16-820 Advanced Computer Vision Assignment 4

Li-Wei Yang
liweiy@andrew.cmu.edu
Section A

Q1.1

$softmax\left(x_i\right) = \frac{e^{x_i}}{\sum e^{x_j}}$, if for all x_i we move by a constant c,
$softmax\left(x_i + c\right) = \frac{e^{x_i+c}}{\sum e^{x_j+c}} = \frac{e^{x_i}e^{c}}{\sum e^{x_j}e^{c}} = \frac{e^{c}e^{x_i}}{e^{c}\sum e^{x_j}} = \frac{e^{x_i}}{\sum e^{x_j}} = softmax\left(x_i\right)$,
so softmax is invariant to translation.
If we choose $c = 0$, we may be taking e^{x_i} on a relative large exponent, this may cause numerical instability. If we choose $c = -\max x_i$, we will shift all x_i to a more numerically stable range thus increase numerical stability.

Q1.2

The range of each element should be [0, 1], and the sum over all elements should be 1, since the sum of all probability should = 1.
One could say that "softmax takes an arbitrary real valued vector x and turns it into a probability distribution over same number of values"
Step 1: exponentially transforms the elements to amplify the difference between elements in the input vector.
Step 2: calculate the sum of amplified elements.
Step 3: calculate the probability of each element in input vector.

Q1.3

Without activation function, a layer of neural network would be $y_i = W_i x_i + b_i$.
If we have n layers,
$y_n = W_n x_n + b_n = W_n(W_{n-1}x_{n-1} + b_{n-1}) + b_n = W_n W_{n-1} x_{n-1} + W_n b_{n-1} + b_n = Ax_1 + b$, where A is the product of W_i and $b = b_n + W_n b_{n-1} + W_n W_{n-1} b_{n-2} + ...$, and x_1 is the input on first layer.
$y_n = Ax_1 + b$ is the same as a general linear regression problem, $y = ax + b$.

Q1.4

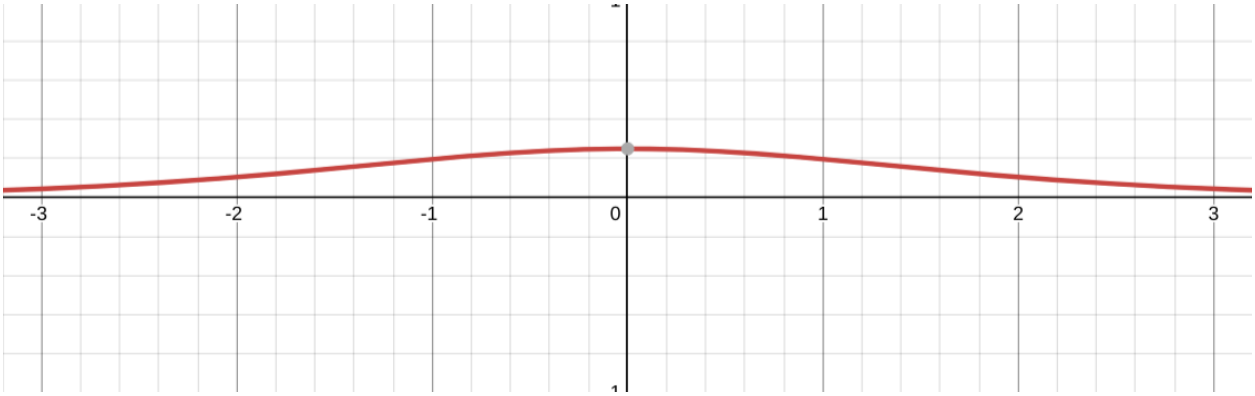
$\sigma(x) = \frac{1}{1+e^{-x}}$, differentiate by x, $\sigma'(x) = \frac{d}{dx}\left(1+e^{-x}\right)^{-1} = -\left(1+e^{-x}\right)^{-2} * e^{-x} * (-1) = \frac{e^{-x}}{(1+e^{-x})^2}$
$\sigma'(x) = \frac{e^{-x}}{(1+e^{-x})^2} = \sigma(x) * \frac{e^{-x}}{(1+e^{-x})} = \sigma(x)\left(1-\sigma(x)\right)$

Q1.5

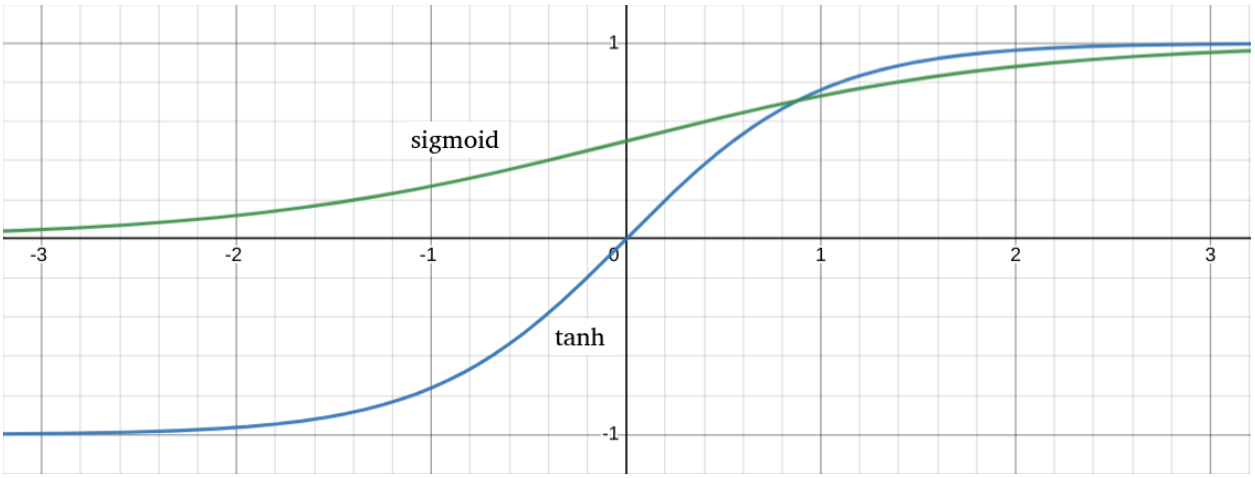
$y = Wx + b$
$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial W} = \delta x^T$, $\delta x^T \in \mathbb{R}^{k \times d}$
$\frac{\partial J}{\partial x} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial x} = W^T\delta$, $W^T\delta \in \mathbb{R}^{d \times 1}$
$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial b} = \delta$, $\delta \in \mathbb{R}^{k \times 1}$

Q1.6

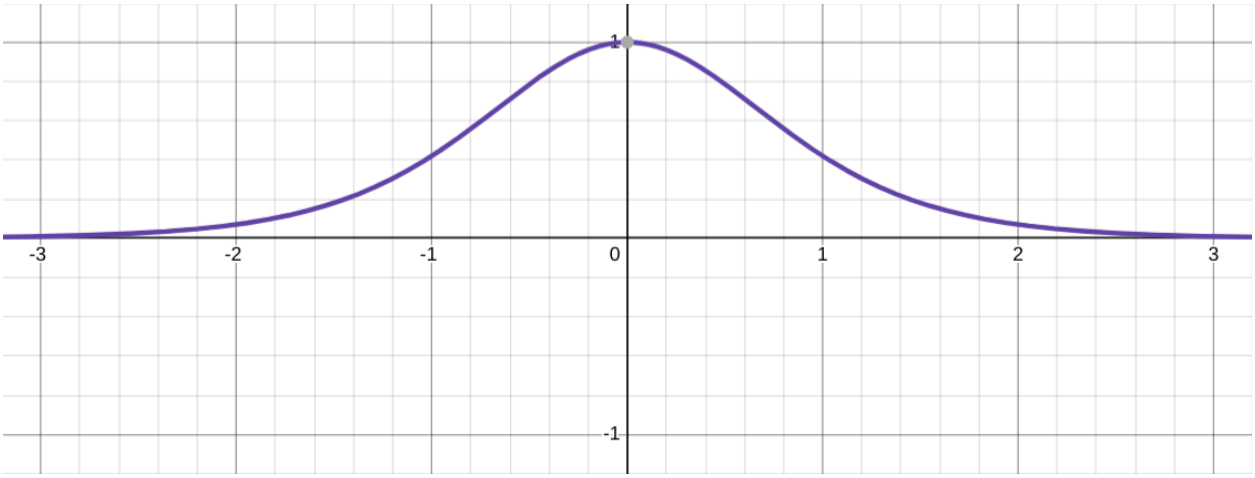
$\sigma'(x) = \frac{e^{-x}}{(1+e^{-x})^2}$
Plot $\sigma'(x)$:



The value of sigmoid gradient ranges from 0 to 0.25, if we stack multiple layers of sigmoid, the small multiple may cause the gradient to vanish.
Plot sigmoid and tanh:



The output range of sigmoid: (0, 1)
The output range of tanh: (-1, 1)
We prefer tanh because tanh can be negative, so if x is negative, the negative effect can be pass on.



tanh has maximum value of 1, thus the gradient would vanish slower.
$\sigma(x) = \frac{1}{1+e^{-x}}$
$\frac{1-e^{-x}}{1+e^{-x}} = 2\sigma(x) - 1$
$\tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}} = 2\sigma(2x) - 1$

Q2.1.1

A zero-initialized network would multiply input with zeros, thus the output would also be zeros. Because all the output are the same, this would be bad for gradient descent.

Q2.1.2

```
def initialize_weights(in_size, out_size, params, name=""):
    # Standard deviation
    init_range = np.sqrt(6/(in_size+out_size))
    W = np.random.uniform(-init_range, init_range, (in_size, out_size))
    b = np.zeros(out_size)

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

Q2.1.3

We initialize layers with random number can help us find different result and often leads us to better result.
We scale the initialization depend on layer size because this can keep the activation value in desired range. If we do not scale the initialization, higher layer would have higher activations as shown in Fig. 6 of [1].

Q2.2.1

```
############################## 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)
    """

    # get the layer parameters
    W = params["W" + name]
    b = params["b" + name]

    pre_act = 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
```

Q2.2.2

```
############################## Q 2.2.2  ##############################
# x is [examples,classes]
# softmax should be done for each row
def softmax(x):
    res = np.zeros(x.shape)
    for i, x_i in enumerate(x):
        c = -np.max(x_i)
        x_shifted = x_i + c
        res[i] = np.exp(x_shifted)/np.sum(np.exp(x_shifted))
    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 = 0, 0
    examples, classes = y.shape

    loss = -np.sum(y*np.log(probs))
    for label, prob in zip(y, probs):
        if np.argmax(label) == np.argmax(prob):
            acc += 1
    acc = acc/examples
    return loss, acc
```

Q2.3

```
############################## 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

    # Examples, classes
    n, k = delta.shape
    derivative = activation_deriv(post_act)
    # Examples, k
    # print(derivative.shape)

    # [Examples x D]
    # print(X.shape)

    # D*n*n*k = D*k
    grad_W = np.dot(X.T, derivative*delta)

    # n*k*k*D = n*D
```

```
grad_X = np.dot(derivative*delta, W.T)

# 1*n*n*k = (k,)
grad_b = np.dot(np.ones((1, n)), derivative*delta).flatten()

assert grad_W.shape == W.shape
assert grad_X.shape == X.shape
assert grad_b.shape == b.shape

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

Q2.4

```
############################## 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 = []
    n, d = x.shape
    # print(n)
    # print(batch_size)
    indices = np.random.randint(0, n, size=(int(n/batch_size), batch_size))
    # print(indices)
    for idx in indices:
        x_batch = x[idx, :]
        y_batch = y[idx, :]
        batches.append((x_batch, y_batch))
```

