Q1.1

1. $softmax(x_i) = \frac{e^{x_i}}{\sum e^{x_j}}$
2. $softmax(x_i + c) = \frac{e^{x_i+c}}{\Sigma_j e^{x_j+c}} = \frac{e^{x_i}e^c}{\Sigma_j e^{x_i}e^c}$
3. $softmax(x_i + c) = \frac{e^{x_i}}{\Sigma_j e^{x_j}} = softmax(x_i)$

In (1) we have the general form of the softmax equation. Translating by a scalar c, allows us to expand terms (2). In (2) we also find that by using basic exponent rules the scalar term actually falls out. This allows us to assert in (3) that the softmax is invariant to translation as $softmax(x_i + c) = softmax(x_i)$.

Often it is a good idea to use c $=$ -max(x_i) because this scales the terms by shifting them all by the maximum value, such that the largest value will turn to 1 ($e^{xi\_max-xi\_max} = 1$), and the rest will be scaled between 0-1. With this in mind, it is not hard to see that this method of shifting can prevent some numerical overflow and promote stability.

Q1.2

1. Values from softmax range from 0-1, with the resultant sum being 1.
2. One could say that “softmax takes an arbitrary real valued vector x and turns it into a [***probability distribution]***
3. Steps of Softmax
    a. Take the exponential of each outcome to find the outcome frequency
    b. Find the sum of all the outcome frequencies
    c. Normalize by the sum of the outcome frequencies to find the total outcome frequency (the probability of each occurance)

Q1.3

Generally, in a forward pass of a fully-connected layer we would use the equation: $y = wx + b$

Applying this to between two fully-connected layers without an activation looks like the process derived below:

1. $y_l = w_l x_l + b_l$
2. $y_l = w_l(w_{l-1}x_{l-1} + b_{l-1}) + b_l$
3. $y_l = w_l w_{l-1} x_{l-1} + w_l b_{l-1} + b_l$
4. $y_l = w' x_{l-1} + b'$
5. $y_l = wx + b$

At step (5) we can observe that we ended up with the same general form as we did when we started, showing that a multi-layer network without activiations is equivalent to doing a linear regression problem.

Q1.4

The derivative of the sigmoid function is given by the following formula:

$$\frac{d}{dx}\sigma(x) = \sigma(x) \cdot (1 - \sigma(x))$$

where $\sigma(x)$ is the sigmoid function. This formula can be derived by applying the chain rule to the sigmoid function, which is defined as:

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

First, we can rewrite the derivative using the definition of the sigmoid function:

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

Next, we can apply the chain rule to take the derivative of the fraction. The derivative of the numerator is simply 1, and the derivative of the denominator is given by the following:

$$\frac{d}{dx}(1+e^{-x}) = \frac{d}{dx}1 + \frac{d}{dx}e^{-x} = 0 - e^{-x} = -e^{-x}$$

Therefore, the derivative of the sigmoid function is given by:

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

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

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

Q1.5

The gradient of the loss function J with respect to y can be written as:

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

Since y is a function of W, x, and b, we can use the chain rule to compute the partial derivatives of J with respect to W, x, and b.

First, let's find the partial derivative of J with respect to W. We can write:

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

Take the derivative of y wrt W:

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

Therefore, the partial derivative of J with respect to W is given by:

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

Next, let's find the partial derivative of J with respect to x. We can write:

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

Find the derivative of y wrt x:

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

Therefore, the partial derivative of J with respect to x is given by:

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

Finally, let's find the partial derivative of J with respect to b. We can write:

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

Take the derivative of y wrt b:

$$\frac{\partial y}{\partial b} = 1$$

Therefore, the partial derivative of J with respect to b is given by:

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y} \cdot 1$$

To summarize, we have found that the partial derivatives of the loss function J with respect to W, x, and b are given by:

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

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

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

In matrix form, these equations can be written as:

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

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

$$\frac{\partial J}{\partial b} = 1 \cdot \frac{\partial J}{\partial y}$$

Where X, W, 1 are all matrices.

Q1.6

1. The sigmoid can only output values between 0,1 and its derivate can only output values between 0, 0.25. If we are propagating these small values sequentially over many layers we can expect the gradients to continually decrease until they "vanish".

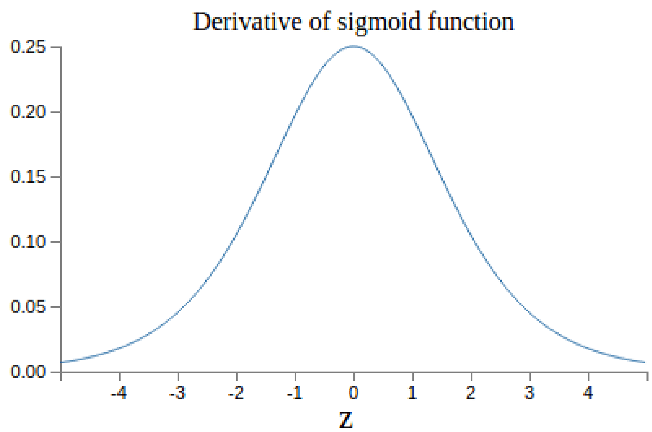


Figure 1. Plot of sigmoid derivative

2. The sigmoid is capable of outputting between (0,1) while tanh can output between (-1, 1). During training if you obtain a network with a lot of 0 outputs the sigmoid would also output a lot of signals that map to 0, which may cause the subsequent gradient to go to 0 and the learning for the network to stall.
3. As is apparent given figure 2, the derivate of tanh produces a higher maximum peak than the sigmoid derviative: leading to a greater descent and update of the weights & biases during learning.

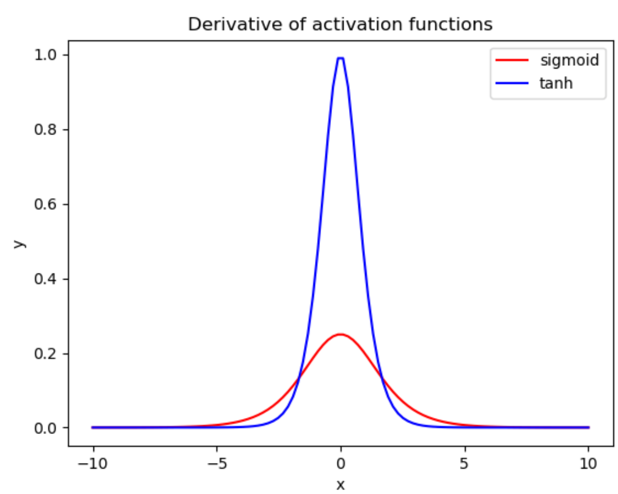


Figure 2. Plot of tanh derivative

4. The tanh and sigmoid are defined as:

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

Here is the numerator of the tanh:

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

Here is the denominator of the tanh:

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

Therefore, the tanh function can be written in terms of the sigmoid function as:

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

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

This shows that the tanh function can be expressed in terms of the sigmoid function.

Q2.1.1

It is not a good idea to initialize a network with all zeros for one because if all the weights are the same the neurons will end up learning the same thing, it will lead to symmetry across the network. It also may prevent the network from being able to generalize new data because if it ends up outputting the same constant value it won't have learned any new patters from the data

Q2.1.2

```python
def initialize_weights(in_size, out_size, params, name=''):
    W, b = None, None

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

    var = np.sqrt(6 / (in_size + out_size))

    W = np.random.randn(in_size, out_size) * var
    b = np.zeros(shape=out_size)

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

Q2.1.3

By imposing randomness without the system we can help to maximize the potential for a local minima to be reached, and often more quickly. The weights are instantiated based on the size of the layer due to the fact that the variance of the distribution should be unqiue to that layer so that any issues regarding exploding/vanishing gradients can be mitigated. This is best visualized with the graph from the 6th page of the pdf given on "Understanding the difficulty of training deep feedforward neural networks" (below)

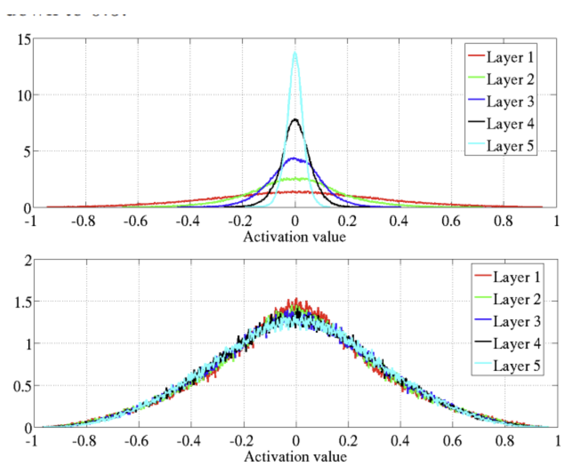


Figure 3. Standard activation values for tanh (top) and normalized initialization (bottom)

Q2.2.1

```python
def sigmoid(x):

    # Simply the equation for a sigmoid activation function
    res = 1 / (1 + np.exp(-x))

    return res

############################## Q 2.2.1 ##############################


def forward(X, params, name='', activation=sigmoid):
    """
    Do a forward pass

    Keyword arguments:
    X -- input vector [Examples x D]
    params -- a dictionary containing parameters
    name -- name of the layer
    activation -- the activation function (default is sigmoid)
    """
    pre_act, post_act = None, None
    # get the layer parameters
    W = params['W' + name]
    b = params['b' + name]

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

    """
    1. Forward prop without activation: W.T * X + b
    2. Pass through the activation function
    """

    pre_act = np.dot(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

```python
def softmax(x):
    res = None

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

    """
    1. For numerical stability, shift all values over so the maximum is now 0
        - expand the dims so you can broadcast, need a singleton dimension
    2. Exponentiate all the xvals
    3. Take a cumulative sum acros the rows, so you can softmax each example
    4. Get the softmax result to make a probability vector
        - Gives prob. that a particular input is of a particular class
    """

    # 1.
    x_shift = x - np.expand_dims(np.amax(x, axis=1), axis=1)
    # 2.
    x_exp = np.exp(x_shift)
    # 3.
    x_sum = np.sum(x_exp, axis=1, keepdims=True)
    # 4.
    res = x_exp / x_sum

    return res
```

Q2.2.3

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

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

    """
    1. Use cross-entropy loss function
    2. Find which label was accurately guessed
    """

    # 1.
    loss = - np.sum((y * np.log(probs)))
    # 2.
    prob_labels = np.argmax(probs, axis=1)
    y_labels = np.argmax(y, axis=1)
    acc = (y_labels == prob_labels).astype(int)
    acc = np.sum(acc) / acc.shape[0]

    return loss, acc
```

Q2.3

```python
def backwards(delta, params, name='', activation_deriv=sigmoid_deriv):
    """
    Do a backwards pass

