# 16-720 Computer Vision: Homework 5 (Fall 2022)

# Neural Networks for Recognition

Haejoon Lee

## 1 Theory

**Q1.1 Theory** [**3 points**] Prove that softmax is invariant to translation, that is

$$softmax(x) = softmax(x + c) \quad \forall c \in \mathbb{R}.$$

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

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

Often we use $c = -\max x_i$. Why is that a good idea? (Tip: consider the range of values that numerator will have with $c = 0$ and $c = -\max x_i$)

$$\frac{e^{x_i+c}}{\sum_i e^{x_i+c}} = \frac{e^c e^{x_i}}{e^c \sum_i e^{x_i}} = \frac{e^{x_i}}{\sum_i e^{x_i}}$$

Thus, softmax is invariant to translation.

By use -max(xi), we can confine the range of numerator to from 0 to 1 to prevent overflow.

**Q1.2 Theory** [**3 points**] Softmax can be written as a three step processes, with $s_i = e^{x_i}$, $S = \sum s_i$ and $softmax(x_i) = \frac{1}{S} s_i$.

- As $x \in \mathbb{R}^d$, what are the properties of $softmax(x)$, namely what is the range of each element? What is the sum over all elements?
- One could say that *"softmax takes an arbitrary real valued vector $x$ and turns it into a _____"*.
- Can you see the role of each step in the multi-step process now? Explain them.

1) Range of each element: 0~1, sum over all elements: 1
2) "Probability score"
3) First, map x_i to positive. Then, sum all for the next step. Lastly, divide s_i over the sum to calculate probability.

**Q1.3 Theory** [**3 points**] Show that multi-layer neural networks without a non-linear activation function are equivalent to linear regression.

$$
\begin{aligned}
h_1 &= W_1 x_1 + b_1 \\
h_2 &= W_2 h_1 + b_2 \\
&= W_2 W_1 x_1 + W_2 b_1 + b_2 \\
&= W^* x_1 + b^*
\end{aligned}
$$

**Q1.4 Theory** [**3 points**] Given the sigmoid activation function $\sigma(x) = \frac{1}{1+e^{-x}}$, derive the gradient of the sigmoid function and show that it can be written as a function of $\sigma(x)$ (without having access to $x$ directly)

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

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

**Q1.5 Theory** [**12 points**] Given $y = Wx + b$ (or $y_i = \sum_{j=1}^{d} x_j W_{ij} + b_i$), and the gradient of some loss $J$ with respect $y$, show how to get $\frac{\partial J}{\partial W}$, $\frac{\partial J}{\partial x}$ and $\frac{\partial J}{\partial b}$. Be sure to do the derivatives with scalars and re-form the matrix form afterwards. Here is some notional suggestions.

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

1)

$$\frac{\partial J}{\partial y_i} = \delta_i$$

$$\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial y_i} = \delta_i x_j$$

$$\begin{bmatrix} \delta_1 x_1 & \cdots & \delta_1 x_d \\ \vdots & \ddots & \vdots \\ \delta_1 x_i & \cdots & \delta_k x_d \end{bmatrix} \rightarrow \frac{\partial J}{\partial W} = \delta x^T$$

2)

$$\frac{\partial J}{\partial x_j} = \frac{\partial J}{\partial y} \frac{\partial y}{\partial x_j} = \delta * [W_{1j}, \cdots, W_{kj}]$$

$$\frac{\partial J}{\partial x} = W^T \delta$$

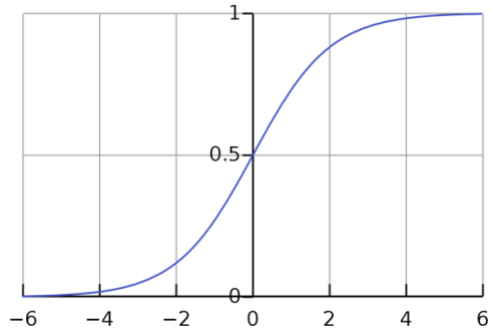
3)

$$\frac{\partial J}{\partial b_i} = \frac{\partial J}{\partial y_i} \frac{\partial y_i}{\partial b_i} = \delta_i \rightarrow \frac{\partial J}{\partial b} = \delta$$

**Q1.6 Theory [4 points]** When the neural network applies the elementwise activation function (such as sigmoid), the gradient of the activation function scales the backpropogation update. This is directly from the chain rule, $\frac{d}{dx}f(g(x)) = f'(g(x))g'(x)$.

1. Consider the sigmoid activation function for deep neural networks. Why might it lead to a "vanishing gradient" problem if it is used for many layers (consider plotting Q1.4)?
2. Often it is replaced with $\tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$. What are the output ranges of both tanh and sigmoid? Why might we prefer tanh ?
3. Why does $\tanh(x)$ have less of a vanishing gradient problem? (plotting the derivatives helps! for reference: $\tanh'(x) = 1 - \tanh(x)^2$)
4. tanh is a scaled and shifted version of the sigmoid. Show how $\tanh(x)$ can be written in terms of $\sigma(x)$.

1)




When the input for sigmoid goes to big positive or negative value, the gradient of the function approach to zero.

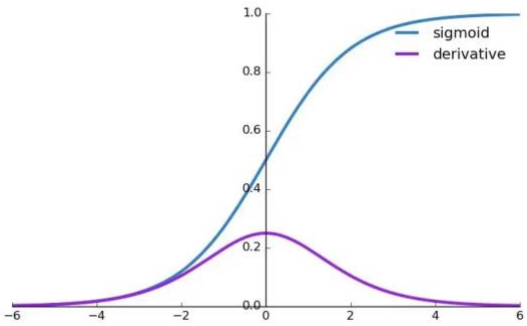
2)

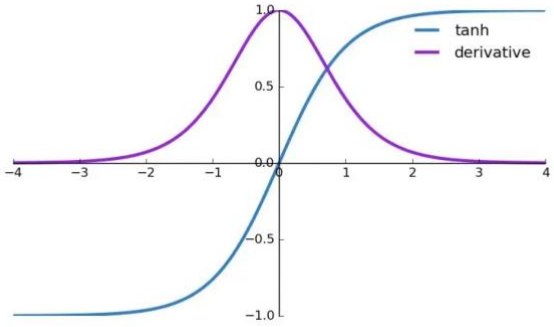
Sigmoid: 0 ~ 1

Tanh: -1 ~ 1

Since the Tanh is zero-centered, it can produce minus (-) gradient values for weight update.

3)




As shown above, tanh provides bigger gradient values than sigmoid.

4)

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

**Q2.1.1 Theory [3 points]** Why is it not a good idea to initialize a network with all zeros? If you imagine that every layer has weights and biases, what can a zero-initialized network output after training?

If we set all zero for weight initialization, gradient will be all the same for each layer leading to the same update during training.

