CMU 16-720: Homework 5

Joao Vitor Dias Monteiro
jmonteir@andrew.cmu.edu
December 4, 2022

Q1.1: Prove that softmax is invariant to translation. Oftenr we use $c = -max\ x_i$. Why is that a good idea?

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \times \frac{e^{c}}{e^{c}} = \frac{e^{x_i+c}}{\sum_{j=1}^{n} e^{x_j+c}} = softmax(x_i + c)$$

Often $c = -max\ x_i$ is used for the purpose of numerical stability since in that case e^{x_j+c} will be at most one, thus guaranteeing the denominator will not have large values due to the exponentials.

Q1.2: What is the range of $softmax(x_i)$? What is $\sum_{i=1}^{N} softmax(x_i)$?

- The range of $softmax(x_i)$ is between 0 and 1. The sum of all $softmax(x_i)$ terms is 1.
- One could say that "softmax takes an arbitrary real values vector and turns it into a vector of numbers between zero and one".
- $s_i = e^{x_i}$ maps x_i into a positive number , $S = \sum s_i$ computes the sum of all s_i and $softmax(x_i) = (1/S)s_i$ maps s_i to a number between 0 and 1.

Q1.3: Show that multi-layer neural networks without a non-linear activation function are equivalent to linear regression.

Without loss of generality, consider a neural network with 3 layers:

$$\hat{y} = h_3(\boldsymbol{W}_3 h_2(\boldsymbol{W}_2 h_1(\boldsymbol{W}_1 x + \boldsymbol{b}_1) + \boldsymbol{b}_2) + \boldsymbol{b}_3)$$

where h_i is the activation function, $\boldsymbol{W} \in \mathbb{R}^{k_i \times k_{i-1}}$ is the weight matrix, $\boldsymbol{b} \in \mathbb{R}^{k_i}$ is the bias term, k_i is the number of nodes and subscript i means that all of these are associated with the i-th layer. Lastly, $x \in \mathbb{R}^{k_0}$ is the input features and $\hat{y}$ the corresponding prediction by this multilayer neural network.

Now, let h_i be a linear function, that is

$$h_i(z) = \boldsymbol{M}_i z$$

where $\boldsymbol{M}_i \in \mathbb{R}^{k_i \times k_i}$. Notice that

$$h_1(\boldsymbol{W}_1 x + \boldsymbol{b}_1) = \boldsymbol{M}_1(\boldsymbol{W}_1 x + \boldsymbol{b}_1) = \boldsymbol{M}_1\boldsymbol{W}_1 x + \boldsymbol{M}_1\boldsymbol{b}_1 = \boldsymbol{W}_1^* x + \boldsymbol{b}_1^*$$

$$h_2(\boldsymbol{W}_2 h_1(\boldsymbol{W}_1 x + \boldsymbol{b}_1) + \boldsymbol{b}_2) = \boldsymbol{M}_2(\boldsymbol{W}_2\boldsymbol{W}_1^* x + \boldsymbol{W}_2\boldsymbol{b}_1^* + \boldsymbol{b}_2) = \boldsymbol{W}_2^* x + \boldsymbol{b}_2^*$$

$$\hat{y} = h_3(\boldsymbol{W}_3(\boldsymbol{W}_2^* x + \boldsymbol{b}_2^*)) = \boldsymbol{M}_3(\boldsymbol{W}_3\boldsymbol{W}_2^* x + \boldsymbol{W}_3\boldsymbol{b}_2^* + \boldsymbol{b}_3) = \boldsymbol{W}_3^* x + \boldsymbol{b}_3^*$$

Therefore $\hat{y}$ is a linear transformation of x thus proving that if h_i's are linear activation functions the neural network becomes a linear regression.

Q1.4: Given the sigmoid 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 $\sigma(x)$.

$$\begin{aligned}
\frac{d}{dx}\sigma(x) &= \frac{0 \times (1 - e^{-x}) - 1 \times (-e^{-x})}{(1 + e^{-x})^2} \\
&= \left(\frac{e^{-x}}{1 + e^{-x}}\right) \times \frac{1}{1 + e^{-x}} \\
&= \left(1 - \frac{1}{1 + e^{-x}}\right) \times \frac{1}{1 + e^{-x}} \\
&= [1 - \sigma(x)]\sigma(x)
\end{aligned}$$

Q1.5: Given $y = Wx + b$ 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 $\frac{\partial J}{\partial b}$.

By using chain rule we can get

$$\frac{\partial J}{\partial W_{ij}} = \left(\frac{\partial J}{\partial y}\right) \frac{\partial y}{\partial W_{ij}} = \delta_i x_j$$

Therefore,

$$\frac{\partial J}{\partial W} = \begin{bmatrix} \delta_1 x_1 & \delta_1 x_2 & \cdots & \delta_1 x_d \\ \delta_2 x_1 & \delta_2 x_2 & \cdots & \delta_2 x_d \\ \vdots & \vdots & \vdots & \vdots \\ \delta_k x_1 & \delta_k x_2 & \cdots & \delta_k x_d \end{bmatrix} = \delta x^T$$

Similarly,

$$\frac{\partial J}{\partial x} = \left[\left(\frac{\partial J}{\partial y}\right)^T \frac{\partial y}{\partial x}\right]^T = \left(\delta^T W\right)^T = W^T \delta$$

and

$$\frac{\partial J}{\partial b} = \left[\left(\frac{\partial J}{\partial y}\right)^T \frac{\partial y}{\partial b}\right]^T = \left(\delta^T I_k\right)^T = \delta$$

where I_k is a $k \times k$ identity matrix.