    Keyword arguments:
    delta -- errors to backprop
    params -- a dictionary containing parameters
    name -- name of the layer
    activation_deriv -- the derivative of the activation_func
    """
    grad_X, grad_W, grad_b = None, None, None
    # everything you may need for this layer
    W = params['W' + name]
    b = params['b' + name]
    X, pre_act, post_act = params['cache_' + name]

    # do the derivative through activation first
    # (don't forget activation_deriv is a function of post_act)
    # then compute the derivative W, b, and X
    ##########################
    ##### your code here #####
    ##########################

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

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

Q2.4

```python
def get_random_batches(x, y, batch_size):
    batches = []
    ##########################
    ##### your code here #####
    ##########################

    # shuffled the data
    shuffled_x, shuffled_y = shuffled_data(x, y)
    # get the number of batches
    num_batches = x.shape[0] / batch_size
    # split the batches into number of batches
    xbatches = np.split(shuffled_x, num_batches)
    ybatches = np.split(shuffled_y, num_batches)
    # for the splot arrays, zip them together, then append them to batches
    for xbatch, ybatch in zip(xbatches, ybatches):
        batches.append((xbatch, ybatch))

    return batches


def shuffled_data(x, y):
    shuffled_ind = np.random.permutation(x.shape[0])
    shuffled_x = x[shuffled_ind, :]
    shuffled_y = y[shuffled_ind, :]
    return shuffled_x, shuffled_y
```

Q2.5

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

        if "W" in k:
            params_copy = copy.deepcopy(params)

            for i in range(v.shape[0]):
                for j in range(v.shape[1]):

                    v[i, j] += eps

                    # Add epsilon
                    h1 = forward(x, params_copy, 'layer1')
                    probs = forward(h1, params_copy, 'output', softmax)
                    loss_pluseps, acc_minuseps = compute_loss_and_acc(y, probs)

                    v[i, j] -= 2*eps
                    # Subtract Epsilon
                    h1 = forward(x, params_copy, 'layer1')
                    probs = forward(h1, params_copy, 'output', softmax)
                    loss_minuseps, acc_minuseps = compute_loss_and_acc(
                        y, probs)

                    # compute derivative
                    params[k][i, j] = (loss_pluseps-loss_minuseps)/(2*eps)

                    # reset grad
                    v[i, j] += eps

        elif "b" in k:
            params_copy = copy.deepcopy(params)

            for i in range(v.shape[0]):

                v[i] += eps

                # Add epsilon & do forward prop
                h1 = forward(x, params_copy, 'layer1')
                probs = forward(h1, params_copy, 'output', softmax)
                loss_pluseps, acc_minuseps = compute_loss_and_acc(y, probs)

                v[i] -= 2*eps
                # Subtract Epsilon & do forward prop
                h1 = forward(x, params_copy, 'layer1')
                probs = forward(h1, params_copy, 'output', softmax)
                loss_minuseps, acc_minuseps = compute_loss_and_acc(y, probs)

                # compute derivative
                params[k][i] = (loss_pluseps-loss_minuseps)/(2*eps)

                # reset grad
                v[i] += eps
```

Q3.1: Code

```python
import string
import pickle
import numpy as np
import scipy.io
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid
from nn import *

train_data = scipy.io.loadmat('../data/nist36_train.mat')
valid_data = scipy.io.loadmat('../data/nist36_valid.mat')
test_data = scipy.io.loadmat('../data/nist36_test.mat')

train_x, train_y = train_data['train_data'], train_data['train_labels']
valid_x, valid_y = valid_data['valid_data'], valid_data['valid_labels']
test_x, test_y = test_data['test_data'], test_data['test_labels']


if False:  # view the data
    np.random.shuffle(train_x)
    for crop in train_x:
        plt.imshow(crop.reshape(32, 32).T, cmap="Greys")
        plt.show()

max_iters = 50
# pick a batch size, learning rate
batch_size = 64
learning_rate = 1e-3
hidden_size = 64
##########################
##### your code here #####
##########################


batches = get_random_batches(train_x, train_y, batch_size)
batch_num = len(batches)

params = {}

# initialize layers
initialize_weights(train_x.shape[1], hidden_size, params, "layer1")
initialize_weights(hidden_size, train_y.shape[1], params, "output")
layer1_W_initial = np.copy(params["Wlayer1"])  # copy for Q3.3

```

Q3.1: Code

```
train_loss = []
valid_loss = []
train_acc = []
valid_acc = []
for itr in range(max_iters):
    # record training and validation loss and accuracy for plotting
    h1 = forward(train_x, params, 'layer1')
    probs = forward(h1, params, 'output', softmax)
    loss, first_acc = compute_loss_and_acc(train_y, probs)
    train_loss.append(loss/train_x.shape[0])
    train_acc.append(first_acc)
    h1 = forward(valid_x, params, 'layer1')
    probs = forward(h1, params, 'output', softmax)
    loss, acc = compute_loss_and_acc(valid_y, probs)
    valid_loss.append(loss/valid_x.shape[0])
    valid_acc.append(acc)

    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)

        for k, v in sorted(list(params.items())):
            if 'grad' in k:
                name = k.split('_')[1]
                # val = val + lr * val_dot
                params[name] -= learning_rate * params[k]

    avg_acc /= len(batches)

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

    # if itr % 100 == 0:
    #     learning_rate *= 0.9

# record final training and validation accuracy and loss
h1 = forward(valid_x, params, 'layer1')
probs = forward(h1, params, 'output', softmax)
loss, acc = compute_loss_and_acc(valid_y, probs)
valid_loss.append(loss/valid_x.shape[0])
valid_acc.append(acc)

h1 = forward(train_x, params, 'layer1')
probs = forward(h1, params, 'output', softmax)
loss, acc = compute_loss_and_acc(train_y, probs)
train_loss.append(loss/train_x.shape[0])
train_acc.append(acc)
```

Q3.1: Results

- Hyperparameters:
    - Learning Rate: 2e-3
    - Batch Size: 64
    - Epochs: 50
- Results:
    - Training Accuracy: 83%
    - Validation Accuracy: 75.1%
    - Testing Accuracy: 75.6%

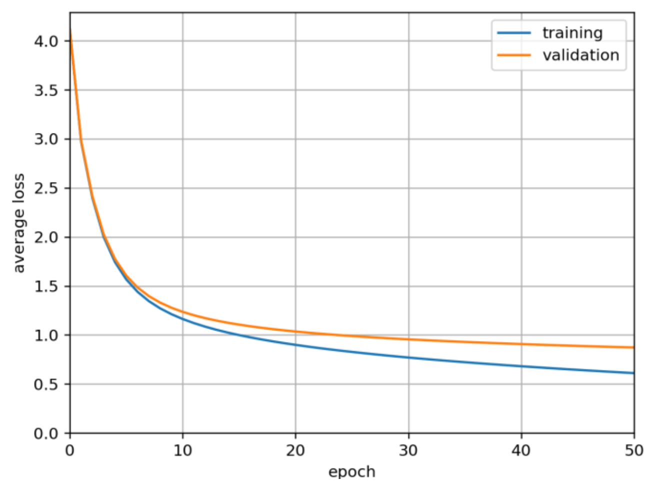


Figure 4. Loss for training and validation sets

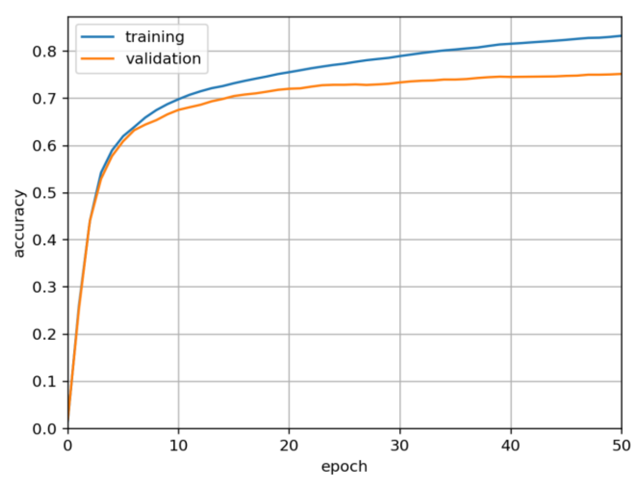


Figure 5. Accuracy for training and validation sets

Q3.2

Learning Rate 10x Smaller:

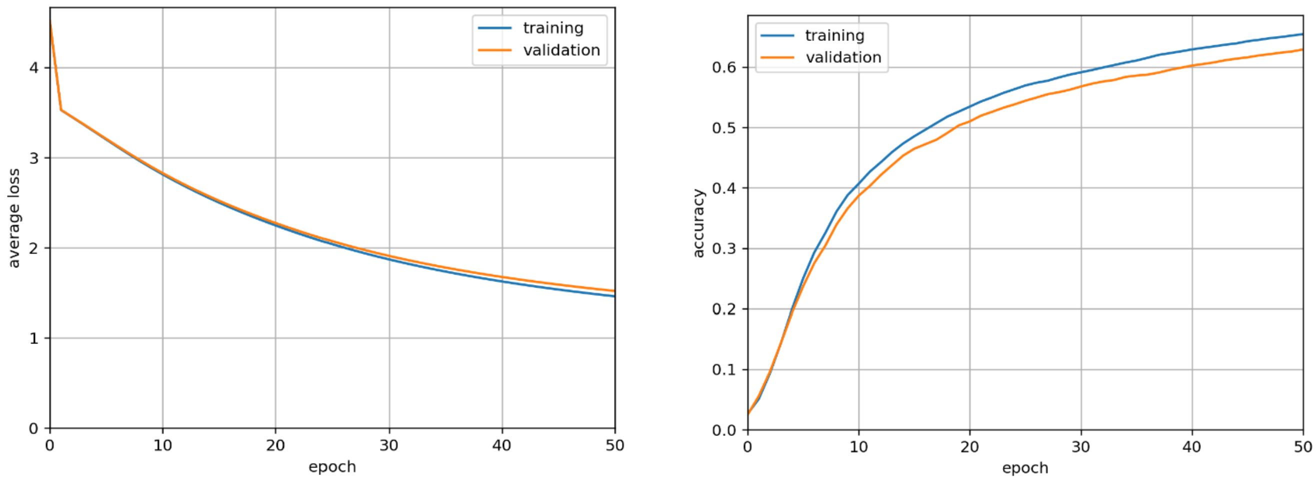


Figure 6. Loss & Accuracy for learning rate = 2e-4

- Hyperparameters:
    - Learning Rate: 2e-4
    - Batch Size: 64
    - Epochs: 50
- Results:
    - Training Accuracy: 65%
    - Validation Accuracy: 62.9%
    - Testing Accuracy: 63.9%

Q3.2

Learning Rate 10x Larger:

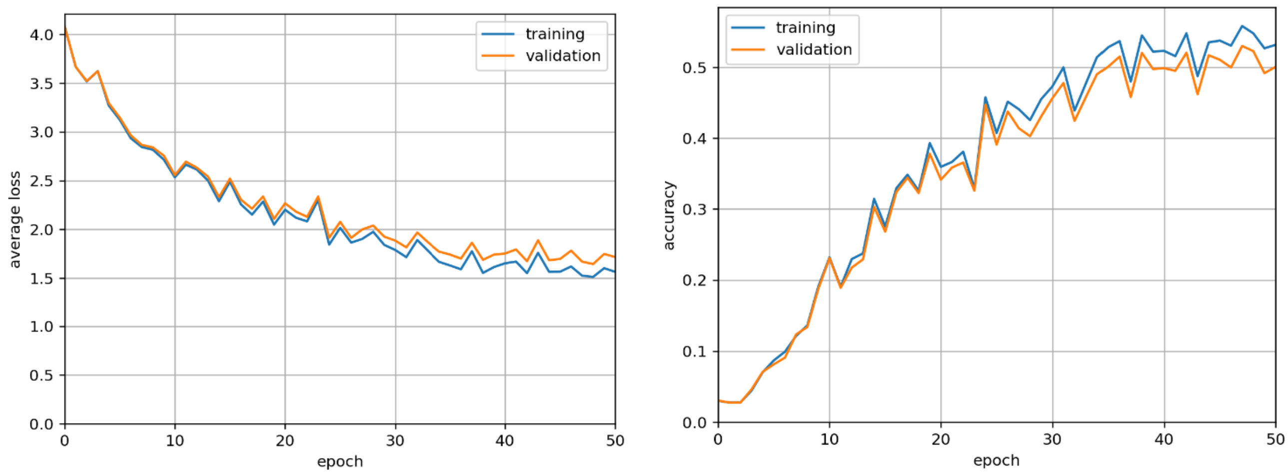


Figure 7. Loss & Accuracy for learning rate = 2e-2

- Hyperparameters:
  - Learning Rate: 2e-2
  - Batch Size: 64
  - Epochs: 50
- Results:
  - Training Accuracy: 55%
  - Validation Accuracy: 50%
  - Testing Accuracy: 51.3%

Comments:

- In the learning rate of 2e-4, we observe a smooth curve with a gradual increase in accuracy, however the learning isn't fast enough to plateau before the imposed number of epochs.
- In the learning rate of 2e-2 there is a much higher oscillation in learning and loss curves, seemingly producing a highly erractic learning profile. This also resulted in a much loser accuracy for all three data sets than the smaller learning rates.
- Between these two datasets and the original learning rate, the original still produced the best results on the test set at an accuracy of 75.6%. However, just looking at the altered learning rates in Q3.2, the slower learning rate performed better (63.9% vs 51.3% accuracies.

Q3.3

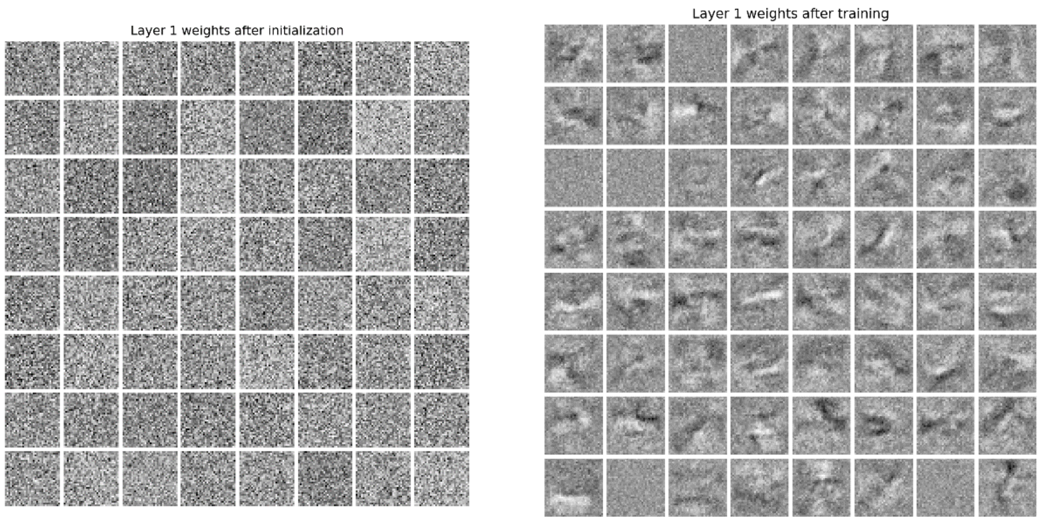


Figure 8. Weights after initialization (left) and after training (right)

Comments:

The network used to generate figure 8 used the hyperparameters given in Q3.1.1. Visually, the weights given after initialization have no visably recognized patterns – just looks like pure white noise. After training the weights are beginning to show resemblance of learning shape or form, hinting at the fact that they are gaining information about how to accurately predict the MNIST dataset.

Q3.4 Code

