16-820 Advanced Computer Vision: Homework 4 (Fall 2024)
Deep Learning

Ryan Wu (Andrew ID: weihuanw)

Wednesday November 6, 2024

Q1.1
Prove that softmax is invariant to translation, that is, softmax$(x) =$ **softmax**$(x + c), \quad \forall c \in \mathbb{R}$.

The softmax function is defined as:

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

With the given added translation constant $c \in \mathbb{R}$, the softmax function can be define as:

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

e^c can be factored and canceled out, which proves that softmax is invariant to translation:

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

Choosing $c = -\max x_i$ is beneficial for numerical stability in the softmax function. It makes sure that the exponential term is bounded between [0, 1], which prevents overflow issues.

Q1.2
Softmax can be written as a three-step process, 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.

The range of each element is between [0, 1]. The sum over all elements is equal to 1.

One could say that "softmax takes an arbitrary real-valued vector x and turns it into a probability distribution.

First step, $s_i = e^{x_i}$, performs exponentiation on input x_i, which ensures all s_i are positive for probability interpretation. Second step, $S = \sum s_i$, performs summation of all the exponentiated s_i, which serves as a normalization factor for the overall probability distribution. Third step, softmax$(x_i) = \frac{1}{S} s_i$, performs the normalization by dividing s_i with S, ensuring that the resulting values are in the range [0, 1] and sums to 1, thus forming a valid probability distribution.

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

The linear regression model is defined as:

$$Y = \beta X + b \tag{1}$$

A multi-layer neural network's structure is defined as:

$$a_i = W_i x + b_i \tag{2}$$

The hidden layers a_i without a non-linear activation function are defined as:

$$a_{i-1} = W_{i-1} a_i + b_{i-1} \tag{3}$$

Substituting (2) into (3), we get:

$$a_{i-1} = W_{i-1}(W_i x + b_i) + b_{i-1} \implies a_{i-1} = W_{i-1} W_i x + W_{i-1} b_i + b_{i-1} \tag{4}$$

We can introduce the weight matrix $\beta_{conbined}$ and the bias vector $b_{combined}$ into (4):

$$\beta_{conbined} = W_{i-1} W_i, \quad b_{combined} = W_{i-1} b_i + b_{i-1}$$

$$a_{i-1} = \beta_{conbined} x + b_{combined} \tag{5}$$

The network's output a_{i-1} (5) is equivalent to linear regression in (1).

Q1.4
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).

The derivative of $\sigma(x)$ is:

$$\frac{d\sigma(x)}{dx} = \frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right) = \frac{0 \cdot (1+e^{-x}) - 1 \cdot (-e^{-x})}{(1+e^{-x})^2} = \frac{e^{-x}}{(1+e^{-x})^2}$$

Since

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

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

The derivative can be expressed in terms of $\sigma(x)$:

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

Q1.5
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 to y, show how to get $\frac{\partial J}{\partial W}$, $\frac{\partial J}{\partial x}$, and $\frac{\partial J}{\partial b}$. Be sure to compute the derivatives with scalars and then re-form the matrix form afterward. Here are some notational suggestions: $\frac{\partial J}{\partial y} = \delta \in \mathbb{R}^{k\times 1}$, $W \in \mathbb{R}^{k\times d}$, $x \in \mathbb{R}^{d\times 1}$, $b \in \mathbb{R}^{k\times 1}$.

$[\frac{\partial J}{\partial W}]$

The chain rule states that:

$$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial W}$$

Since $y_i = \sum_{j=1}^{d} x_j W_{ij} + b_i$, $\frac{\partial y_i}{\partial W_{ij}}$ becomes:

$$\frac{\partial y_i}{\partial W_{ij}} = x_j \implies \frac{\partial y}{\partial W} = x^T$$

$\frac{\partial J}{\partial W}$ becomes:

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

$[\frac{\partial J}{\partial x}]$

The chain rule states that:

$$\frac{\partial J}{\partial x} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial x}$$

Since $y_i = \sum_{j=1}^{d} x_j W_{ij} + b_i$, $\frac{\partial y_i}{\partial x_j}$ becomes:

$$\frac{\partial y_i}{\partial x_j} = W_{ij} \implies \frac{\partial y}{\partial x} = W^T$$

$\frac{\partial J}{\partial x}$ becomes:

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

$[\frac{\partial J}{\partial b}]$

The chain rule states that:

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial b}$$

Since $y = Wx + b$, $\frac{\partial y}{\partial b}$ becomes:

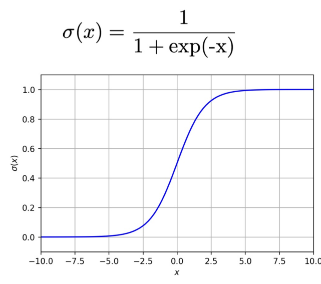
$$\frac{\partial y}{\partial b} = I$$

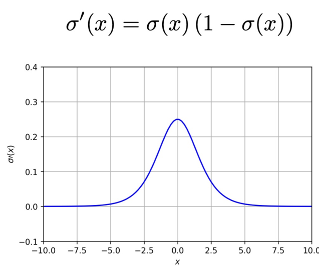
$\frac{\partial J}{\partial b}$ becomes:

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y} I = \delta$$

Q1.6
1. Consider the sigmoid activation function in 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 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 function. Show how $\tanh(x)$ can be written in terms of $\sigma(x)$.

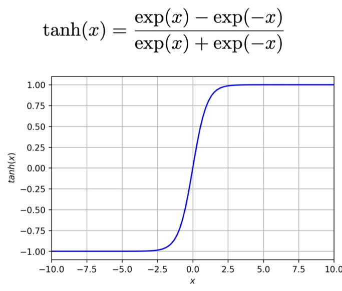
1. From the sigmoid activation function's gradient plot below (right), we can observe that the maximin gradient values occurs at $\sigma'(x = 0) = 0.25$ and all other gradient values of x approaches 0, which can lead to a vanishing gradient problem.