Q1.6: Various questions about vanishing gradient problem, and comparison between tanh and sigmoid

1. The gradient of sigmoid activation function is a number between 0 and 1. Thus, in a deep neural networks with many layers the gradient in the backpropagation update will be the product of many numbers between 0 and 1 which leads to the vanish gradient problem (i.e. gradient becoming close to zero).
2. The sigmoid function ranges between 0 and 1, while the tanh function ranges between -1 and 1. We might prefer tanh because tanh is symmetric around 0.
3. Because the derivative of tanh around zero (where the normalized input data would be) is larger than the derivative of the sigmoid.
4. $tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}} = (1-e^{-2x})\frac{1}{1+e^{-2x}} = (1-e^{-2x})\sigma(2x) = \sigma(2x) - \sigma(2x)(1+e^{-2x}-1) = \sigma(2x) - \sigma(2x)\left(\frac{1}{\sigma(2x)} - 1\right) = 2\sigma(2x) - 1$

Q2.1.1: 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?

Because when initialize a network with all zeros different units (i.e. neurons) will have the same update in the gradient during the backpropagation. Consequently, the weights and biases associated with those distinct units will be the same, and thus reducing considerably the representation power of the neural network model.

Q2.1.2: Implement function initialize_weights in python/nn.py. Include your code in the writeup.

Figure 1 shows the code snippet of the function initialize_weights.

```python
def initialize_weights(in_size,out_size,params,name=''):
    upper_limit = np.sqrt(6/(in_size+out_size))

    W = np.random.uniform(low=-upper_limit, high=upper_limit, size=(in_size, out_size))
    b = np.zeros(out_size)

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

Figure 1: Code snippet of the function initialize_weights

Q.2.1.3: Why do we initialize with random numbers? Why do we scale the initialization depending on layer size?

We initialize a neural network with random numbers to break the symmetry between distinct units and avoid that distinct units have the same (or similar) update in the gradient during backpropagation and consequently reducing the representation power of the neural network.

The standard initialization (Equation 1 in the paper) causes the variance of the back-propagated gradient to be dependent on the layer. In particular, the variance of the back-propagated gradient gets smaller as it is propagated downwards, and consequently there is more risk of having vanishing gradient problems. However, the authors found that such decreasing back-propagated gradients is not observed when scaling the initialization according to the layer size.

Q2.2.1: In python/nn.py implement function sigmoid. Include your code in the writeup.

Figure 2 shows the code snippet of the function sigmoid.

```
def sigmoid(x):
    res = 1/(1+np.exp(-x))

    return res
```

Figure 2: Code snippet of the function sigmoid

Q2.2.2: In python/nn.py implement the softmax function. Include your code in the writeup.

Figure 3 shows the code snippet of the function softmax.

```
def softmax(x):

    c = -np.max(x, axis = 1)
    c_matrix = np.repeat(c, x.shape[1]).reshape(x.shape)
    exp_x_plus_c = np.exp(x + c_matrix)

    denominator = np.repeat(np.sum(exp_x_plus_c, axis = 1),
                            x.shape[1]).reshape(x.shape)

    res = exp_x_plus_c/denominator

    return res
```

Figure 3: Code snippet of the function softmax

Q.2.2.3: In python/nn.py, implement the function compute_loss_and_acc. Include your code in the writeup.

Figure 4 shows the code snippet of the function compute_loss_and_acc.

```python
def compute_loss_and_acc(y, probs):

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

    n_correct = np.sum(np.argmax(y, axis = 1) == np.argmax(probs, axis = 1))
    acc = n_correct /y.shape[0]

    return loss, acc
```

Figure 4: Code snippet of the function compute_loss_and_acc

Q2.3: In python/nn.py, implement function backwards. Include your code in the writeup.

Figure 5 shows the code snippet of the function backwards.

```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

    dE_dh = delta * activation_deriv(post_act)
    grad_X = np.matmul(dE_dh, W.T)
    grad_W = np.matmul(X.T, dE_dh)
    grad_b = dE_dh.sum(axis=0)

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

Figure 5: Code snippet of the function backwards

Q2.4: In python/nn.py implement function get_random_batches. In python/run_q2.py, write a training loop that iterates over the batches, does foward and backward propagation, and applies gradient update. Include your code in the writeup.

Figure 6 shows the code snippet of the function get_random_batches.

```python
def get_random_batches(x,y,batch_size):
    batches = []

    nbatches = int(x.shape[0]/batch_size)
    batch_index = np.repeat(range(nbatches),batch_size)

    rd = x.shape[0] % batch_size
    if rd > 0:
        batch_index = np.hstack((batch_index,np.repeat(nbatches, rd)))
        nbatches += 1

    np.random.shuffle(batch_index)

    for i in range(nbatches):
        index = (batch_index == i)
        batches.append((x[index,:], y[index,:]))

    return batches
```

Figure 6: Code snippet of the function get_random_batches

Figure 7 shows the code snippet of the training loop (in python/run_q2.py) that iterates over the batches, does foward and backward propagation, and applies gradient update.

```python
# Q 2.4
batches = get_random_batches(x,y,5)
# print batch sizes
print([_[0].shape[0] for _ in batches])
batch_num = len(batches)

# WRITE A TRAINING LOOP HERE
max_iters = 500
learning_rate = 1e-3
# with default settings, you should get loss < 35 and accuracy > 75%
for itr in range(max_iters):
    total_loss = 0
    avg_acc = 0
    for xb,yb in batches:

        # forward
        h1 = forward(xb, params,'layer1')
        probs = forward(h1,params,'output',softmax)

        # loss
        # be sure to add loss and accuracy to epoch totals
        loss, acc = compute_loss_and_acc(yb, probs)
        total_loss += loss
        avg_acc += acc

        # backward
        delta1 = probs - yb
        delta2 = backwards(delta1,params,'output',linear_deriv)
        _ = backwards(delta2,params,'layer1',sigmoid_deriv)

        # apply gradient
        # gradients should be summed over batch samples
        params['Woutput'] -= learning_rate* params['grad_Woutput']
        params['boutput'] -= learning_rate* params['grad_boutput']
        params['Wlayer1'] -= learning_rate* params['grad_Wlayer1']
        params['blayer1'] -= learning_rate* params['grad_blayer1']


    avg_acc /=batch_num


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

Figure 7: Code snippet of the training loop (in python/run_q2.py) that iterates over the batches, does foward and backward propagation, and applies gradient update.

Q2.5: In python/nn.py implement a numerical gradient checker. Include your code in the writeup.

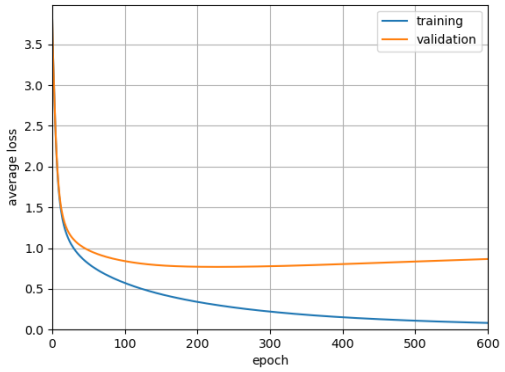
Figure 8 shows the code snippet of the numerical gradient checker (in python/run_q2.py).

