exactly the same as q2!
        # your loss is now squared error
        # 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
        #   so you should be able to write your loop without any special conditions

        h1 = forward(xb, params, 'layer1', relu)
        h2 = forward(h1, params, 'layer2', relu)
        h3 = forward(h2, params, 'layer3', relu)
        probs = forward(h3, params, 'output', sigmoid)

        # Loss and accuracy
        loss = np.sum((xb - probs)**2)
        total_loss += loss

        # Backward pass
        delta = 2*(probs-xb)
        delta = backwards(delta, params, 'output', sigmoid_deriv)
        delta = backwards(delta, params, 'layer3', relu_deriv)
        delta = backwards(delta, params, 'layer2', relu_deriv)
        backwards(delta, params, 'layer1', relu_deriv)

        # Apply gradient
        for layer in ['output','layer1','layer2','layer3']:
            params['m_W' + layer] = 0.9*params['m_W' + layer] - learning_rate * params['grad_W' + layer]
            params['W' + layer] += params['m_W' + layer]
            params['m_b' + layer] = 0.9*params['m_b' + layer]  - learning_rate * params['grad_b' + layer]
            params['b' + layer]+= params['m_b' + layer]
```

Problem 5.2



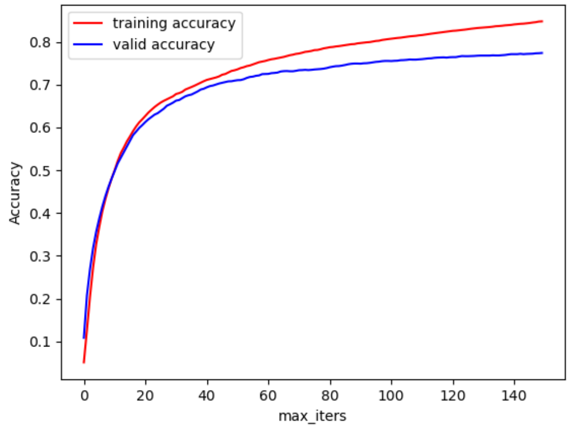
We can see that initially there is a drastic drop in the total loss of the system, but as the number of epochs increase, themomentum of convergence decreases and the total loss starts decreases slowly. Also, since we are changing the learning rateas iterations increase, we also slow down the convergence of the system, if the learning rate becomes relatively small.

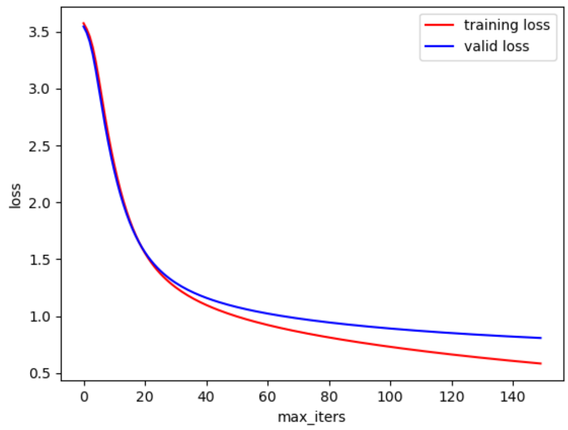
Problem 5.3.1

Problem 5.3.2

The average PSNR over the validation data set is **14.581606637265628**

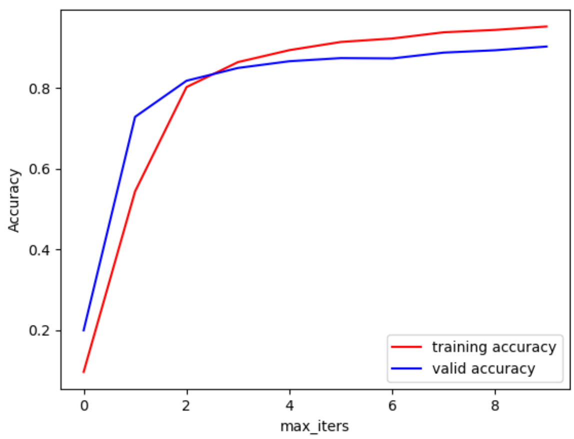
Problem 6.1.1

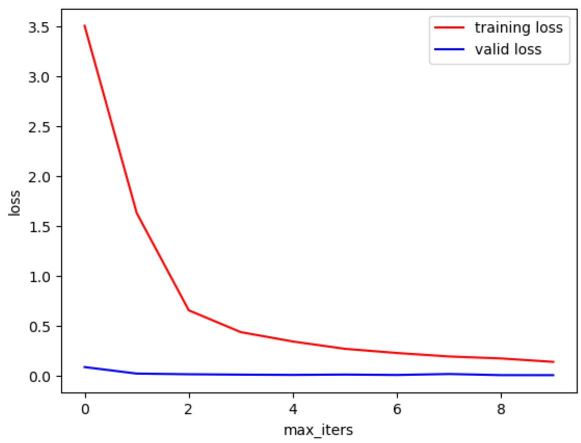




Validation accuracy: 0.7741666666666667

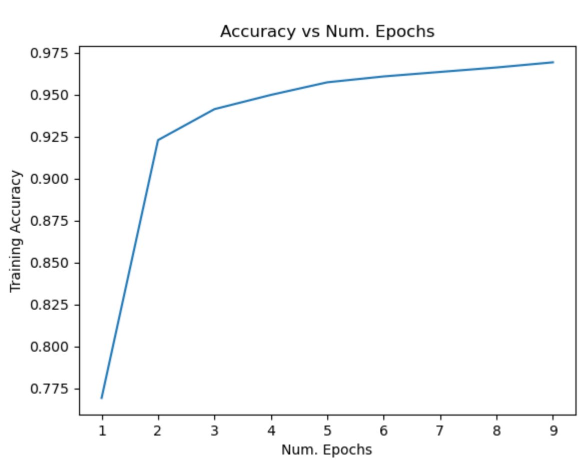
Problem 6.1.2

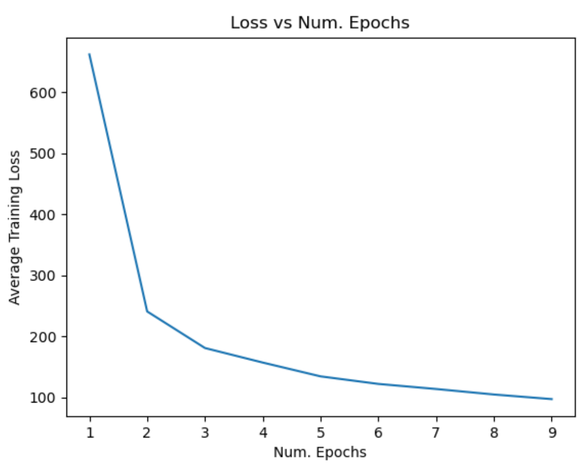




Validation accuracy: 0.9027777777777778

Problem 6.1.3





Problem 6.2