```python
# Q3.4
confusion_matrix = np.zeros((test_y.shape[1], test_y.shape[1]))

# compute confusion matrix
##########################
##### your code here #####
##########################

for i in range(0, test_y.shape[0]):
    # find the current row's prediction and ground truth label
    pred = np.argmax(test_y[i, :])
    true = np.argmax(test_probs[i, :])

    confusion_matrix[pred, true] += 1
```

Q3.4 Results:

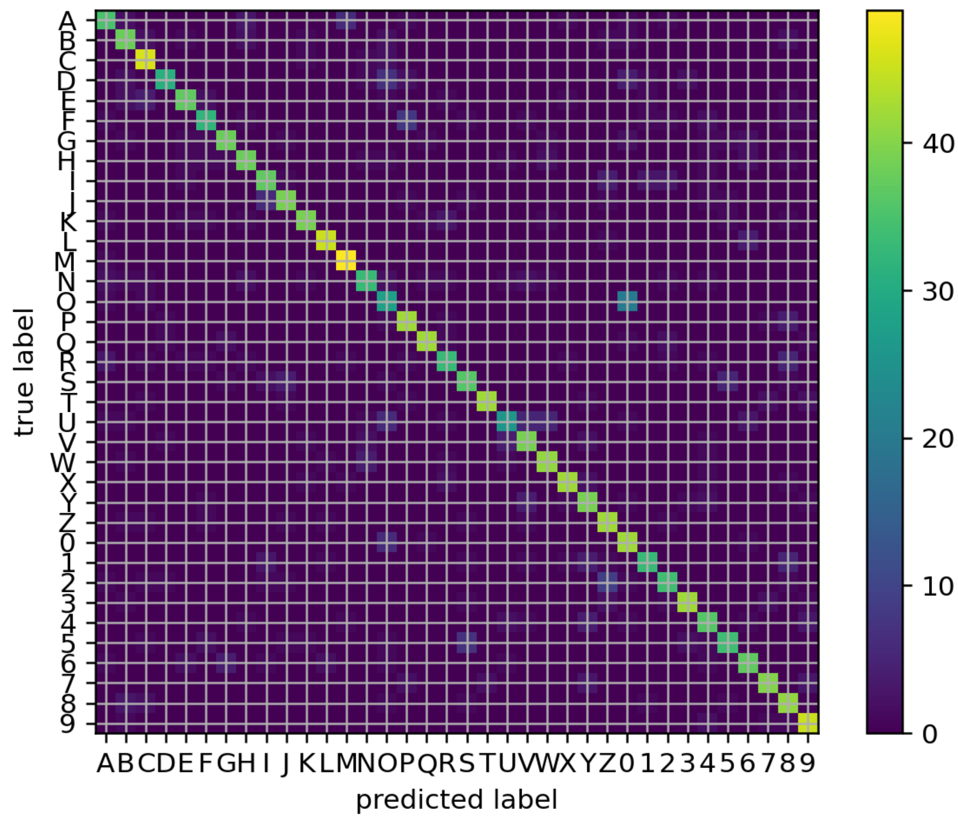


Figure 9. Confusion Matrix for testing data on the MNIST dataset

Comments

Here, the hyperparameters used were the same as in Q3.1 (lr = 2e-3, batch size = 64, epochs = 50). The most confused character seemed to be "0" versus "O", which is more than intuitive. Some other lesser confused characters were "2" and "Z", "S" and "5", and "F" and "P". These are reasonable results to obtain, as they are virtually the same penstroke, and depending on two wrote the F/P they are generally similar besides being closed in the off-vertical position.

Q4.1

Assumptions:

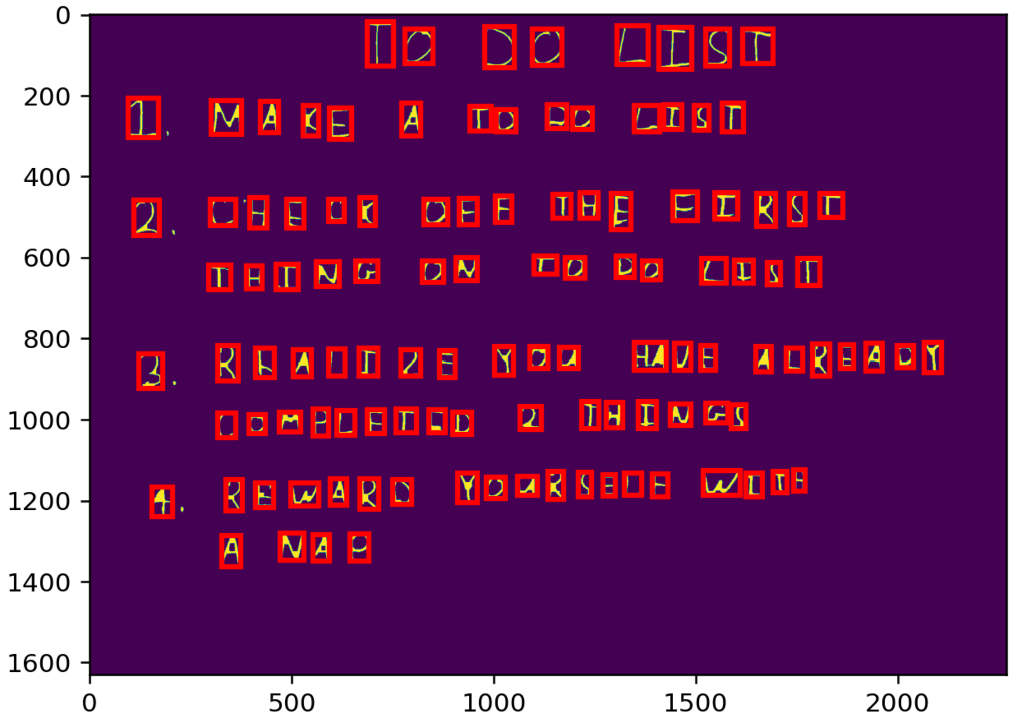
1. The letters written are fully connected
    a. As seen through the O, E, T if the letters are not fully connected there would be multiple bounding boxed surrounding a singular letter
2. The letters should not be fully connected
    a. Our classification would not work for letters that are connected, say if they were written in cursive as the letters are by nature not standalone

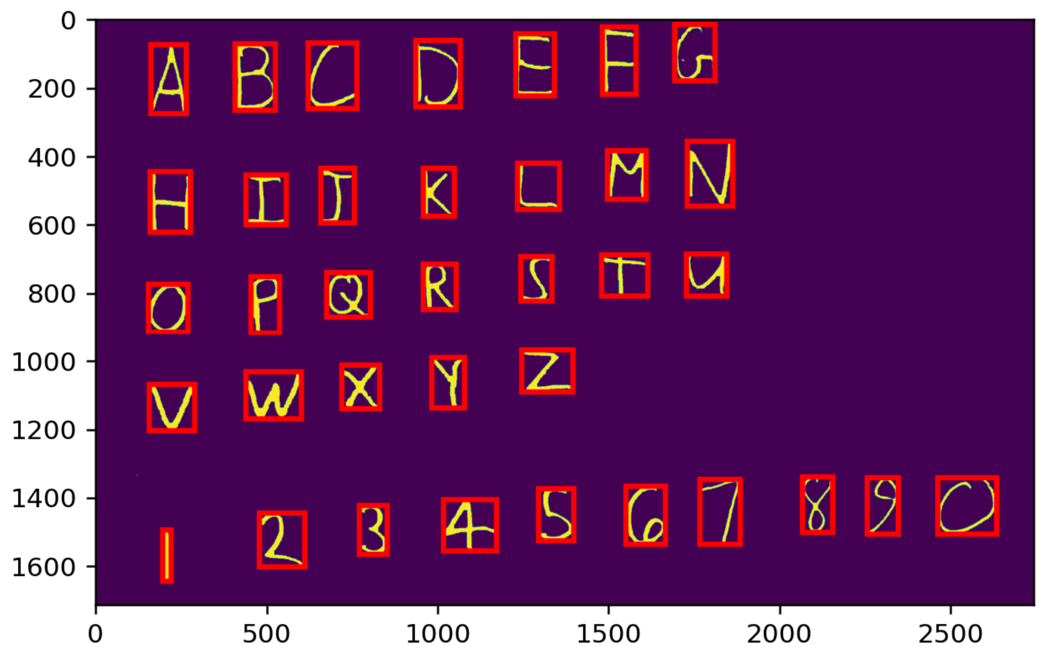
Figure 10. Example of assumptions made by our model

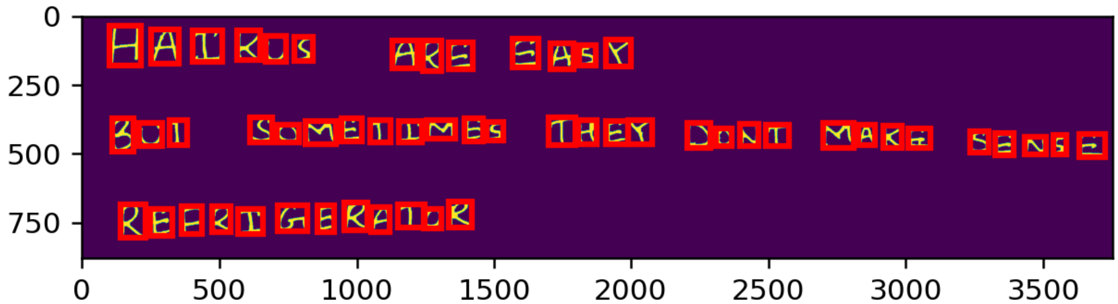
Q4.2: Code

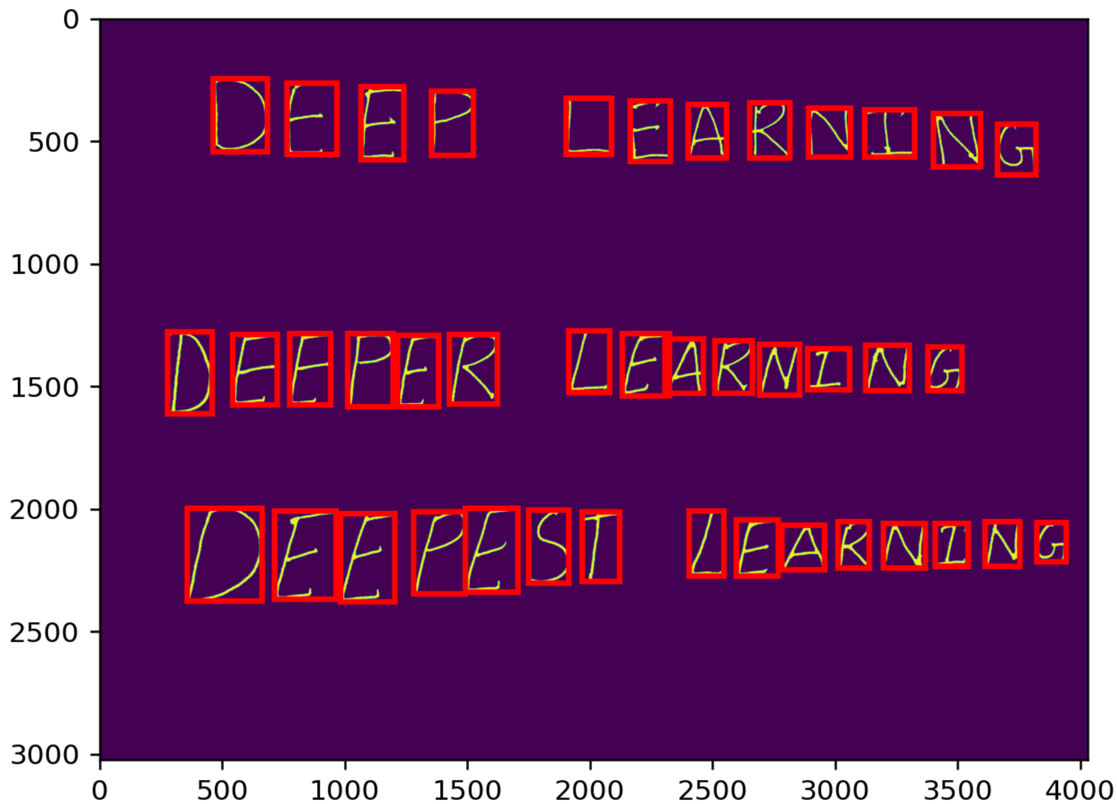
```
def findLetters(image):
    bboxes = []
    bw = None
    # insert processing in here
    # one idea estimate noise -> denoise -> greyscale -> threshold -> morphology -> label -> skip small boxes
    # this can be 10 to 15 lines of code using skimage functions

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

    # denoise the image
    # image = skimage.restoration.denoise_tv_chambolle(
    #     image, weight=0.1, channel_axis=-1)
    # change the image to gray
    image = skimage.color.rgb2gray(image)

    # apply threshold
    thresh = skimage.filters.threshold_otsu(image)
    bw = skimage.morphology.closing(
        image <= thresh, skimage.morphology.square(10)).astype(np.float32)

    # remove artifacts connected to image border
    cleared = skimage.segmentation.clear_border(bw)

    # label image regions
    label_image = skimage.measure.label(cleared)
    # to make the background transparent, pass the value of `bg_label`,
    # and leave `bg_color` as `None` and `kind` as `overlay`
    image_label_overlay = skimage.color.label2rgb(
        label_image, image=image, bg_label=0)

    # fig, ax = plt.subplots(figsize=(10, 6))
    # ax.imshow(image_label_overlay)

    for region in skimage.measure.regionprops(label_image):
        # take regions with large enough areas
        if region.area >= 300:
            bboxes.append(region.bbox)

    return bboxes, bw
```

Q4.3


TO DO LIST
1. MAKE A TO DO LIST