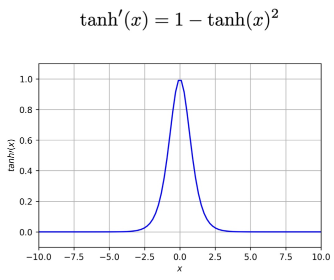




2. The output range for tanh is [-1, 1] . The output range for sigmoid is [0, 1]. One might prefer tanh since the function outputs values centered around zero. which may have better convergence during training and less of a vanishing gradient problem.

3. From the tanh activation function's gradient plot below (right), we can observe that the maximin gradient values occurs at $tanh'(x = 0) = 1$, which is larger than the sigmoid function (0.25), leading to less of a vanishing problem.





4. Since

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

tanh(x) can be expressed as:

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

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?

A zero-initialized network will lead to every neuron in each layer learning the same features and producing identical outputs. This occurs because, with zero weights and biases, the neurons are symmetric and essentially no learning is being done.

Q2.1.2
In `python/nn.py`, implement a function to initialize one layer's weights with Xavier initialization [1], where Var$[w] = \frac{2}{n_{\text{in}}+n_{\text{out}}}$, 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
def initialize_weights(in_size, out_size, params, name=""):
    W, b = None, None

    # Xavier initialization
    limit = np.sqrt(6) / np.sqrt(in_size + out_size)
    W = np.random.uniform(-limit, limit, (in_size, out_size))

    # bias initialization (1D array)
    b = np.zeros(out_size)

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

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

We initialize with random weights to ensure every neuron in each layer learns different features, which is critical for meaningful model training. We scale the initialization depending on layer size to ensure proper gradient propagation throughout the network.

Q2.2.1

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

```python
def sigmoid(x):
    res = None
    X = np.clip(x, -500, 500) # prevent overflow warning

    res = 1 / (1 + np.exp(-x))

    return res
```

```python
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]

    # compute the pre-activation
    pre_act = np.dot(X, W) + b

    # compute the post-activation
    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
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
def softmax(x):
    res = None

    # subtract the max for numerical stability
    x = x - np.max(x, axis=1, keepdims=True)

    # compute the softmax
    res = np.exp(x) / np.sum(np.exp(x), axis=1, keepdims = True)

    return res
```

Q2.2.3

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:

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

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

```python
def compute_loss_and_acc(y, probs):
    loss, acc = None, None

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

    # compute the accuracy
    acc = np.sum(np.argmax(y, axis=1) == np.argmax(probs, axis=1)) / y.shape[0]

    return loss, acc
```

Q2.3

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
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

    # compute the derivative (delta)
    delta = delta * activation_deriv(post_act)

    # compute the gradient of W
    grad_W = np.dot(X.T, delta)

    # compute the gradient of b
    grad_b = np.sum(delta, axis=0)

    # compute the gradient of X
    grad_X = np.dot(delta, W.T)

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

Q2.4
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 ##############################
def get_random_batches(x, y, batch_size):
    batches = []

    # shuffle the data
    num_data = x.shape[0]
    indices = np.arange(num_data)
    np.random.shuffle(indices)

    # split the data into batches
    for i in range(0, num_data, batch_size):
        batch_indices = indices[i:i+batch_size]
        batches.append((x[batch_indices], y[batch_indices]))

    return batches
```

```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
        for k in params.keys():
            if "grad_" in k:
                params[k.replace("grad_", "")] -= learning_rate * params[k]

    # divide avg by batch_num
    avg_acc /= batch_num

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

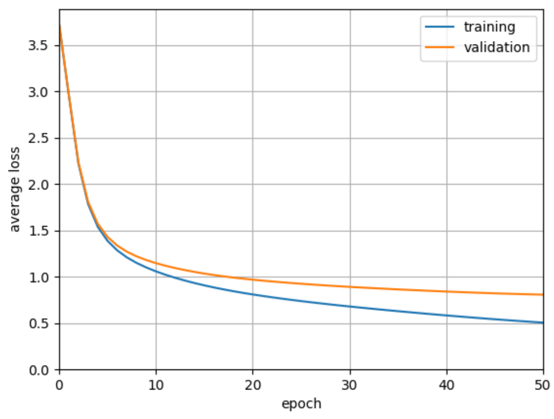
Q2.5

In python/run_q2.py, implement a numerical gradient checker. Instead of using the analytical gradients computed from the chain rule, add ϵ 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.