```
Starting epoch 22 / 24
Train accuracy:  0.8661764705882353
Val accuracy:  0.788235294117647
Starting epoch 23 / 24
Train accuracy:  0.8544117647058823
Val accuracy:  0.7617647058823529
Starting epoch 24 / 24
Train accuracy:  0.8838235294117647
Val accuracy:  0.8117647058823529
```

```
Train Epoch: 21, Loss: 2.314396
Train Epoch: 21, Loss: 2.284284
Train Epoch: 21, Loss: 2.378585
Test set: Average loss: 2.1330, Accuracy: 25.88%
Train Epoch: 22, Loss: 2.102735
Train Epoch: 22, Loss: 2.061395
Train Epoch: 22, Loss: 2.200077
Test set: Average loss: 2.1505, Accuracy: 25.88%
Train Epoch: 23, Loss: 2.257583
Train Epoch: 23, Loss: 2.289290
Train Epoch: 23, Loss: 1.847931
Test set: Average loss: 2.2278, Accuracy: 20.59%
```

The fine-tuned network and the scratch network are implemented in q7-finetune.py. The maximum validation accuracy for the finetuned network is **81.176%** and the maximum validation accuracy for the scratch trained network is **25.88%**. Wecan see that the finetuned network outperforms the scratch trained network.

Since the model is using pre-trained weights (from ImageNet dataset = very large number of images for it to learn from), it has better understanding (more semantic information of how images in scenes and flowers in such scenes can look like and can build better representations internally for the image - making it easier for it to identify and classify the image.