**Q2.1.2 Code [3 points]** In `python/nn.py`, implement a function to initialize one layer's weights with Xavier initialization [1], where $Var[w] = \frac{2}{n_{in}+n_{out}}$ where $n$ is the dimensionality of the vectors and you use a **uniform distribution** to sample random numbers (see eq 16 in the paper). **Include your code in the writeup.**

```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

    ########################
    ##### your code here #####
    ########################
    W = np.random.uniform(-np.sqrt(6)/np.sqrt(in_size + out_size), np.sqrt(6)/np.sqrt(in_size + out_size), (in_size, out_size))
    b = np.zeros((out_size))

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

**Q2.1.3 Theory [3 points]** Why do we initialize with random numbers? Why do we scale the initialization depending on layer size (see near Fig 6 in the paper)?

Random initialization: to break symmetry of weights updates

Scaling: to preserve variance of activation at each layer and backpropagated gradients

**Q2.2.1 Code [4 points]** In `python/nn.py`, implement sigmoid, along with forward propagation for a single layer with an activation function, namely $y = \sigma(XW + b)$, returning the output and intermediate results for an $N \times D$ dimension input $X$, with examples along the rows, data dimensions along the columns. **Include your code in the writeup.**

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

    res = 1.0 / (1.0 + np.exp(-x))

    return res
```

```python
############################## 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 #####
    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 Code [3 points]** In `python/nn.py`, implement the softmax function. Be sure to use the numerical stability trick you derived in Q1.1 softmax. **Include your code in the writeup.**

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

    ##########################
    ##### your code here #####
    max_term = np.max(x, axis=1, keepdims=True)
    exp_term = np.exp(x-max_term)
    res = exp_term / np.sum(exp_term, axis=1, keepdims=True)
    ##########################

    return res
```

**Q2.2.3 Code [3 points]** In `python/nn.py`, write a function to compute the accuracy of a set of labels, along with the scalar loss across the data. The loss function generally used for classification is the cross-entropy loss.

$$L_{\mathbf{f}}(\mathbf{D}) = -\sum_{(\mathbf{x},\mathbf{y})\in\mathbf{D}} \mathbf{y}\cdot\log(\mathbf{f}(\mathbf{x}))$$

Here $\mathbf{D}$ is the full training dataset of data samples $\mathbf{x}$ ($N \times 1$ vectors, N = dimensionality of data) and labels $\mathbf{y}$ ($C \times 1$ one-hot vectors, C = number of classes), and $\mathbf{f} : \mathbb{R}^N \to [0,1]^C$ is the classifier. The log is natural log. **Include your code in the writeup.**

```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, None

    ##########################
    ##### your code here #####
    prediction = np.argmax(probs, axis=1)

    mask_true = y.astype(np.bool)
    probs_true = probs[mask_true]
    loss = 0.0 - np.sum(np.log(probs_true))

    mask_correct = mask_true[np.arange(prediction.shape[0]), prediction]
    num_correct = np.sum(mask_correct)
    acc = float(num_correct) / mask_correct.shape[0]
    ##########################

    return loss, acc
```

**Q2.3 Code [7 points]** In `python/nn.py`, write a function to compute backpropogation for a single layer, given the original weights, the appropriate intermediate results, and given gradient with respect to the loss. You should return the gradient with respect to $X$ so you can feed it into the next layer. As a size check, your gradients should be the same dimensions as the original objects. **Include your code in the writeup.**

```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 #####
    delta_pre = activation_deriv(post_act) * delta
    grad_W = X.transpose() @ delta_pre
    grad_b = np.sum(delta_pre, axis=0, keepdims=True)
    grad_b = grad_b[0,:] # (1, N) -> (N, )
    grad_X = delta_pre @ W.transpose()
    ##########################

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

**Q2.4 Code [5 points]** In `python/nn.py`, write a function that takes the entire dataset (`x` and `y`) as input and splits it into random batches. In `python/run_q2.py`, write a training loop that iterates over the batches, does forward and backward propagation, and applies a gradient update. The provided code samples batches only once, but it is also common to sample new random batches at each epoch. You may optionally try both strategies and note any difference in performance. **Include your code in the writeup.**

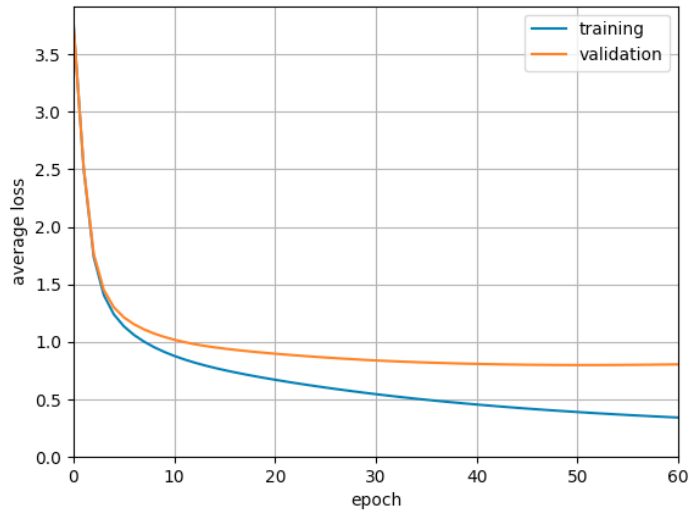
```python
############################## Q 2.4 ##############################
# split x and y into random batches
# return a list of [(batch1_x,batch1_y)...]
def get_random_batches(x,y,batch_size):
    batches = []
    ########################
    ##### your code here #####
    num_data = x.shape[0]
    idxs_random = list(np.random.permutation(num_data))

    bp = 0
    while bp < num_data:
        idxs_batch = idxs_random[bp:bp+batch_size]
        batches.append((x[idxs_batch, :], y[idxs_batch, :]))
        bp += batch_size
    ########################
    return batches
```

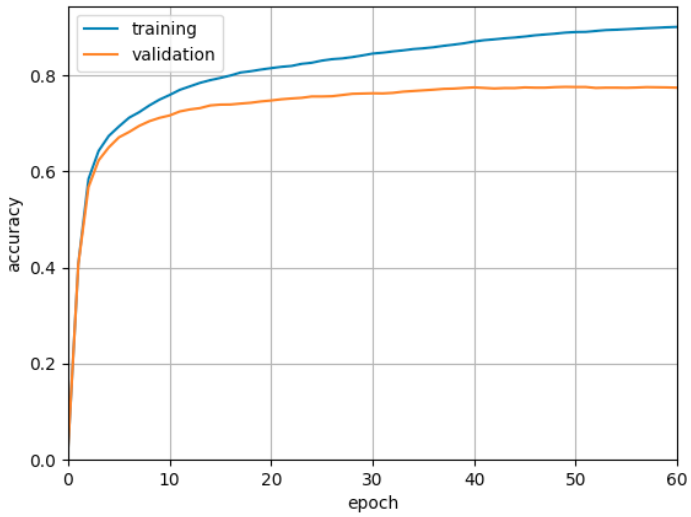
**Q2.5** [**5 points**] In `python/run_q2.py`, implement a numerical gradient checker. Instead of using the analytical gradients computed from the chain rule, add $\epsilon$ offset to each element in the weights, and compute the numerical gradient of the loss with central differences. Central differences is just $\frac{f(x+\epsilon)-f(x-\epsilon)}{2\epsilon}$. Remember, this needs to be done for each scalar dimension in all of your weights independently. This should help you check your gradient code, so there is no need to show the result, but do **include your code in the writeup.**

```
# Q 2.5 should be implemented in this file
# you can do this before or after training the network.