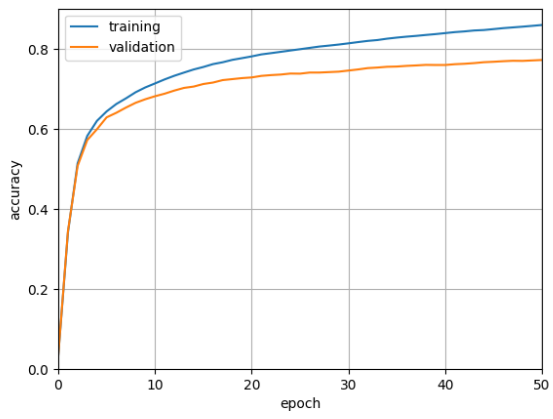
```python
# Q 2.5 should be implemented in this file
# you can do this before or after training the network.

# 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

    # loop over all parameters (W and b)
    for i in range(v.size):
        # add epsilon, run the network, get the loss
        params[k].flat[i] += eps
        h1 = forward(x, params, "layer1")
        probs = forward(h1, params, "output", softmax)
        loss_add, _ = compute_loss_and_acc(y, probs)

        # subtract 2*epsilon, run the network, get the loss
        params[k].flat[i] -= 2 * eps
        h1 = forward(x, params, "layer1")
        probs = forward(h1, params, "output", softmax)
        loss_sub, _ = compute_loss_and_acc(y, probs)

        # restore the original parameter value
        params[k].flat[i] += eps

        # compute derivative with central diffs
        params["grad_" + k].flat[i] = (loss_add - loss_sub) / (2 * eps)

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

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. Hint: Use fixed random seeds to improve reproducibility.

The network was able to achieve 77.25% accuracy on the validation set with a batch size of 32 and learning rate of 0.003. Below are the plot results:



(a) The cross-entropy loss plot.

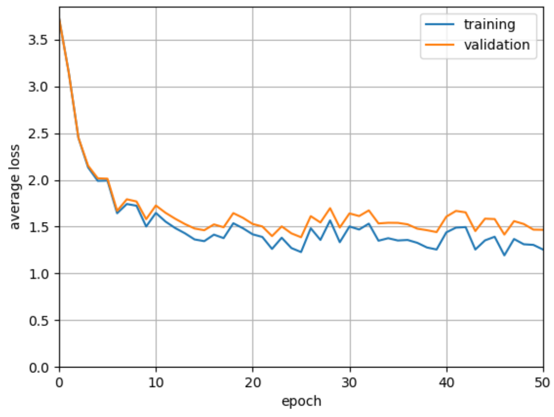


(b) The accuracy plot

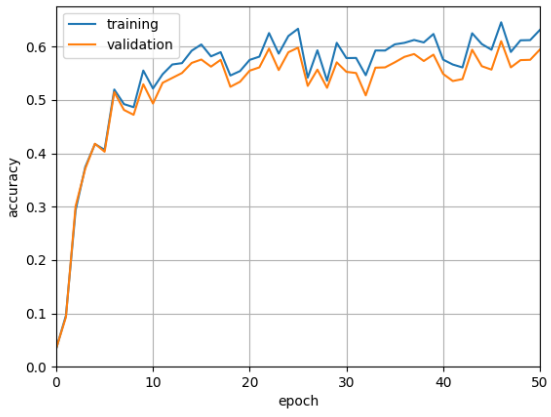
Q3.2
Use the script to train three networks: one with your tuned learning rate, one with 10 times that learning rate, and one with one tenth of that learning rate. Include all six plots in your writeup (two will be the same from the previous question). Comment on how the learning rates affect the training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

From the learning rate study below, we can observe that a higher learning rate will introduce oscillation in the loss and accuracy plots, results in a lower accuracy prediction. On the other hand, a lower learning rate introduced a smoother and better matched loss and accuracy plots. However, the network was unable to converge within the epoch, thus also produced a lower accuracy prediction. The best parameters for the network are: epoch of 50; batch size of 32; learning rate of 0.003, which produced 77.25 % accuracy.

[10x Learning Rate]
The network was able to achieve around 59.44% accuracy on the validation set with a batch size of 32 and learning rate of 0.03. Below are the plot results:

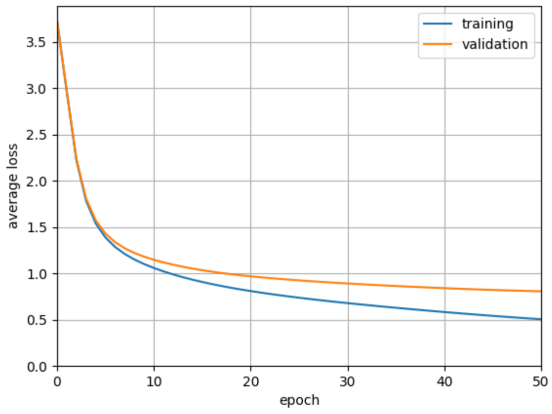


(a) The cross-entropy loss plot.

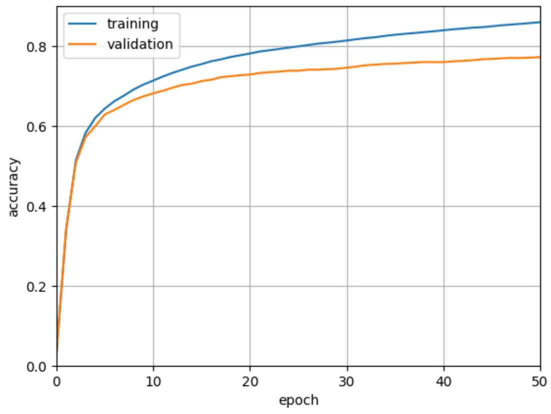


(b) The accuracy plot

[Previous Learning Rate]
The network was able to achieve 77.25% accuracy on the validation set with a batch size of 32 and learning rate of 0.003. Below are the plot results:

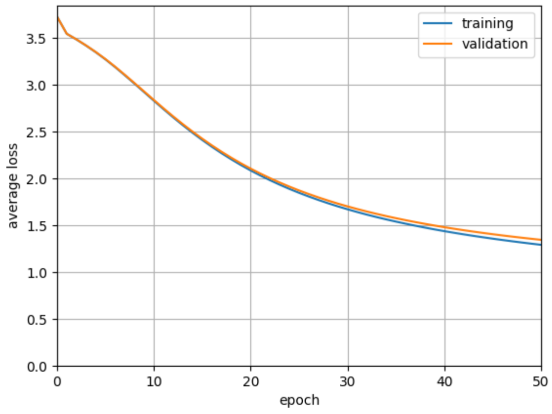


(a) The cross-entropy loss plot.

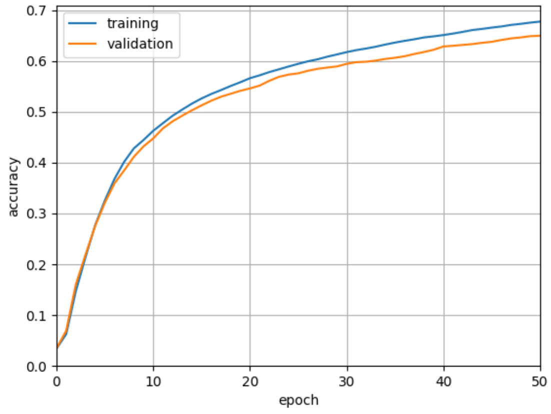