```python
# 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 k in ['Wlayer1', 'Woutput']:
        for i in range(params[k].shape[0]):
            for j in range(params[k].shape[1]):
                value_backup = params[k][i,j]
                #   add epsilon
                params[k][i,j] += eps

                #   run the network
                h1 = forward(x, params,'layer1')
                probs = forward(h1,params,'output',softmax)

                #   get the loss
                loss_plus_eps, _ = compute_loss_and_acc(y, probs)

                #   subtract 2*epsilon
                params[k][i,j] -= 2*eps

                #   run the network
                h1 = forward(x, params,'layer1')
                probs = forward(h1,params,'output',softmax)

                #   get the loss
                loss_minus_eps, _ = compute_loss_and_acc(y, probs)

                #   restore the original parameter value
                params[k][i,j] = value_backup

                #   compute derivative with central diffs
                num_gradient = (loss_plus_eps - loss_minus_eps)/(2*eps)


                if k == 'Wlayer1':
                    params['grad_Wlayer1'][i,j] = num_gradient
                else:
                    params['grad_Woutput'][i,j] = num_gradient
    if k in ['blayer1', 'boutput']:

        for i in range(params[k].shape[0]):
            value_backup = params[k][i]

            #   add epsilon
            params[k][i] += eps

            #   run the network
            h1 = forward(x, params,'layer1')
            probs = forward(h1,params,'output',softmax)

            #   get the loss
            loss_plus_eps, _ = compute_loss_and_acc(y, probs)

            #   subtract 2*epsilon
            params[k][i] -= 2*eps

            #   run the network
            h1 = forward(x, params,'layer1')
            probs = forward(h1,params,'output',softmax)

            #   get the loss
            loss_minus_eps, _ = compute_loss_and_acc(y, probs)

            #   restore the original parameter value
            params[k][i] = value_backup

            num_gradient = (loss_plus_eps - loss_minus_eps)/(2*eps)
            #   compute derivative with central diffs