# compute gradients using forward and backward
h1 = forward(x,params,'layer1')
probs = forward(h1,params,'output',softmax)
loss, acc = compute_loss_and_acc(y, probs)
delta1 = probs - y
delta2 = backwards(delta1,params,'output',linear_deriv)
backwards(delta2,params,'layer1',sigmoid_deriv)

# save the old params
import copy
params_orig = copy.deepcopy(params)

# compute gradients using finite difference
eps = 1e-6
for k,v in params.items():
    if '_' in k:
        continue
    # we have a real parameter!
    # 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

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

    if v.ndim == 1:
        # print('v.shape is ',v.shape)
        # continue
        grad_tensor = params['grad_' + k]

        for i in range(len(v)):
            vij_orig = v[i]
```

```
            v[i] = vij_orig + eps

            h1 = forward(x, params, 'layer1')
            probs = forward(h1, params, 'output', softmax)
            loss_plus, acc = compute_loss_and_acc(y, probs)

            v[i] = vij_orig - eps

            h1 = forward(x, params, 'layer1')
            probs = forward(h1, params, 'output', softmax)
            loss_minus, acc = compute_loss_and_acc(y, probs)

            v[i] = vij_orig

            diff_grad = (loss_plus - loss_minus) / (2*eps)

            grad_tensor[i] = diff_grad

    else:
        rows, cols = v.shape[0:2]
        # print('v.shape is ',v.shape)
        # continue
        grad_tensor = params['grad_' + k]

        for i in range(rows):
            for j in range(cols):

                vij_orig = v[i, j]

                v[i, j] = vij_orig - eps
                h1 = forward(x, params, 'layer1')
                probs = forward(h1, params, 'output', softmax)
                loss_minus, acc = compute_loss_and_acc(y, probs)

                v[i, j] = vij_orig + eps
                h1 = forward(x, params, 'layer1')
                probs = forward(h1, params, 'output', softmax)
                loss_plus, acc = compute_loss_and_acc(y, probs)

                v[i, j] = vij_orig
                diff_grad = (loss_plus - loss_minus) / (2*eps)
                grad_tensor[i, j] = diff_grad

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

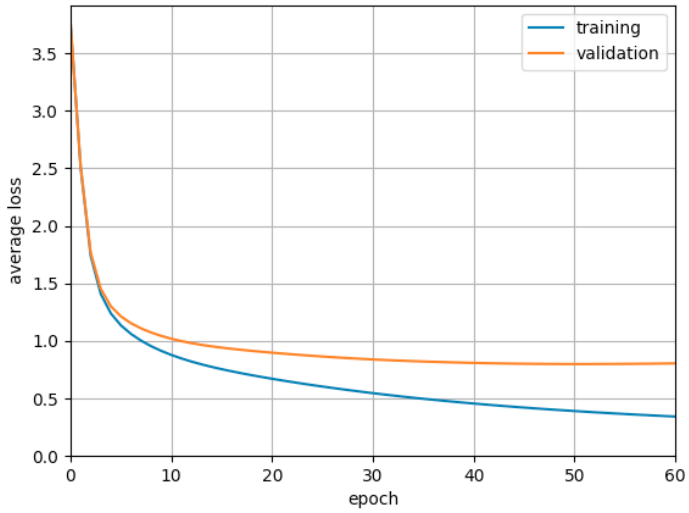
**Q3.1 Code [5 points]** Train a network from scratch. Use a single hidden layer with 64 hidden units, and train for at least 50 epochs. The script will generate two plots: one showing the accuracy on both the training and validation set over the epochs, and the other showing the cross-entropy loss averaged over the data. Tune the batch size and learning rate to get an accuracy on the validation set of at least 75%. Include the plots in your writeup. Hint: Use fixed random seeds to improve reproducability.

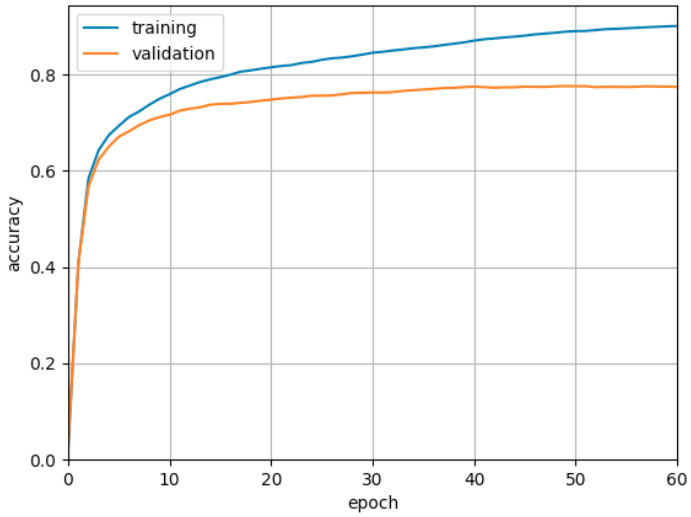


**Q3.2 Writeup [3 points]** Use the script to train three networks, one with your tuned learning rate, one with 10 times that learning rate, and one with one tenth that learning rate. Include all six plots in your writeup (two will be the same from the previous question). Comment on how the learning rates affect the training, and report the final accuracy of the best network on the test set. Hint: Use fixed random seeds to improve reproducability.

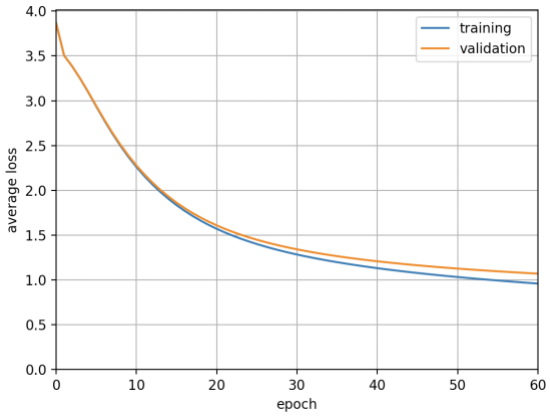
1) Learning rate: 0.005. I achieved the best performance with this learning rate.