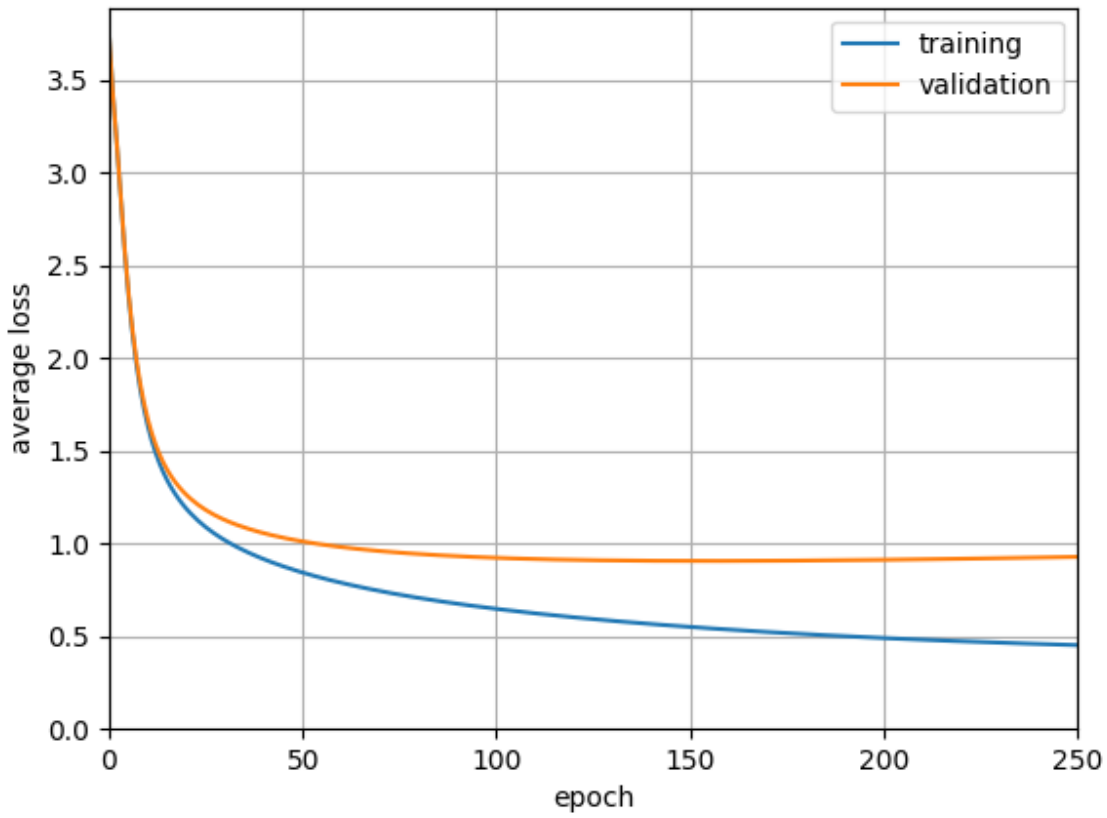
Q2.5

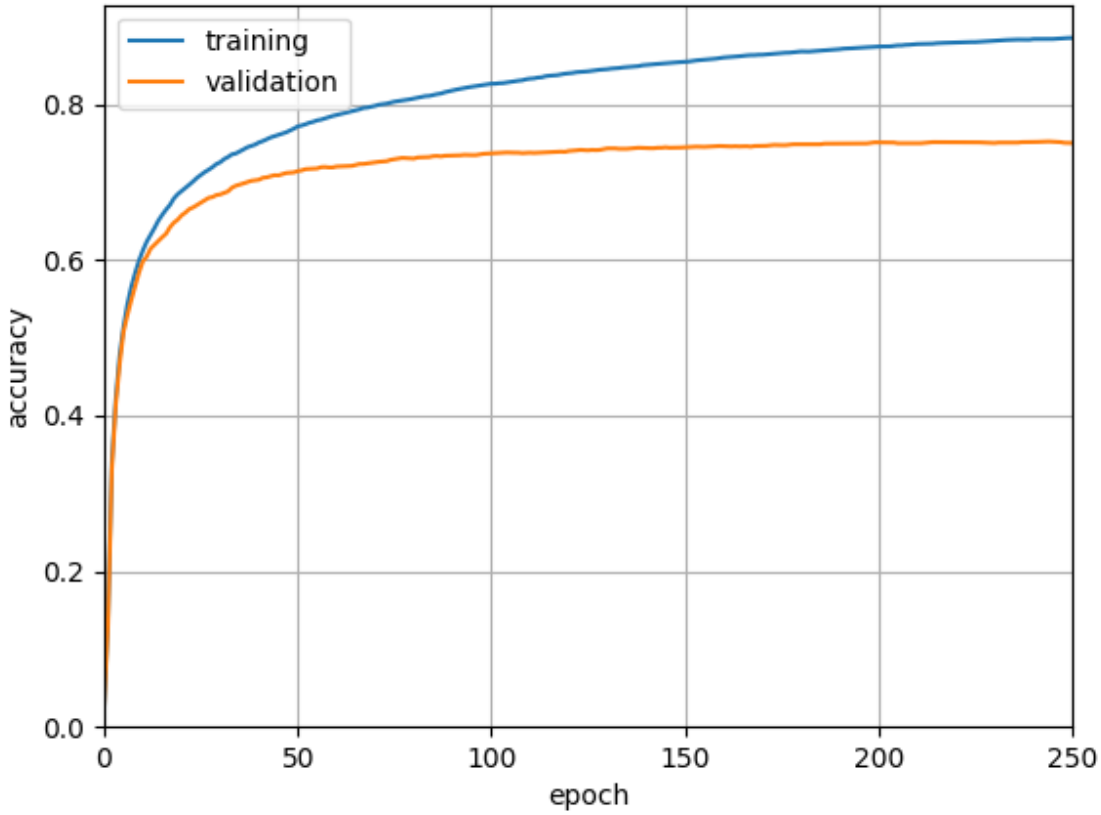
```
# compute gradients using finite difference
eps = 1e-6
for k, v in params.items():
    if "_" in k:
        # skip gradient params
        continue
    # for each value inside the parameter
    #   add epsilon
    #   run the network
    #   get the loss
    #   subtract 2*epsilon
    #   run the network
    #   get the loss
    #   restore the original parameter value
    #   compute derivative with central diffs
    if k[0] == 'W':
        dimensions, classes = v.shape
        for d in range(dimensions):
            for c in range(classes):
                v[d, c] += eps
                #   run the network
                h1 = forward(x, params, "layer1")
                probs = forward(h1, params, "output", softmax)
                loss_add, acc = compute_loss_and_acc(y, probs)
                v[d, c] -= 2*eps
                h1 = forward(x, params, "layer1")
                probs = forward(h1, params, "output", softmax)
                loss_sub, acc = compute_loss_and_acc(y, probs)
                # restore
                v[d, c] += eps
                params['grad_' + k][d, c] = (loss_add-loss_sub)/(2*eps)
    if k[0] == 'b':
        dimensions = v.shape[0]
        for d in range(dimensions):
            v[d] += eps
            #   run the network
            h1 = forward(x, params, "layer1")
            probs = forward(h1, params, "output", softmax)
            loss_add, acc = compute_loss_and_acc(y, probs)
            v[d] -= 2*eps
            h1 = forward(x, params, "layer1")
```

```
probs = forward(h1, params, "output", softmax)
loss_sub, acc = compute_loss_and_acc(y, probs)
# restore
v[d] += eps
params['grad_' + k][d] = (loss_add-loss_sub)/(2*eps)
```

Q3.1

max_iters = 250, batch_size = 128, learning_rate = 1e-3, hidden_size = 64
Validation accuracy: 0.7505555555555555, Test accuracy: 0.7627777777777778