            if k == 'blayer1':
                params['grad_blayer1'][i] = num_gradient
            else:
                params['grad_boutput'][i] = num_gradient
total_error = 0
for k in params.keys():
    if 'grad_' in k:
        # relative error
        den = np.maximum(np.abs(params[k]),np.abs(params_orig[k]))
        err = np.abs(params[k] - params_orig[k])/den
        err = err.sum()
        print('{} {:.2e}'.format(k, err))
        total_error += err
# should be less than 1e-4
print('total {:.2e}'.format(total_error))
```

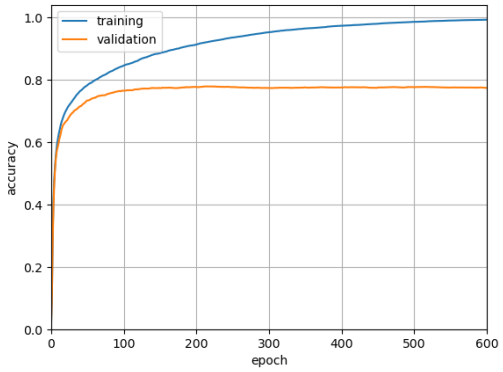
Figure 8: Code snippet of the numerical gradient checker (in python/run_q2.py).

Q3.1: 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.

I trained a neural network with a single hidden layer (64 hidden units) for 600 epochs. I used a batch size of 16 and learning rate equal to 0.001. The final validation accuracy was 77.4%, and the final test accuracy was 78.3%. Figures 9 (a)-(b) show respectively the loss and accuracy over the epochs, both for training and validation sets.



(a) Loss by epoch step



(b) Accuracy by epoch step

Figure 9: Loss and accuracy over the epochs for training (blue lines) and validation (orange lines) datasets

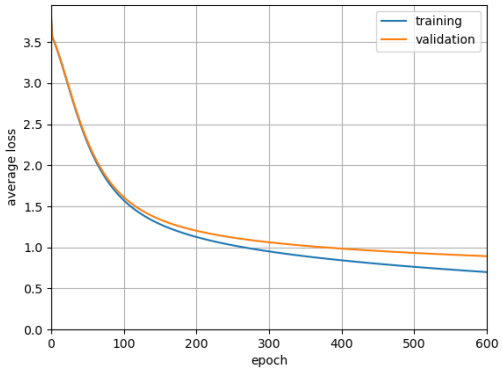
Q3.2: Use the script to train three networks, one with your tuned learning rate, one with 10 times that learning rate, and one with one tenth that learning rate. Include all six plots in your writeup. Comment on how the learning rates affect the training, and report the final accuracy of the best network on the test set.

Table 1 shows the validation and test accuracies after training the same neural network architecture for 600 epochs using distinct learning rates: 0.0001, 0.001 and 0.01. The best test accuracy was observed when using the learning rate 0.001, while the worst test accuracy was observed when using tthe learning rate 0.0001.

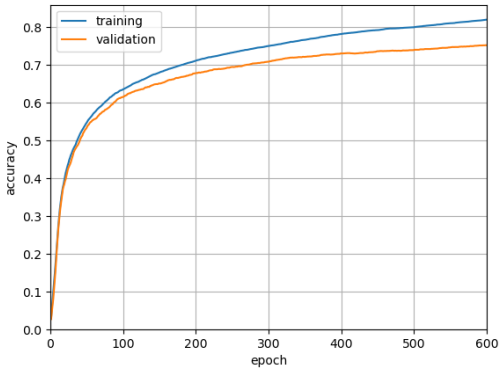
Table 1: Validation and test accuracies after training same neural network architecture for 600 epochs using distinct learning rates: 0.0001, 0.001 and 0.01

	Learning Rate		
	0.0001	0.001	0.01
Validation	75.3%	77.4%	% 75.3%
Test	75.4%	78.3%	% 75.7%

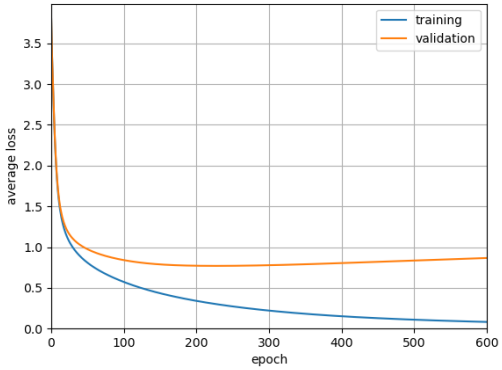
Figures 10 (a)-(f) show loss and accuracy over the epochs for training (blue lines) and validation (orange lines) datasets for three different learning rates: 0.0001, 0.001 and 0.01. The higher the training rate the more rapidly was the drop in the loss in the earlier epochs (both training and validation). Similarly, the higher the learning rate the faster was the increase in accuracy (both training and validation). However, as seen in Figure 10(e), if learning rates are too large then the gradient descent algorithm can jump over areas with lower loss function.



