Homework 5: Neural Network for Recognition

Srividya Gandikota
sgandiko
`sgandiko@andrew.cmu.edu`

1 Theory

1.1 Prove that softmax is invariant to translation, that is softmax(x) = softmax(x + c).

Given that we know a vector x, for each index i, we know that softmax$(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$, we tend to use c = -max x_i because x + c is only going be a value between two things, either equal to 0 or a negative number. This one feature is helpful in ensuring that the results will not experience instability because we make sure that the largest value in x becomes 0 after we adjust.

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

$$\downarrow$$

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

$$\downarrow$$
$$\text{softmax}(x) = \text{softmax}(x + c)$$

1.2 Softmax can be written as a three step processes:

1. As x $\in$ Rd, what are the properties of softmax(x), namely what is the range of each element? What is the sum over all elements?
   - The values for the softmax range are between 0 and 1, while the sum over all elements is equal to 1.
2. One could say that "softmax takes an arbitrary real valued vector x and turns it into a ____."
   - Softmax takes an arbitrary real valued vector x and turns it into a probability distribution.
3. Can you see the role of each step in the multi-step process now? Explain them.
   - The step of taking the exponential of each value is to find the outcome frequency, the sum is taken to find the total frequency, and the normalization or division is taken to get the probability of each occurrence.

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

When looking at a forward pass of a fully connected layer, we need to use the equation y = wx + b. When we take this and apply it to a multa layer process, we see that the ending equation is in the same form as the initial one, which is helpful in proving that the multi layer neural network without a non linear activation function is equivalient to linear regression.

$$\begin{gathered} y_l = w_l x_l + b_l \\ = w_l(w_{l-1}x_{l-1} + b_{l-1}) + b_l \\ = w' x_{l-1} + b' \\ \downarrow \\ y_l = wx + b \end{gathered}$$

1.4 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 σ(x) can be found below.

$$
\begin{gathered}
\frac{d}{dx}\sigma(x) = (\sigma(x))(1 - \sigma(x)) \\
\text{If } \sigma(x) = \frac{1}{1+e^{-x}},\ \frac{d}{dx}\sigma(x) = \frac{d}{dx}\frac{1}{1+e^{-x}} \\
\frac{d}{dx}\frac{1}{1+e^{-x}} = \frac{d}{dx}1 + \frac{d}{dx}e^{-x} = -e^{-x} \\
\downarrow \\
\frac{d}{dx}\sigma(x) = \frac{1}{1+e^{-x}}\frac{-1}{1+e^{-x}} = \frac{1}{1+e^{-x}}^{2} \\
\downarrow \\
\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x))
\end{gathered}
$$

1.5 Given y = W x + b and the gradient of some loss J with respect y, show how to get the three partial derivatives below.

Given y = Wx + b and the gradient of some loss J with respect y, the derivations for the three partial derivatives $\frac{\partial J}{\partial W}$, $\frac{\partial J}{\partial x}$, and $\frac{\partial J}{\partial b}$can be found below. The values are taken the transpose of in order to make the matrix multiplication math work out.

- $\frac{\partial J}{\partial W}$

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

- $\frac{\partial J}{\partial x}$

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

- $\frac{\partial J}{\partial b}$

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial b}$$
$$\frac{\partial J}{\partial y} = \sigma$$
$$\frac{\partial y}{\partial b} = 1$$
$$\downarrow$$
$$\frac{\partial J}{\partial b} = \sigma$$

1.6 When the neural network applies the elementwise activation function (such as sigmoid), the gradient of the activation function scales the backpropogation update.

1. Consider the sigmoid activation function for deep neural networks. Why might it lead to a "vanishing gradient" problem if it is used for many layers (consider plotting Q1.4)?

   - The sigmoid activation function only has outputs between 0,1 and its derivate only outputs between 0 and 0.25. This means that if we are use these small values over the multi-layers, we see the gradients to continually decrease until they disappear.

2. Often it is replaced with tanh(x) = $\frac{1-e-2x}{1+e-2x}$. What are the output ranges of both tanh and sigmoid? Why might we prefer tanh ?

   - The output range of the sigmoid function is between 0 and 1, while the output of the tanh function is between -1 and 1. We might prefer the tanh function because we can avoid the vanishing gradient and also have the ability to converge faster because the gradient can go both positive and negative.

3. Why does tanh(x) have less of a vanishing gradient problem? (plotting the derivatives helps! for reference: tanh'(x) = 1 - $tanh(x)^2$).

   - Since the derivative of the tanh function has a range between 0 and 1, it has a four times larger range to work with instead of the sigmoid function, meaning that the gradient would drop slower and vanish less than the alternate.

4. tanh is a scaled and shifted version of the sigmoid. Show how tanh(x) can be written in terms of σ(x).

   - Given that $\sigma(x) = \dfrac{1}{1+e^{-x}}$, we can use that as $\dfrac{1-e^{-x}}{1+e^{-x}} = 2\sigma(x) - 1$. This can simplify down to tanh(x) so we can see that $tanh(x) = \dfrac{1-e^{-2x}}{1+e^{-2x}} = 2\sigma(2x) - 1$.

2 Implement a Fully Connected Network

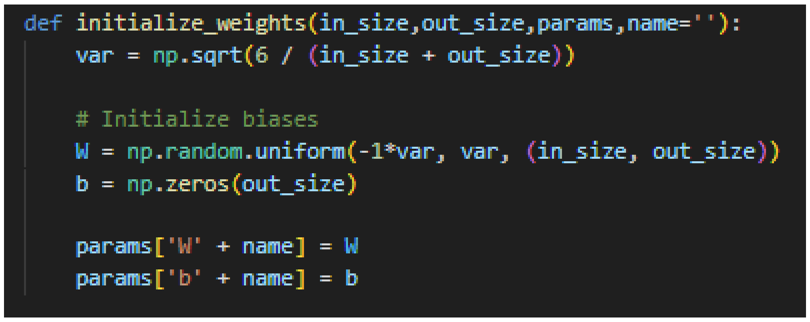
2.1 Network Initialization

2.1.1 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?

It is typically not a great idea to initialize a network with all zeros because if all of the weights are 0, all the neurons in the layer will end up matching or learning the same thing. This means that in the end, the network will be symmetric. If you initialize a network with all zeros and train on any dataset, the output would basically be like applying a linear transformation on the entire thing.

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

The implementation of the initialization function can be found in Figure 1:



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

    # Initialize biases
    W = np.random.uniform(-1*var, var, (in_size, out_size))
    b = np.zeros(out_size)

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

Figure 1: Xavier Initialization

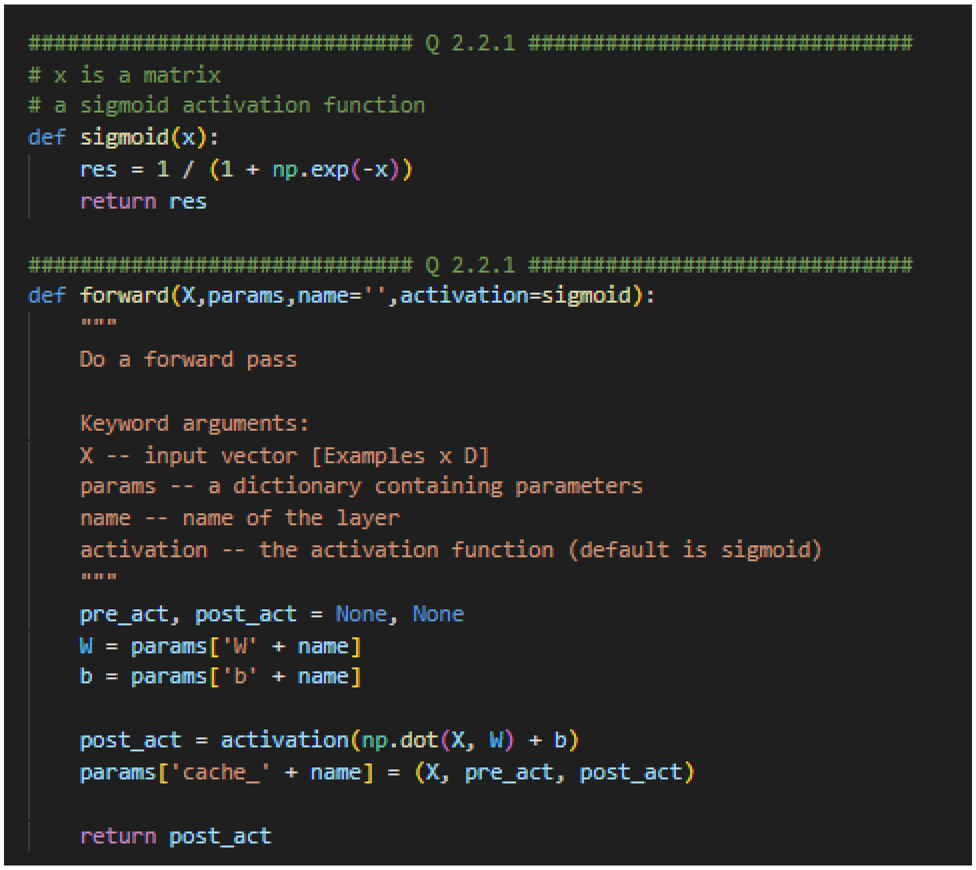
2.1.3 Why do we initialize with random numbers? Why do we scale the initialization depending on layer size (see near Fig 6 in the paper)?

Initializing the network with random numbers can help avoid getting the same computations from each layer or the symmetry mentioned before. This method also helps the random initialization avoid stopping at the local minima instead of the absolute minima. We scale the initialization depending on layer size as a method to help mitigate vanishing gradients and to also the variance at where we want during propagation.

2.2 Forward Propagation

2.2.1 In python/nn.py, implement sigmoid, along with forward prop- agation for a single layer with an activation function, namely $y = \sigma(XW + b)$, returning the output and intermediate results for an $N \times D$ dimension input X, with examples along the rows, data dimensions along the columns.