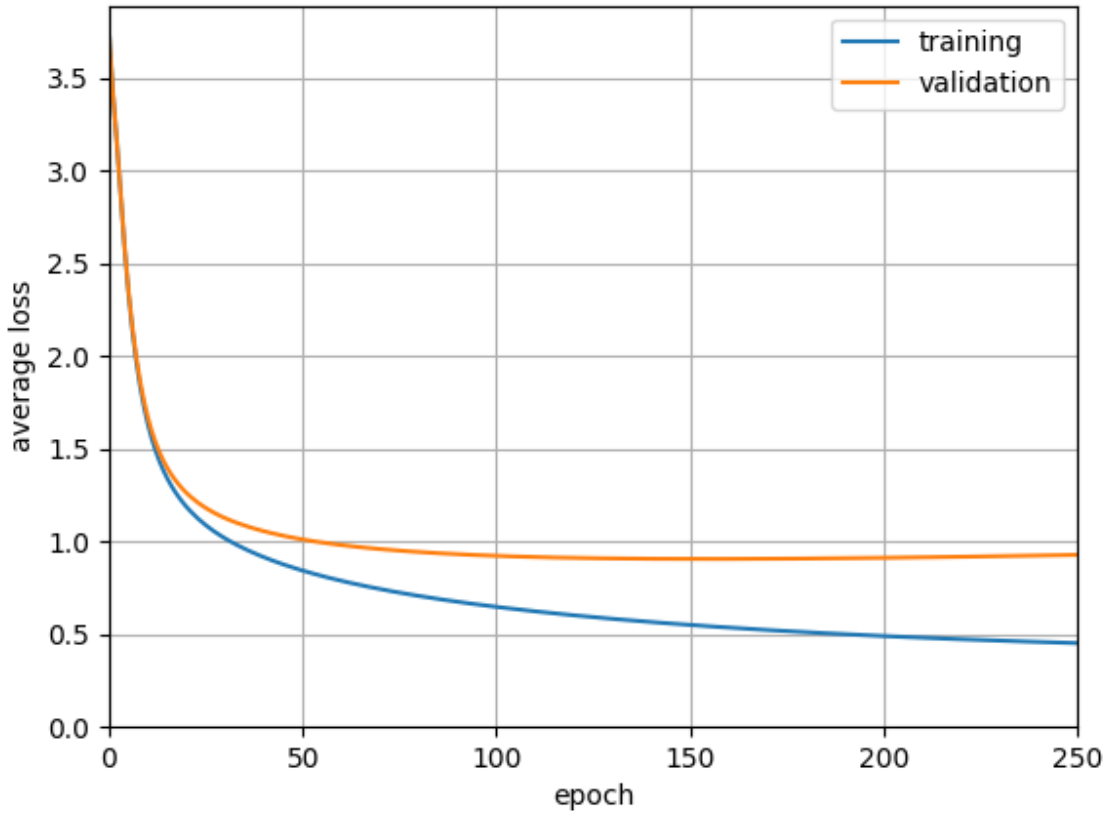


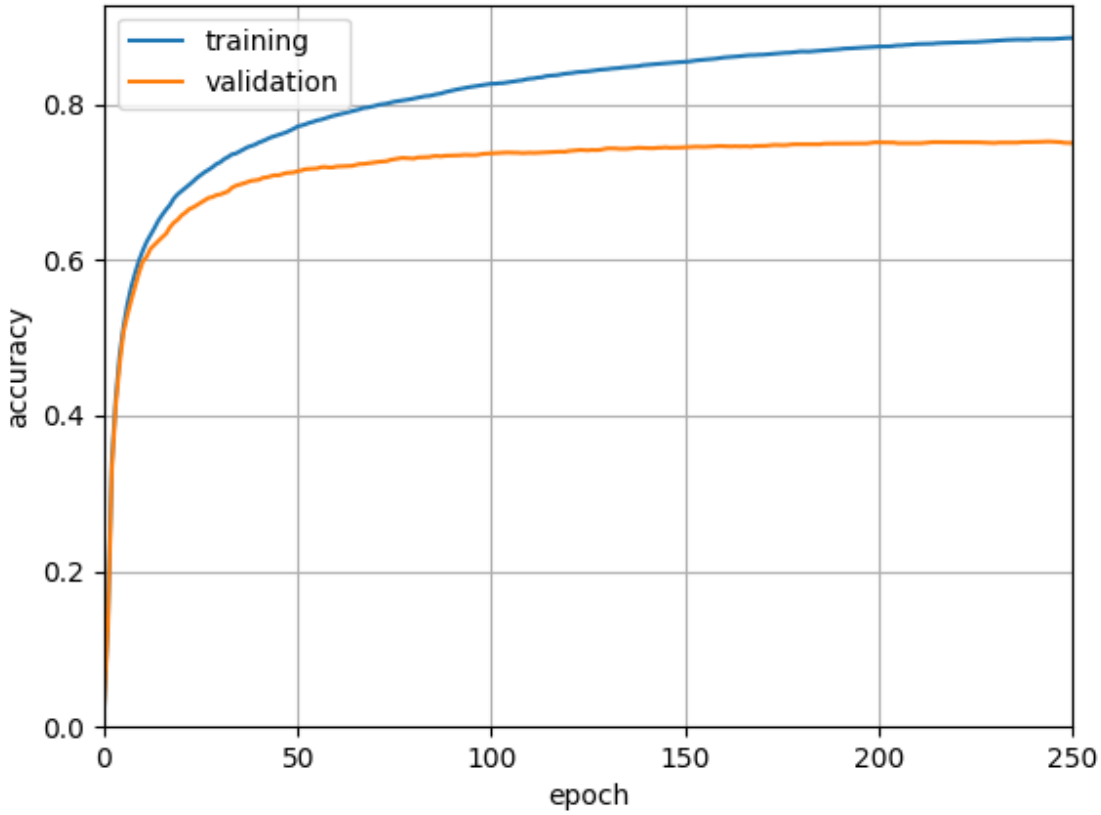

training
validation
accuracy
epoch
0.0
0.2
0.4
0.6
0.8
0
50
100
150
200
250

Q3.2

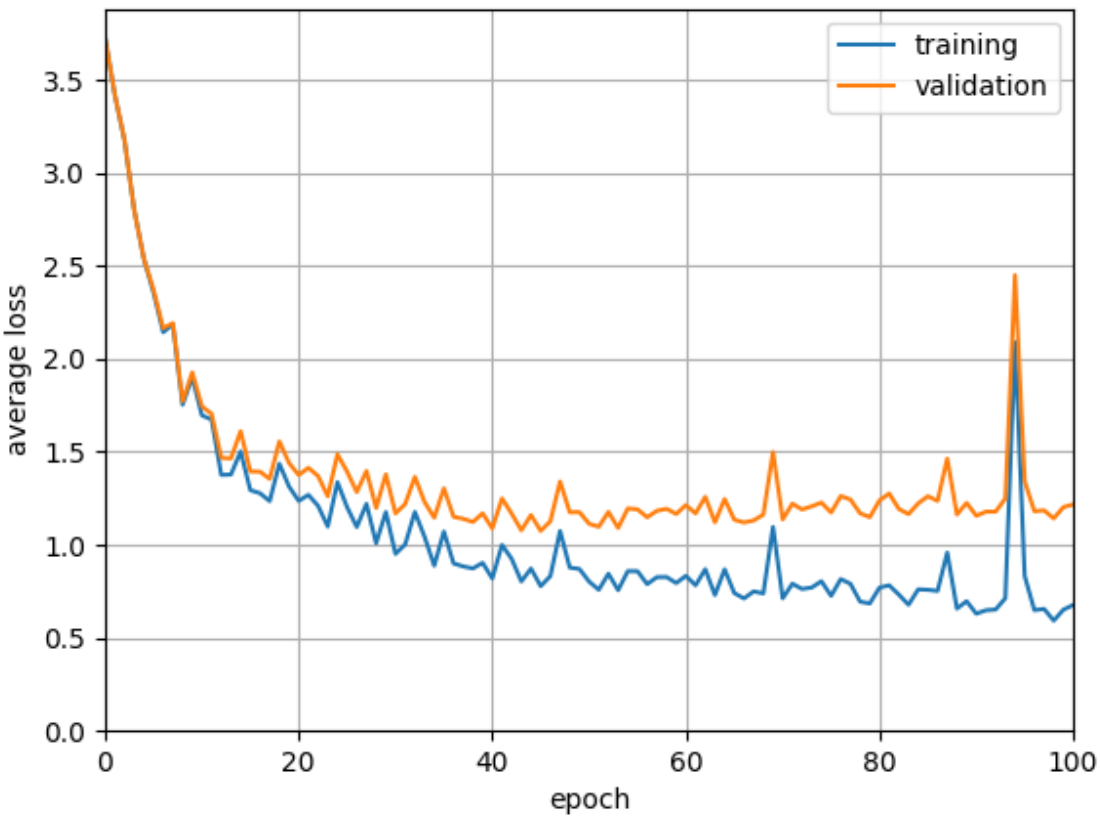
Train for 100 epoch, best test accuracy = 0.7522, from network with learning_rate = 1e-3.
With a larger learning rate, everytime the network would update more toward the gradient, and thus causing a more noisy learning curve. On the contrary, with smaller learning rate, the network would update less toward the gradient, causing a smoother learning curve, sometimes not update enough to reach good result.