2. CHECK OFF THE FIRST THING ON TO DO LIST
3. REALIZE YOU HAVE ALREADY COMPLETED 2 THINGS
4. REWARD YOURSELF WITH A NAP
0 200 400 600 800 1000 1200 1400 1600
0 500 1000 1500 2000


A B C D E F G
H I J K L M N
O P Q R S T U
V W X Y Z
1 2 3 4 5 6 7 8 9 0
0 200 400 600 800 1000 1200 1400 1600
0 500 1000 1500 2000 2500


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


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

Q4.4: Code

```
for img in os.listdir('../images'):
    im1 = skimage.img_as_float(
        skimage.io.imread(os.path.join('../images', img)))
    bboxes, bw = findLetters(im1)

    plt.imshow(bw)
    for bbox in bboxes:
        minr, minc, maxr, maxc = bbox
        rect = matplotlib.patches.Rectangle((minc, minr), maxc - minc, maxr - minr,
                                            fill=False, edgecolor='red', linewidth=2)
        plt.gca().add_patch(rect)
    plt.show()
    # find the rows using..RANSAC, counting, clustering, etc.
    ##########################
    ##### your code here #####
    ##########################

    def sorted_arr(arr):

        # find the xvals
        xvals = []
        for i in arr:
            xvals.append(i[1])
        xvals.sort()
        # use the sorted x vals to sort the original array
        result = []
        for i in xvals:
            for j in arr:
                if i == j[1]:
                    result.append(j)
        return result

    # make the rows by finding the box's whose centroids are within the
    # ref template
    rows = []
    realtime_arr = [bboxes[0]]
    for i in range(1, len(bboxes)):
        curr_bbox = bboxes[i]
        box_center_y = (curr_bbox[2] + curr_bbox[0])/2
        ref_bbox = realtime_arr[-1]
        if box_center_y < ref_bbox[0] or box_center_y > ref_bbox[2]:
            rows.append(realtime_arr)
            realtime_arr = []
            realtime_arr.append(curr_bbox)
        else:
            realtime_arr.append(curr_bbox)
    rows.append(realtime_arr)
```

Q4.4: Code

```python
def get_image(bbox):
    # get the box from the image
    y1, x1, y2, x2 = bbox
    ex = bw[y1:y2, x1:x2]
    width = x2 - x1
    height = y2 - y1
    # pad the image based on if its height or width is larger
    if height > width:
        pad_amt = int(height/2)
        ex = np.pad(ex, (pad_amt, pad_amt), "constant")
    elif width > height:
        pad_amt = int(width/2)
        ex = np.pad(ex, (pad_amt, pad_amt), "constant")
    # plt.imshow(ex)
    # plt.show()

    # our image is reversed bw, so swap black and white
    where_0 = np.where(ex == 0)
    where_1 = np.where(ex == 1)
    ex[where_0] = 1
    ex[where_1] = 0
    # plt.imshow(ex)
    # plt.show()

    # erode (since image is bw, we have to erode to have the effect of dialation)
    if img == "04_deep.jpg":
        ex = cv2.erode(ex, np.ones((14, 14)), iterations=1)
    elif img == "02_letters.jpg":
        ex = cv2.erode(ex, np.ones((4, 4)), iterations=1)
    else:
        ex = cv2.erode(ex, np.ones((7, 7)), iterations=1)
    # resize to 32x32 and transpose in one step
    ex = cv2.resize(ex.T, (32, 32))