The implementation of the sigmoid/forward propagation function can be found in Figure 2:

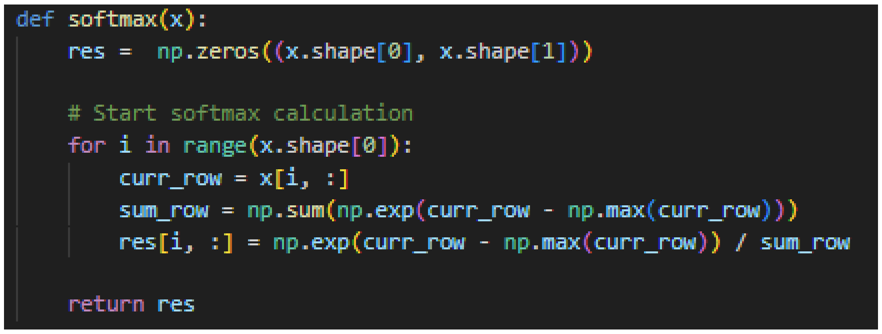


```
############################## Q 2.2.1 ##############################
# x is a matrix
# a sigmoid activation function
def sigmoid(x):
    res = 1 / (1 + np.exp(-x))
    return res

############################## Q 2.2.1 ##############################
def forward(X,params,name='',activation=sigmoid):
    """
    Do a forward pass

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

    post_act = activation(np.dot(X, W) + b)
    params['cache_' + name] = (X, pre_act, post_act)

    return post_act
```

Figure 2: Sigmoid and Forward Propagation Function

2.2.2 In python/nn.py, implement the softmax function. Be sure to use the numerical stability trick you derived in Q1.1 softmax.

The implementation of the softmax function can be found in Figure 3:

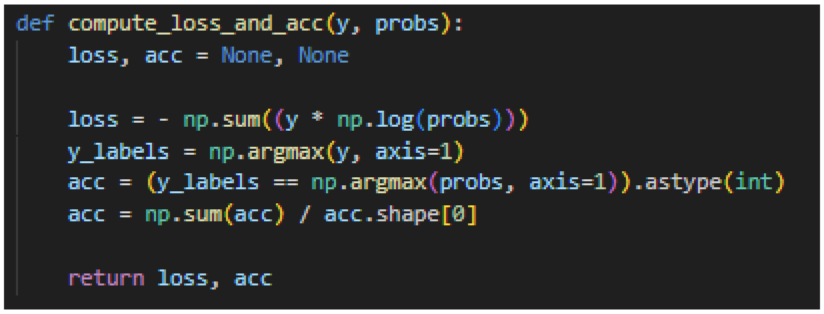


```python
def softmax(x):
    res =  np.zeros((x.shape[0], x.shape[1]))

    # Start softmax calculation
    for i in range(x.shape[0]):
        curr_row = x[i, :]
        sum_row = np.sum(np.exp(curr_row - np.max(curr_row)))
        res[i, :] = np.exp(curr_row - np.max(curr_row)) / sum_row

    return res
```

Figure 3: Softmax Function

2.2.3 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 implementation of the accuracy and loss function can be found in Figure 4:

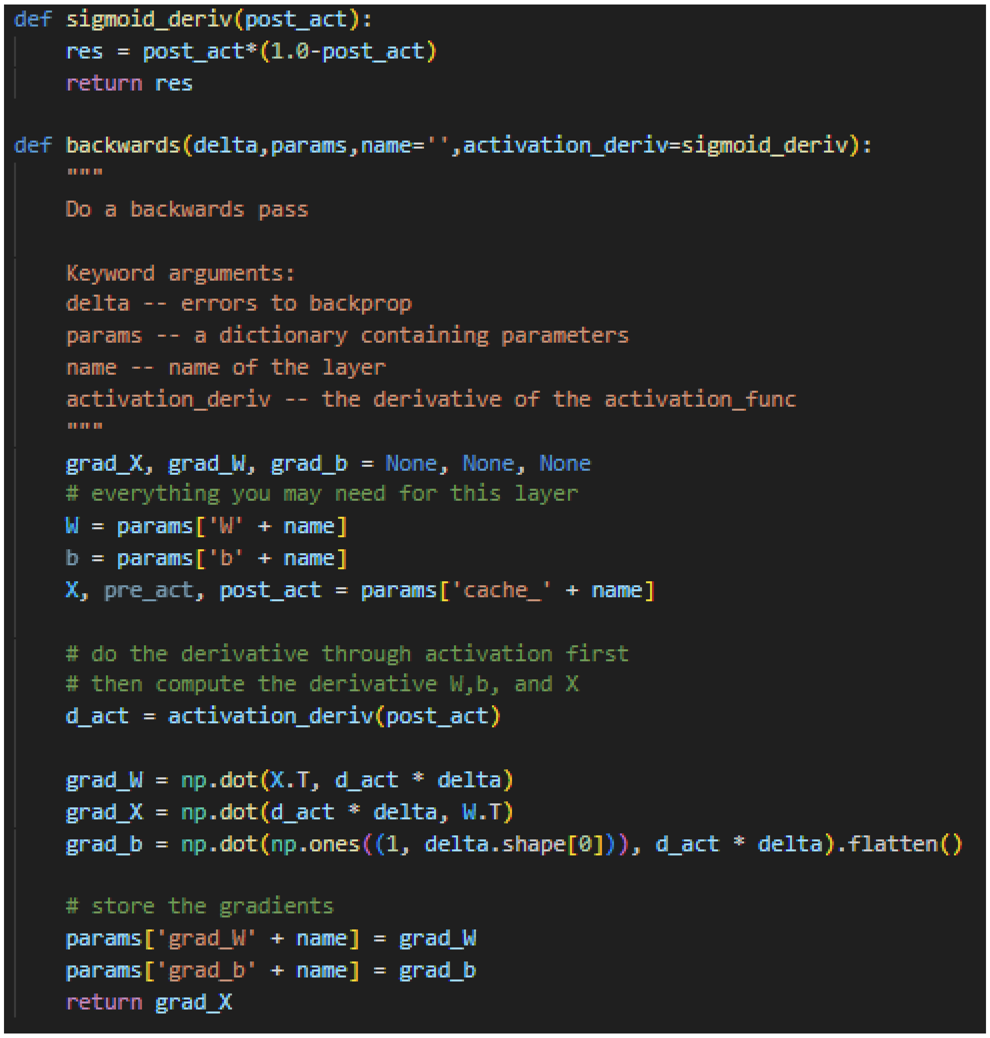


```python
def compute_loss_and_acc(y, probs):
    loss, acc = None, None

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

    return loss, acc
```

Figure 4: Compute Loss and Accuracy Function

2.3 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.

The implementation of the backpropagation function can be found in Figure 5:

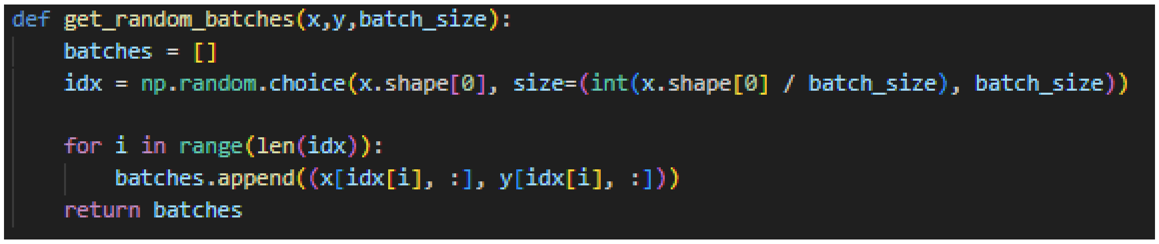


```python
def sigmoid_deriv(post_act):
    res = post_act*(1.0-post_act)
    return res

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

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

    # do the derivative through activation first
    # then compute the derivative W,b, and X
    d_act = activation_deriv(post_act)

    grad_W = np.dot(X.T, d_act * delta)
    grad_X = np.dot(d_act * delta, W.T)
    grad_b = np.dot(np.ones((1, delta.shape[0])), d_act * delta).flatten()

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

Figure 5: Backpropagation Function

2.4 In python/nn.py, write a function that takes the entire dataset (x and y) as input, and then split the dataset into random batches. In python/run q2.py, write a training loop that iterates over the random batches, does forward and backward propagation, and applies a gradient update step.

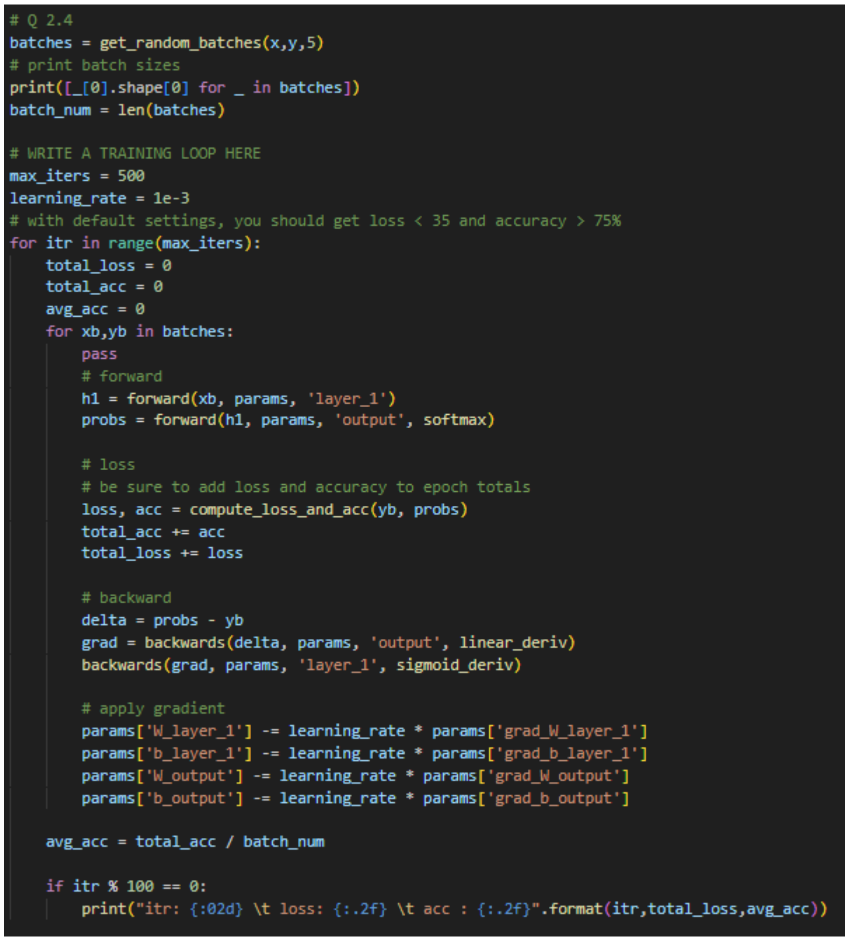
The implementation of the random batch splitting function can be found in Figure 6:



```python
def get_random_batches(x,y,batch_size):
    batches = []
    idx = np.random.choice(x.shape[0], size=(int(x.shape[0] / batch_size), batch_size))

    for i in range(len(idx)):
        batches.append((x[idx[i], :], y[idx[i], :]))
    return batches
```

Figure 6: Split into Batches Function

The implementation of the training loop function can be found in Figure 7:

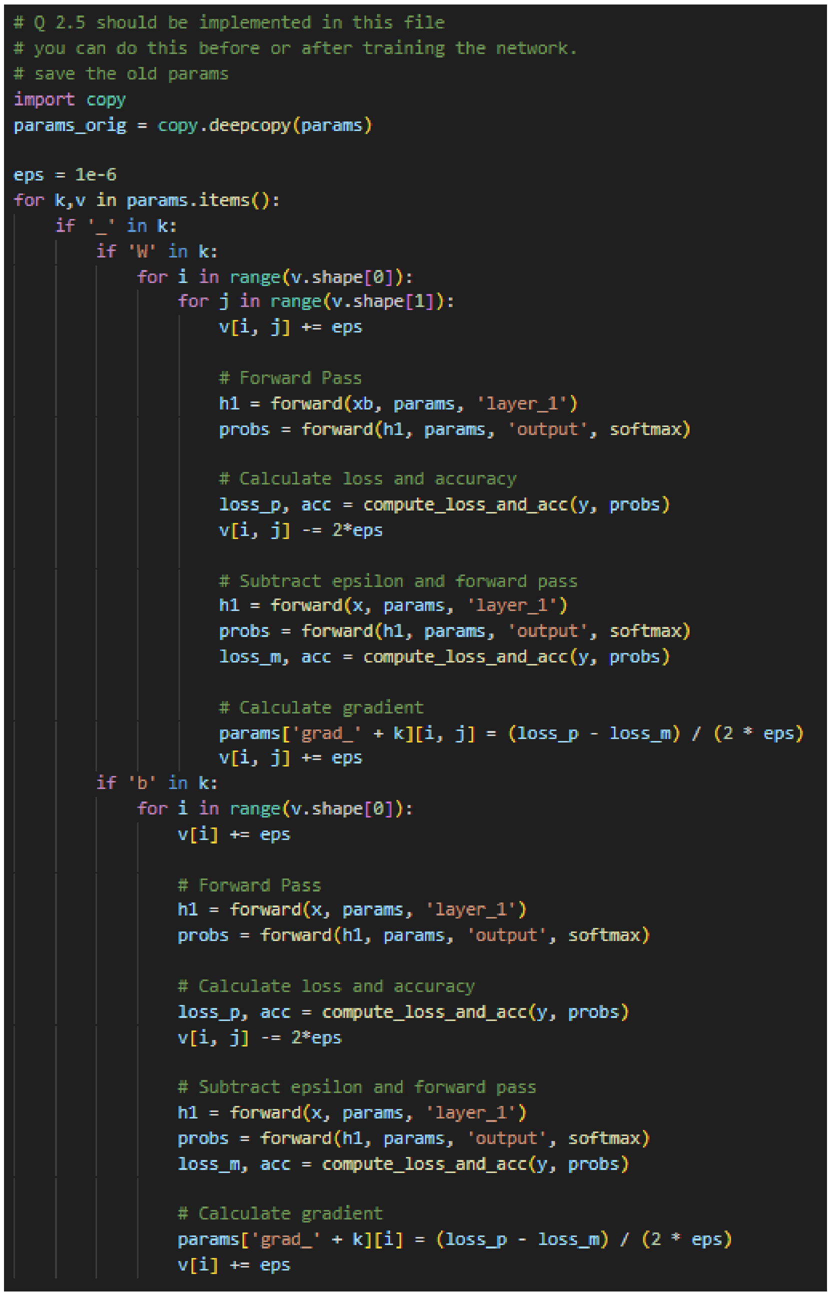


```python
# Q 2.4
batches = get_random_batches(x,y,5)
# print batch sizes
print([_[0].shape[0] for _ in batches])
batch_num = len(batches)

# 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
    total_acc = 0
    avg_acc = 0
    for xb,yb in batches:
        pass
        # forward
        h1 = forward(xb, params, 'layer_1')
        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_acc += acc
        total_loss += loss

        # backward
        delta = probs - yb
        grad = backwards(delta, params, 'output', linear_deriv)
        backwards(grad, params, 'layer_1', sigmoid_deriv)

        # apply gradient
        params['W_layer_1'] -= learning_rate * params['grad_W_layer_1']
        params['b_layer_1'] -= learning_rate * params['grad_b_layer_1']
        params['W_output'] -= learning_rate * params['grad_W_output']
        params['b_output'] -= learning_rate * params['grad_b_output']

    avg_acc = total_acc / batch_num

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

Figure 7: Training Loop Function

2.5 In python/run q2.py, implement a numerical gradient checker. Instead of using the analytical gradients computed from the chain rule, add ∈ offset to each element in the weights, and compute the numerical gradient of the loss with central differences.

The implementation of the numerical gradient checker function can be found in Figure 8:



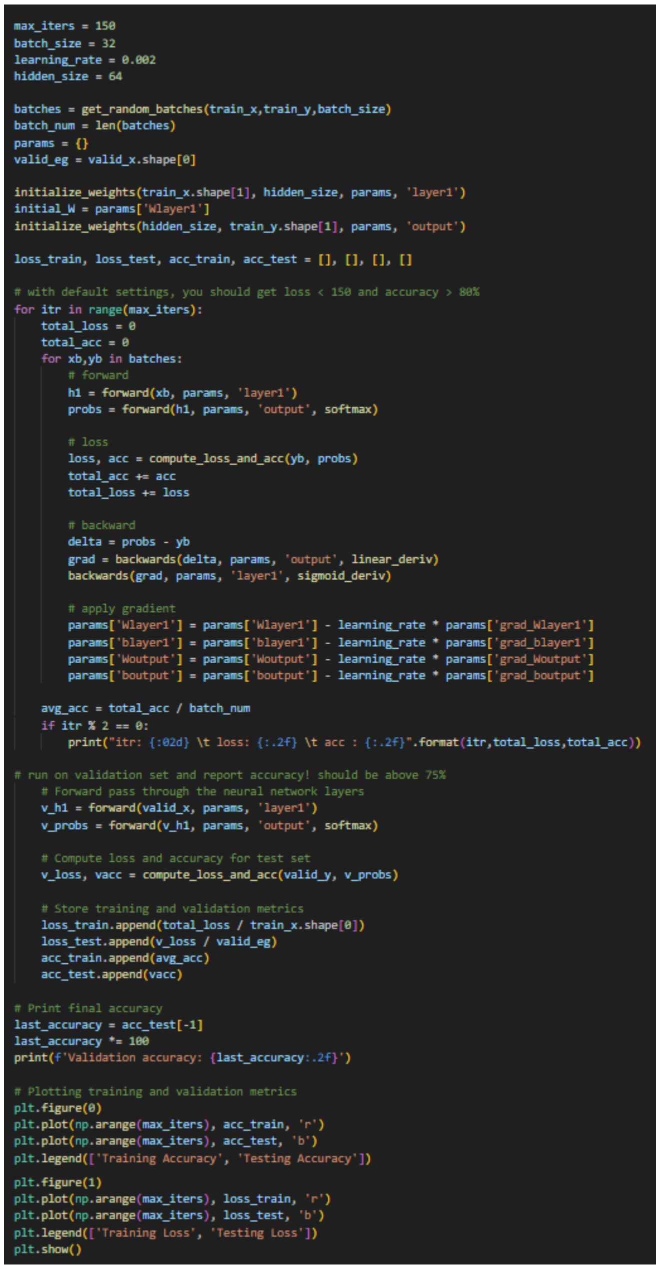
```python
# Q 2.5 should be implemented in this file
# you can do this before or after training the network.
# save the old params
import copy
params_orig = copy.deepcopy(params)

eps = 1e-6
for k,v in params.items():
    if '_' in k:
        if 'W' in k:
            for i in range(v.shape[0]):
                for j in range(v.shape[1]):
                    v[i, j] += eps

                    # Forward Pass
                    h1 = forward(xb, params, 'layer_1')
                    probs = forward(h1, params, 'output', softmax)

                    # Calculate loss and accuracy
                    loss_p, acc = compute_loss_and_acc(y, probs)
                    v[i, j] -= 2*eps

                    # Subtract epsilon and forward pass
                    h1 = forward(x, params, 'layer_1')
                    probs = forward(h1, params, 'output', softmax)
                    loss_m, acc = compute_loss_and_acc(y, probs)

                    # Calculate gradient
                    params['grad_' + k][i, j] = (loss_p - loss_m) / (2 * eps)
                    v[i, j] += eps
        if 'b' in k:
            for i in range(v.shape[0]):
                v[i] += eps

                # Forward Pass
                h1 = forward(x, params, 'layer_1')
                probs = forward(h1, params, 'output', softmax)

                # Calculate loss and accuracy
                loss_p, acc = compute_loss_and_acc(y, probs)
                v[i, j] -= 2*eps

                # Subtract epsilon and forward pass
                h1 = forward(x, params, 'layer_1')
                probs = forward(h1, params, 'output', softmax)
                loss_m, acc = compute_loss_and_acc(y, probs)

                # Calculate gradient
                params['grad_' + k][i] = (loss_p - loss_m) / (2 * eps)
                v[i] += eps
```

Figure 8: Numerical Gradient Checker Function

3 Training Model

3.1 Train a network from scratch. Use a single hidden layer with 64 hidden units, and train for at least 30 epochs.

The implementation of the network can be found in Figure 9 below.