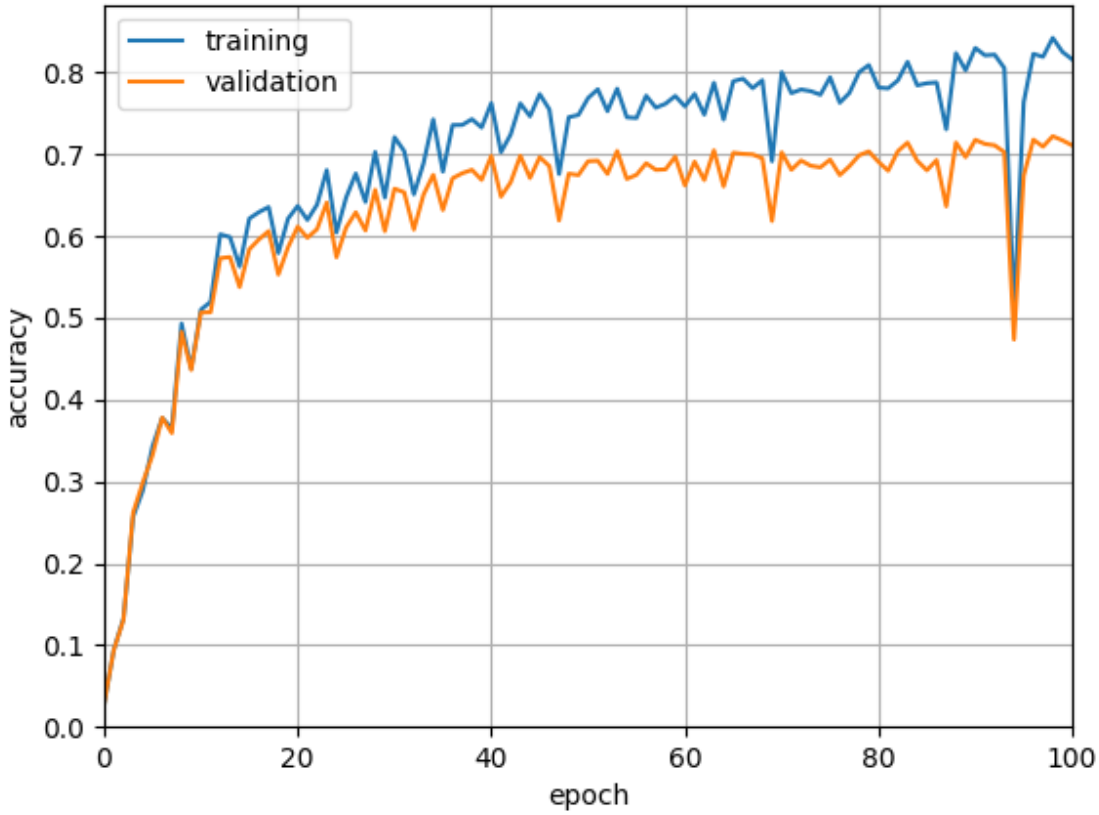
learning_rate = 1e-3



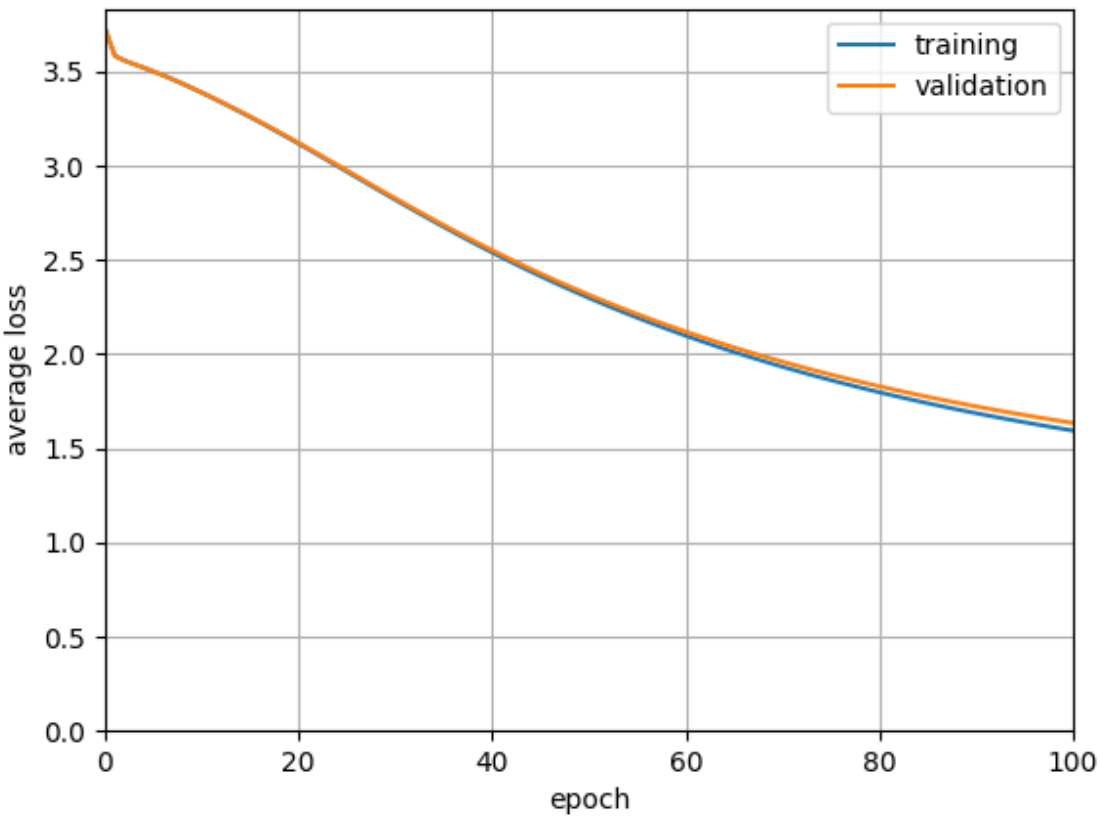


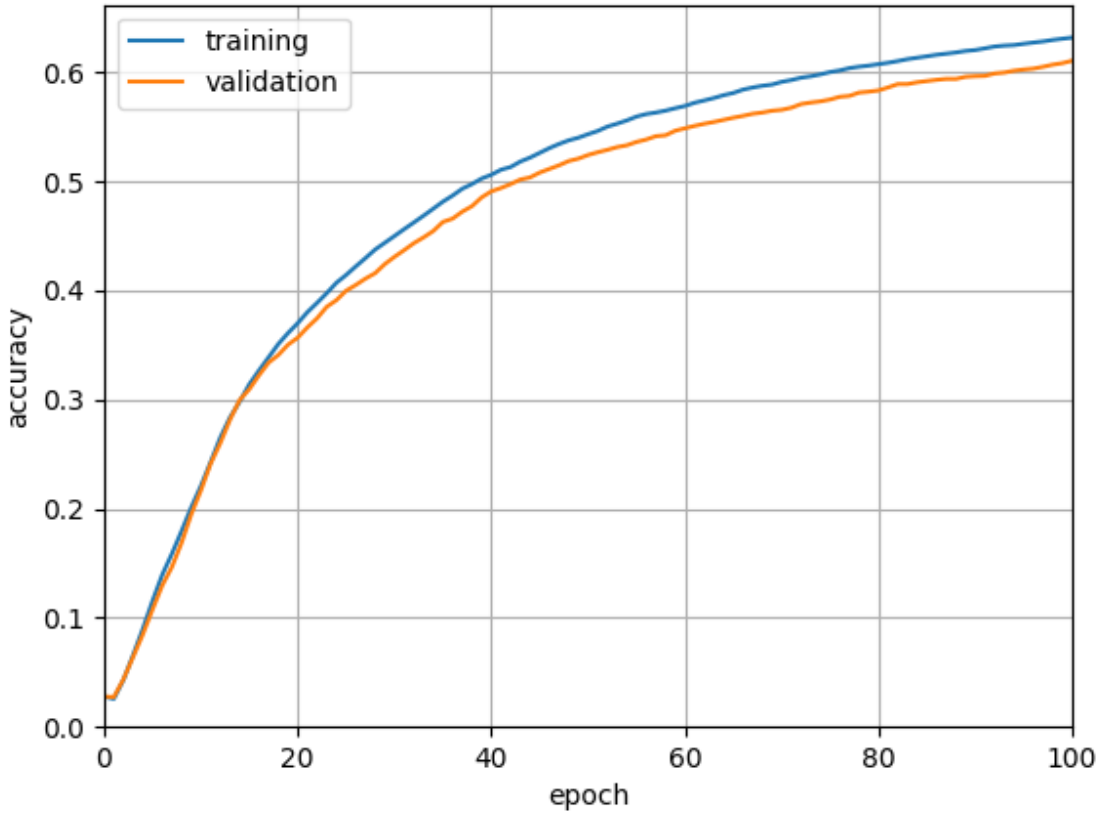
10 times, learning_rate = 1e-2


training
validation
average loss
epoch
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
20
40
60
80
100

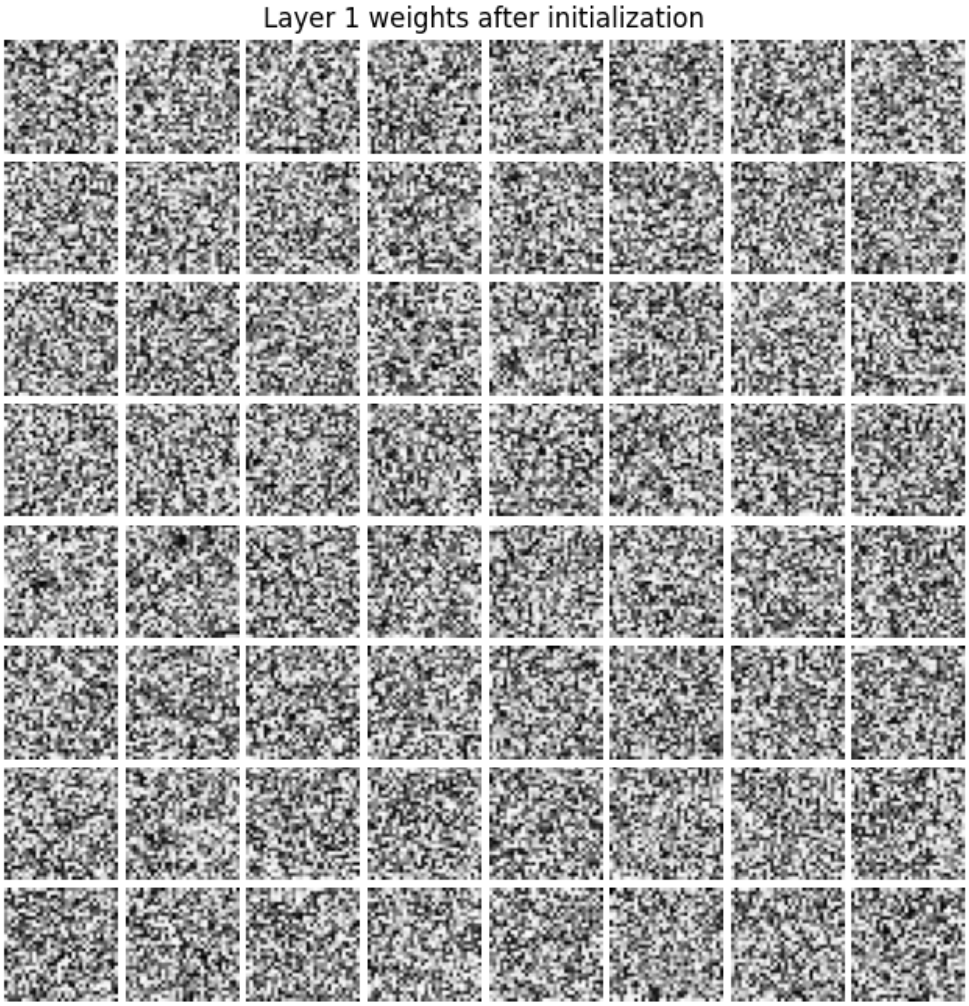


0.1 times, learning_rate = 1e-4


training
validation
average loss
epoch
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
20
40
60
80
100

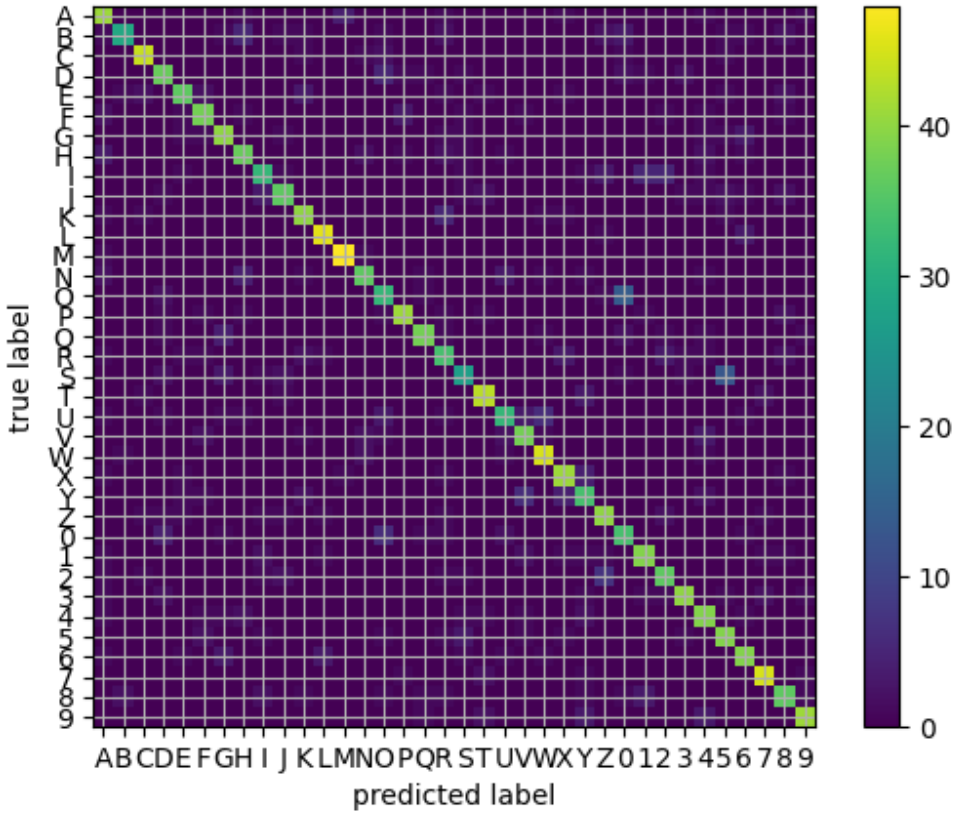

training
validation
accuracy
epoch
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0
20
40
60
80
100

Q3.3



Just after initialization, the weights are just uniform distribution noise. As we train the weights, we could see the weights start to represent some pattern.

Q3.4

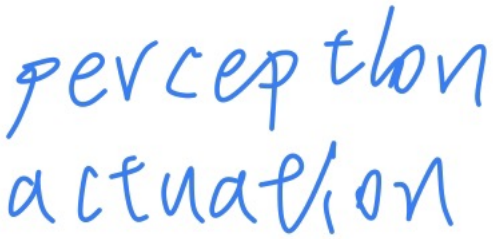


From the confusion matrix, we can see the network is having a hard time distinguish between '0' and 'O', as well as '5' and 'S'.

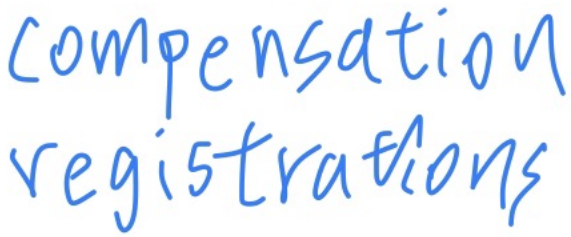
Q4.1

The assumption comes from the data we train the network, we train the network letter by letter, but in real writing sometimes we connect letters together. Another assumption is that the writing style is close to the training data.
Consider the note I took during system engineering class:

connected words



special writing style



Sometimes I connect t and i, this may be hard to separate and be consider in same bounding box, also I have unique writing style on 's', see final 's' in word 'registrations', this will be potential misclassify letter.

Q4.2

findLetters in q4.py
I do not use gaussian blur and denoise, they make my performance drop. I add some padding on the bounding box in case the algorithm crops the letter. For 4th pictures, I have to dilate more to get a good result, this is because the word is rather small compare to the space it takes up.