    # if img == "02_letters.jpg":
    #     plt.imshow(ex)
    #     plt.show()

    # flatten to get proper input size
    ex = ex.flatten()
    return ex

# for each row, sort the boxes and get the image the bbox represents
for row in range(len(rows)):
    rows[row] = sorted_arr(rows[row])
    for letter in range(len(rows[row])):
        rows[row][letter] = get_image(rows[row][letter])

# load the weights
# run the crops through your neural network and print them out
import pickle
import string
letters = np.array(
    [_ for _ in string.ascii_uppercase[:26]] + [str(_) for _ in range(10)])
params = pickle.load(open('q3_weights.pickle', 'rb'))
##########################
##### your code here #####
##########################
```

Q4.4: Code

```python
# create a dict that maps probs to characters
map_char = {0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F', 6: 'G', 7: 'H', 8: 'I', 9: 'J',
            10: 'K', 11: 'L', 12: 'M', 13: 'N', 14: 'O', 15: 'P', 16: 'Q', 17: 'R', 18: 'S', 19: 'T',
            20: 'U', 21: 'V', 22: 'W', 23: 'X', 24: 'Y', 25: 'Z', 26: '0', 27: '1', 28: '2', 29: '3',
            30: '4', 31: '5', 32: '6', 33: '7', 34: '8', 35: '9'}

for row in rows:

    # turn row into np array (letters x 1024)
    row = np.vstack(row)

    # do a forward pass in the row
    h1 = forward(row, params, 'layer1')
    probs = forward(h1, params, 'output', softmax)

    # create one long string to add guesses to
    row_letters = ""
    for i in range(probs.shape[0]):
        # for each letter in the row find the guess
        guess = np.argmax(probs[i, :])
        # turn the guess into a character
        row_letters += map_char[guess]
    # print the row
    print(row_letters)
print("\n")
```

Q4.4: Results

Ground Truth:	Predicted:	Added Spaces
TO DO LIST 1 MAKE A TO DO LIST 2 CHECK OFF THE FIRST THING ON TO DO LIST 3 REALIZE YOU HAVE ALREADY COMPLETED 2 THINGS 4 REWARD YOURSELF WITH A NAP	T0CCLIST IHAKEATOD0LIST 2CHECKOFETHFFIRST THINGONTODOLXST 2R2ALI2EYOUHVEALR2ADY COAPLET2D2YHIN0S 4REWAXDYOURSELF4ITH ANAP	T0 CC LIST I HAKE A TO D0 LIST 2 CHECK OFE THF FIRST THING ON TO DO LXST 2 R2ALI2E YOU HVE ALR2ADY COAPLET2D 2 YHIN0S 4 REWAXD YOURSELF 4ITH ANAP
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 1 2 3 4 5 6 7 8 9 0	XSCCCFC HIJKCKU CYUXST4 V4XYI XXS4SUJXYC	X S C C C F C H I J K C K U C Y U X S T 4 V 4 X Y I X X S 4 S U J X Y C
HAIKUS ARE EASY BUT SOMETIMES THEY DON'T MAKE SENSE REFRIDGERATOR	HAIRUSAREEASY BLTSOMETIHESTHEYUONTMAK2SENSR REFRI6ERATOR	HAIRUS ARE EASY BLT SOMETIHES THEY UONT MAK2 SENSR REFRI6ERATOR
DEEP LEARNING DEEPER LEARNING DEEPEST LEARNING	CFCYCEARNI44 UEEYFYLEARKIN6 UFFYESTLEARNING	CFCY CEARNI44 UEEYFY LEARKIN6 UFFYEST LEARNING

Comments:

The network seemed to perform fairly well for all samples, the worst performing by inspection was that of the letters image file. However, it did have accuracy in guessing some letters like in the second row (H, I, J, K…). I tdid not perform well on the numbers outside of four, but some of the mistakes were reasonble, like the guess of an S for 5 and a C for 0. The rest the images had reasonable estimations of the sentences. An interesting note is that I had to dialate the images differently, as different sample images had different pen strokes, some of the images required more dialation to more closely match the testing data set that the network was trained on.

Q6.1.1: Code

```
train_data = scipy.io.loadmat('../data/nist36_train.mat')
valid_data = scipy.io.loadmat('../data/nist36_valid.mat')
test_data = scipy.io.loadmat('../data/nist36_test.mat')

train_x, train_y = train_data['train_data'], train_data['train_labels']
valid_x, valid_y = valid_data['valid_data'], valid_data['valid_labels']
test_x, test_y = test_data['test_data'], test_data['test_labels']


# layer dims
examples = train_x.shape[0]
input_size = train_x.shape[1]
classes = train_y.shape[1]

# for the y data, change to examples x 1
train_y = np.argmax(train_y, 1)
valid_y = np.argmax(valid_y, 1)
test_y = np.argmax(test_y, 1)

# visualize
# ex = train_x[0, :]
# ex = ex[:, np.newaxis]
# print(ex.shape)
# ex = np.resize(ex, (32, 32))
# cv2.imshow("", ex)
# cv2.waitKey(0)
# cv2.destroyAllWindows()

# Generate the tensors from np arrays
train_x = torch.from_numpy(train_x).to(torch.float32)
train_y = torch.from_numpy(train_y)

valid_x = torch.from_numpy(valid_x).to(torch.float32)
valid_y = torch.from_numpy(valid_y)

test_x = torch.from_numpy(test_x).to(torch.float32)
test_y = torch.from_numpy(test_y)

# Create a tensor dataset object so we can create a dataloader object
train_ds = TensorDataset(train_x, train_y)
valid_ds = TensorDataset(valid_x, valid_y)
test_ds = TensorDataset(test_x, test_y)

# Create a data loader object for the loop
train_data = DataLoader(train_ds, batch_size=32,
                        shuffle=True)
valid_data = DataLoader(valid_ds, batch_size=32,
                        shuffle=True)
test_data = DataLoader(test_ds, batch_size=32,
                       shuffle=True)
```

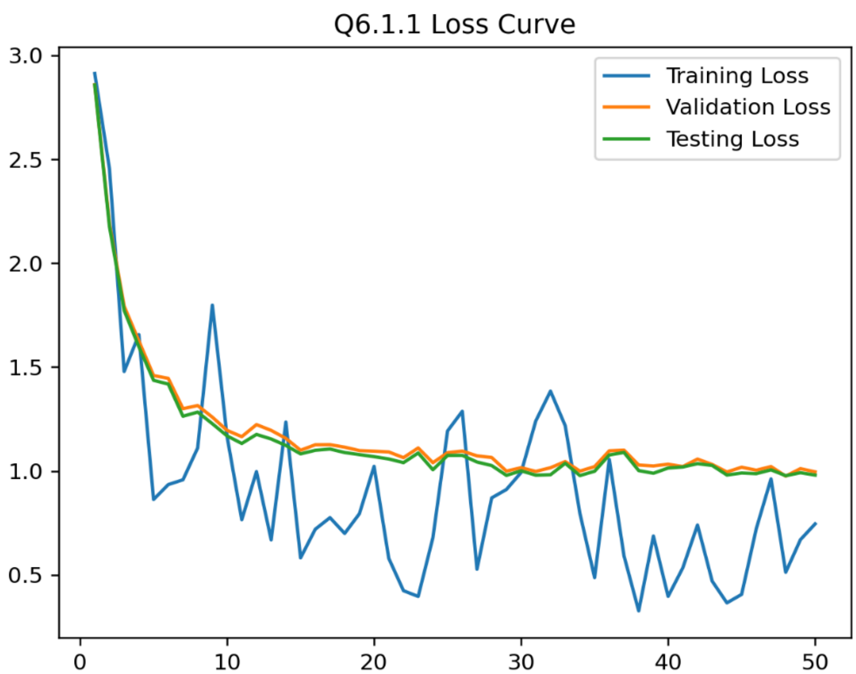
Q6.1.1: Code

```
max_iters = 50
# pick a batch size, learning rate
batch_size = 64
learning_rate = 2e-3
hidden_size = 64

model = torch.nn.Sequential(
    torch.nn.Linear(input_size, hidden_size),
    torch.nn.Sigmoid(),
    torch.nn.Linear(hidden_size, classes),
    torch.nn.LogSoftmax(dim=1))

print(model)

loss_funct = nn.NLLLoss()

optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

nn_params = {"training acc": 0,
             "training loss": 0,
             "validation acc": 0,
             "validation loss": 0,
             "testing acc": 0,
             "testing loss": 0
             }

nn_output = np.zeros(shape=(6, max_iters))

for i in range(max_iters):

    # ============== data training ====================
    iter_acc = 0
    iter_loss = 0

    for batch_train in train_data:

        batch_x, batch_y = batch_train

        optimizer.zero_grad()

        y_pred = model(batch_x)

        loss = loss_funct(y_pred, batch_y)
        iter_loss = loss.item()

        loss.backward()

        optimizer.step()

        _, predicted = torch.max(y_pred.data, 1)
        iter_acc += torch.sum(predicted == batch_y).item()

    iter_acc /= examples

    nn_params["training acc"] = (nn_params['training acc'] + iter_acc) / 2
    nn_params['training loss'] = iter_loss
```

6.1.1: Code

```
    # =========================== validation data ===========
    iter_acc = 0
    iter_loss = 0

    y_pred = model(valid_x)

    loss = loss_funct(y_pred, valid_y)
    iter_loss = loss.item()

    _, predicted = torch.max(y_pred.data, 1)

    iter_acc += torch.sum(predicted == valid_y).item()
    iter_acc /= valid_x.size(0)

    nn_params["validation acc"] = (nn_params['validation acc'] + iter_acc) / 2
    nn_params['validation loss'] = iter_loss

    # ===================== testing data ====================

    iter_acc = 0
    iter_loss = 0

    y_pred = model(test_x)

    loss = loss_funct(y_pred, test_y)
    iter_loss = loss.item()

    _, predicted = torch.max(y_pred.data, 1)

    iter_acc += torch.sum(predicted == test_y).item()
    iter_acc /= test_x.size(0)

    nn_params["testing acc"] = (nn_params['testing acc'] + iter_acc) / 2
    nn_params['testing loss'] = iter_loss

    print(
        f"Iter: {i} | Training Acc: {round(nn_params['training acc']*100, 2)}

    nn_output[0, i] = nn_params['training acc']
    nn_output[1, i] = nn_params['validation acc']
    nn_output[2, i] = nn_params['testing acc']
    nn_output[3, i] = nn_params['training loss']
    nn_output[4, i] = nn_params['validation loss']
    nn_output[5, i] = nn_params['testing loss']