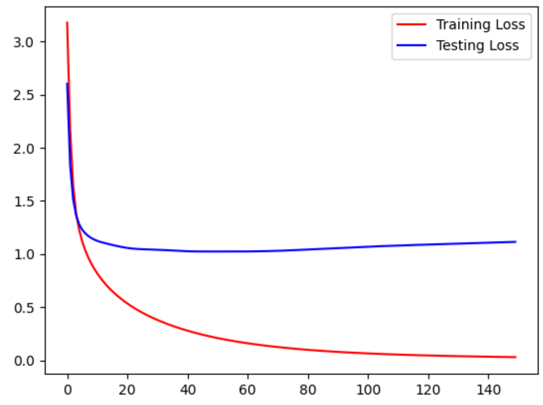
```python
max_iters = 150
batch_size = 32
learning_rate = 0.002
hidden_size = 64

batches = get_random_batches(train_x,train_y,batch_size)
batch_num = len(batches)
params = {}
valid_eg = valid_x.shape[0]

initialize_weights(train_x.shape[1], hidden_size, params, 'layer1')
initial_W = params['Wlayer1']
initialize_weights(hidden_size, train_y.shape[1], params, 'output')

loss_train, loss_test, acc_train, acc_test = [], [], [], []

# with default settings, you should get loss < 150 and accuracy > 80%
for itr in range(max_iters):
    total_loss = 0
    total_acc = 0
    for xb,yb in batches:
        # forward
        h1 = forward(xb, params, 'layer1')
        probs = forward(h1, params, 'output', softmax)

        # loss
        loss, acc = compute_loss_and_acc(yb, probs)
        total_acc += acc
        total_loss += loss

        # backward
        delta = probs - yb
        grad = backwards(delta, params, 'output', linear_deriv)
        backwards(grad, params, 'layer1', sigmoid_deriv)

        # apply gradient
        params['Wlayer1'] = params['Wlayer1'] - learning_rate * params['grad_Wlayer1']
        params['blayer1'] = params['blayer1'] - learning_rate * params['grad_blayer1']
        params['Woutput'] = params['Woutput'] - learning_rate * params['grad_Woutput']
        params['boutput'] = params['boutput'] - learning_rate * params['grad_boutput']

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

# run on validation set and report accuracy! should be above 75%
    # Forward pass through the neural network layers
    v_h1 = forward(valid_x, params, 'layer1')
    v_probs = forward(v_h1, params, 'output', softmax)

    # Compute loss and accuracy for test set
    v_loss, vacc = compute_loss_and_acc(valid_y, v_probs)

    # Store training and validation metrics
    loss_train.append(total_loss / train_x.shape[0])
    loss_test.append(v_loss / valid_eg)
    acc_train.append(avg_acc)
    acc_test.append(vacc)

# Print final accuracy
last_accuracy = acc_test[-1]
last_accuracy *= 100
print(f'Validation accuracy: {last_accuracy:.2f}')

# Plotting training and validation metrics
plt.figure(0)
plt.plot(np.arange(max_iters), acc_train, 'r')
plt.plot(np.arange(max_iters), acc_test, 'b')
plt.legend(['Training Accuracy', 'Testing Accuracy'])

plt.figure(1)
plt.plot(np.arange(max_iters), loss_train, 'r')
plt.plot(np.arange(max_iters), loss_test, 'b')
plt.legend(['Training Loss', 'Testing Loss'])
plt.show()
```

Figure 9: Network Implementation

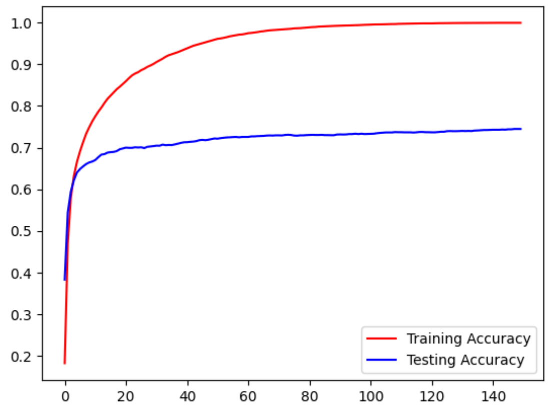
The hyperparameters used to tune the results are below in Table 1, along with the results in Figure 10 below. This testing set resulted in an accuracy of 75.58%.

Parameter	**Value**
Epochs	150
Batch Size	32
Learning Rate	0.002
Hidden Size	64

Table 1: Neural Network Training Parameters



(a) Loss for Network

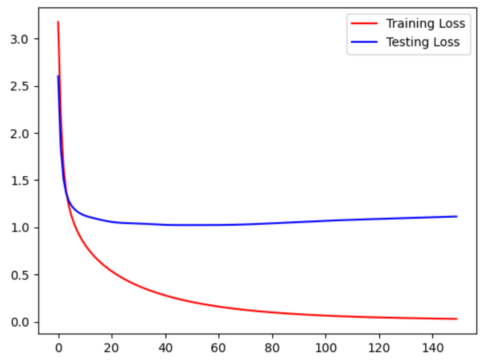


(b) Accuracy for Network

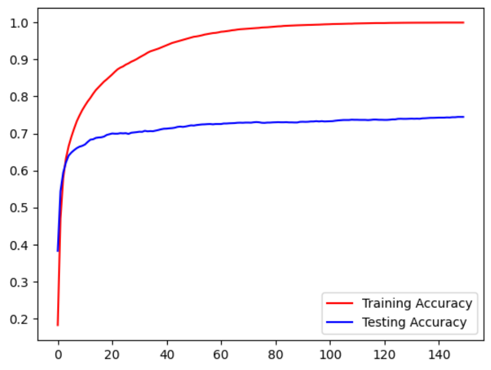
Figure 10: Loss and Accuracy Figures for Base LR

3.2 Use your modified training script to train three networks: one with your best learning rate lr, 10 * lr, 0.1 * lr. Include all 6 plots in your writeup showing 3 plots for accuracy, 3 for loss. Comment on how the learning rates affect the training, and report the final accuracy of the best network on the test set.

- Base LR: 0.002



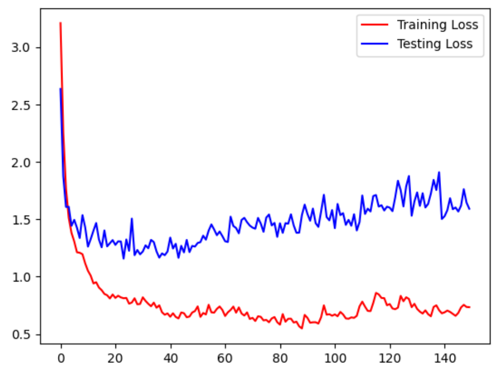
(a) Loss for Network



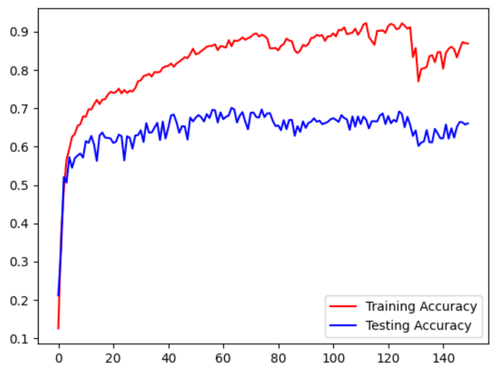
(b) Accuracy for Network

Figure 11: Loss and Accuracy Figures for Base LR

- Base LR: 0.02



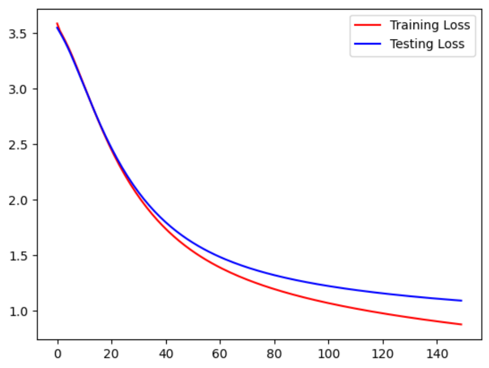
(a) Loss for Network



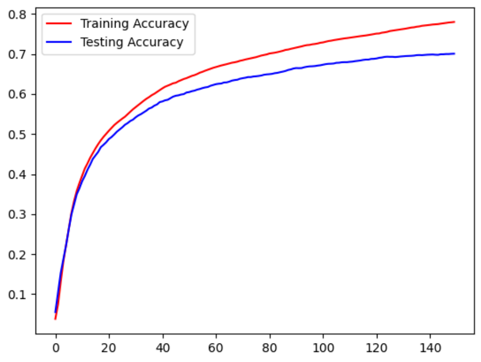
(b) Accuracy for Network

Figure 12: Loss and Accuracy Figures for 10LR

- Base LR: 0.0002



(a) Loss for Network



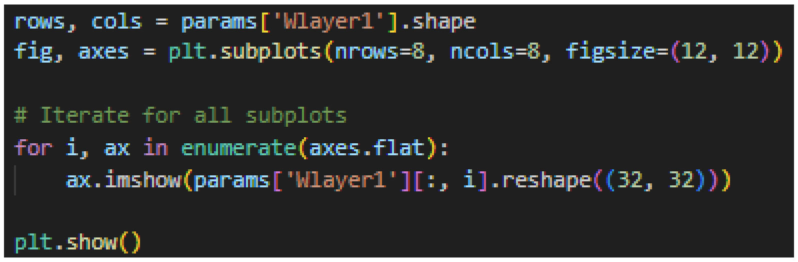
(b) Accuracy for Network

Figure 13: Loss and Accuracy Figures for 0.1LR

When we look at the accuracy between the three learning rates, the best learning rate is still for LR = 0.002 which an accuracy of 75.58%, the second best was LR = 0.0002 with an accuracy of 70.06% and the worst scenario was the one that had LR = 0.02 and an accuracy of 66.08%. What we can see here is that if the learning rate increases to 0.02 as seen in Figure 12, there is significant oscillation for the loss which also cause the accuracy to drop and oscillate as well. On the other hand, if the learning rate drops down to 0.002, we wee that the curve is very smooth with a gradual slope increase/decrease but we notice here that the graph never reaches steady state or plateau because the number of iterations is not sufficient for the small learning rate.