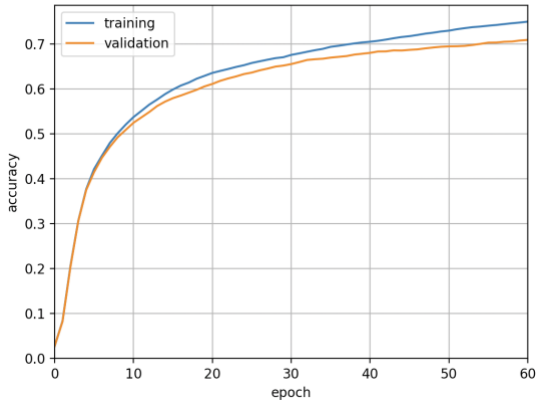


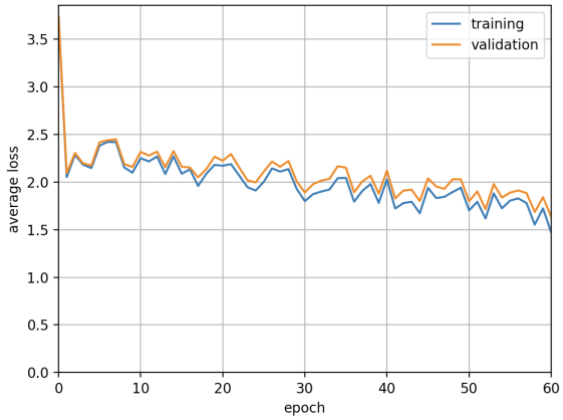

```
Validation accuracy:  0.7741666666666667
Test accuracy:  0.77
```

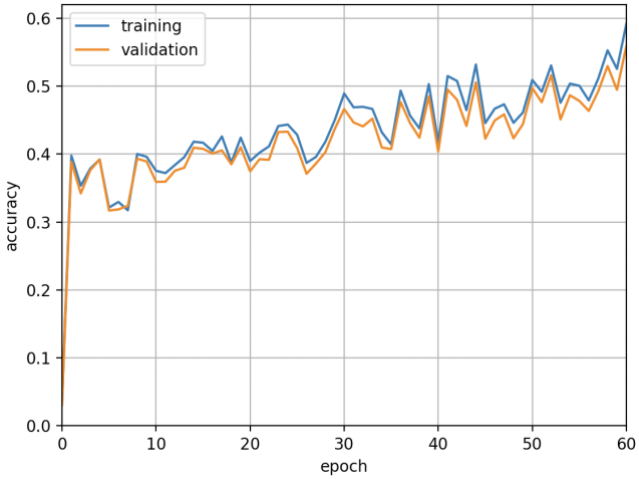
2) Learning rate: 0.0005



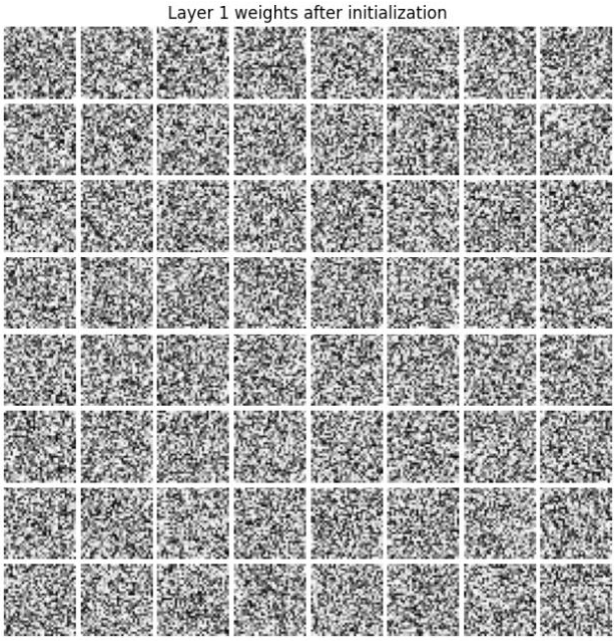

3) Learning rate: 0.05






By decreasing learning rate, training got slower and performances between on training set and validation set were almost the same. When learning rate was increased, training was unstable, performance couldn't be achieved as much high as before.

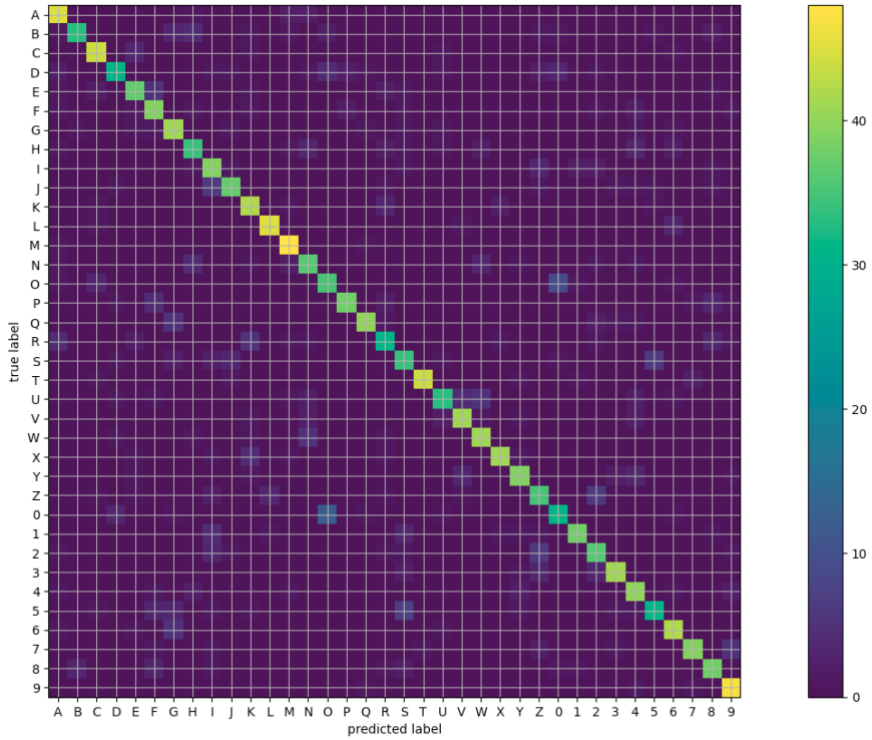
**Q3.3 Writeup [3 points]** The script will visualize the first layer weights as 64 32x32 images, both immediately after initialization and after fully training. Include both visualizations in your writeup. Comment on the learned weights and compare them to the initialized weights. Do you notice any patterns?




D


After training, the weights became blurring and edge capturing filters. Some of them clearly looks some a piece of a letter.

**Q3.4 Writeup [3 points]** Visualize and include the confusion matrix of the test data for your best model. Comment on the top few pairs of classes that are most commonly confused.




From the result, we can find few pairs of classes that the network confused

1st: 0 and O

2nd: S and 5

3rd: z and 2

All of them are similar letters and the result is understandable.

**Q4.1 Theory [4 points]** The method outlined above is pretty simplistic, and while it works for the given text samples, it makes several assumptions. What are two big assumptions that the sample method makes? In your writeup, include two example images where you expect the character detection to fail (for example, miss valid letters, misclassify letters or respond to non-letters).