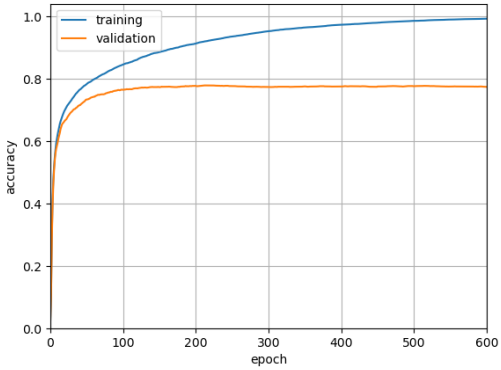
(a) Loss by epoch step - learning rate: 0.0001



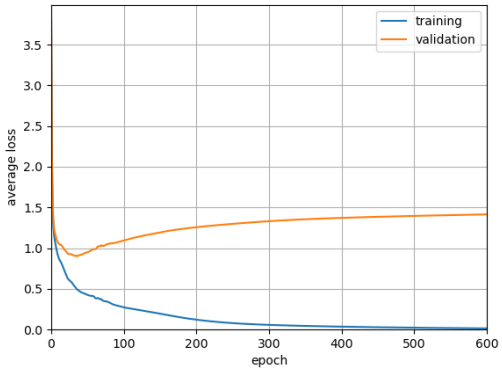
(b) Accuracy by epoch step - learning rate: 0.0001



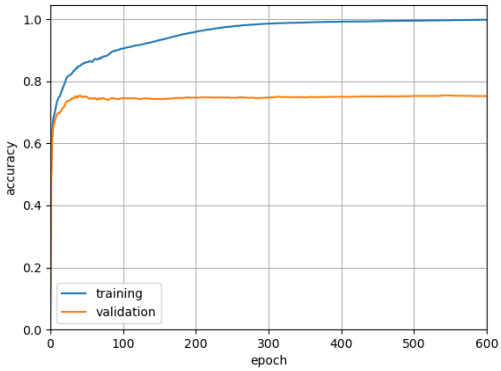
(c) Loss by epoch step - learning rate: 0.001



(d) Accuracy by epoch step - learning rate: 0.001



(e) Loss by epoch step - learning rate: 0.01

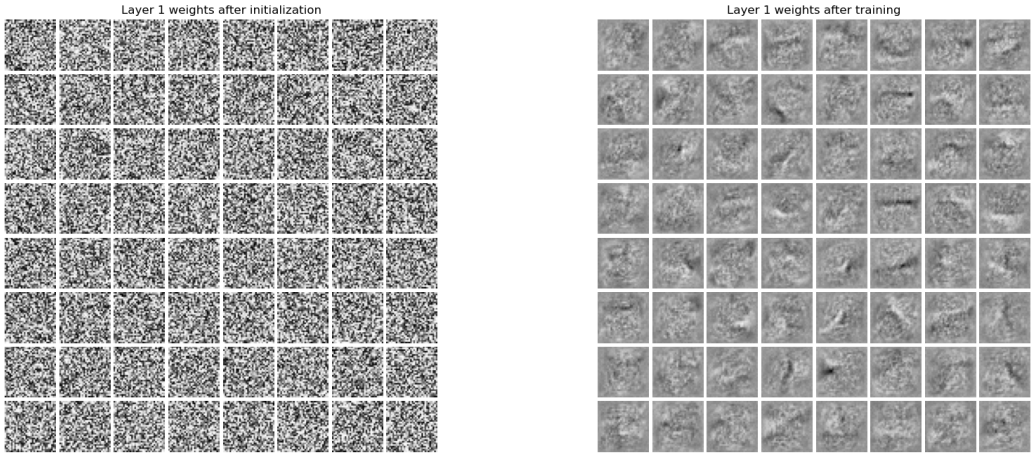


(e) Accuracy by epoch step - learning rate: 0.01

Figure 10: Loss and accuracy over the epochs for training (blue lines) and validation (orange lines) datasets for three different learning rates: 0.00001 (a, b), 0.0001 (c, d) and 0.001 (e, f)

Q3.3: The script will visualize the first layer weights as 64 32×32 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?

Figures 11(a)-(b) show the first layer weights after initialization but before training (a), and the first layer weights after training. Right after initialization the weights look random, after training there are patterns in the weights that resemble digits and letters.



(a) After initialization but before training **(b)** After training

Figure 11: First layer weights after initialization but before training (a) and after training (b)

Q.3.4: 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.

Figure 12 shows the confusion matrix of the test data for the best model trained. Notice the there few spots off the diagonal that are not zero (i.e. not dark blue), which indicates the most misclassified pairs. In particular, the most commonly confused pairs observed were: (O, 0), (5, S) and (2, Z). This makes sense, as these characters do look alike and depending on the handwriting even humans sometimes misrecongnize them.

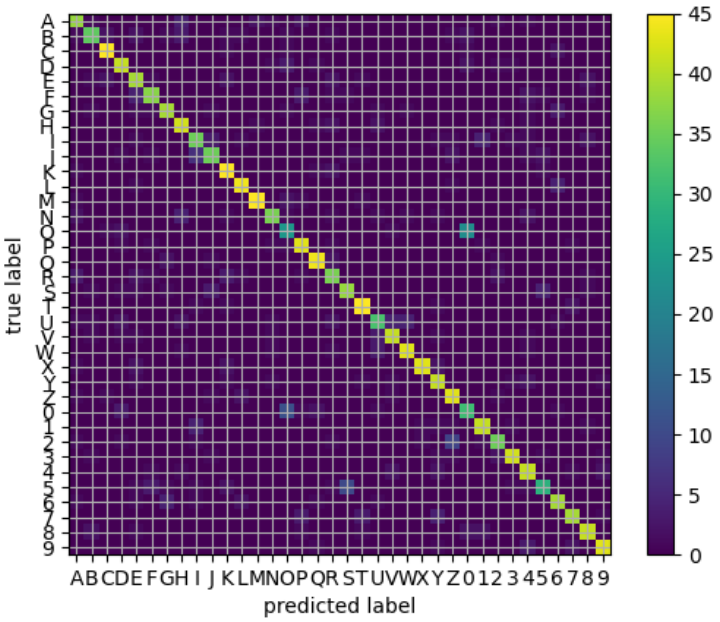


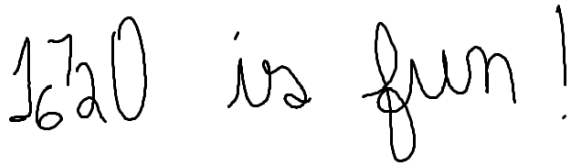
Figure 12: Confusion matrix of the test data

Q4.1: 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).

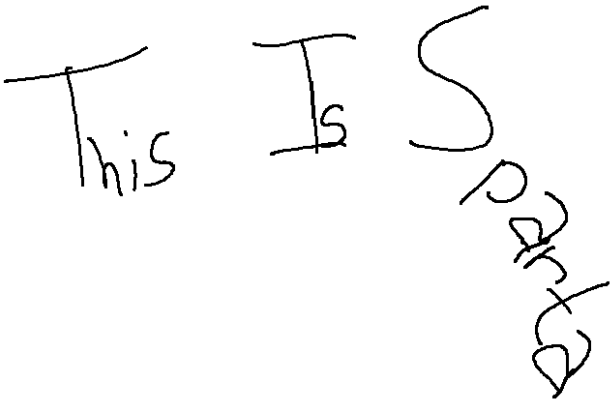
The extracting text method described in the handout makes the following assumptions:

- characters from the same line are aligned horizontally
- there is enough white space between characters (both vertically and horizontally)
- characters in the same row have similar size
- characters are similar to the ones in the training dataset

Figures 13(a)-(b) show examples where I expect the character detection algorithm to fail.



(a) Example 1



(a) Example 2

Figure 13: Example images where character detection algorithm is expected to fail

Q4.2: In n python/q4.py, implement the function findLetters. Include your code in the writeup.

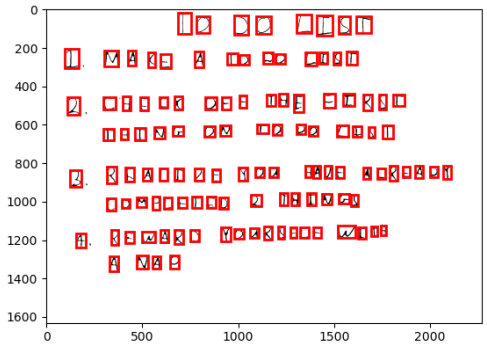
Figure 14 shows the code snippet of the function findLetters.