(b) The accuracy plot

[0.1x Learning Rate]
The network was able to achieve around 64.94% accuracy on the validation set with a batch size of 32 and learning rate of 0.0003. Below are the plot results:



(a) The cross-entropy loss plot.



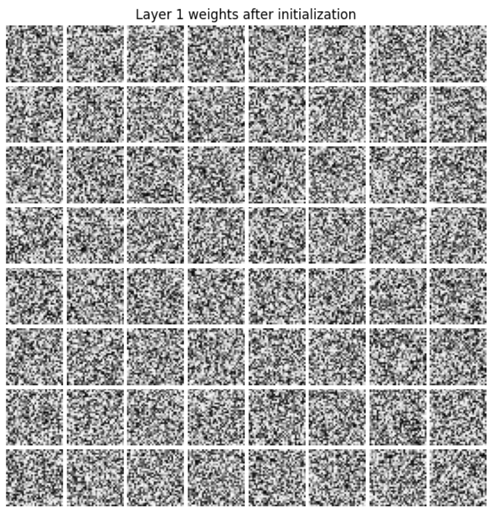
(b) The accuracy plot

Q3.3

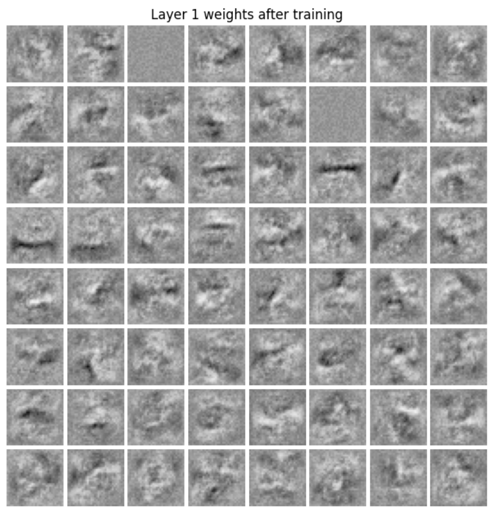
The script will visualize the first layer weights as 64 32x32 images, both immediately after initialization and after fully training. Include both visualizations in your writeup. Comment on the learned weights and compare them to the initialized weights. Do you notice any patterns?

From the visualization below, the initial first layer weights (a) can be seen as noisy, random with no meaningful patten. The trained first layer weights (b) can be seen to have distinct patterns with meaningful features detected.

Using epochs 50, a batch size of 32, and learning rate of 0.003, the visualization of the first later weights both immediately after initialization and after fully training are shown below:



(a) Layer 1 weights after initialization.

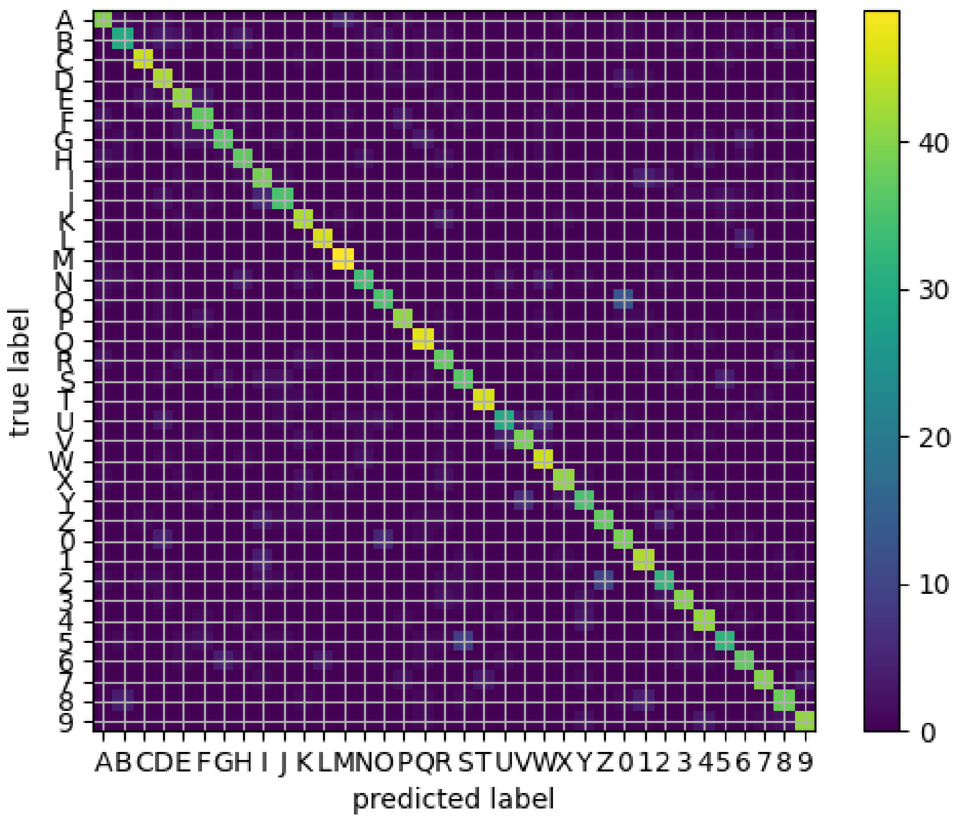


(b) Layer 1 weights after training.

Q3.4
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.

From the confusion matrix visualization below, we can observe that o and 0; z and 2, s and 5 are the top few pairs of classes that are most commonly confused. This is due to the similarity in shapes between the pair.

Using epochs 50, a batch size of 32, and learning rate of 0.003, the visualization of the confusion matrix is as follow:

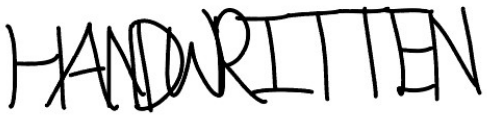


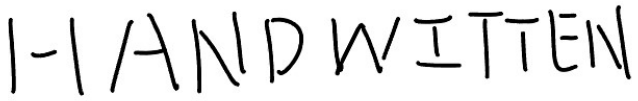
The confusion matrix.

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 two big assumptions are: handwritten characters are well separated and spaced, and each element of a single handwritten characters are not properl. Below are the two example images where one can expect the character detection will fail:


(a) Handwritten characters not separated nor spaced.

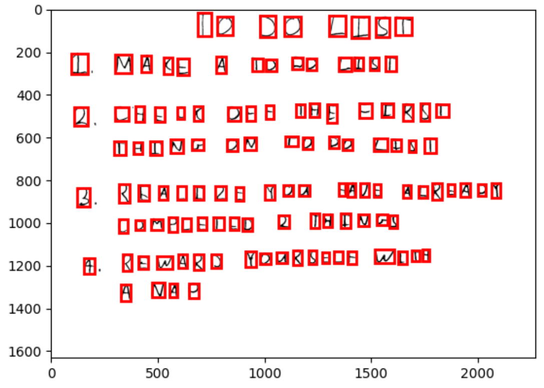

(b) Individual handwritten characters not connected.

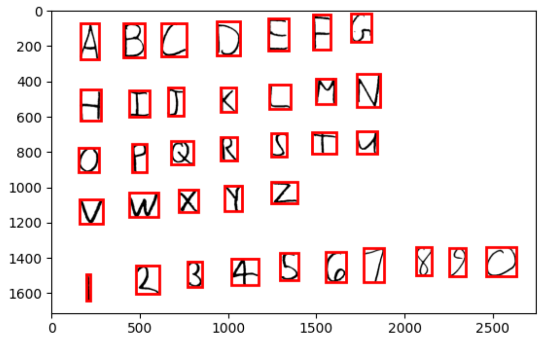
Q4.2

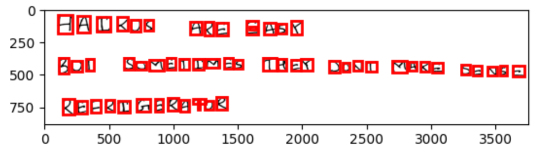
In python/q4.py, implement the function to find letters in the image. Given an RGB image, this function should return bounding boxes for all of the located handwritten characters in the image, as well as a binary black-and-white version of the image im. Each row of the matrix should contain [y1,x1,y2,x2] 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.

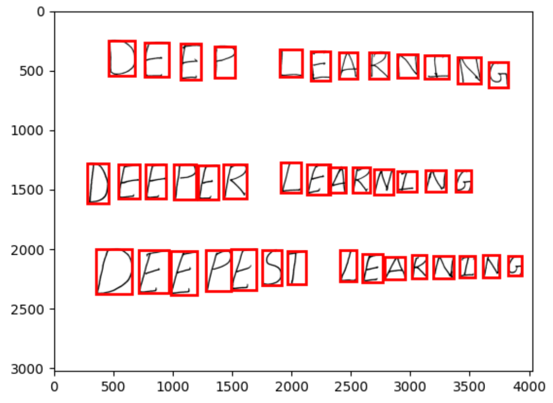
```python
def findLetters(image):
    bboxes = []
    bw = None
    # insert processing in here
    # one idea estimate noise -> denoise -> grayscale -> threshold -> morphology -> label ->
        skip small boxes
    # this can be 10 to 15 lines of code using skimage functions