Assumption:

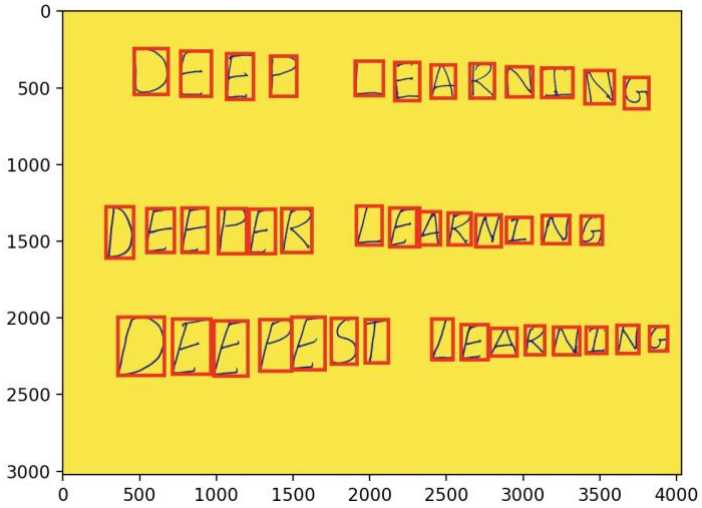
1) Letters are placed separately. Shouldn't be connected.
2) The size of letters is consistent. Small bounding boxes will be skipped.

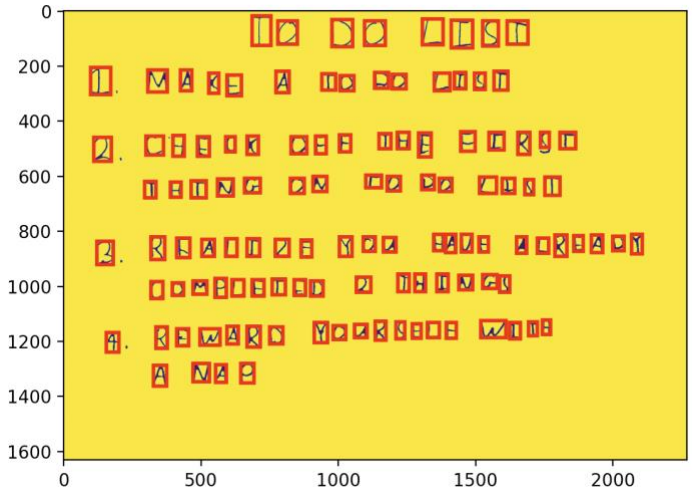
Examples:

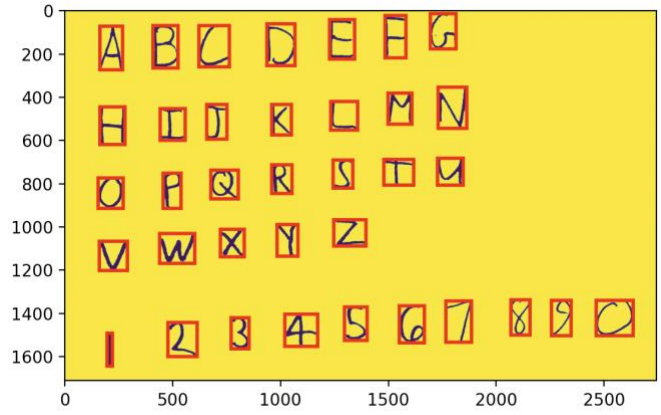
**Q4.2 Code [10 points]** In `python/q4.py`, implement the function to find letters in the image. Given an RGB image, this function should return bounding boxes for all of the located handwritten characters in the image, as well as a binary black-and-white version of the image `im`. Each row of the matrix should contain `[y1,x1,y2,x2]` the positions of the top-left and bottom-right corners of the box. The black and white image should be floating point, 0 to 1, with the characters in black and background in white. Hint: Since we read text left to right, top to bottom, we can use this to cluster the coordinates. **Include your code in the writeup.**

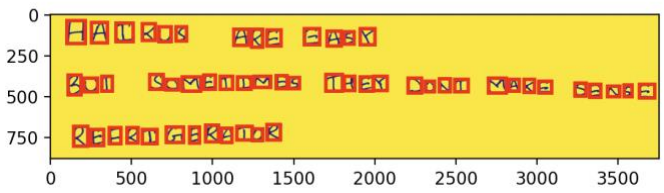
```python
# takes a color image
# returns a list of bounding boxes and black_and_white image
def findLetters(image):
    bboxes = []
    bw = None
    # insert processing in here
    # one idea estimate noise -> denoise -> greyscale -> threshold -> morphology -> label -> skip small boxes
    # this can be 10 to 15 lines of code using skimage functions

    ##########################
    ##### your code here #####
    ##########################
    image = skimage.restoration.denoise_bilateral(image, multichannel=True) # Denoising

    gray = skimage.color.rgb2gray(image) # Grayscale

    thresh = skimage.filters.threshold_otsu(gray) # Thresholding

    bw = skimage.morphology.closing(gray<thresh, skimage.morphology.square(7)) # Morphology

    label_image = skimage.morphology.label(bw,connectivity=2) # Label

    #Skip small boxes
    props_region = skimage.measure.regionprops(label_image)
    total_area = 0
    for region in props_region:
        total_area += region.area
    mean_area = total_area / len(props_region)

    for region in props_region:
        if region.area > 0.5 * mean_area:
            bboxes.append(region.bbox)

    bw = (~bw).astype(np.float)

    return bboxes, bw
```

**Q4.3 Writeup [5 points]** Using `python/run_q4.py`, visualize all of the located boxes on top of the binary image to show the accuracy of your `findLetters(..)` function. Include all the resulting images in your writeup.






0
250
500
750
0
500
1000
1500
2000
2500
3000
3500

**Q4.4 Code/Writeup [8 points]** In `python/run_q4.py`, you will now load the image, find the character locations, classify each one with the network you trained in **Q3.1**, and return the text contained in the image. Be sure you try to make your detected images look like the images from the training set. Visualize them and adjust accordingly. If you find that your classifier performs poorly, consider dilation under skimage morphology to make the letters thicker.

Your solution is correct if you can correctly detect approximately 100% of the letters and classify approximately 80% of the letters in each of the sample images.

Run your `run_q4` on all of the provided sample images in `images/`. Include the extracted text in your writeup. It is fine if your code ignores spaces, but if so, please add them manually in the writeup.