# plotting
x = np.arange(1, max_iters + 1)

plt.figure(1)
plt.title("Q6.1.1 Accuracy Curve")
plt.plot(x, nn_output[0, :] * 100, label="Training Accuracy")
plt.plot(x, nn_output[1, :] * 100, label="Validation Accuracy")
plt.plot(x, nn_output[2, :] * 100, label="Testing Accuracy")
plt.legend()
```

Q6.1.1: Results



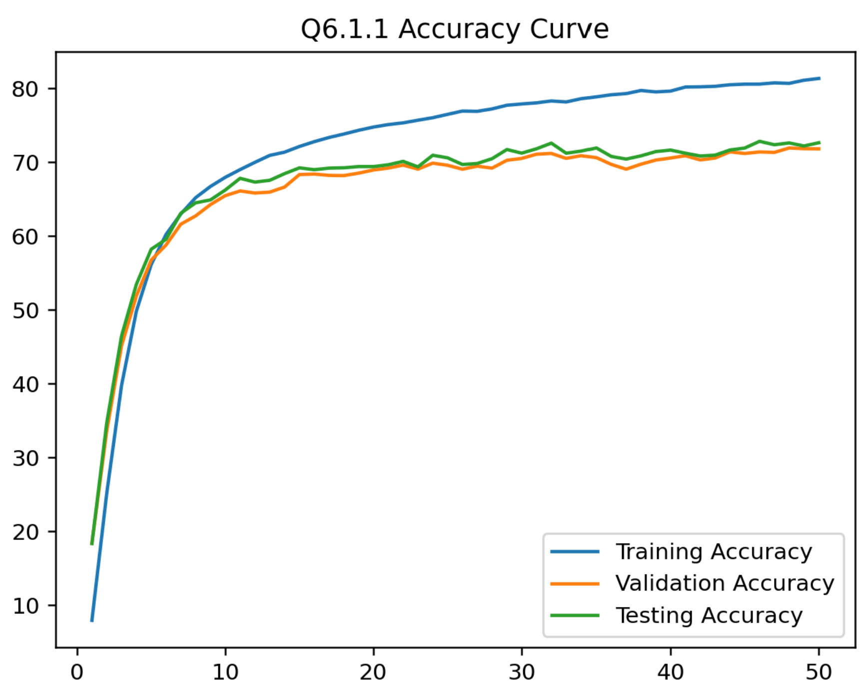
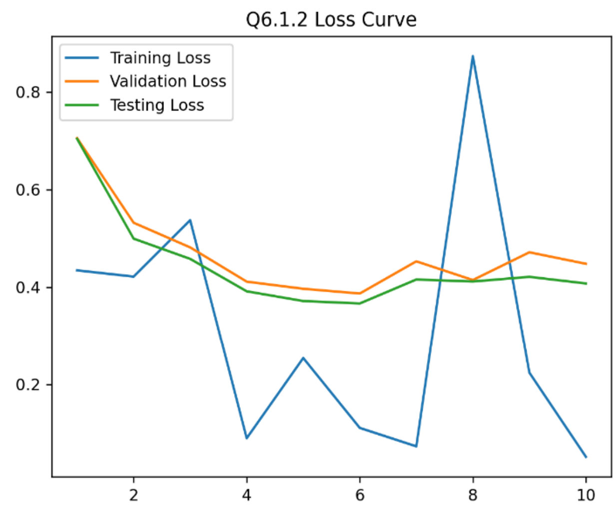


Figure 11. Loss & Accuracy curves for fully connected network recreated in Pytorch

Q6.1.2: Results



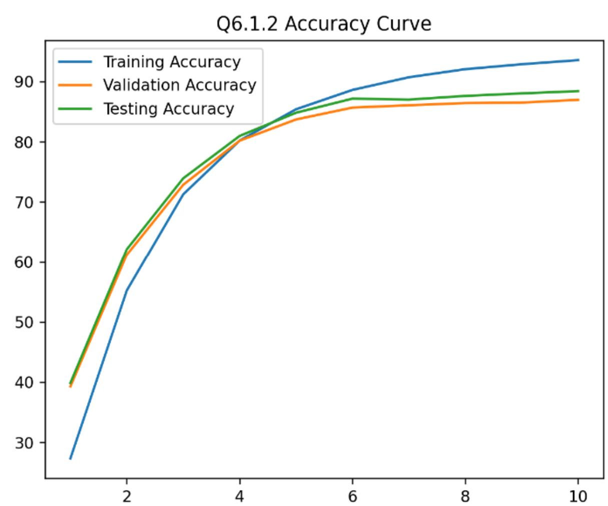


Figure 12. Loss & Accuracy curves for ConvNet on MNIST data

Comments:

Integrating convolutional layers, with the same hyperparamters (lr = 2e-3, batch size = 64, epochs = 50), allowed this neural net to converge to a higher lever of accuracy than the fully connected network five times quicker (10 epochs versus 50 epochs).

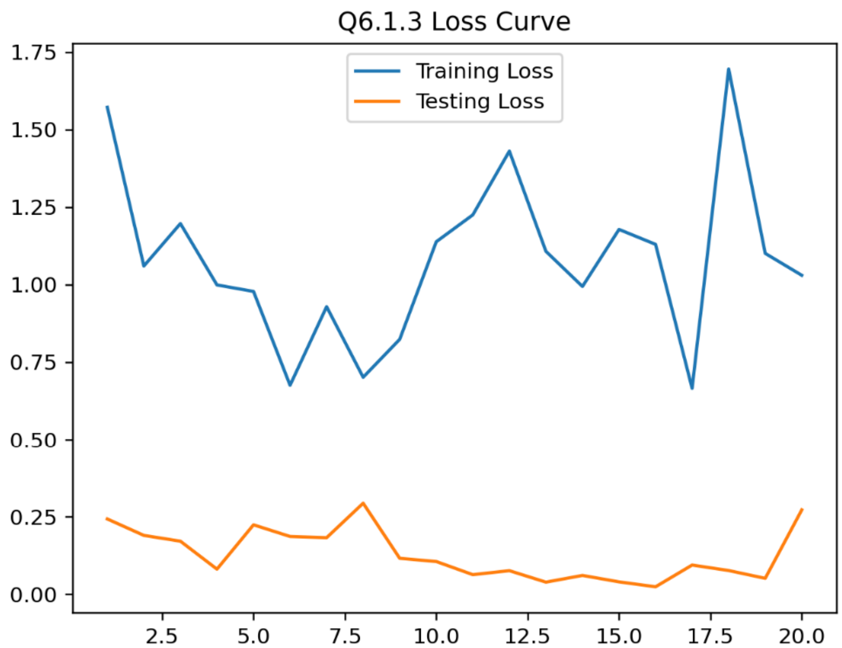
Q6.1.2: Code

```
max_iters = 10
# pick a batch size, learning rate
batch_size = 64
learning_rate = 2e-3
hidden_size = 150

model = torch.nn.Sequential(
    torch.nn.Conv2d(in_channels=1, out_channels=7, kernel_size=3),
    torch.nn.ReLU(),
    torch.nn.MaxPool2d(kernel_size=2, stride=2),
    torch.nn.Conv2d(in_channels=7, out_channels=13, kernel_size=5),
    torch.nn.ReLU(),
    torch.nn.MaxPool2d(kernel_size=2, stride=2),
    torch.nn.Flatten(),
    torch.nn.Linear(325, hidden_size),
    torch.nn.ReLU(),
    torch.nn.Linear(hidden_size, classes),
    torch.nn.LogSoftmax(dim=1))

loss_funct = nn.NLLLoss()

optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

nn_params = {"training acc": 0,
             "training loss": 0,
             "validation acc": 0,
             "validation loss": 0,
             "testing acc": 0,
             "testing loss": 0
             }

nn_output = np.zeros(shape=(6, max_iters))

for i in range(max_iters):

    # ============== data training ===================
    iter_acc = 0
    iter_loss = 0

    for batch_train in train_data:

        batch_x, batch_y = batch_train

        batch_x = batch_x.view(-1, 1, 32, 32)

        optimizer.zero_grad()

        y_pred = model(batch_x)

        loss = loss_funct(y_pred, batch_y)
        iter_loss = loss.item()

        loss.backward()

        optimizer.step()

        _, predicted = torch.max(y_pred.data, 1)
        iter_acc += torch.sum(predicted == batch_y).item()

    iter_acc /= examples

    nn_params["training acc"] = (nn_params['training acc'] + iter_acc) / 2
    nn_params['training loss'] = iter_loss
```

Q6.1.2: Code

```
# ============================ validation data ===========
iter_acc = 0
iter_loss = 0

valid_x = valid_x.view(-1, 1, 32, 32)

y_pred = model(valid_x)

loss = loss_funct(y_pred, valid_y)
iter_loss = loss.item()

_, predicted = torch.max(y_pred.data, 1)

iter_acc += torch.sum(predicted == valid_y).item()
iter_acc /= valid_x.size(0)

nn_params["validation acc"] = (nn_params['validation acc'] + iter_acc) / 2
nn_params['validation loss'] = iter_loss

# ======================= testing data =====================

iter_acc = 0
iter_loss = 0

test_x = test_x.view(-1, 1, 32, 32)

y_pred = model(test_x)

loss = loss_funct(y_pred, test_y)
iter_loss = loss.item()

_, predicted = torch.max(y_pred.data, 1)

iter_acc += torch.sum(predicted == test_y).item()
iter_acc /= test_x.size(0)

nn_params["testing acc"] = (nn_params['testing acc'] + iter_acc) / 2
nn_params['testing loss'] = iter_loss

print(
    f"Iter: {i} | Training Acc: {round(nn_params['training acc']*100, 2)}

nn_output[0, i] = nn_params['training acc']
nn_output[1, i] = nn_params['validation acc']
nn_output[2, i] = nn_params['testing acc']
nn_output[3, i] = nn_params['training loss']
nn_output[4, i] = nn_params['validation loss']
nn_output[5, i] = nn_params['testing loss']
```

Q6.1.3: Results



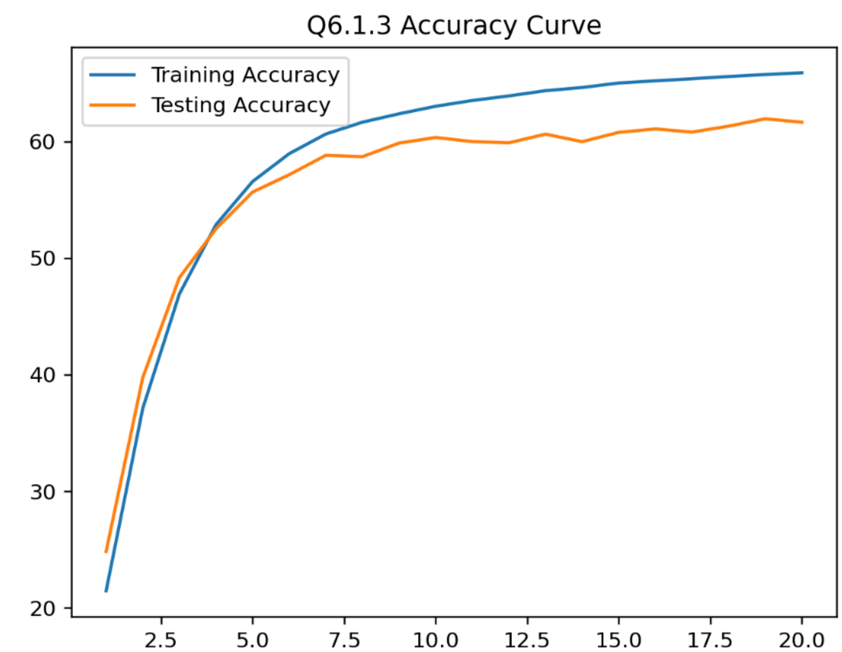


Figure 13. Loss & Accuracy curve for CIFAR-10 ConvNet

Q6.1.3: Code