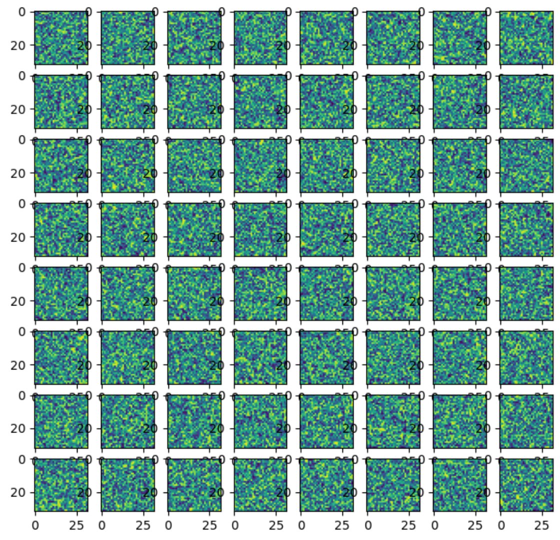
3.3 Visualize the first layer weights that your network learned as 64 32x32 images (using reshape and ImageGrid). Compare these to the network weights immediately after initialization. Include both visualizations in your writeup. Comment on the learned weights. Do you notice any patterns?

The implementation of the visualization for weights can be found in Figure 14 below.

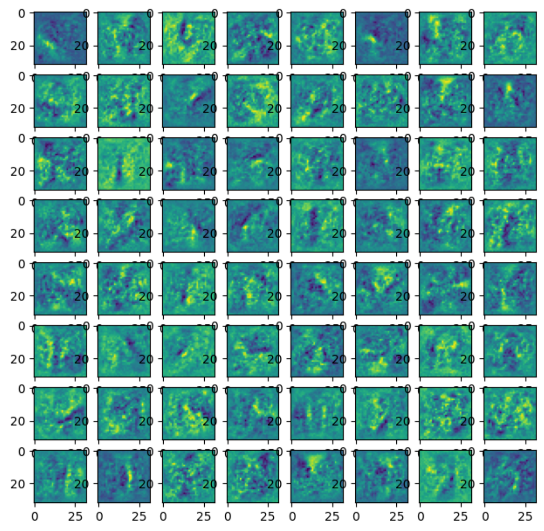


```
rows, cols = params['Wlayer1'].shape
fig, axes = plt.subplots(nrows=8, ncols=8, figsize=(12, 12))

# Iterate for all subplots
for i, ax in enumerate(axes.flat):
    ax.imshow(params['Wlayer1'][:, i].reshape((32, 32)))

plt.show()
```

Figure 14: Network Implementation



(a) Initialization Visualization



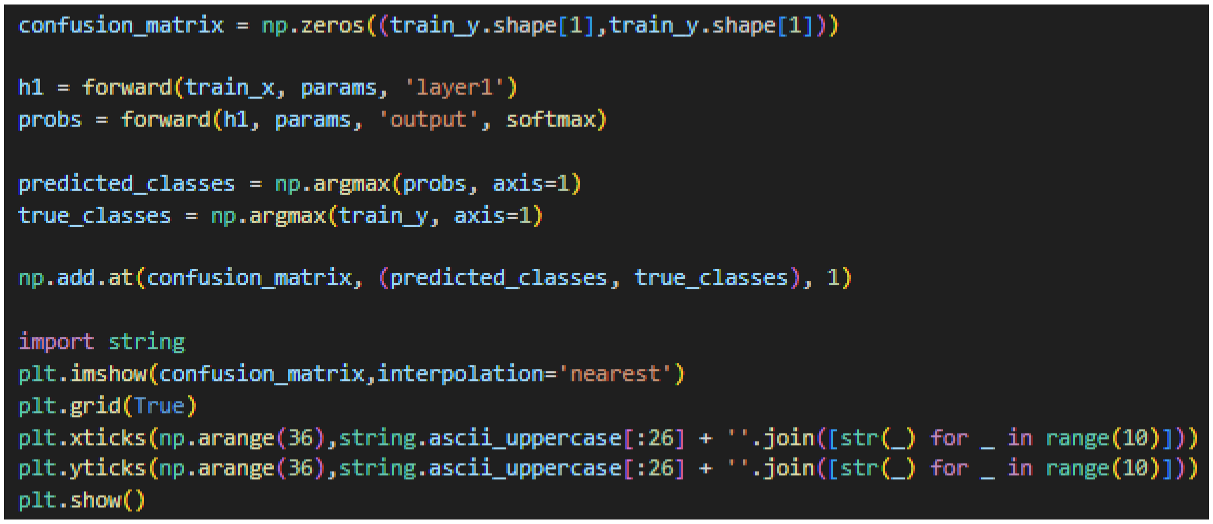
(b) Trained Visualization

Figure 15: Visualization of Layer 1 Weights

In Figure 15, we can see the comparison of visualization for the weights of the first layer during initialization and after training. The initialized weights have a very random distribution and scattered due to the initialization with randomized uniform distribution. But after the system is fully trained for layer 1, we can see that the new weights resemble blobs or shapes on the right with more information stored within them.

3.4 Visualize the confusion matrix of the test data for your best model. Comment on the top few pairs of classes that are most commonly confused.

The implementation of the confusion matrix can be found in Figure 16 below.



```
confusion_matrix = np.zeros((train_y.shape[1],train_y.shape[1]))

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

predicted_classes = np.argmax(probs, axis=1)
true_classes = np.argmax(train_y, axis=1)

np.add.at(confusion_matrix, (predicted_classes, true_classes), 1)

import string
plt.imshow(confusion_matrix,interpolation='nearest')
plt.grid(True)
plt.xticks(np.arange(36),string.ascii_uppercase[:26] + ''.join([str(_) for _ in range(10)]))
plt.yticks(np.arange(36),string.ascii_uppercase[:26] + ''.join([str(_) for _ in range(10)]))
plt.show()
```

Figure 16: Confusion Matrix Implementation

The confusion matrix can be found in Figure 17 below. Some of the top few pairs of classes that are most commonly confused seem to be those with similar shapes such as 0 and O, 2 and Z, and 5 and S. However, compared to the other confused characters, 0 and O have a much higher confusion rate compared to the others.

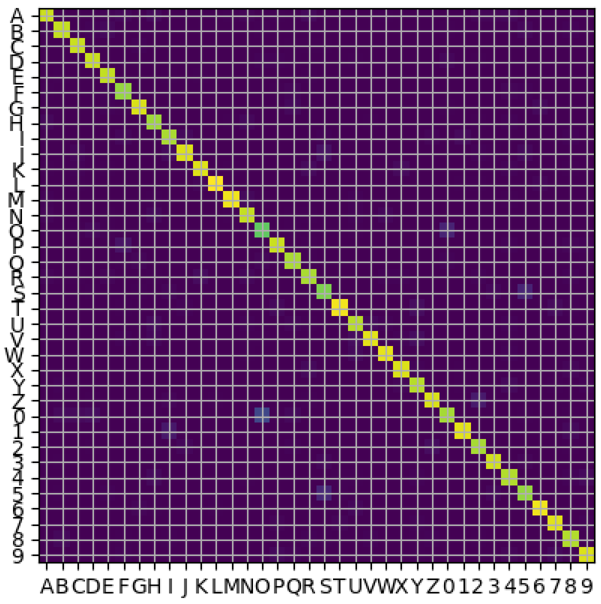


Figure 17: Confusion Matrix

4 Extract Text from Images

4.1 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 (either miss valid letters, or respond to non-letters).

The two big assumptions that the sample method takes is that the individual letters are all not connected to each other and and that each component of each letter is fully connected. If we present cases where these are not true, we may expect to see detection failure. If we look at the top scenario of Figure 18, the letters have components within that do not connect properly, whereas the bottom scenario has letters connected to each other as its failure point.