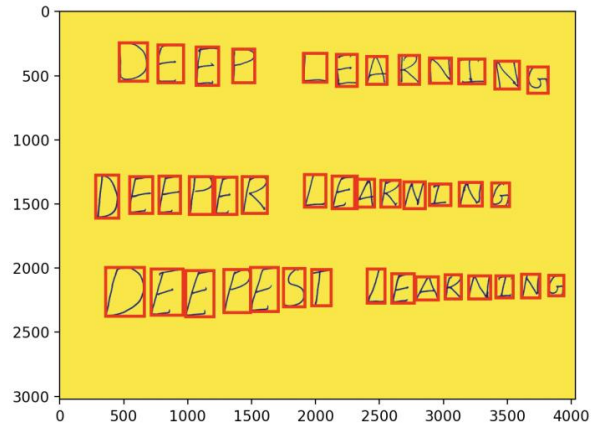
```
(cv) HAEJOON@haejoons-mbp python % /Users/
1.0
File name: 04_deep.jpg
DEBPLBARMING
DEBP8RLEARHJNG
D88P8SPLEARNING
1.0
File name: 01_list.jpg
T0DJLIST
INAXEATQ2ULISF
2CHFCKUFFTHEFIR8T
7HINGQNTQOQLIST
3REALIZEYQUHAUEALR6ADT
CQMPLFTJDJIHINGS
9RFWARDYQURGELFWITB
ANAP
1.0
File name: 02_letters.jpg
ABCDBFG
HI3KLMN
QPQRSTV
VWXYZ
1Z3GS6789J
1.0
File name: 03_haiku.jpg
HAIKUSAREHAGX
BUTSQMBTIMBGTHEXDQWTMAKGSHNGE
RBGRIGERATQR
```

File name: 04_deep.jpg

DEBP LBARMING

DEBP8R LEARHJNG

D88P8SP LEARNING




File name: 01_list.jpg

T0 DJ LIST

I NAXE A TQ2U LISF

2 CHFCK UFF THE FIR8T

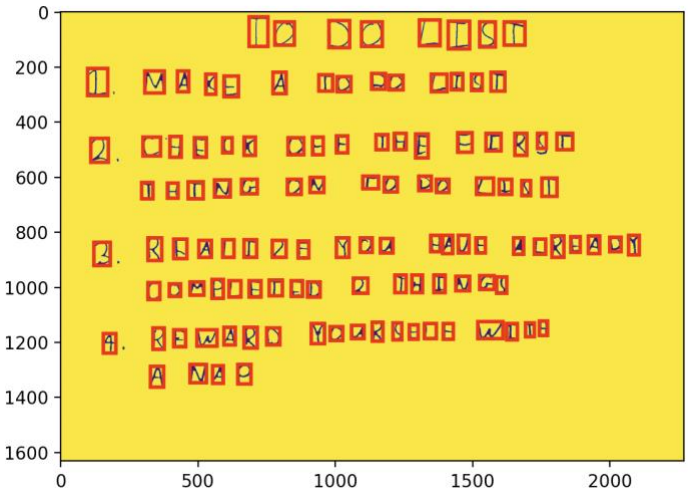
7HING QN TQ OQ LIST

3 REALIZE YQU HAUE ALR6ADT

CQMPLFTJD J IHINGS

9 RFWARD YQURGELF WITB

A NAP



File name: 02_letters.jpg

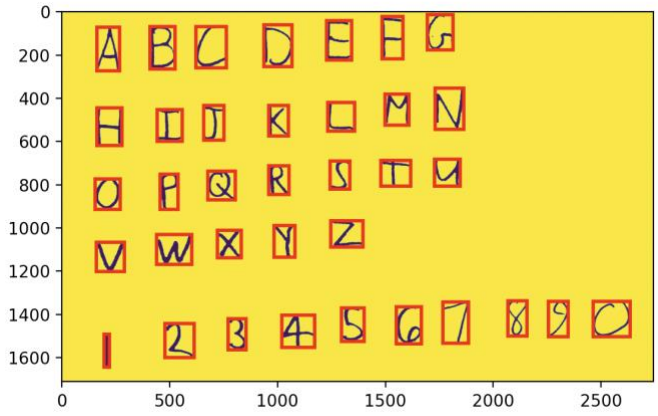
ABCDBFG

HI3KLMN

QPQRSTV

VWXYZ

1Z3GS6789J

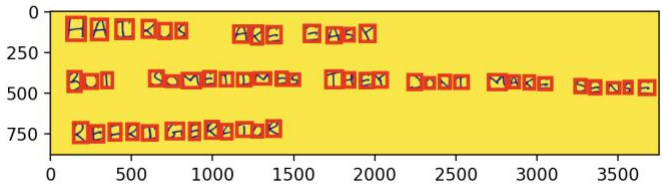



File name: 03_haiku.jpg

HAIKUS ARE HAGX

BUT SQMBTIMBG THEX DQWT MAKG SHNGE

RBGRIGERATQR



## 5.1 Building the Autoencoder

**Q5.1.1 (Extra Credit) Code [5 points]** Due to the difficulty in training auto-encoders, we have to move to the $relu(x) = max(x, 0)$ activation function. It is provided for you in `util.py`. Implement an autoencoder where the layers are

- 1024 to 32 dimensions, followed by a ReLU
- 32 to 32 dimensions, followed by a ReLU
- 32 to 32 dimensions, followed by a ReLU
- 32 to 1024 dimensions, followed by a sigmoid (this normalizes the image output for us)

The loss function that you're using is total squared error for the output image compared to the input image (they should be the same!). **Include your code in the writeup.**

```
# Q5.1 & Q5.2
# initialize layers here
#########################
##### your code here #####
#########################

in_size = train_x.shape[1] # 1024
initialize_weights(in_size, hidden_size, params, 'layer1')
initialize_weights(hidden_size, hidden_size,params, 'hidden1')
initialize_weights(hidden_size, hidden_size,params, 'hidden2')
initialize_weights(hidden_size, in_size,params, 'output')

# Make new params (m_Wlayer1, ...) for momentum
keys = [k for k in params.keys()]
for k in keys:
    params['m_' + k] = np.zeros(params[k].shape)

# 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

        #########################
        ##### your code here #####
        #########################
        pass
```

```
            h1 = forward(X=xb, params=params, name='layer1', activation=relu)
            h2 = forward(X=h1, params=params, name='hidden1', activation=relu)
            h3 = forward(X=h2, params=params, name='hidden2', activation=relu)
            x_recon = forward(X=h3, params=params, name='output', activation=sigmoid)

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

            delta1 = 2*(x_recon-xb)
            delta2 = backwards(delta=delta1, params=params, name='output', activation_deriv=sigmoid_deriv)
            delta3 = backwards(delta=delta2, params=params, name='hidden2', activation_deriv=relu_deriv)
            delta4 = backwards(delta=delta3, params=params, name='hidden1', activation_deriv=relu_deriv)
            backwards(delta=delta4, params=params, name='layer1', activation_deriv=relu_deriv)

            # Momentum update
            for k in params.keys():
                if '_' not in k:
                    params['m_'+k] = 0.9 * params['m_'+k] - learning_rate * params['grad_'+k]
                    params[k] += params['m_'+k]

    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

# plot loss curve
plt.plot(range(len(losses)), losses)
plt.xlabel("epoch")
plt.ylabel("average loss")
plt.xlim(0, len(losses)-1)
plt.ylim(0, None)
plt.grid()
plt.show()
```