    # Denoise (Gaussian blur)
    denoised = skimage.filters.gaussian(image, sigma=1)
    # Denoise (Bilateral filter)
    # denoised = skimage.restoration.denoise_bilateral(image, sigma_color=0.05,
        sigma_spatial=15, channel_axis=-1)

    # Convert to greyscale
    greyscale = skimage.color.rgb2gray(denoised)

    # Threshold
    threshold = skimage.filters.threshold_otsu(greyscale)
    binary = greyscale < threshold

    # Morphology
    bw = skimage.morphology.closing(binary, skimage.morphology.square(5))

    # Label
    label_image = skimage.measure.label(bw, background=0, connectivity=2)

    # Skip small boxes
    for region in skimage.measure.regionprops(label_image):
        if region.area < 100:
            continue

        minr, minc, maxr, maxc = region.bbox
        bboxes.append((minr, minc, maxr, maxc))

    return bboxes, bw
```

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.


(a) Visualization for 01_list.jpg.


(b) Visualization for 02_letters.jpg.


(c) Visualization for 03_haiku.jpg.


(d) Visualization for 04_deep.jpg.

Q4.4
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 act 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 detect all and correctly 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.

```
# Extracted Text for 01_list.jpg
      TO DD LIST
I MAFE A TD 20 LIST
I CHECK OFE THE FIRST
  T4INF ON TO DO CTST
3 RFALI2E YOU HAVE AKREADY
  CDMPLDTDD 2 THINFS
T 8EWAXD YOURSECE WITH
  A NAP
```

```
# Extracted Text for 02_letters.jpg
A B C D E F G
H I I K L M N
O P Q R S T U
V W X Y Z
1 Z 3 F S G 7 X 7 0
```

```
# Extracted Text for 03_haiku.jpg
HAIKUS ARE EHASX
BUT SOMETIMES TAEY OONT MAKE SENSE
REFRIGERATOR
```