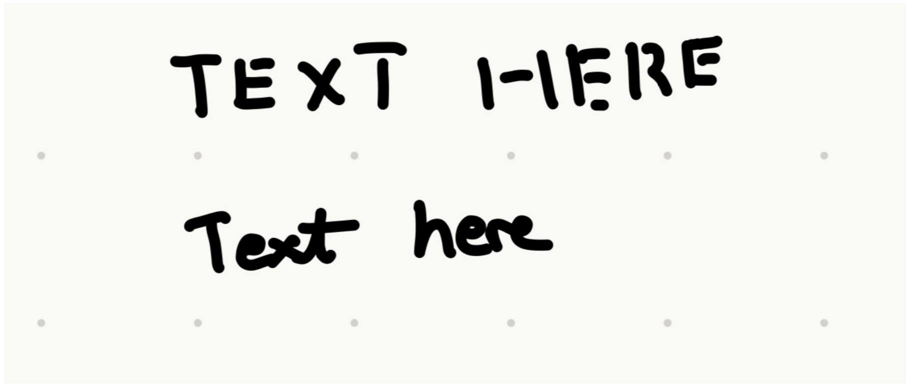
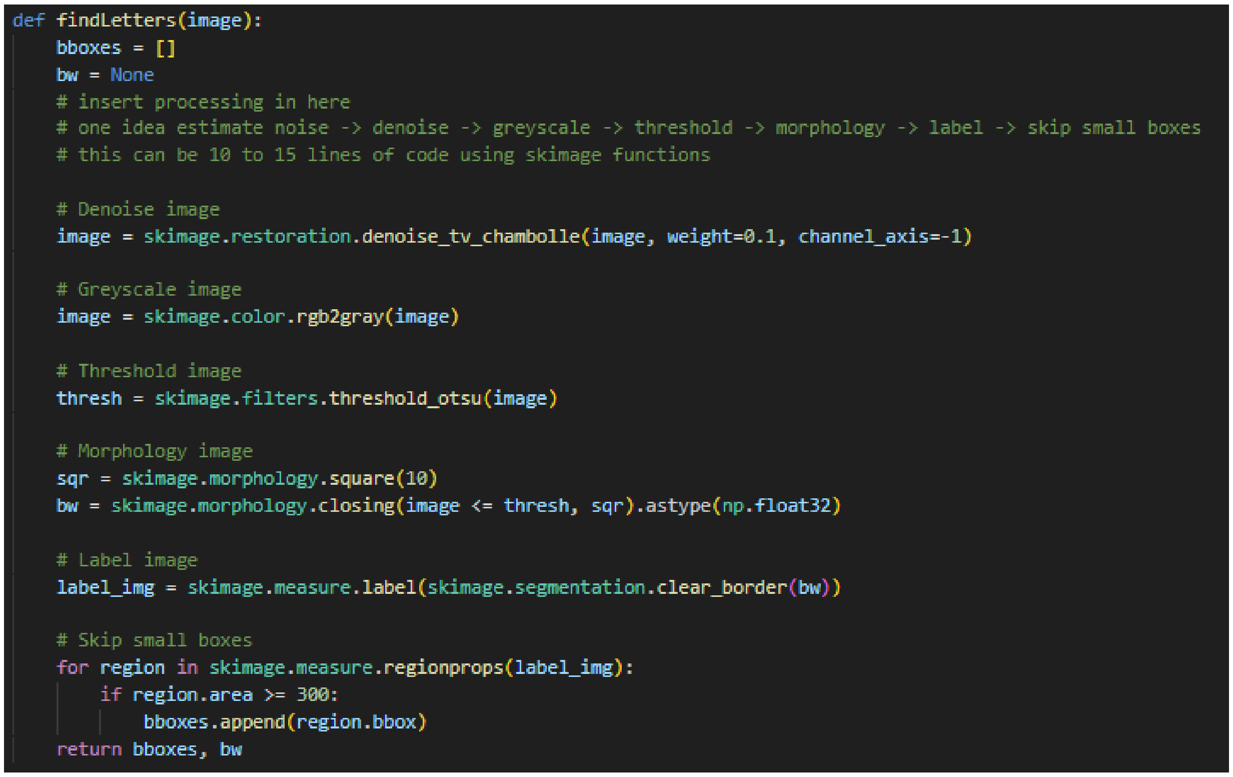


Figure 18: Failure Points for Detection

4.2 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.

The implementation of the findLetters function can be found in Figure 19 below.



```python
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

    # Denoise image
    image = skimage.restoration.denoise_tv_chambolle(image, weight=0.1, channel_axis=-1)

    # Greyscale image
    image = skimage.color.rgb2gray(image)

    # Threshold image
    thresh = skimage.filters.threshold_otsu(image)

    # Morphology image
    sqr = skimage.morphology.square(10)
    bw = skimage.morphology.closing(image <= thresh, sqr).astype(np.float32)

    # Label image
    label_img = skimage.measure.label(skimage.segmentation.clear_border(bw))

    # Skip small boxes
    for region in skimage.measure.regionprops(label_img):
        if region.area >= 300:
            bboxes.append(region.bbox)
    return bboxes, bw
```

Figure 19: find Letters Implementation

4.3 Run findLetters(..) on all of the provided sample images in images/. Plot all of the located boxes on top of the image to show the accuracy of your findLetters(..) function. Include all the result images in your writeup.