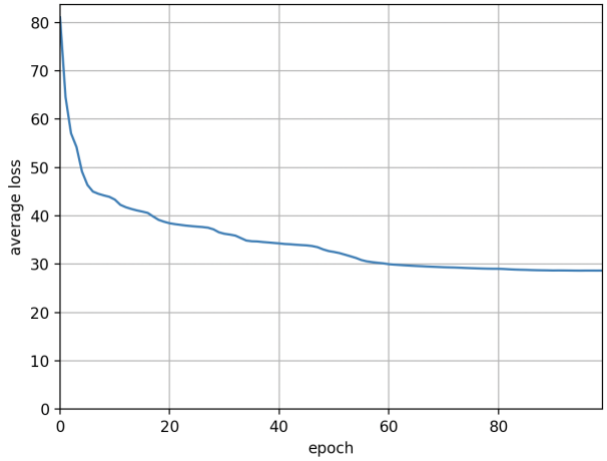
**Q5.1.2 (Extra Credit) Code [5 points]** To help even more with convergence speed, we will implement momentum. Now, instead of updating $W = W - \alpha \frac{\partial J}{\partial W}$, we will use the update rules $M_W = 0.9M_W - \alpha \frac{\partial J}{\partial W}$ and $W = W + M_W$. To implement this, populate the parameters dictionary with zero-initialized momentum accumulators, one for each parameter. Then simply perform both update equations for every batch. **Include your code in the writeup.**

Same code for Q5.1.1

```
# Q5.1 & Q5.2
# initialize layers here
#########################
##### your code here #####
#########################

in_size = train_x.shape[1] # 1024
initialize_weights(in_size, hidden_size, params, 'layer1')
initialize_weights(hidden_size, hidden_size,params, 'hidden1')
initialize_weights(hidden_size, hidden_size,params, 'hidden2')
initialize_weights(hidden_size, in_size,params, 'output')

# Make new params (m_Wlayer1, ...) for momentum
keys = [k for k in params.keys()]
for k in keys:
    params['m_' + k] = np.zeros(params[k].shape)

# 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

        #########################
        ##### your code here #####
        #########################
        pass
        h1 = forward(X=xb, params=params, name='layer1', activation=relu)
        h2 = forward(X=h1, params=params, name='hidden1', activation=relu)
        h3 = forward(X=h2, params=params, name='hidden2', activation=relu)
        x_recon = forward(X=h3, params=params, name='output', activation=sigmoid)

        loss = np.sum((x_recon-xb)**2)
        total_loss += loss
```

```
            delta1 = 2*(x_recon-xb)
            delta2 = backwards(delta=delta1, params=params, name='output', activation_deriv=sigmoid_deriv)
            delta3 = backwards(delta=delta2, params=params, name='hidden2', activation_deriv=relu_deriv)
            delta4 = backwards(delta=delta3, params=params, name='hidden1', activation_deriv=relu_deriv)
            backwards(delta=delta4, params=params, name='layer1', activation_deriv=relu_deriv)

            # Momentum update
            for k in params.keys():
                if '_' not in k:
                    params['m_'+k] = 0.9 * params['m_'+k] - learning_rate * params['grad_'+k]
                    params[k] += params['m_'+k]

        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

# plot loss curve
plt.plot(range(len(losses)), losses)
plt.xlabel("epoch")
plt.ylabel("average loss")
plt.xlim(0, len(losses)-1)
plt.ylim(0, None)
plt.grid()
plt.show()
```

**Q5.2 (Extra Credit) Writeup/Code [5 points]** Using the provided default settings, train the network for 100 epochs. Plot the training loss curve and include it in the writeup. What do you observe?




The average loss saturated after 80 epochs even during training.

**Q5.3.1 (Extra Credit) Writeup/Code [5 points]** Now let's evaluate how well the autoencoder has been trained. Select 5 classes from the total 36 classes in the validation set and for each selected class include in your report 2 validation images and their reconstruction. What differences do you observe in the reconstructed validation images compared to the original ones?

The reconstructed images were at the similar shape with original images but much blurrier.

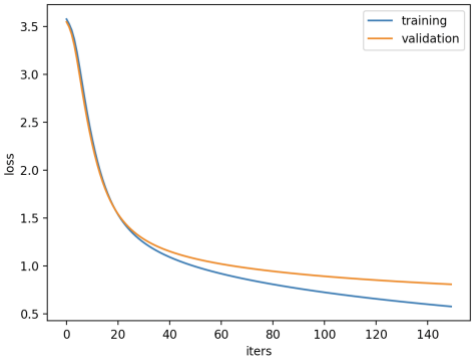
**Q5.3.2 (Extra Credit) Writeup [5 points]** Let's evaluate the reconstruction quality using Peak Signal- to-noise Ratio (PSNR). PSNR is defined as

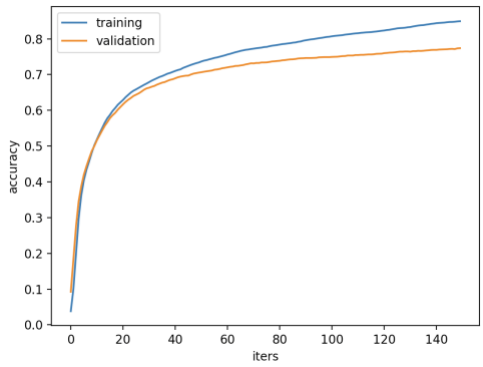
$$\text{PSNR} = 20 \times \log_{10}(\text{MAX}_I) - 10 \times \log_{10}(\text{MSE}) \tag{1}$$

where $\text{MAX}_I$ is the maximum possible pixel value of the image, and MSE (mean squared error) is computed across all pixels. Said another way, maximum refers to the brightest overall sum (maximum positive value of the sum). You may use skimage.metrics.peak_signal_noise_ratio for convenience. Report the average PSNR you get from the autoencoder across all images in the validation set (it should be around 15).

```
PSNR: 15.819120654450426
```

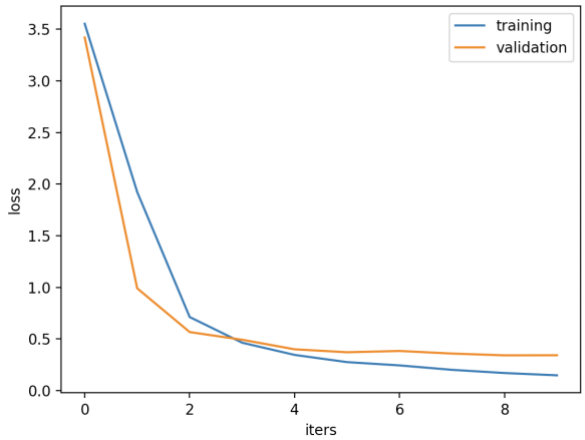
**Q6.1.1 Code/Writeup [5 points]** Re-write and re-train your fully-connected network on the included NIST36 in PyTorch. Plot training accuracy and loss over time.

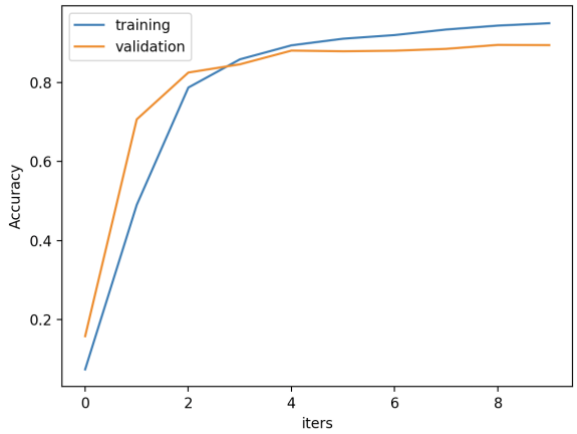



```
Validation accuracy:  0.7738888888888888
```

**Q6.1.2 Code/Writeup [5 points]** Train a convolutional neural network with PyTorch on the included NIST36 dataset. Compare its performance with the previous fully-connected network.