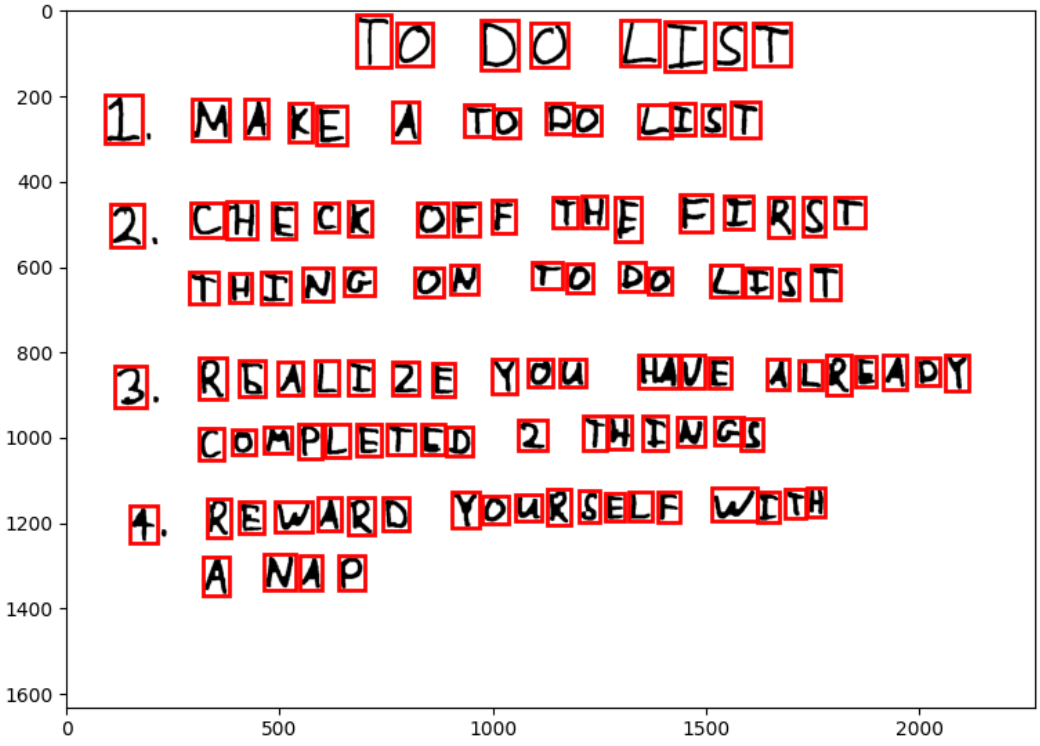
```
# takes a color image
# returns a list of bounding boxes and black_and_white image
def findLetters(image, n_img):
    bboxes = []
    bw = None

    if n_img == 3:
        d = 25
        ext = 20
    else:
        d = 5
        ext = 5

    # 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

    # image = gaussian(image, sigma=1.5)
    # image = skimage.restoration.denoise_wavelet(image, sigma=2)
    image = rgb2gray(image)
    thresh = threshold_otsu(image)
    bw = closing(image < thresh, square(12))
    # fig, ax = plt.subplots(figsize=(10, 6))
    # ax.imshow(bw)
    # plt.show()
    bw = dilation(bw, footprint=square(d))

    cleared = clear_border(bw)
    labels = label(cleared)

    for props in regionprops(labels):
        if props.area >= 500:
            minr, minc, maxr, maxc = props.bbox
            ext_bbox = [minr-ext, minc-ext, maxr+ext, maxc+ext]
            bboxes.append(ext_bbox)
    # turn foreground to black and background to white
```

```
    bw = 1.0 - bw
    return bboxes, bw
```

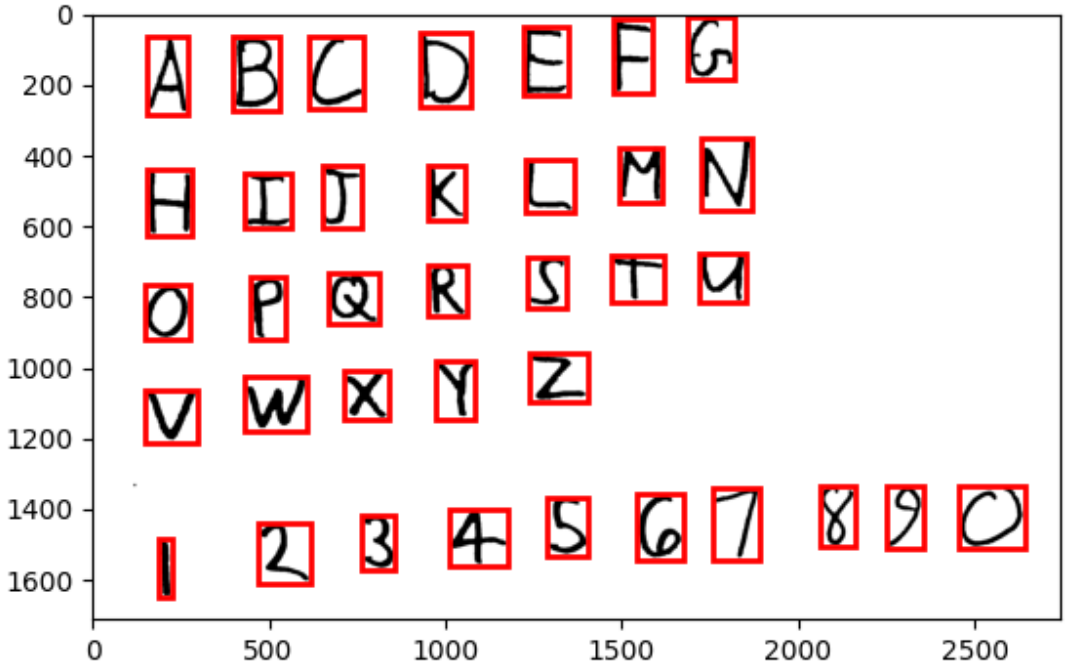
Find rows and arrange letters in sequence in run_q4.py

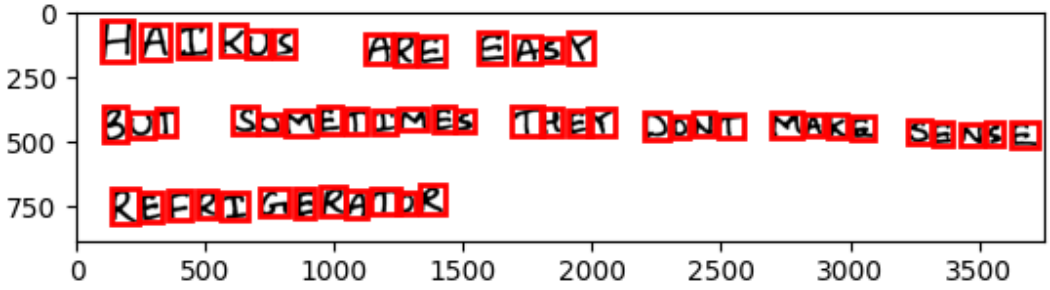
```
# find the rows using..RANSAC, counting, clustering, etc.
row_num = 0
cur_maxr = bboxes[0][2]
bboxes_row_list = []
bboxes_row_list.append([])
bboxes.sort(key=lambda box: box[2])
for bbox in bboxes:
    minr, minc, maxr, maxc = bbox
    # char_image = bw[minr:maxr, minc:maxc]
    # plt.imshow(char_image)
    # plt.show()
    if minr > cur_maxr:
        row_num += 1
        cur_maxr = maxr
        bboxes_row_list.append([])
    bboxes_row_list[row_num].append(bbox)
print("row_num", row_num+1)
for row in bboxes_row_list:
    row.sort(key=lambda box: box[3])
```

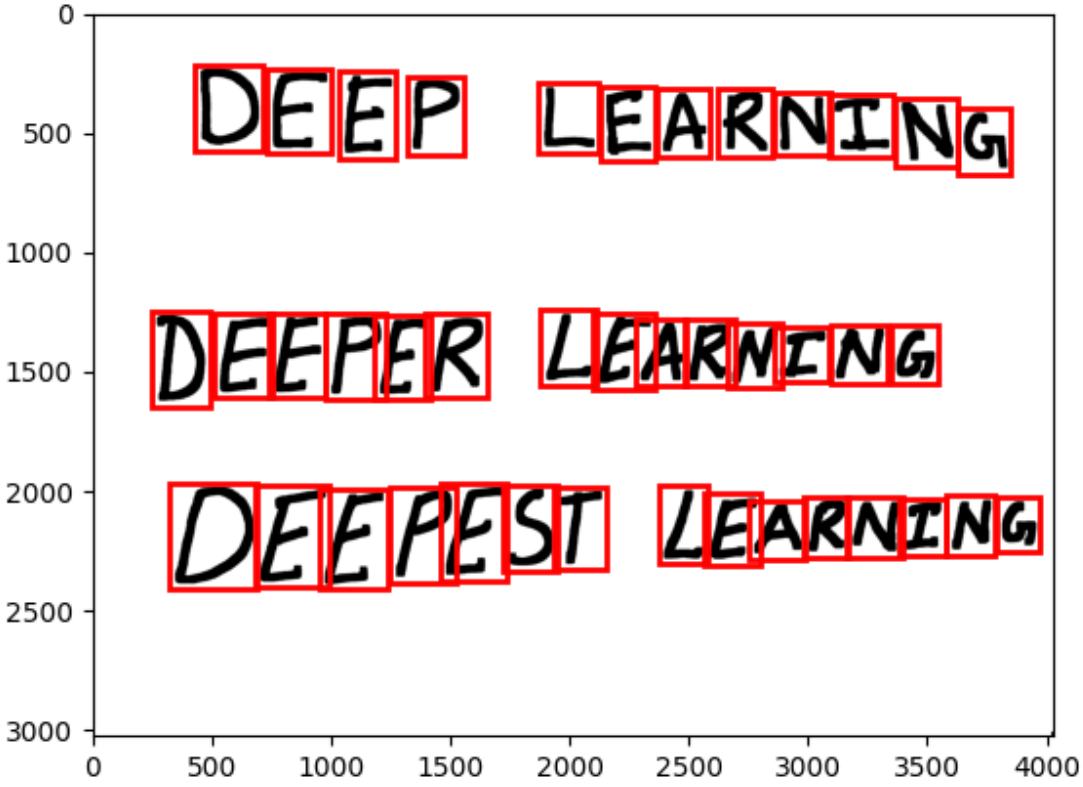
Q4.3



The 'ha' in 'have' in point 3 is considered to be in the same bounding box. This will make the network hard to classify.


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


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


DEEP LEARNING
DEEPER LEARNING
DEEPEST LEARNING
0
500
1000
1500
2000
2500
3000
3500
4000

Q4.4

Accuracy on four pictures: 77.05%, 75.61%, 80.70%, 81.82%.
Overall, all results have accuracy greater than 75%, and the retrieved strings are:
1.
T0 DQ LI5T
I MAKE A T0 D0 LIST
2 CH2CK QFE THE FIRST
THING 0N T0 D0 LIST
3 R2ALIZE Y0U UUE ALR6ADY
C0MPLETLD 2 THINGS
4 REWARD Y0URSELF WITH
A NAP

2.
ABCDHFG
MIJKLMN
QPQRSTU
VWXYZ
8Z3GBG78PD

3.
HAIKGS ARE HASY
BUT SQMETIMES THEX DDNT MAK2 SGNGE
RBFRIGBRAT0R

4.
DEHPLBARMING
DHEPERLBARHING
DEKPESTLEARNIHG

Q5.1.1

Initialization

```
initialize_weights(1024, 32, params, "layer1")
initialize_weights(32, 32, params, "layer2")
initialize_weights(32, 32, params, "layer3")
initialize_weights(32, 1024, params, "output")
```

Forward

```
h1 = forward(xb, params, "layer1", relu)
h2 = forward(h1, params, "layer2", relu)
h3 = forward(h2, params, "layer3", relu)
out = forward(h3, params, "output", sigmoid)
```

Loss

```
loss = np.sum((xb - out)**2)
```

Backward

```
delta1 = 2*(out-xb)
delta2 = backwards(delta1, params, "output", sigmoid_deriv)
delta3 = backwards(delta2, params, 'layer3', relu_deriv)
delta4 = backwards(delta3, params, 'layer2', relu_deriv)
delta5 = backwards(delta4, params, 'layer1', relu_deriv)
```