```python
def findLetters(image):
    bboxes = []
    bw = None

    #estimate noise
    estNoise = skimage.restoration.estimate_sigma(image = image,
                                                  average_sigmas = True,
                                                  multichannel = True,
                                                  channel_axis = 2)
    #denoise
    denoiseImg = skimage.restoration.denoise_wavelet(image = image,
                                                     sigma = estNoise,
                                                     channel_axis = 2)
    #grayscale
    gs_img = skimage.color.rgb2gray(denoiseImg)

    #threshold
    thresh = skimage.filters.threshold_otsu(gs_img)

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

    cleared_image = skimage.segmentation.clear_border(bw)

    #label
    label_image = skimage.measure.label(cleared_image)

    #distribution of region areas
    region_areas = []
    for region in skimage.measure.regionprops(label_image):
        region_areas.append(region.area)

    #get a threshold for "small" boxes
    avg_region = np.mean(np.array(region_areas))
    std_region = np.std(np.array(region_areas))
    small_area_threshold = avg_region - 2.5*std_region

    for region in skimage.measure.regionprops(label_image):
        # take regions with large enough areas
        if region.area >= small_area_threshold:
            bboxes.append(region.bbox)

    return bboxes, bw.astype(float)
```

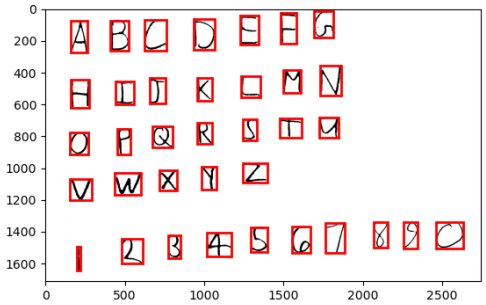
Figure 14: Code snippet of the function findLetters.

Q4.3: 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.

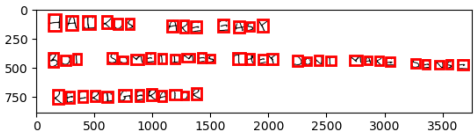
Figures 15(a)-(d) show the located bounding boxes for all sample images. All characters were captured properly, except the periods in the figure 01_list.jpg.



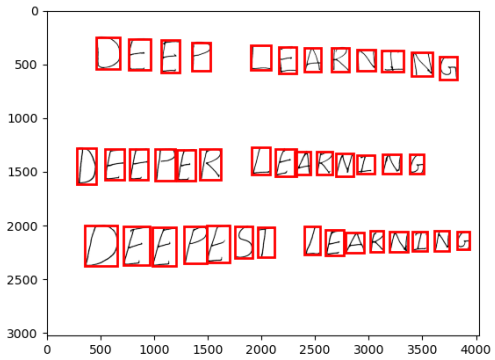
(a) Image 01_list.jpg



(b) Image 02_letters.jpg



(c) Image 03_haiku.jpg



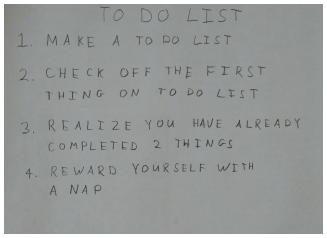
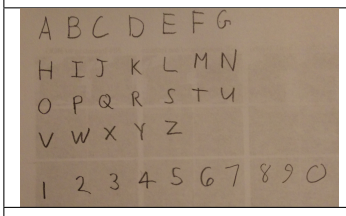
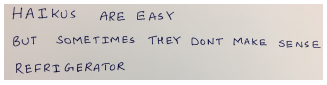
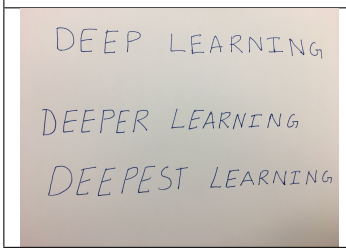
(d) Image 04_deep.jpg

Figure 15: Bounding boxes using function findLetters

Q.4: Run your run_q4.py 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.

Table 2 shows the characters extracted for each sample image. All had accuracy greater than 80%.

Table 2: Character extracting results

Image	Text Extracted	Accuracy
	T0 D0 LIST I HAKE A TO DO 6ZST 2 CHKCK OFF YHE FZRST TH2HE OH TO DO L8ST 3 RKALZZE YOU HAVE ALREADY COMPLETDD Z YHINOS 4 REWARD YOURSELF WIYH A NAP	81.7%
	A B C D E F G H I J K L M N 0 P Q R S T W V W X Y Z 3 Z 3 Q 5 G 7 8 9 O	80.6%
	HAIKUS ARE EASY BUT SOMETIMES TREY D0NT MAKE SEMSE REFRIGERAT0R	92.6%
	DEEF LEARMING DHHPER LEARHING DFEFFST LEARNING	80.5%

Q5.1.1: Implement the autoencoder specified in the handout. Include your code in the writeup.

Figure 16 shows the code snippet of the autoencoder specified in the handout.

```
# Q5.1 & Q5.2
# initialize layers here
initialize_weights(train_x.shape[1], 32, params, "layer1")
initialize_weights(32, 32, params, "layer2")
initialize_weights(32, 32, params, "layer3")
initialize_weights(32, train_x.shape[1], params, "output")

for l in ['layer1', 'layer2', 'layer3', 'output']:
    params['MW_'+l] = np.zeros(params['W'+l].shape)
    params['Mb_'+l] = np.zeros(params['b'+l].shape)

# should look like your previous training loops
losses = []
for itr in range(max_iters):
    total_loss = 0
    for xb,_ in batches:
        # forward
        h1 = forward(xb, params,'layer1', relu)
        h2 = forward(h1, params,'layer2', relu)
        h3 = forward(h2, params,'layer3', relu)
        output_pred = forward(h3,params,'output', sigmoid)

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

        # backward
        delta1 = output_pred - xb
        delta2 = backwards(delta1,params,'output',linear_deriv)
        delta3 = backwards(delta2,params,'layer3',relu_deriv)
        delta4 = backwards(delta3,params,'layer2',relu_deriv)
        _ = backwards(delta4,params,'layer1',relu_deriv)

        # apply gradient
        for l in ['layer1', 'layer2', 'layer3', 'output']:
            params['MW_'+l] = 0.9*params['MW_'+l] - learning_rate*params['grad_W'+l]
            params['W'+l] += params['MW_'+l]
            params['Mb_'+l] = 0.9*params['Mb_'+l] - learning_rate*params['grad_b'+l]
            params['b'+l] += params['Mb_'+l]

    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
```

Figure 16: Code snippet of the autoencoder specified in the handout.

Q5.1.2: Implement momentum. Include your code in the writeup.

Figure 17 shows the code snippet of the autoencoder (with momentum) specified in the handout.