```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

batch_size = 10

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)


trainloader = DataLoader(trainset, batch_size=batch_size,
                         shuffle=True)


batch_size = 10

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = DataLoader(testset, shuffle=False)
classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

# establish params
max_iters = 20
# pick a batch size, learning rate
learning_rate = 1e-3
hidden_size = 64

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

model = torch.nn.Sequential(
    torch.nn.Conv2d(in_channels=3, out_channels=6, kernel_size=5),
    torch.nn.ReLU(),
    torch.nn.MaxPool2d(kernel_size=2, stride=2),
    torch.nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5),
    torch.nn.ReLU(),
    torch.nn.MaxPool2d(kernel_size=2, stride=2),
    torch.nn.Flatten(),
    torch.nn.Linear(16*25, 20),
    torch.nn.ReLU(),
    torch.nn.Linear(20, 10),
    torch.nn.LogSoftmax(dim=1)).to(device)

loss_funct = nn.NLLLoss()

optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

Q6.1.3: Code

```
nn_params = {"training acc": 0,
             "training loss": 0,
             "validation acc": 0,
             "validation loss": 0,
             "testing acc": 0,
             "testing loss": 0
             }

nn_output = np.zeros(shape=(6, max_iters))

for i in range(max_iters):

    print("Begin Iteration:", i)

    # ============== data training ===================
    iter_acc = 0
    iter_loss = 0

    for batch_train in trainloader:

        batch_x, batch_y = batch_train

        optimizer.zero_grad()

        y_pred = model(batch_x)

        loss = loss_funct(y_pred, batch_y)
        iter_loss = loss.item()

        loss.backward()

        optimizer.step()

        _, predicted = torch.max(y_pred.data, 1)
        iter_acc += torch.sum(predicted == batch_y).item()

    iter_acc /= trainset.data.shape[0]

    nn_params["training acc"] = (nn_params['training acc'] + iter_acc) / 2
    nn_params['training loss'] = iter_loss

    print("Training Iteration", i, "Complete.")

    # ====================== testing data ====================

    iter_acc = 0
    iter_loss = 0

    for batch_test in testloader:

        test_x, test_y = batch_test

        y_pred = model(test_x)

        loss = loss_funct(y_pred, test_y)
        iter_loss = loss.item()

        _, predicted = torch.max(y_pred.data, 1)

        iter_acc += torch.sum(predicted == test_y).item()

    iter_acc /= testset.data.shape[0]

    print("Testing Iteration", i, "Complete.")

    nn_params["testing acc"] = (nn_params['testing acc'] + iter_acc) / 2
    nn_params['testing loss'] = iter_loss
```

Q6.1.4: Results

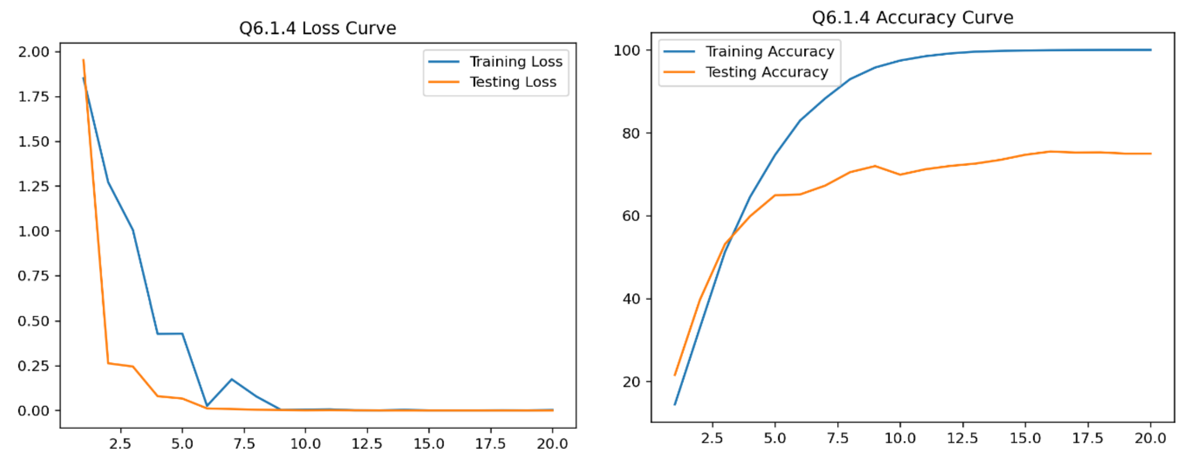


Figure 14. Loss & Accuracy curves for ConvNet on SUN Dataset

Comments:

- Neural Network Approach to Learning:
    - Hyperparameters:
        - Learning Rate: 1e-3
        - Batch Size: 128
        - Epochs: 20
    - Results:
        - Training Accuracy: 99.9%
        - Validation Accuracy: 74.9%
- Classical BOW Results:
    - 67.2%
- Here we saw that the neural network approach performed roughly 8% better than the classical approach the same when you just consider the testing data, but blew the classical approach out of the water in terms of the training data. Since I just used the data set from HW, I only had 160 samples to learn on for the training set. This may have led to the lack of performance on the testing data, if I had had more time, I would've liked to find a way to obtain more training data and added more generalizations to the data so that it would have been able to transfer its performance from training to testing and not overfit to the training set.

Q6.1.4: Code

```
batch_size = 128

testval_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225]),
    transforms.Resize((256, 256))
])


train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225]),
    transforms.Resize((256, 256))
])


device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# ==================== Load Data =====================

# ? Training Data
trainset = torchvision.datasets.ImageFolder(
    root='../data/oxford-flowers17/train', transform=train_transform)


trainloader = DataLoader(trainset, batch_size=batch_size,
                         shuffle=True, pin_memory=True)

# ? Testing Data
testset = torchvision.datasets.ImageFolder(
    root='../data/oxford-flowers17/test', transform=testval_transform)
testloader = DataLoader(testset, pin_memory=True)

# ? Validation Data
valset = torchvision.datasets.ImageFolder(
    root='../data/oxford-flowers17/val', transform=testval_transform)
valloader = DataLoader(valset, pin_memory=True)

# establish params
max_iters = 50
# pick a batch size, learning rate
hidden_size = 64
learning_rate = 1e-3
classes = 17

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

Q6.1.4: Code

```python
model = torch.nn.Sequential(
    torch.nn.Conv2d(in_channels=3, out_channels=6, kernel_size=5),
    torch.nn.ReLU(),
    torch.nn.MaxPool2d(kernel_size=2, stride=2),
    torch.nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5),
    torch.nn.ReLU(),
    torch.nn.MaxPool2d(kernel_size=2, stride=2),
    torch.nn.Flatten(),
    torch.nn.Linear(59536, 800),
    torch.nn.ReLU(),
    torch.nn.Linear(800, classes),
    torch.nn.ReLU(),
).to(device)

loss_funct = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

nn_params = {"training acc": 0,
             "training loss": 0,
             "validation acc": 0,
             "validation loss": 0,
             "testing acc": 0,
             "testing loss": 0
             }

nn_output = np.zeros(shape=(6, max_iters))

for i in range(max_iters):

    # ============== data training ===================
    iter_acc = 0
    iter_loss = 0

    for batch_train in trainloader:

        batch_x, batch_y = batch_train

        optimizer.zero_grad()

        y_pred = model(batch_x)

        loss = loss_funct(y_pred, batch_y)
        iter_loss = loss.item()

        loss.backward()

        optimizer.step()

        _, predicted = torch.max(y_pred.data, 1)
        iter_acc += torch.sum(predicted == batch_y).item()

    iter_acc /= len(trainset.imgs)

    nn_params["training acc"] = (nn_params['training acc'] + iter_acc) / 2
    nn_params['training loss'] = iter_loss

    print(f"Training Iteration {i} complete")
```

Q6.1.4: Code

```
# ========================== validation data ===========
iter_acc = 0
iter_loss = 0

for batch_val in valloader:

    batch_x, batch_y = batch_val

    y_pred = model(batch_x)

    loss = loss_funct(y_pred, batch_y)
    iter_loss = loss.item()

    optimizer.step()

    _, predicted = torch.max(y_pred.data, 1)
    iter_acc += torch.sum(predicted == batch_y).item()

iter_acc /= len(valset.imgs)

nn_params["validation acc"] = (nn_params['validation acc'] + iter_acc) / 2
nn_params['validation loss'] = iter_loss

print(f"Validation Iteration {i} complete")

# ====================== testing data ====================

iter_acc = 0
iter_loss = 0

for batch_test in testloader:

    batch_x, batch_y = batch_test

    y_pred = model(batch_x)

    loss = loss_funct(y_pred, batch_y)
    iter_loss = loss.item()

    _, predicted = torch.max(y_pred.data, 1)
    iter_acc += torch.sum(predicted == batch_y).item()

iter_acc /= len(testset.imgs)

nn_params["testing acc"] = (nn_params['testing acc'] + iter_acc) / 2
nn_params['testing loss'] = iter_loss

print(
    f"Iter: {i+1} | Training Acc: {round(nn_params['training acc']*100, 2)} |

nn_output[0, i] = nn_params['training acc']
nn_output[1, i] = nn_params['validation acc']
nn_output[2, i] = nn_params['testing acc']
nn_output[3, i] = nn_params['training loss']
nn_output[4, i] = nn_params['validation loss']
nn_output[5, i] = nn_params['testing loss']
```

Q6.2: SqueezeNet Code

```
batch_size = 128  # 64

testval_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Resize((256, 256)),
])

# train_transform = transforms.Compose([
#     transforms.RandomResizedCrop(224),
#     transforms.RandomHorizontalFlip(),
#     transforms.ToTensor(),
# ])

train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225]),
    transforms.Resize((256, 256))
])


device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# ==================== Load Data =====================

# ? Training Data
trainset = torchvision.datasets.ImageFolder(
    root='../data/oxford-flowers17/train', transform=train_transform)


trainloader = DataLoader(trainset, batch_size=batch_size,
                         shuffle=True)

# ? Testing Data
testset = torchvision.datasets.ImageFolder(
    root='../data/oxford-flowers17/test', transform=testval_transform)
testloader = DataLoader(testset)