```
# Extracted Text for 04_deep.jpg
DCEP LEARMING
DEETFK LEDKNIMG
DFCPFST LEARQIMG
```

Q5.1.1 [Extra Credit]
Due to the difficulty in training autoencoders, we use the activation function relu$(x) = \max(x, 0)$ **provided for you in** `util.py`**. Implement an autoencoder with the following layer architecture:**

- $1024 \rightarrow 32$ **dimensions, followed by a ReLU**
- $32 \rightarrow 32$ **dimensions, followed by a ReLU**
- $32 \rightarrow 32$ **dimensions, followed by a ReLU**
- $32 \rightarrow 1024$ **dimensions, followed by a sigmoid (to normalize the image output)**

The loss function used is the total squared error between the output image and the input image (they should match). Include your code in the writeup.

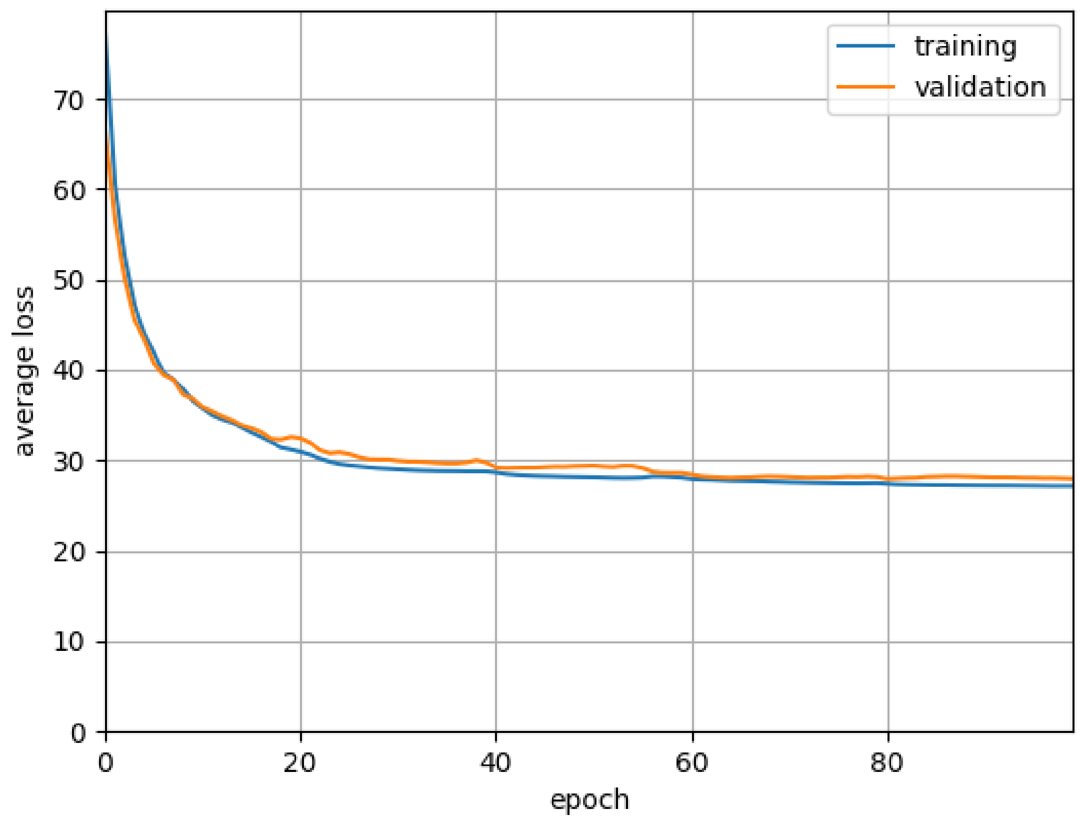
```python
# Q5.1 & Q5.2
# initialize layers
initialize_weights(train_x.shape[1], hidden_size, params, "layer1")
initialize_weights(hidden_size, hidden_size, params, "hidden1")
initialize_weights(hidden_size, hidden_size, params, "hidden2")
initialize_weights(hidden_size, train_x.shape[1], params, "output")
layer1_W_initial = np.copy(params["Wlayer1"])
hidden1_W_initial = np.copy(params["Whidden1"])
hidden2_W_initial = np.copy(params["Whidden2"])
output_W_initial = np.copy(params["Woutput"])

for k in list(params.keys()):
    if "grad" in k:
        name = k.replace("grad_", "")
        params["m_" + name] = np.zeros(params[k].shape)
# should look like your previous training loops
train_loss = []
valid_loss = []

for itr in range(max_iters):
    total_loss = 0
    for xb,_ in batches:
        # forward pass
        h1 = forward(xb, params, "layer1", relu)
        h2 = forward(h1, params, "hidden1", relu)
        h3 = forward(h2, params, "hidden2", relu)
        probs = forward(h3, params, "output", sigmoid)
        # loss (squared error)
        loss = np.sum((probs - xb)**2)
        total_loss += loss
        # backward
        delta1 = 2*(probs - xb)
        delta2 = backwards(delta1, params, "output", sigmoid_deriv)
        delta3 = backwards(delta2, params, "hidden2", relu_deriv)
        delta4 = backwards(delta3, params, "hidden1", relu_deriv)
        backwards(delta4, params, "layer1", relu_deriv)
        # apply gradient, remember to update momentum as well
        for k in list (params.keys()):
            if "grad" in k:
                name = k.replace("grad_", "")
                params["m_" + name] = 0.9 * params["m_" + name] - learning_rate * params[k]
                params[name] += params["m_" + name]

    # append training loss
    train_loss.append(total_loss/train_x.shape[0])
    # validation loss
    h1 = forward(valid_x, params, "layer1", relu)
    h2 = forward(h1, params, "hidden1", relu)
    h3 = forward(h2, params, "hidden2", relu)
    probs = forward(h3, params, "output", sigmoid)
    valid_loss.append(np.sum((probs - valid_x)**2)/valid_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(train_loss)), train_loss, label="training")
plt.plot(range(len(valid_loss)), valid_loss, label="validation")
plt.xlabel("epoch")
plt.ylabel("average loss")
plt.xlim(0, len(train_loss)-1)
plt.ylim(0, None)
plt.legend()
plt.grid()
plt.show()
```

Q5.1.2 [Extra Credit]
To further improve convergence speed, we will implement momentum. Rather than updating weights directly with the formula $W = W - \alpha \frac{\partial J}{\partial W}$, we introduce momentum with the following update rules: $M_W = 0.9 M_W - \alpha \frac{\partial J}{\partial W}$ and $W = W + M_W$. In this approach, initialize the parameters dictionary with zero-initialized momentum accumulators for each parameter. For each batch, apply both update equations. Include your code in the writeup.