Q5.1.2

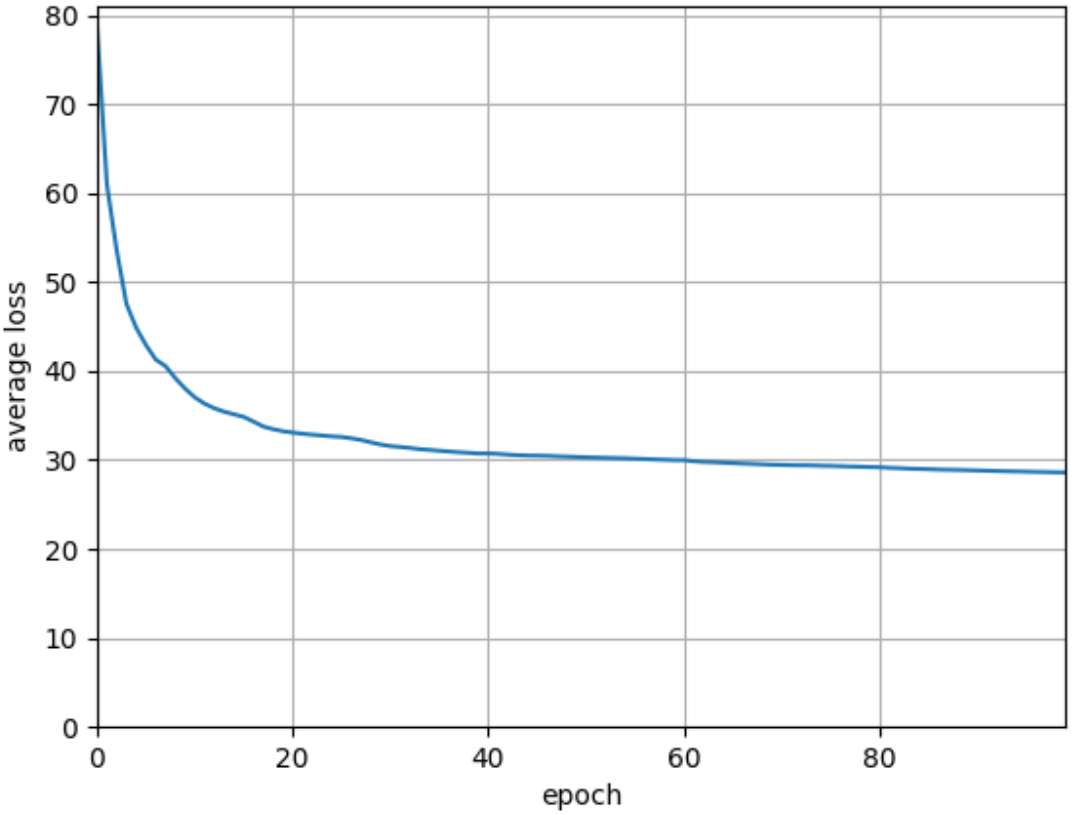
Initialization

```
params["m"+"Wlayer1"] = np.zeros((1024, 32))
params["m"+"blayer1"] = np.zeros(32)
params["m"+"Wlayer2"] = np.zeros((32, 32))
params["m"+"blayer2"] = np.zeros(32)
params["m"+"Wlayer3"] = np.zeros((32, 32))
params["m"+"blayer3"] = np.zeros(32)
params["m"+"Woutput"] = np.zeros((32, 1024))
params["m"+"boutput"] = np.zeros(1024)
```

Update

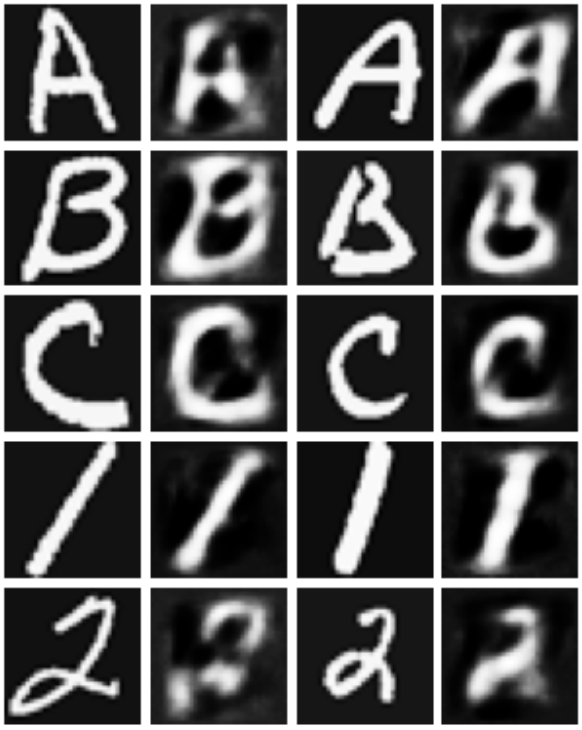
```
# apply gradient, remember to update momentum as well
# v = mu * v - learning_rate * dx # integrate velocity
# x += v # integrate position
mu = 0.9
params["m"+"Wlayer1"] = mu*params["m"+"Wlayer1"] - learning_rate*params["grad_W" +
'layer1']
params["m"+"blayer1"] = mu*params["m"+"blayer1"] - learning_rate*params["grad_b" +
'layer1']
params["m"+"Wlayer2"] = mu*params["m"+"Wlayer2"] - learning_rate*params["grad_W" +
'layer2']
params["m"+"blayer2"] = mu*params["m"+"blayer2"] - learning_rate*params["grad_b" +
'layer2']
params["m"+"Wlayer3"] = mu*params["m"+"Wlayer3"] - learning_rate*params["grad_W" +
'layer3']
params["m"+"blayer3"] = mu*params["m"+"blayer3"] - learning_rate*params["grad_b" +
'layer3']
params["m"+"Woutput"] = mu*params["m"+"Woutput"] - learning_rate*params["grad_W" +
'output']
params["m"+"boutput"] = mu*params["m"+"boutput"] - learning_rate*params["grad_b" +
'output']

params["Wlayer1"] += params["m"+"Wlayer1"]
params["blayer1"] += params["m"+"blayer1"]
params["Wlayer2"] += params["m"+"Wlayer2"]
params["blayer2"] += params["m"+"blayer2"]
params["Wlayer3"] += params["m"+"Wlayer3"]
params["blayer3"] += params["m"+"blayer3"]
params["Woutput"] += params["m"+"Woutput"]
params["boutput"] += params["m"+"boutput"]
```

Q5.2



With momentum, the loss curve is quite smooth, it proves that momentum would help the training converge.

Q5.3.1



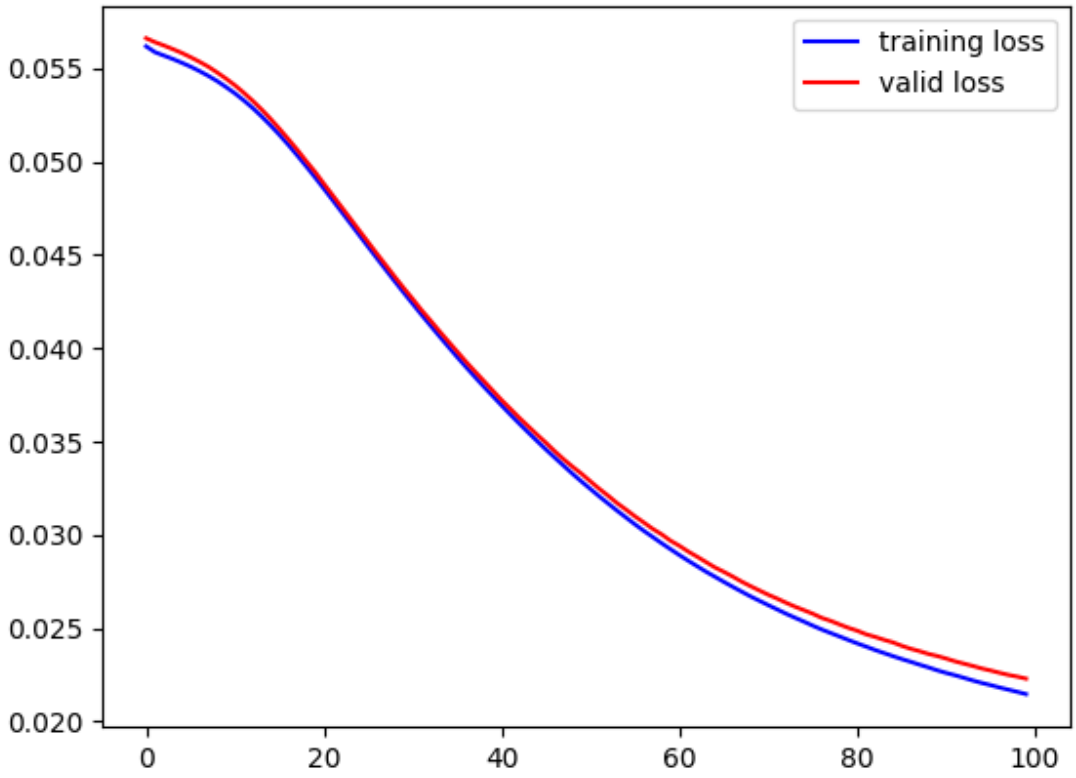
The reconstructed version is the blurred version of the original one. With only limited information, autoencoder can roughly reconstruct the shapes of the letters.

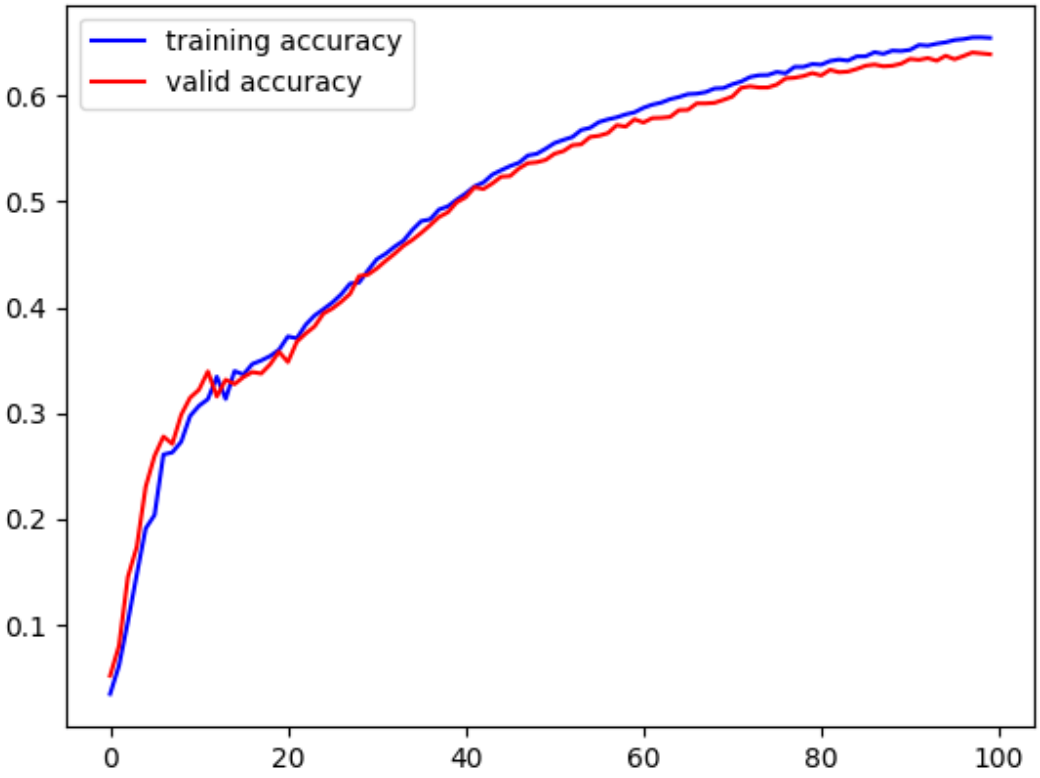
Q5.3.2

With max_iters = 100, batch_size = 36, learning_rate = 3e-5, Average PSNR=15.920232203834017