# ? Validation Data
valset = torchvision.datasets.ImageFolder(
    root='../data/oxford-flowers17/val', transform=testval_transform)
valloader = DataLoader(valset)

# establish params
max_iters = 30
# pick a batch size, learning rate
learning_rate = 1e-5

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

Q6.2 SqueezeNet Code:

```
model = squeezenet1_1(pretrained=True).to(device=device)

for param in model.parameters():
    param.requires_grad = False

# replace the last layer so that it outputs the correcet number of classes
final_conv = nn.Conv2d(512, 17, kernel_size=1)
model.classifier = nn.Sequential(
    nn.Dropout(p=0.5), final_conv, nn.ReLU(
        inplace=True), nn.AdaptiveAvgPool2d((1, 1))
)

for param in model.classifier.parameters():
    param.requires_grad = True

loss_funct = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

nn_params = {"training acc": 0,
             "training loss": 0,
             "validation acc": 0,
             "validation loss": 0,
             "testing acc": 0,
             "testing loss": 0
             }

nn_output = np.zeros(shape=(6, max_iters))

for i in range(max_iters):

    # ============== data training ===================
    iter_acc = 0
    iter_loss = 0

    for batch_train in trainloader:

        batch_x, batch_y = batch_train

        optimizer.zero_grad()

        y_pred = model(batch_x)

        loss = loss_funct(y_pred, batch_y)
        iter_loss = loss.item()

        loss.backward()

        optimizer.step()

        _, predicted = torch.max(y_pred.data, 1)
        iter_acc += torch.sum(predicted == batch_y).item()

    iter_acc /= len(trainset.imgs)

    nn_params["training acc"] = (nn_params['training acc'] + iter_acc) / 2
    nn_params['training loss'] = iter_loss

    print(f"Training Iteration {i+1} complete")

    # =========================== validation data ==========
```

Q6.2 SqueezeNet Code:

```python
# ============================ validation data ===========
iter_acc = 0
iter_loss = 0

for batch_val in valloader:

    batch_x, batch_y = batch_val

    y_pred = model(batch_x)

    loss = loss_funct(y_pred, batch_y)
    iter_loss = loss.item()

    _, predicted = torch.max(y_pred.data, 1)
    iter_acc += torch.sum(predicted == batch_y).item()

iter_acc /= len(valset.imgs)

nn_params["validation acc"] = (nn_params['validation acc'] + iter_acc) / 2
nn_params['validation loss'] = iter_loss

print(f"Validation Iteration {i+1} complete")

# ======================= testing data =====================

iter_acc = 0
iter_loss = 0

for batch_test in testloader:

    batch_x, batch_y = batch_test

    y_pred - model(batch_x)

    loss = loss_funct(y_pred, batch_y)
    iter_loss = loss.item()

    _, predicted = torch.max(y_pred.data, 1)
    iter_acc += torch.sum(predicted == batch_y).item()

iter_acc /= len(testset.imgs)

nn_params["testing acc"] = (nn_params['testing acc'] + iter_acc) / 2
nn_params['testing loss'] = iter_loss

print(f"Test Iteration {i+1} complete")

print(
    f"Iter: {i+1} | Training Acc: {round(nn_params['training acc']*100, 2)} |
```

Q6.2: Custom Network

```python
batch_size = 128  # 64

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225]),
    transforms.Resize((256, 256))
])

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# =================== Load Data ====================

# ? Training Data
trainset = torchvision.datasets.ImageFolder(
    root='../data/oxford-flowers17/train', transform=transform)


trainloader = DataLoader(trainset, batch_size=batch_size,
                         shuffle=True)

# ? Testing Data
testset = torchvision.datasets.ImageFolder(
    root='../data/oxford-flowers17/test', transform=transform)
testloader = DataLoader(testset)

# ? Validation Data
valset = torchvision.datasets.ImageFolder(
    root='../data/oxford-flowers17/val', transform=transform)
valloader = DataLoader(valset)

# establish params
max_iters = 30
# pick a batch size, learning rate
learning_rate = 1e-3

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

model = torch.nn.Sequential(
    torch.nn.Conv2d(in_channels=3, out_channels=6, kernel_size=5),
    torch.nn.ReLU(),
    torch.nn.MaxPool2d(kernel_size=2, stride=2),
    torch.nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5),
    torch.nn.ReLU(),
    torch.nn.MaxPool2d(kernel_size=2, stride=2),
    torch.nn.Flatten(),
    torch.nn.Linear(59536, 1745),
    torch.nn.ReLU(),
    torch.nn.Linear(1745, 17),
    torch.nn.LogSoftmax(dim=1)).to(device)

loss_funct = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

Q6.2: Custom Network

```python
nn_params = {"training acc": 0,
             "training loss": 0,
             "validation acc": 0,
             "validation loss": 0,
             "testing acc": 0,
             "testing loss": 0
             }

nn_output = np.zeros(shape=(6, max_iters))

for i in range(max_iters):

    # =============== data training ====================
    iter_acc = 0
    iter_loss = 0

    for batch_train in trainloader:

        batch_x, batch_y = batch_train

        optimizer.zero_grad()

        y_pred = model(batch_x)

        loss = loss_funct(y_pred, batch_y)
        iter_loss = loss.item()

        loss.backward()

        optimizer.step()

        _, predicted = torch.max(y_pred.data, 1)
        iter_acc += torch.sum(predicted == batch_y).item()

    iter_acc /= len(trainset.imgs)

    nn_params["training acc"] = (nn_params['training acc'] + iter_acc) / 2
    nn_params['training loss'] = iter_loss

    print(f"Training Iteration {i+1} complete")
```

Q6.2: Custom Network

```
# ============================ validation data ===========
iter_acc = 0
iter_loss = 0

for batch_val in valloader:

    batch_x, batch_y = batch_val

    y_pred = model(batch_x)

    loss = loss_funct(y_pred, batch_y)
    iter_loss = loss.item()

    _, predicted = torch.max(y_pred.data, 1)
    iter_acc += torch.sum(predicted == batch_y).item()

iter_acc /= len(valset.imgs)

nn_params["validation acc"] = (nn_params['validation acc'] + iter_acc) / 2
nn_params['validation loss'] = iter_loss

print(f"Validation Iteration {i+1} complete")

# ====================== testing data =====================

iter_acc = 0
iter_loss = 0

for batch_test in testloader:

    batch_x, batch_y = batch_test

    y_pred = model(batch_x)

    loss = loss_funct(y_pred, batch_y)
    iter_loss = loss.item()

    _, predicted = torch.max(y_pred.data, 1)
    iter_acc += torch.sum(predicted == batch_y).item()

iter_acc /= len(testset.imgs)

nn_params["testing acc"] = (nn_params['testing acc'] + iter_acc) / 2
nn_params['testing loss'] = iter_loss

print(f"Test Iteration {i+1} complete")

print(
    f"Iter: {i+1} | Training Acc: {round(nn_params['training acc']*100, 2)} |
```

Q6.2 Results

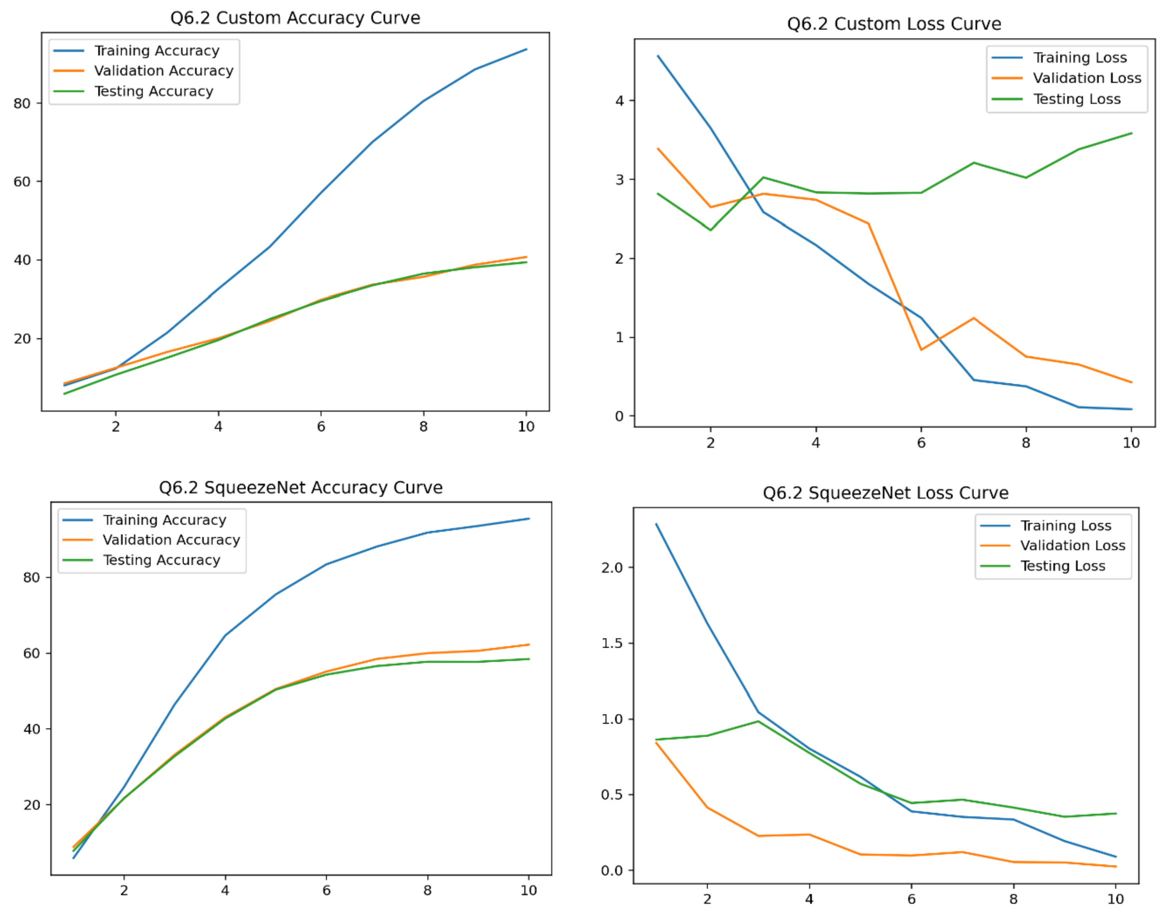


Figure 15. Loss & Accuracy Curves for SqueezeNet (Bottom) and custom ConvNet (Top)

Comments

- Both networks were run for 10 epochs, in which the SqueezeNet performed on the test and validation accuracy. It seems that the SqueezeNet was able to generalize better to nontraining style data, displaying its robustness in learning.
- SqueezeNet Accuracies:
    - Train: 95.4%
    - Test: 58.4
    - Validation: 62.2%
- Custom Network:
    - Train: 93.6%
    - Test: 36.8%
    - Validation: 37.6%