```
# Q5.1 & Q5.2
# initialize layers here
initialize_weights(train_x.shape[1], 32, params, "layer1")
initialize_weights(32, 32, params, "layer2")
initialize_weights(32, 32, params, "layer3")
initialize_weights(32, train_x.shape[1], params, "output")

for l in ['layer1', 'layer2', 'layer3', 'output']:
    params['MW_'+l] = np.zeros(params['W'+l].shape)
    params['Mb_'+l] = np.zeros(params['b'+l].shape)

# should look like your previous training loops
losses = []
for itr in range(max_iters):
    total_loss = 0
    for xb,_ in batches:
        # forward
        h1 = forward(xb, params,'layer1', relu)
        h2 = forward(h1, params,'layer2', relu)
        h3 = forward(h2, params,'layer3', relu)
        output_pred = forward(h3,params,'output', sigmoid)

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

        # backward
        delta1 = output_pred - xb
        delta2 = backwards(delta1,params,'output',linear_deriv)
        delta3 = backwards(delta2,params,'layer3',relu_deriv)
        delta4 = backwards(delta3,params,'layer2',relu_deriv)
        _ = backwards(delta4,params,'layer1',relu_deriv)

        # apply gradient
        for l in ['layer1', 'layer2', 'layer3', 'output']:
            params['MW_'+l] = 0.9*params['MW_'+l] - learning_rate*params['grad_W'+l]
            params['W'+l] += params['MW_'+l]
            params['Mb_'+l] = 0.9*params['Mb_'+l] - learning_rate*params['grad_b'+l]
            params['b'+l] += params['Mb_'+l]

    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
```

Figure 17: Code snippet of the autoencoder (with momentum) specified in the handout.

Q5.2: 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?

Figure 18 shows the encoder training loss curve. The averge loss drops fairly quickly in the first 5 epochs and then gradually declines afterwards.

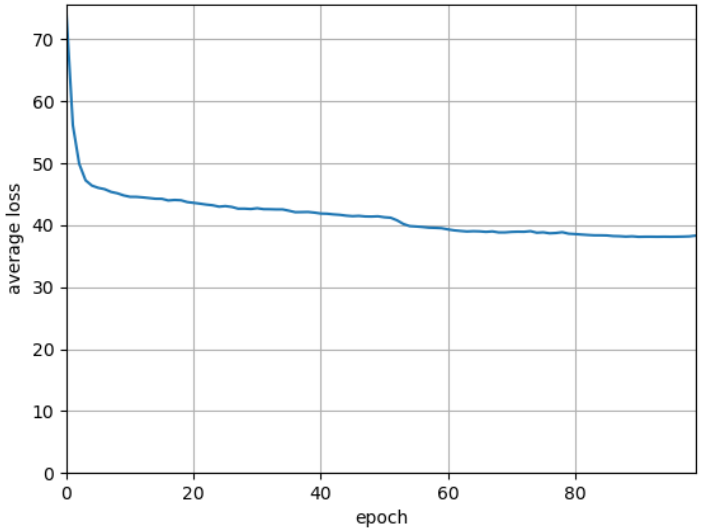


Figure 18: Training loss curve

5.3.1: 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?

Figure 19 shows 10 validations images (2 for each class) for the classes A, B, C, 1 and 2. The columns 1 and 3 represent the original images, while columns 2 and 4 are the corresponding reconstructed versions. The reconstructed versions are blurry, and also seems to misrepresent some characters (e.g. reconstruction version of 2 looks like a Z).

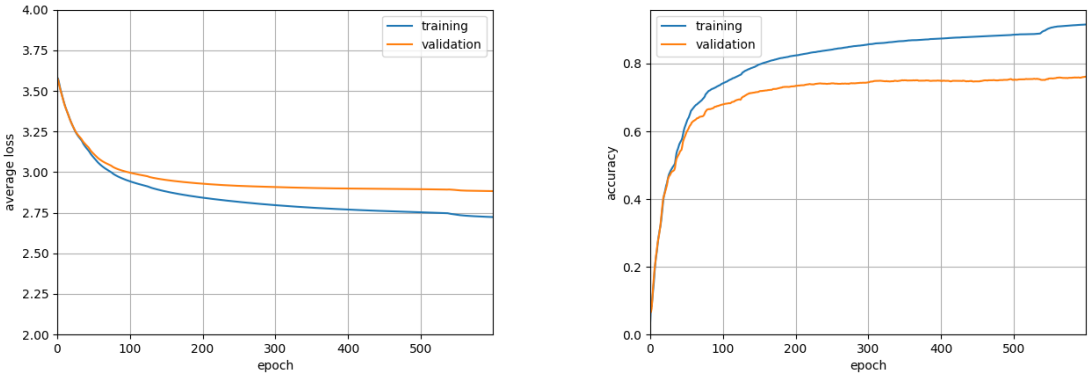
Figure 19: Auto encoder - Original (columns 1 and 3) versus reconstructed (columns 2 and 4) images

Q5.3.2: Report the average PSNR you get from the autoencoder across all images in the validation set

The average PSNR was 14.55.

Q6.1.1: Re-write and re-train your fully-connected network on the included NIST36 in PyTorch. Plot training accuracy and loss over time.

Figures 20 (a)-(b) show respectively the loss and accuracy over the epochs using PyTorch, both for training and validation sets.



(a) Loss by epoch step **(b)** Accuracy by epoch step

Figure 20: Loss and accuracy over the epochs for training (blue lines) and validation (orange lines) datasets - using PyTorch

Figure 21 shows the code snippet of the fully-connected network implemented using PyTorch.

```
#define model
hidden_size = 64
model = torch.nn.Sequential(nn.Linear(train_x.shape[1], hidden_size),
                            nn.Sigmoid(),
                            nn.Linear(hidden_size, train_y.shape[1]),
                            nn.Softmax(1))
```

Figure 21: Code snippet of the fully-connected network implemented using PyTorch

Q6.1.2: Train a convolutional neural network with PyTorch on the included NIST36 dataset. Compare its performance with the previous fully-connected network.

I implemented a convolutional neural network fairly similar to the Lenet model (see Figure 22). The validation accuracy of the fully-connected network was 77.4% while for the CNN model was 84.5%.

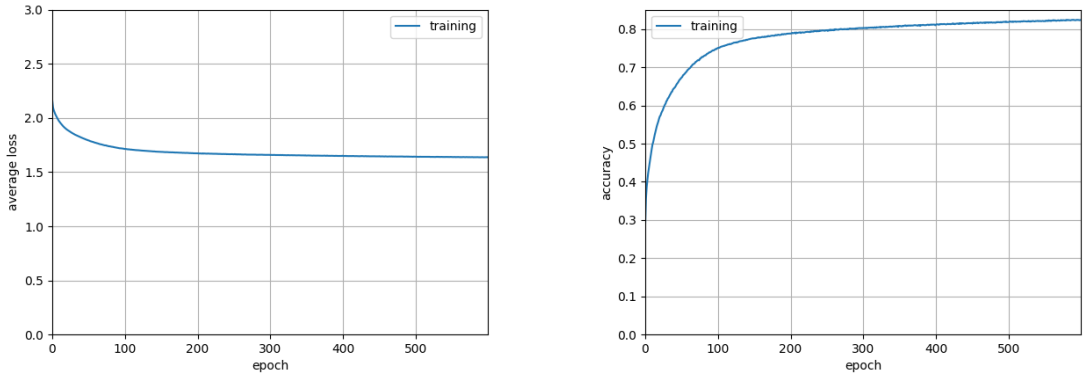
Figure 22 shows the code snippet of the CNN implemented using PyTorch.