- Image 1

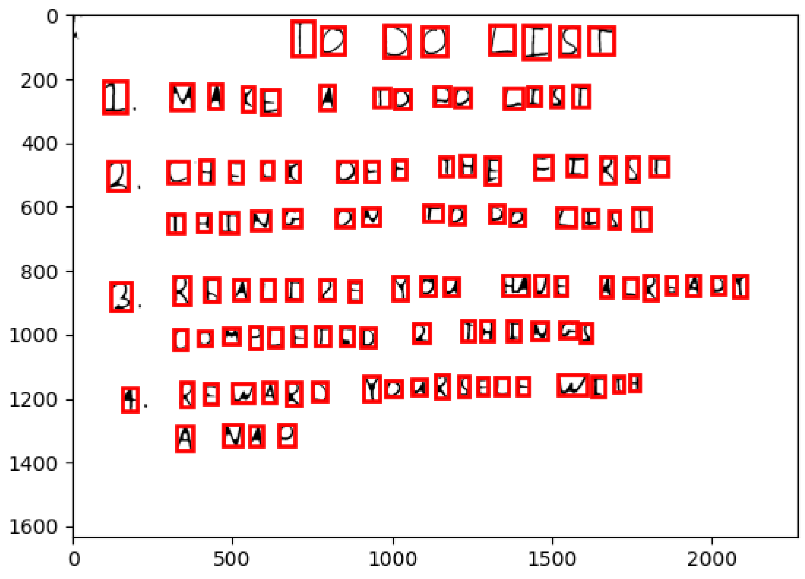


Figure 20: Image 1 Letter Bounding Boxes

- Image 2

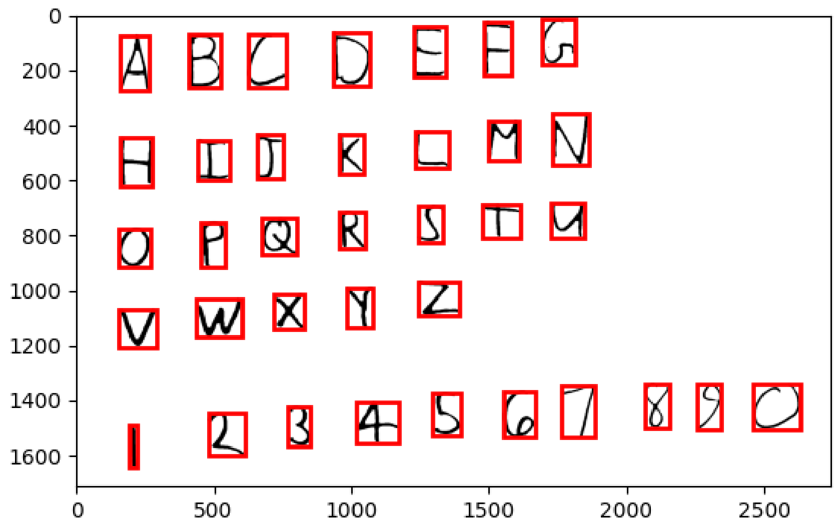


Figure 21: Image 2 Letter Bounding Boxes

- Image 3

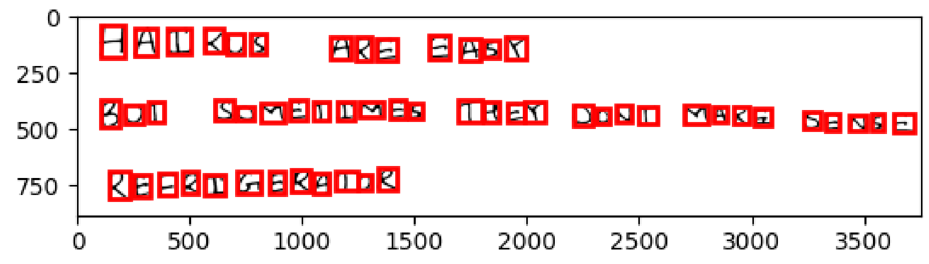


Figure 22: Image 3 Letter Bounding Boxes

- Image 4

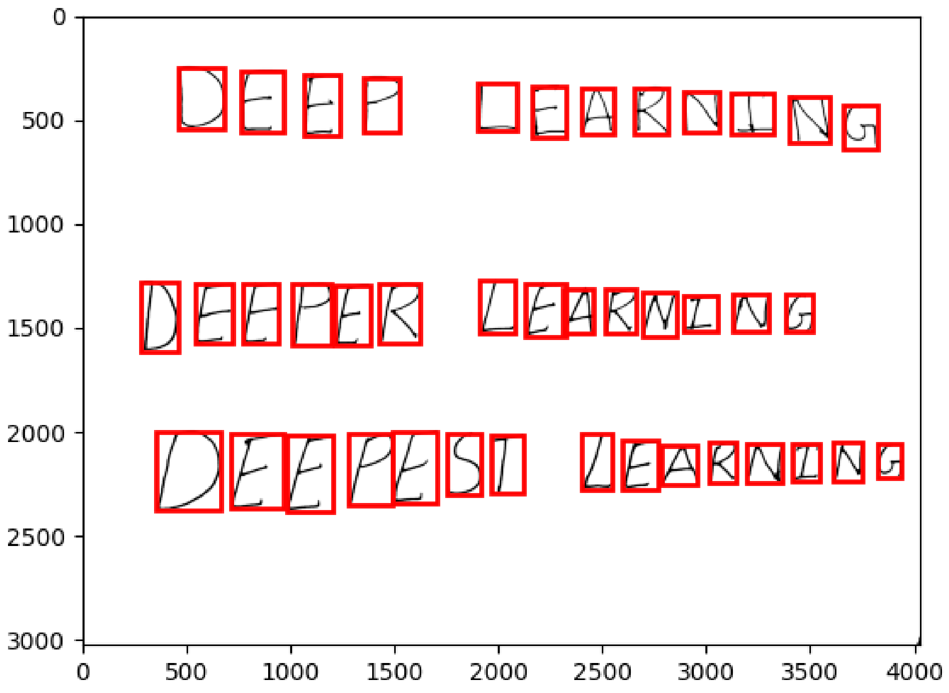
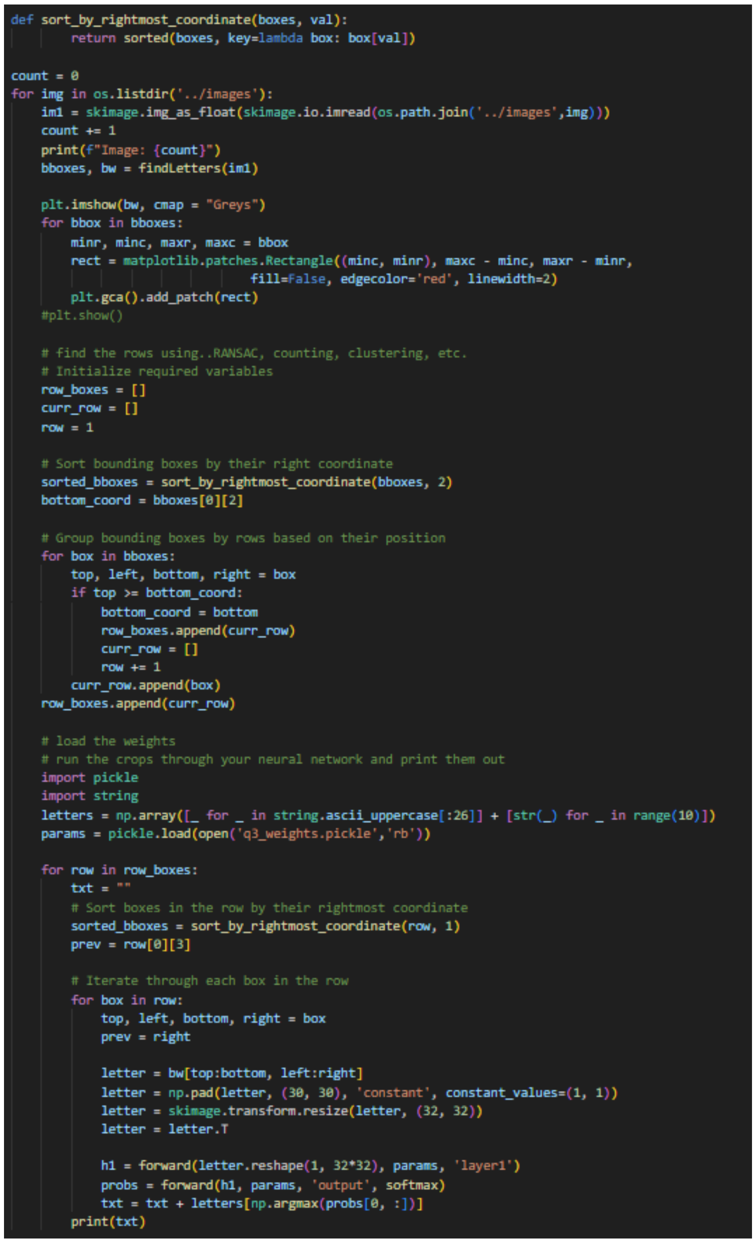


Figure 23: Image 4 Letter Bounding Boxes

4.4 Use python/run q4.py for this question. Now you will 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.

The implementation of the text translation function can be found in Figure 24 below:



```python
def sort_by_rightmost_coordinate(boxes, val):
        return sorted(boxes, key=lambda box: box[val])

count = 0
for img in os.listdir('../images'):
    im1 = skimage.img_as_float(skimage.io.imread(os.path.join('../images',img)))
    count += 1
    print(f"Image: {count}")
    bboxes, bw = findLetters(im1)

    plt.imshow(bw, cmap = "Greys")
    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.
    # Initialize required variables
    row_boxes = []
    curr_row = []
    row = 1

    # Sort bounding boxes by their right coordinate
    sorted_bboxes = sort_by_rightmost_coordinate(bboxes, 2)
    bottom_coord = bboxes[0][2]

    # Group bounding boxes by rows based on their position
    for box in bboxes:
        top, left, bottom, right = box
        if top >= bottom_coord:
            bottom_coord = bottom
            row_boxes.append(curr_row)
            curr_row = []
            row += 1
        curr_row.append(box)
    row_boxes.append(curr_row)

    # 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'))

    for row in row_boxes:
        txt = ""
        # Sort boxes in the row by their rightmost coordinate
        sorted_bboxes = sort_by_rightmost_coordinate(row, 1)
        prev = row[0][3]

        # Iterate through each box in the row
        for box in row:
            top, left, bottom, right = box
            prev = right

            letter = bw[top:bottom, left:right]
            letter = np.pad(letter, (30, 30), 'constant', constant_values=(1, 1))
            letter = skimage.transform.resize(letter, (32, 32))
            letter = letter.T

            h1 = forward(letter.reshape(1, 32*32), params, 'layer1')
            probs = forward(h1, params, 'output', softmax)
            txt = txt + letters[np.argmax(probs[0, :])]
        print(txt)
```

Figure 24: Text Function

The results for the function can be found in Table 2 below. What we can see here is that at least 50% of the letters were predicted accurately but there are alot of moments when the letters are swapped with similar shaped characters instead.

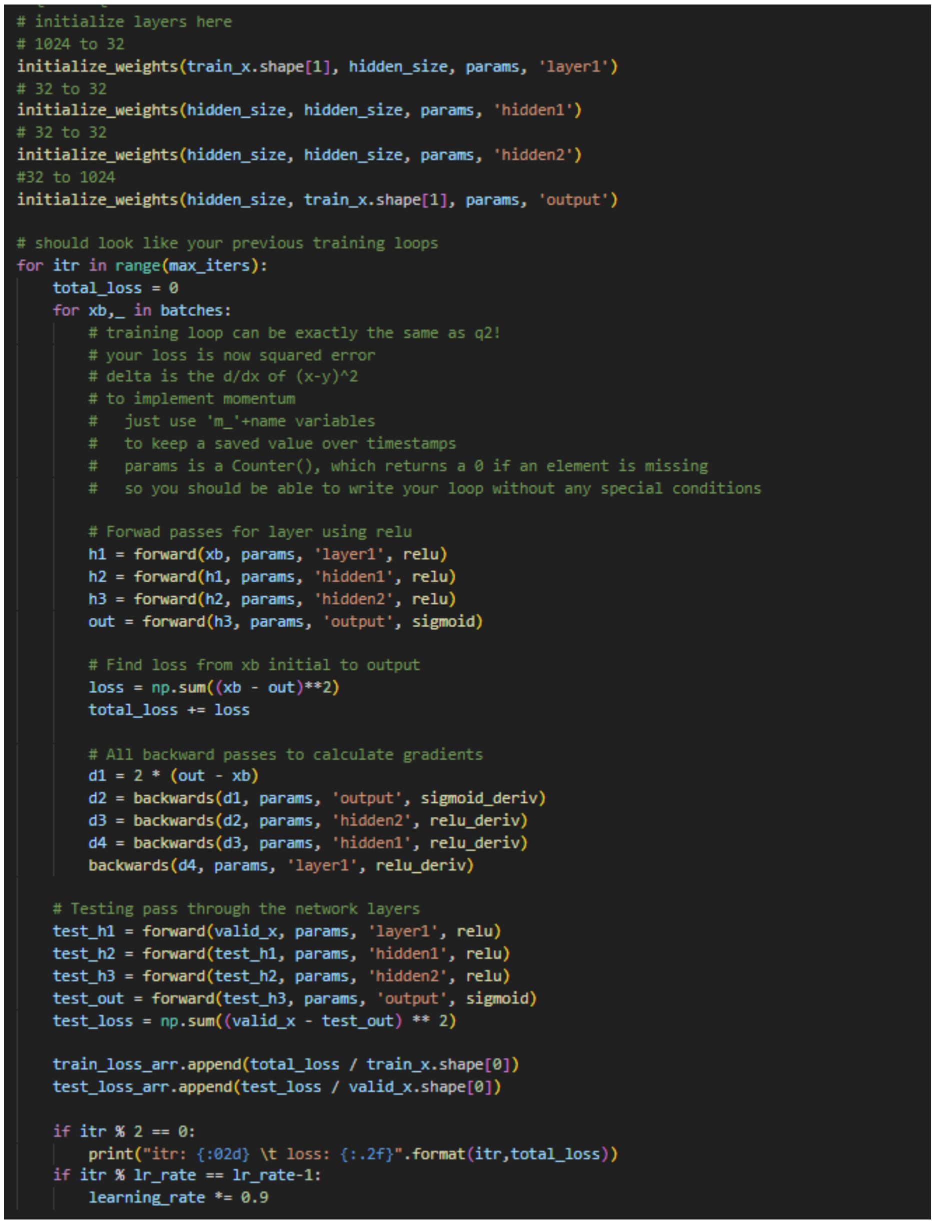
Image	**Text**
1	T0 DO LIST
	I MAKE A TO DO LIST
	2 CHECK OFF THE FIRST
	THING ON TO DO LIST
	2 REALIZE YOU HAVE ALREADY
	COMPLETED 2 THINGS
	4 REWARD YOURSELF WITH
	A NAP
2	A S C C C F C
	H I J K C K U
	0 Y Q X S T 4
	V 4 X Y I
	X X S 4 S U J X Y C
3	HAIRUS ARE EASY
	BLT SOMETIMES TBEX UONT MAKE SEMSE
	REFRIGERATOR
4	DIFFF CEARNI44
	DFFFEK LEAKHIN6
	UFFYVST LEARHIN6

5 Image Compression with Autoencoder

5.1 Building the Autoencoder

5.1.1 Due to the difficulty in training auto-encoders, we have to move to the relu(x) = max(x, 0) activation function. It is provided for you in util.py. Implement an autoencoder where the layers are: 1024 to 32 dimensions, followed by a ReLU, 32 to 32 dimensions, followed by a ReLU, 32 to 32 dimensions, followed by a ReLU, and 32 to 1024 dimensions, followed by a sigmoid (this normalizes the image output for us).