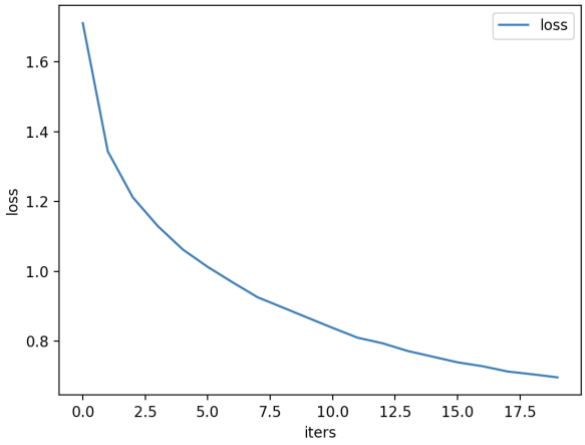


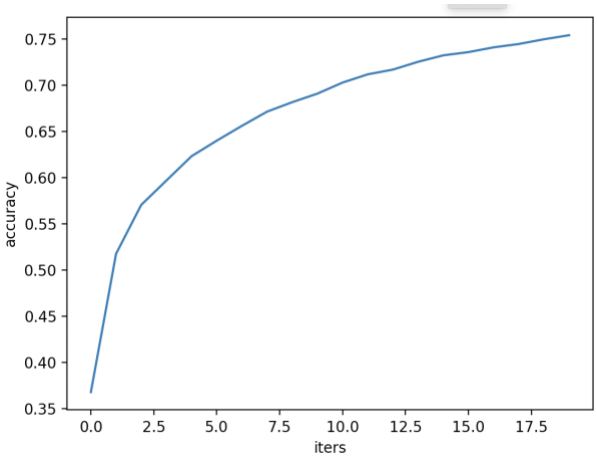




```
Validation accuracy:  0.895
```

Only with 10 iterations (FC: 100 iterations), CNN could achieve higher performance (Validation accuracy: CNN - 89%, FCN – 77%)

**Q6.1.3 Code/Writeup [5 points]** Train a convolutional neural network with PyTorch on CIFAR-10 (`torchvision.datasets.CIFAR10`). Plot training accuracy and loss over time.



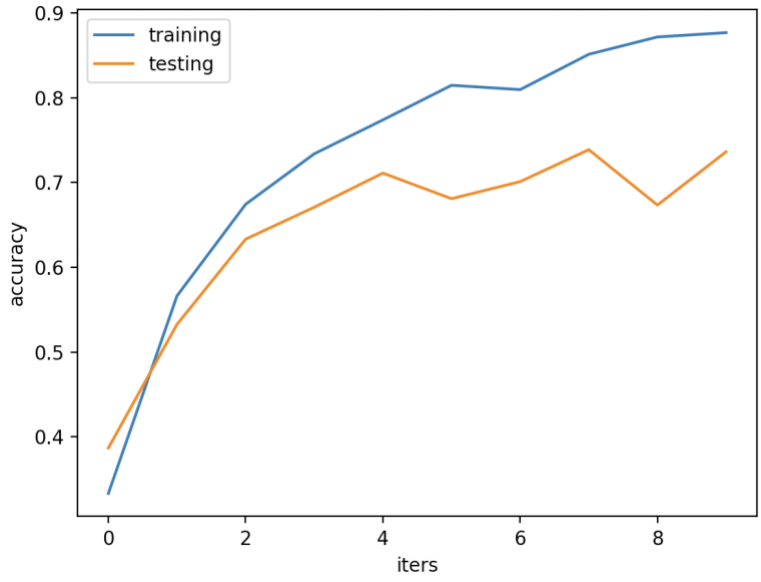



**Q6.1.4 Code/Writeup [10 points]** In Homework 1, we tried scene classification with the bag-of-words (BoW) approach on a subset of the SUN database. Use the same dataset in HW1, and implement a convolutional neural network with PyTorch for scene classification. Compare your result with the one you got in HW1, and briefly comment on it.

**3.1**

| filter_scales | K | Alpha | L | Accuracy |
|---|---|---|---|---|
| [1, 2, 4, 8, 8*sqrt(2)] | 10 | 25 | 2 | 49.75% |
| [1, 2, 4, 8, 8*sqrt(2)] | 10 | 50 | 2 | 51.75% |
| [1, 2, 4, 8, 8*sqrt(2)] | 20 | 50 | 2 | 42.50% |
| [1, 2, 4, 8, 8*sqrt(2)] | 10 | 1000 | 2 | 48.50% |
| [1, 2, 4, 8, 8*sqrt(2)] | 10 | 100 | 2 | 48.25% |
| [1, 2, 4, 8, 8*math.sqrt(2), 16] | 10 | 1000 | 2 | 46.75% |
| [1, 2, 4, 8, 8*sqrt(2)] | 10 | 50 | 4 | 49.00% |
| [1, 2, 4, 8, 8*sqrt(2)] | 200 | 100 | 3 | 57.50% |
| [1, 2, 4, 8, 8*sqrt(2)] | 250 | 150 | 3 | 53.45 |
| [1, 2, 4, 8, 8*sqrt(2)] | 400 | 400 | 2 | **59.25%** |

Above result is from HW1. The highest Accuracy was 59.25%. And the simple CNN, SUNet, could provide 74% accuracy on the same testing dataset as shown below. It is expected to due to that CNN could learn much contextual information and has bigger capacity (more learnable parameters).



**Q6.3 (Extra Credit) Neural Networks in the Real World [20 points]** Download an ImageNet pretrained image classification model of your choice from torchvision. Using this model, pick a single category (out of the 1000 used to train the model) and evaluate the validation performance of this category. You can download the ImageNet validation data from the challenge page by creating an account (top right). Torchvision has a dataloader to help you load and process ImageNet data automatically. Next, find an instance of this selected category in the real world and take a dynamic (i.e with some movement) video of this object. Extract all of the frames from this video and apply your pretrained model to each frame and compare the accuracy of the classifier on your video compared with the images in the validation set. Why might this be? Can you suggest ways to make your model more robust?

First, I tested ResNet50 pretrained on ImageNet dataset on a single validation image from the dataset whose category is 'old English sheepdog'

```
Old English sheepdog: 46.1%
```

It successfully classified the image. Then, I download a video of English sheepdog, extracted 150 frames, and tested the model on them.


```
Accuracy from 150 frames: 42.0%
```

The total accuracy was 42%, which was much lower than the reported performance in the original paper. The reason will be the motion blur from moving and low resolution. If we have to test the 2D model on each video frame, deblurring process will be improve the performance.