```
#define model
model = torch.nn.Sequential(nn.Conv2d(in_channels = 1, out_channels = 6, kernel_size = 5, stride = 1, padding = 0),
                            nn.ReLU(),
                            nn.MaxPool2d(kernel_size = 2, stride = 2, padding = 0),
                            nn.Conv2d(in_channels = 6, out_channels = 16, kernel_size = 5, stride = 1, padding = 0),
                            nn.ReLU(),
                            nn.MaxPool2d(kernel_size = 2, stride = 2, padding = 0),
                            nn.Flatten(),
                            nn.Linear(400, 120),
                            nn.ReLU(),
                            nn.Linear(120, 84),
                            nn.ReLU(),
                            nn.Linear(84, 36),
                            nn.Softmax(1))
```

Figure 22: Code snippet of the CNN implemented using PyTorch - NIST36 dataset

Q6.1.3: Train a convolutional neural network with PyTorch on CIFAR-10 (torchvision.datasets.CIFAR10). Plot training accuracy and loss over time.

I implemented a convolutional neural network fairly similar to the Lenet model (see Figure 24). Figures 23 (a)-(b) show respectively the training loss and accuracy over the epochs for this model.



(a) Loss by epoch step **(b)** Accuracy by epoch step

Figure 23: Training loss and accuracy over the epochs - CIFAR-10 dataset

Figure 24 shows the code snippet of the CNN implemented using PyTorch.

```
model = torch.nn.Sequential(nn.Conv2d(in_channels = 3, out_channels = 6, kernel_size = 5, stride = 1, padding = 0),
                            nn.ReLU(),
                            nn.MaxPool2d(kernel_size = 2, stride = 2, padding = 0),
                            nn.Conv2d(in_channels = 6, out_channels = 16, kernel_size = 5, stride = 1, padding = 0),
                            nn.ReLU(),
                            nn.MaxPool2d(kernel_size = 2, stride = 2, padding = 0),
                            nn.Flatten(),
                            nn.Linear(400, 120),
                            nn.ReLU(),
                            nn.Linear(120, 84),
                            nn.ReLU(),
                            nn.Linear(84, 10),
                            nn.Softmax(1))
```

Figure 24: Code snippet of the CNN implemented using PyTorch - CIFAR-10 dataset

Q6.1.4: 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.

Using a fairly similar LeNet model (see Figure 25), I obtained 45.6% test accuracy. In HW1, the best model I found had test accuracy of 62%. The CNN model tried here was a "quick-and-dirt" approach. I believe it is possible to tweak the model (e.g. add more convolutional layers, use higher resolution) to improve test accuracy.

Figure 25 shows the code snippet of the CNN implemented using PyTorch.

```
#define model
model = torch.nn.Sequential(nn.Conv2d(in_channels = 3, out_channels = 6, kernel_size = 5, stride = 1, padding = 0),
                            nn.BatchNorm2d(6),
                            nn.ReLU(),
                            nn.Dropout(0.5),
                            nn.MaxPool2d(kernel_size = 2, stride = 2, padding = 0),
                            nn.Conv2d(in_channels = 6, out_channels = 16, kernel_size = 5, stride = 1, padding = 0),
                            nn.BatchNorm2d(16),
                            nn.ReLU(),
                            nn.Dropout(0.5),
                            nn.MaxPool2d(kernel_size = 2, stride = 2, padding = 0),
                            nn.Flatten(),
                            nn.Linear(400, 120),
                            nn.ReLU(),
                            nn.Linear(120, 84),
                            nn.ReLU(),
                            nn.Linear(84, 8),
                            nn.Softmax(1))
```

Figure 25: Code snippet of the CNN implemented using PyTorch - SUN dataset

Q6.2: Fine tuning

I skipped this question.

Q6.3: Neural networks in the RealWorld

I downloaded the ResNet50 pretrained image classification model from torchvision, and the 64×64 ImageNet validation data. I chose the category 'Rottweiler'. There were 50 images in the validation set with images of rottweilers. Out of these 50, the Resnet50 model predicted correctly 21 (42.0%) of them. This model misclassified 23 (46.0%) of the 'Rottweiler' images as 'Black-and-tan coonhound' images, which is understandable since both dog breeds have similar colors, and the image resolution is low. When considering top-5 prediction, the ResNet50 accuracy of rottweiler images was 86.0%.

I chose a video of a rotweiller in a cage, which had 270 frames in total and the rotweiller was present in every single frame. Figures 26 (a)-(d) shows four frames of the video.

(a) Frame 0

(b) Frame 90

(c) Frame 180

(d) Frame 270

Figure 26: Frames 0, 90, 180 and 270 of the rotweiller video

ResNet50 did not predict rotweiller for any of the 270 frames. However, interestingly enough, the three most commons predicted labels were: worm fence (39%), prison (29.0%) and steel arch bridge (13.7%). Clearly, the pattern of the dog's cage hugely influences the label prediction. I also looked the top-5 accuracy, but still none of the frames were classified as rotweiller. This shows how important is to have a diversified dataset. One way of improving the model is to do some additional training with data that includes dogs in different contexts (e.g. cages). This can be done with original images, but also with data augmentation transformations.