```python
# Q5.1 & Q5.2
# initialize layers
initialize_weights(train_x.shape[1], hidden_size, params, "layer1")
initialize_weights(hidden_size, hidden_size, params, "hidden1")
initialize_weights(hidden_size, hidden_size, params, "hidden2")
initialize_weights(hidden_size, train_x.shape[1], params, "output")
layer1_W_initial = np.copy(params["Wlayer1"])
hidden1_W_initial = np.copy(params["Whidden1"])
hidden2_W_initial = np.copy(params["Whidden2"])
output_W_initial = np.copy(params["Woutput"])

for k in list(params.keys()):
    if "grad" in k:
        name = k.replace("grad_", "")
        params["m_" + name] = np.zeros(params[k].shape)
# should look like your previous training loops
train_loss = []
valid_loss = []

for itr in range(max_iters):
    total_loss = 0
    for xb,_ in batches:
        # forward pass
        h1 = forward(xb, params, "layer1", relu)
        h2 = forward(h1, params, "hidden1", relu)
        h3 = forward(h2, params, "hidden2", relu)
        probs = forward(h3, params, "output", sigmoid)
        # loss (squared error)
        loss = np.sum((probs - xb)**2)
        total_loss += loss
        # backward
        delta1 = 2*(probs - xb)
        delta2 = backwards(delta1, params, "output", sigmoid_deriv)
        delta3 = backwards(delta2, params, "hidden2", relu_deriv)
        delta4 = backwards(delta3, params, "hidden1", relu_deriv)
        backwards(delta4, params, "layer1", relu_deriv)
        # apply gradient, remember to update momentum as well
        for k in list (params.keys()):
            if "grad" in k:
                name = k.replace("grad_", "")
                params["m_" + name] = 0.9 * params["m_" + name] - learning_rate * params[k]
                params[name] += params["m_" + name]