Q6.1.1

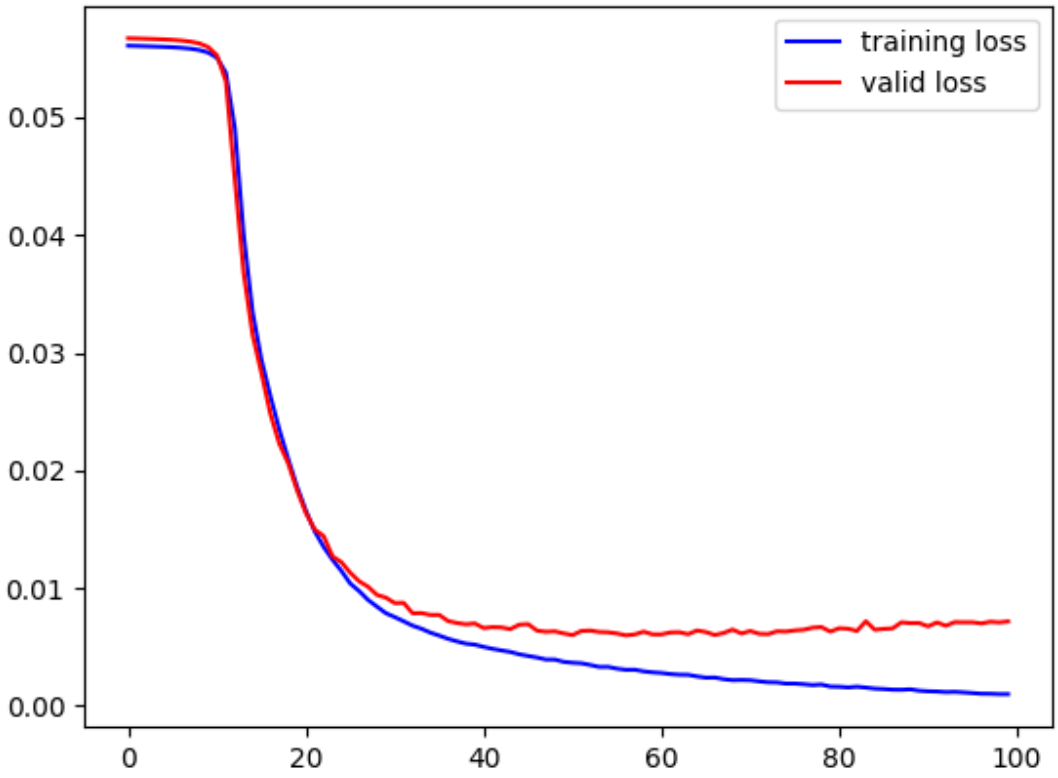
Test acc: 0.64

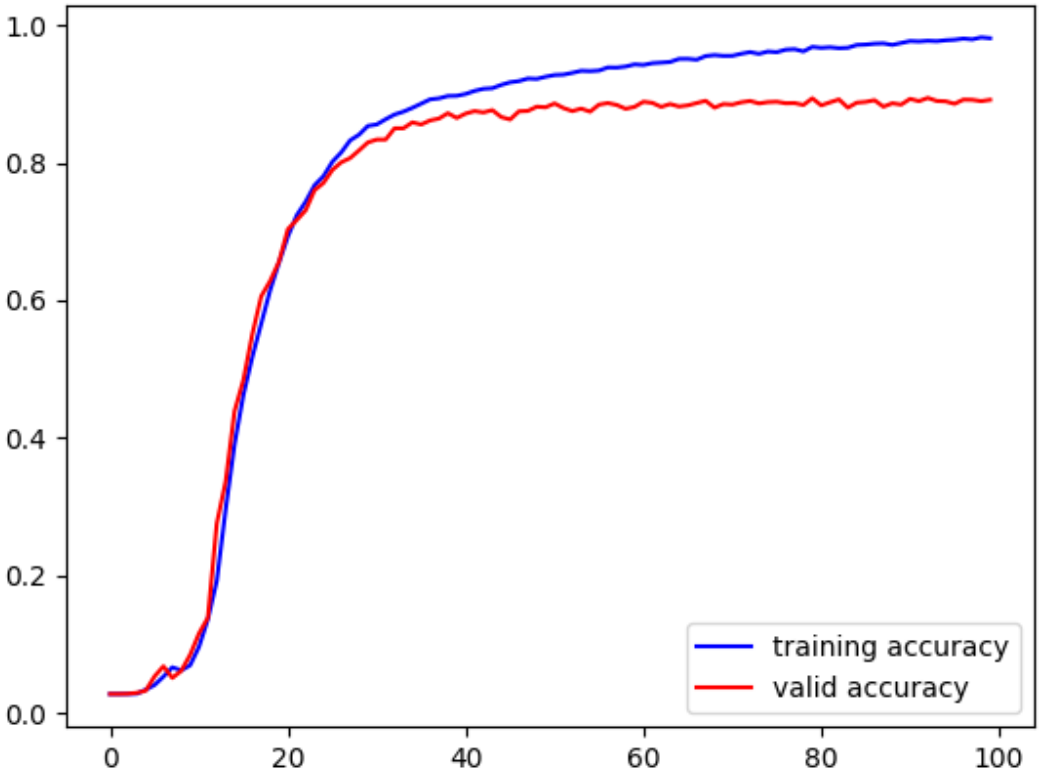



training accuracy
valid accuracy
0.6
0.5
0.4
0.3
0.2
0.1
0
20
40
60
80
100

Q6.1.2

With same hyperparameters, test acc: 0.88

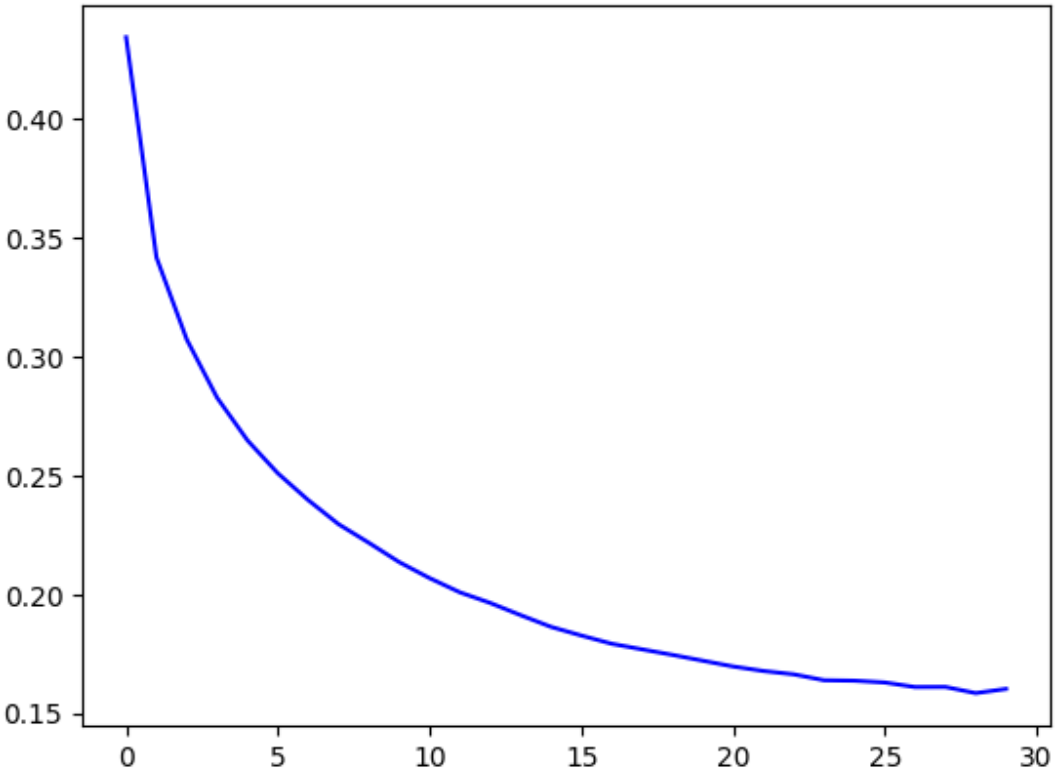




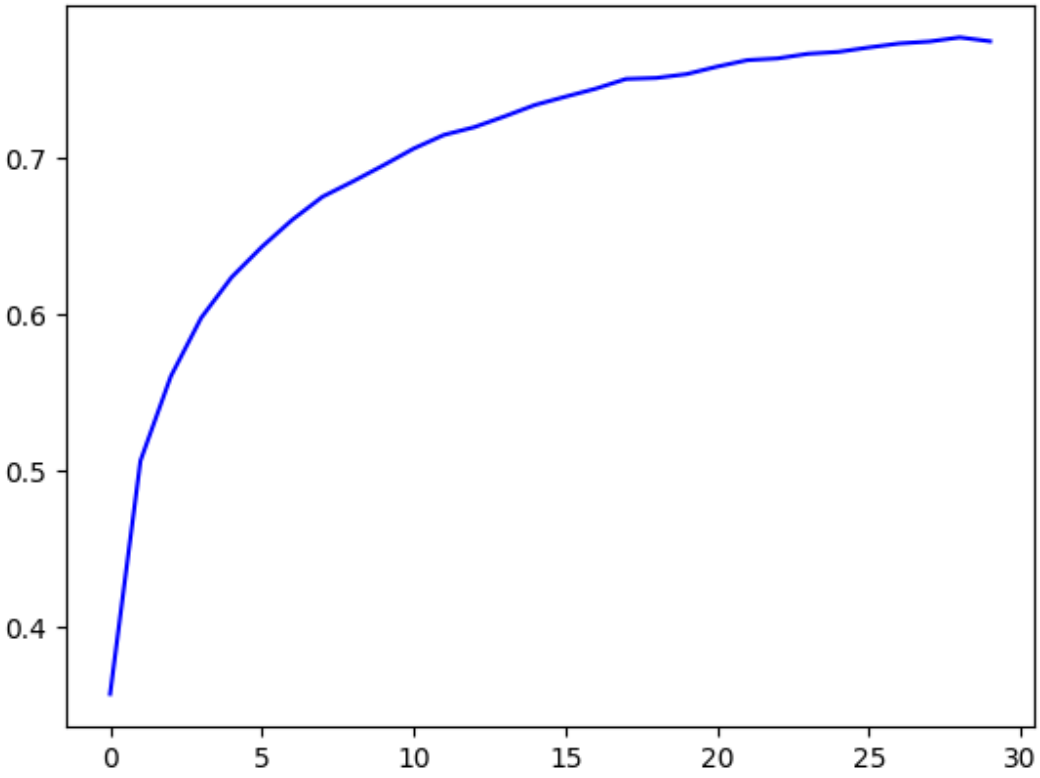
The overall performance is better than previous fully-connected network, and the loss dropping rate is faster after 20 epoch. Because convolutional network shares weights, it would be easy to learn features present in different parts of the image.

Q6.1.3

train loss



train accuracy



Final train accuracy = 77.5%, accuracy of the network on the 10000 test images: 59.11%.