    # append training loss
    train_loss.append(total_loss/train_x.shape[0])
    # validation loss
    h1 = forward(valid_x, params, "layer1", relu)
    h2 = forward(h1, params, "hidden1", relu)
    h3 = forward(h2, params, "hidden2", relu)
    probs = forward(h3, params, "output", sigmoid)
    valid_loss.append(np.sum((probs - valid_x)**2)/valid_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(train_loss)), train_loss, label="training")
plt.plot(range(len(valid_loss)), valid_loss, label="validation")
plt.xlabel("epoch")
plt.ylabel("average loss")
plt.xlim(0, len(train_loss)-1)
plt.ylim(0, None)
plt.legend()
plt.grid()
plt.show()
```

Q5.2 [Extra Credit]
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?

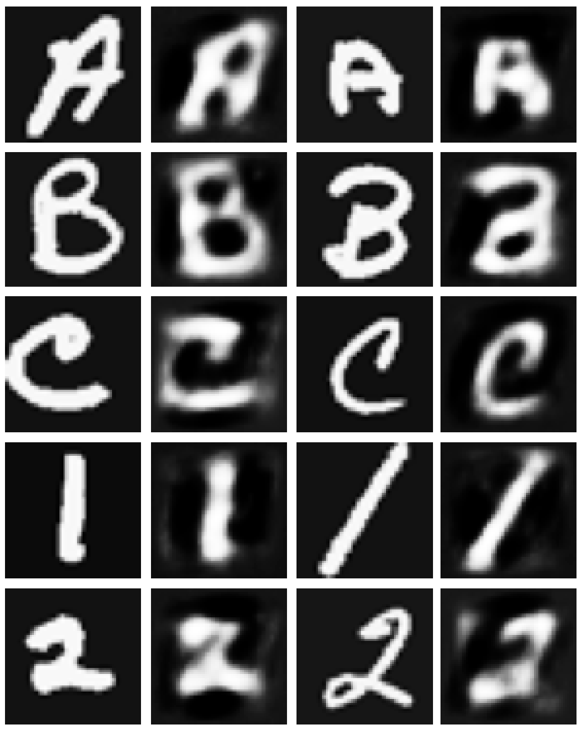
Using the provided default settings, we can observe that training loss and validation loss are both converging smoothly. Both loss plots are close to each other, which indicates a better prediction accuracy with less over-fitting. The training loss curve is as follow:



The cross-entropy loss plot.

Q5.3.1 [Extra Credit]
Now let's evaluate how well the auto-encoder 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?

From the 5 classes visualization, we can observe that the original images are more clear compared to the reconstructed validation images. Moreover, the reconstructed validation images in some cases does not completely reconstruct an one-to-one match to the original. The visualization images are as follows:



The validation images and their reconstruction for 5 selected classes.

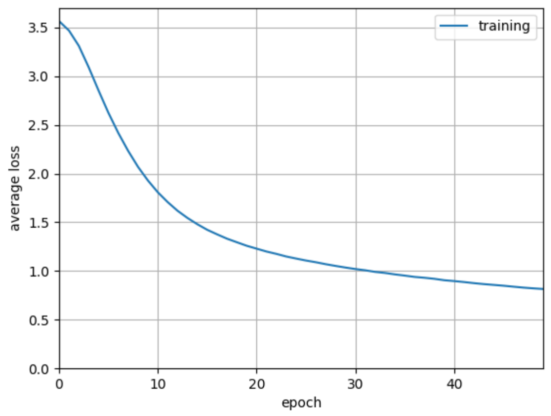
Q5.3.2 [Extra Credit]
Let's evaluate the reconstruction quality using Peak Signal-to-Noise Ratio (PSNR). PSNR is defined as:

$$\mathbf{PSNR} = 20 \times \log_{10}(\mathbf{MAXI}) - 10 \times \log_{10}(\mathbf{MSE})$$

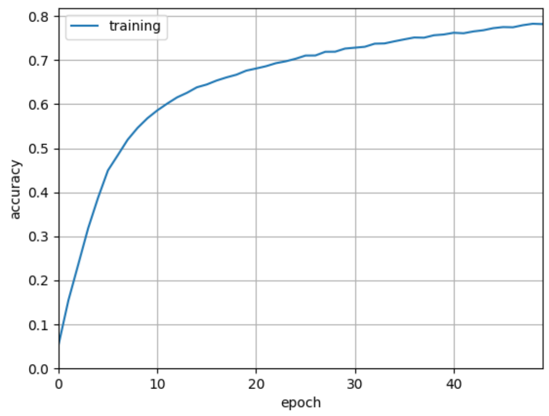
where MAXI is the maximum possible pixel value of the image, and MSE (mean squared error) is computed across all pixels. In other words, MAXI refers to the maximum positive value of the overall sum (brightest overall sum). Report the average PSNR obtained from the autoencoder across all images in the validation set, which should be around 15.

The average PSNR obtained from the auto-encoder across all images in the validation set is around: 15.2798.

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.



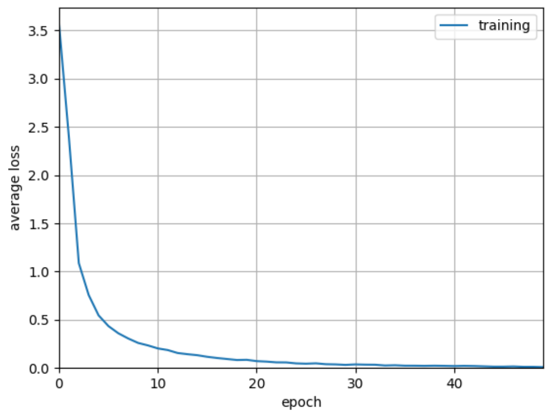
(a) The cross-entropy loss plot.



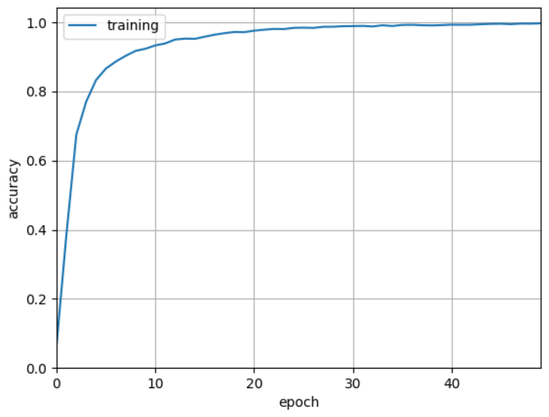
(b) The accuracy plot

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

From the visualization below, we can observe that the convolutional neural network has a lower loss and higher accuracy compared to the fully-connected network. The CNN also converged faster at an earlier epoch. The visualization are as follows:

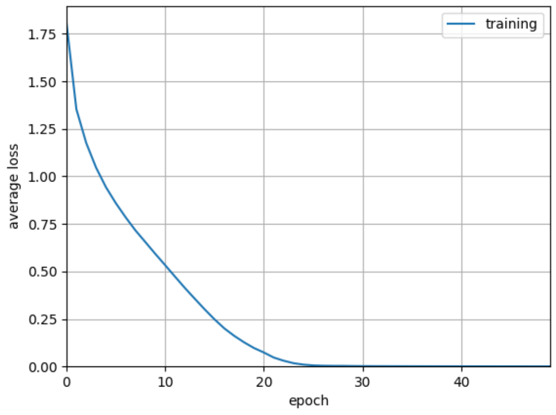


(a) The cross-entropy loss plot.

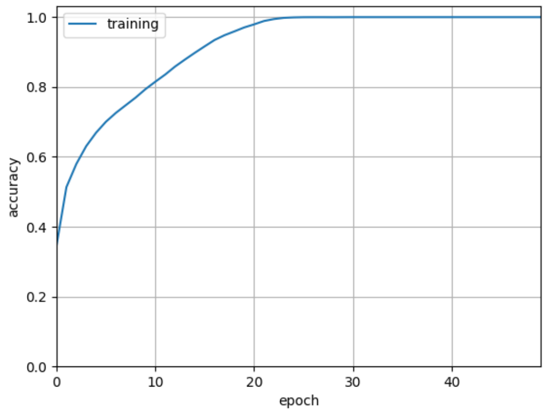


(b) The accuracy plot

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



(a) The cross-entropy loss plot.

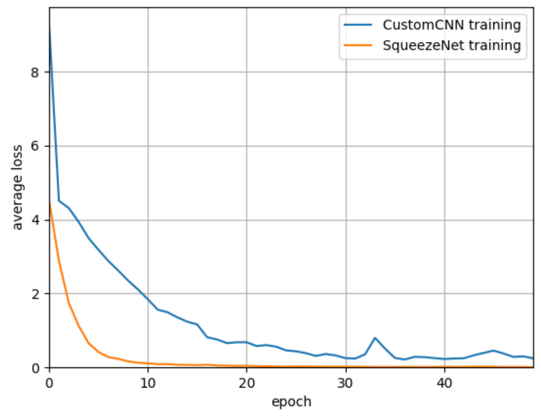


(b) The accuracy plot

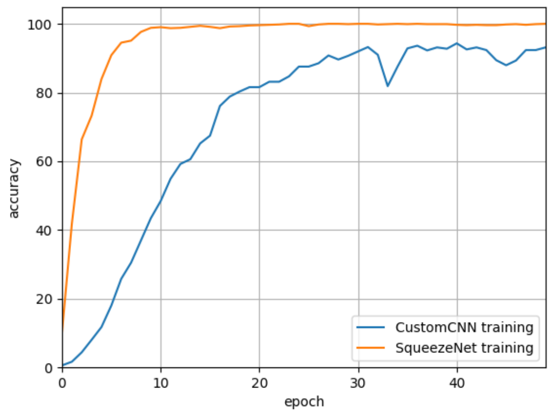
Q6.2 [Extra Credit]

Fine-tune a single layer classifier using pytorch on the flowers 17 (or flowers 102!) dataset using squeezenet1_1, and compare it against an architecture you've designed yourself (for example 3 convolutional layers followed 2 fully connected layers, slide 6) and trained from scratch. How do they compare? We include a script in scripts/ to fetch the flowers dataset and extract it in a way that torchvision.datasets.ImageFolder can consume it, see an example, from data/oxford-flowers17. You should look at how SqueezeNet is defined, and just replace the classifier layer. There exists a pretty good example for fine-tuning in PyTorch

From the visualization using flowers 102 dataset, we can observe that squeezenet1_1 has a lower loss, has a higher accuracy, and converged much faster with smoother curves compared to my custom CNN model. The final loss and accuracy values for the squeezenet1_1 model after 50 epochs are 0.0089 and 100. The final loss and accuracy values for the my custom CNN model after 50 epochs are 0.2483 and 93.1317. The visualization are as follows:



(a) The cross-entropy loss plot.



(b) The accuracy plot

Q6.3 [Extra Credit]
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?

Did not attempt.