Q6.2

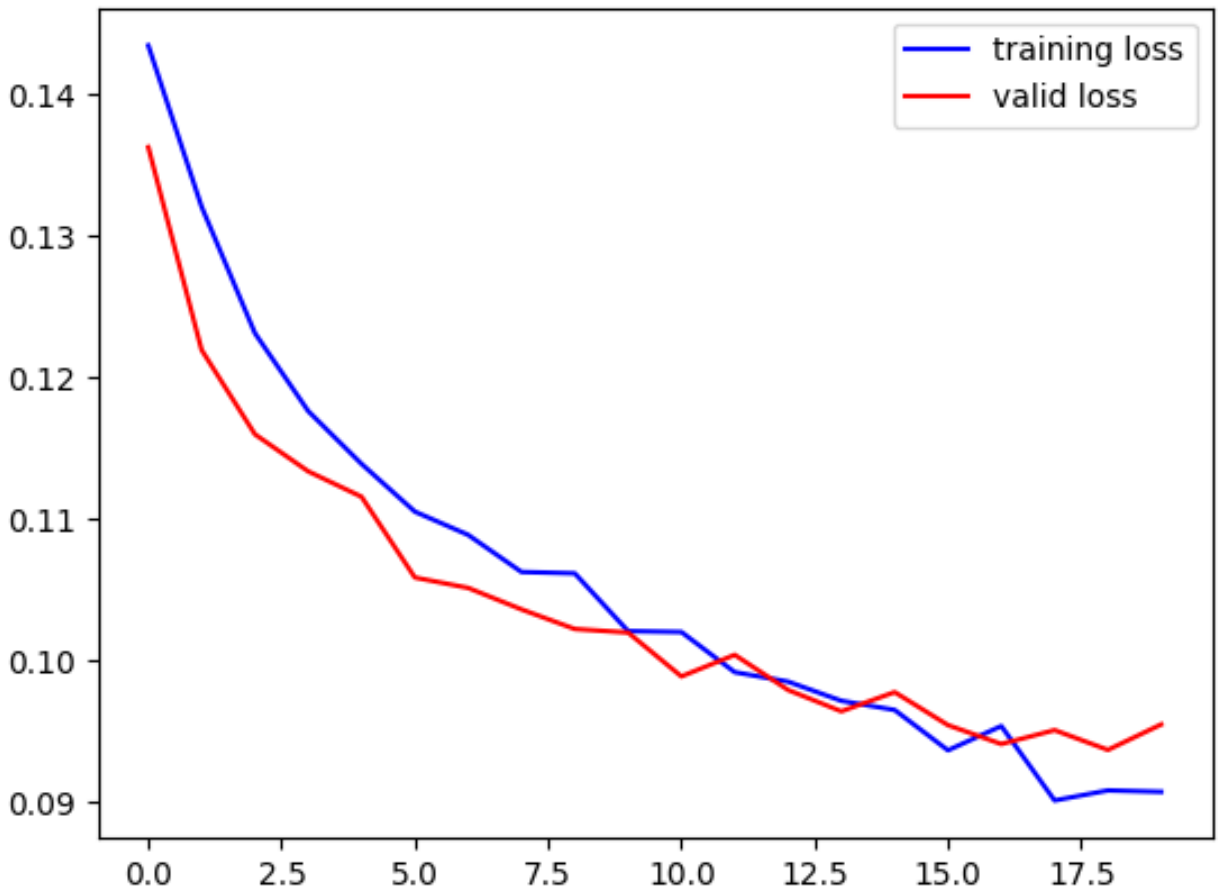
I choose to work on flowers 102 dataset.
Hyperparameters: batch_size = 32, num_epochs = 20, lr = 0.001.
My custom network:

```
# architecture is [CONV-POOL-CONV-POOL-FC-FC]
# [batch_size, 3, 224, 224]
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 9, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(9, 32, 5)
        self.fc1 = nn.Linear(89888, 2048)
        self.fc2 = nn.Linear(2048, 102)

    def forward(self, x):
        # print(x.shape)
        x = self.conv1(x)
        x = self.pool(x)
        # print(x.shape)
        x = self.conv2(x)
        x = self.pool(x)
        # flatten all dimensions except batch
        x = torch.flatten(x, 1)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
conv_net = Net()
```

squeezenet1_1, only modify the classifier blocks according to number of classes:

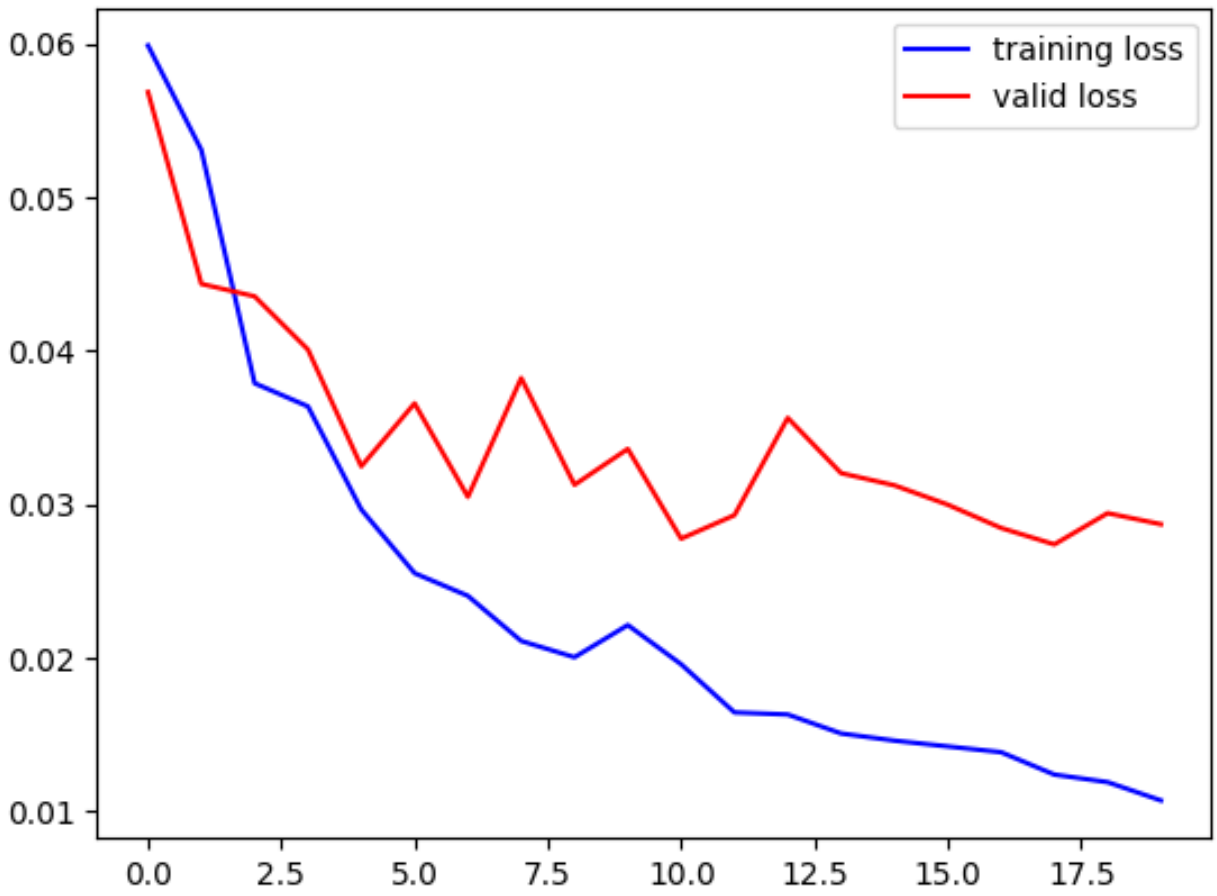
```
s_model.classifier = nn.Sequential(
  nn.Dropout(p=0.5, inplace=False),
  nn.Conv2d(512, 102, kernel_size=(1, 1), stride=(1, 1)),
  nn.ReLU(inplace=True),
  nn.AdaptiveAvgPool2d(output_size=(1, 1)),
)
```

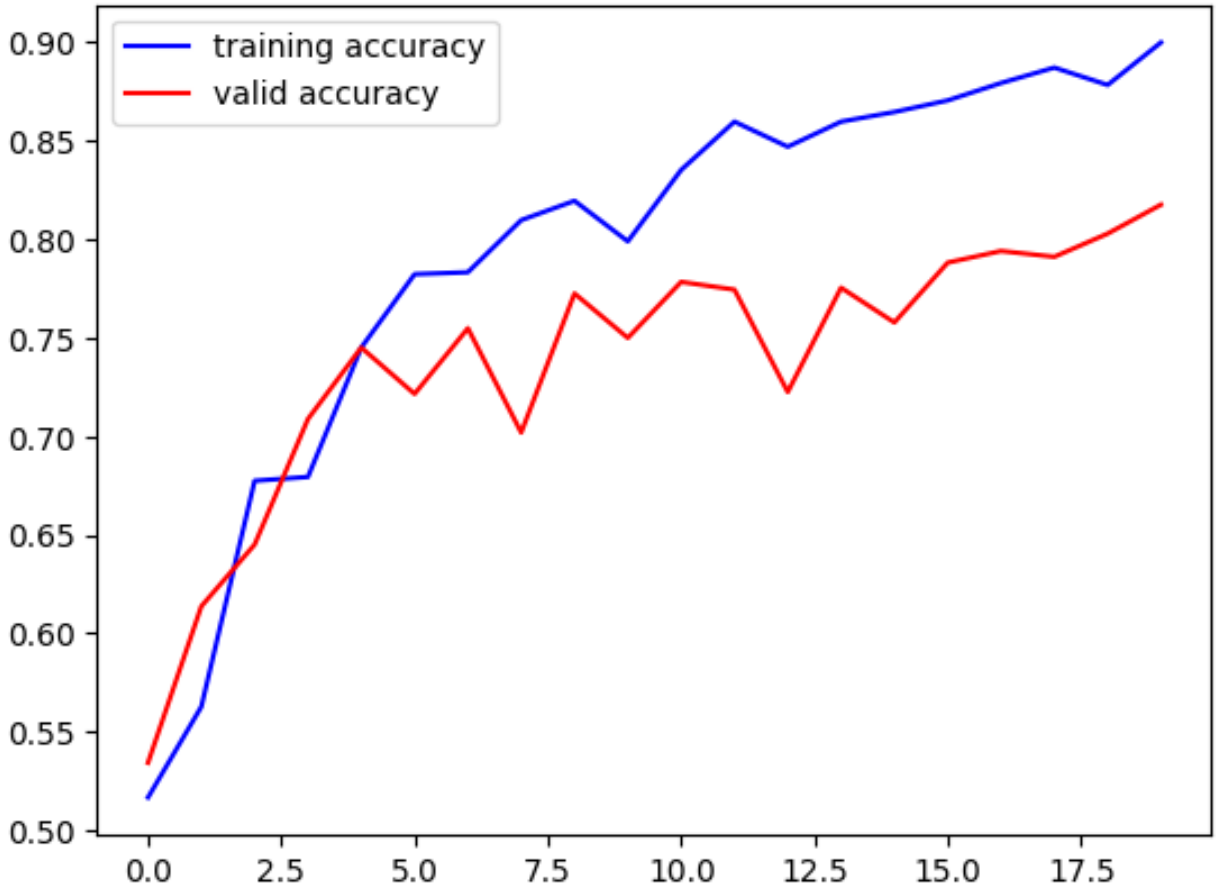
My custom network training and accuracy curve:


training loss
valid loss
0.14
0.13
0.12
0.11
0.10
0.09
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

test accuracy: 21.35%

First fine tune classifiy layer for 5 epoch, then train the network for num_epochs.
Fine-tuned squeezenet1_1 training and accuracy curve:


training loss
valid loss
0.06
0.05
0.04
0.03
0.02
0.01
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5



test accuracy: 75.5%

The fine-tuned squeezenet has better performance, this is understandable since it has more parameters than my custom network, and thus can classify on a complex task with 102 classes.