The implementation of the autoencoder function can be found in Figure 25 below:

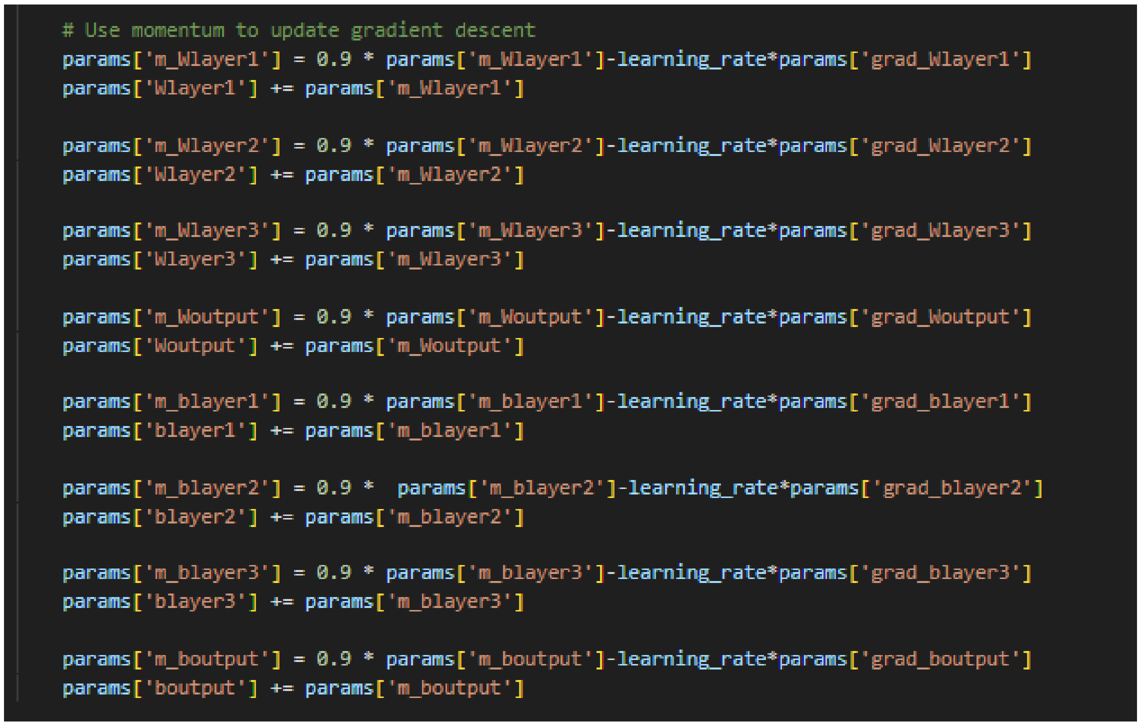


```python
# initialize layers here
# 1024 to 32
initialize_weights(train_x.shape[1], hidden_size, params, 'layer1')
# 32 to 32
initialize_weights(hidden_size, hidden_size, params, 'hidden1')
# 32 to 32
initialize_weights(hidden_size, hidden_size, params, 'hidden2')
#32 to 1024
initialize_weights(hidden_size, train_x.shape[1], params, 'output')

# should look like your previous training loops
for itr in range(max_iters):
    total_loss = 0
    for xb,_ in batches:
        # training loop can be exactly the same as q2!
        # your loss is now squared error
        # delta is the d/dx of (x-y)^2
        # to implement momentum
        #   just use 'm_'+name variables
        #   to keep a saved value over timestamps
        #   params is a Counter(), which returns a 0 if an element is missing
        #   so you should be able to write your loop without any special conditions

        # Forwad passes for layer using relu
        h1 = forward(xb, params, 'layer1', relu)
        h2 = forward(h1, params, 'hidden1', relu)
        h3 = forward(h2, params, 'hidden2', relu)
        out = forward(h3, params, 'output', sigmoid)

        # Find loss from xb initial to output
        loss = np.sum((xb - out)**2)
        total_loss += loss

        # All backward passes to calculate gradients
        d1 = 2 * (out - xb)
        d2 = backwards(d1, params, 'output', sigmoid_deriv)
        d3 = backwards(d2, params, 'hidden2', relu_deriv)
        d4 = backwards(d3, params, 'hidden1', relu_deriv)
        backwards(d4, params, 'layer1', relu_deriv)

    # Testing pass through the network layers
    test_h1 = forward(valid_x, params, 'layer1', relu)
    test_h2 = forward(test_h1, params, 'hidden1', relu)
    test_h3 = forward(test_h2, params, 'hidden2', relu)
    test_out = forward(test_h3, params, 'output', sigmoid)
    test_loss = np.sum((valid_x - test_out) ** 2)

    train_loss_arr.append(total_loss / train_x.shape[0])
    test_loss_arr.append(test_loss / 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
```

Figure 25: Autoencoder Function

5.1.2 To implement this, populate the parameters dictionary with zero-initialized momentum accumulators, one for each parameter. Then simply perform both update equations for every batch.

The implementation of the Momentum Gradient Descent function can be found in Figure 26 below:



```
# Use momentum to update gradient descent
params['m_Wlayer1'] = 0.9 * params['m_Wlayer1']-learning_rate*params['grad_Wlayer1']
params['Wlayer1'] += params['m_Wlayer1']

params['m_Wlayer2'] = 0.9 * params['m_Wlayer2']-learning_rate*params['grad_Wlayer2']
params['Wlayer2'] += params['m_Wlayer2']

params['m_Wlayer3'] = 0.9 * params['m_Wlayer3']-learning_rate*params['grad_Wlayer3']
params['Wlayer3'] += params['m_Wlayer3']

params['m_Woutput'] = 0.9 * params['m_Woutput']-learning_rate*params['grad_Woutput']
params['Woutput'] += params['m_Woutput']

params['m_blayer1'] = 0.9 * params['m_blayer1']-learning_rate*params['grad_blayer1']
params['blayer1'] += params['m_blayer1']

params['m_blayer2'] = 0.9 *  params['m_blayer2']-learning_rate*params['grad_blayer2']
params['blayer2'] += params['m_blayer2']

params['m_blayer3'] = 0.9 * params['m_blayer3']-learning_rate*params['grad_blayer3']
params['blayer3'] += params['m_blayer3']

params['m_boutput'] = 0.9 * params['m_boutput']-learning_rate*params['grad_boutput']
params['boutput'] += params['m_boutput']
```

Figure 26: Momentum Gradient Descent Function

5.2 Training the Autoencoder

5.2.1 Using the provided default settings, train the network for 100 epochs. What do you observe in the plotted training loss curve as it progresses?

The results of trianing the network for 100 epochs can be in Figure 27 below. What we observe here is that as the number of iterations or epochs increases, the loss decreases sharply and slightly jagged until about 20 iterations and the decline curve smooths out. One thing to consider or be wary about is to make sure we don't overfit the entire model.

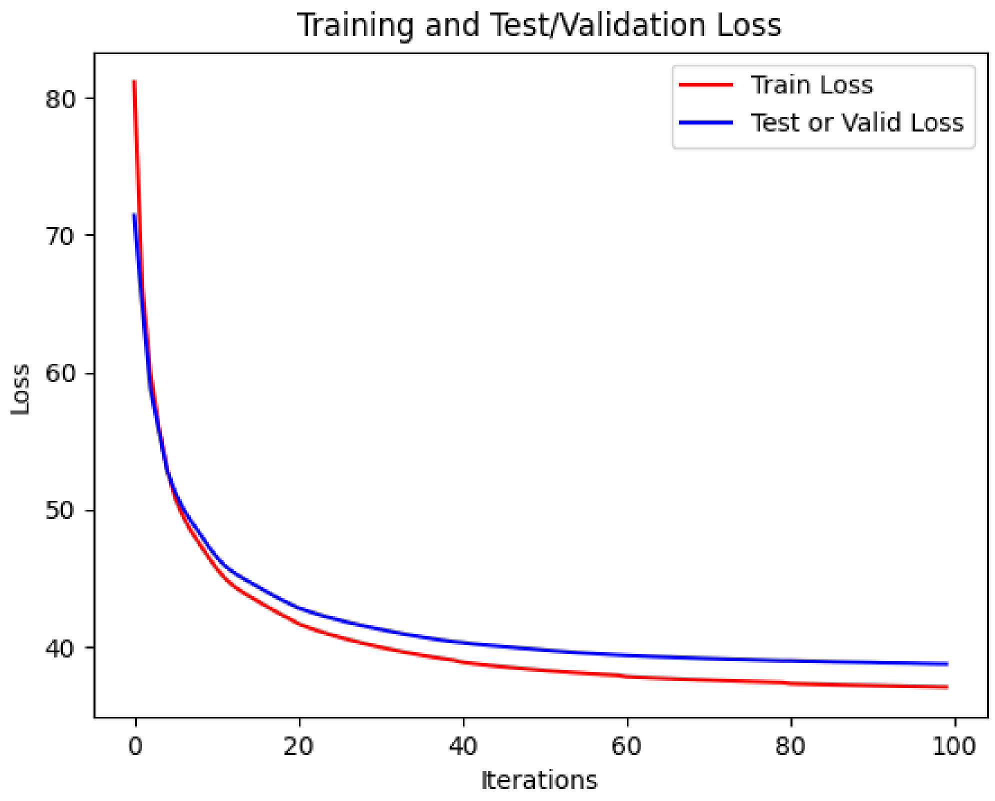


Figure 27: Autoencoder Training Results

5.3 Evaluating the Autoencoder

5.3.1 Now let's evaluate how well the autoencoder has been trained. Select 5 classes from the total 36 classes in the validation set and for each selected class include in your report 2 validation images and their reconstruction. What differences do you observe that exist in the reconstructed validation images, compared to the original ones?

- Z:

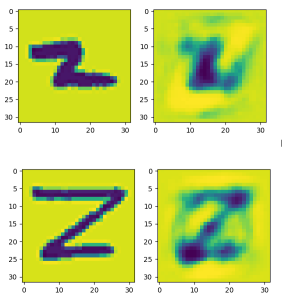


Figure 28: Class Z Results

- 3:

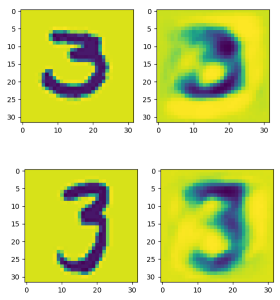


Figure 29: Class 3 Results

- W:

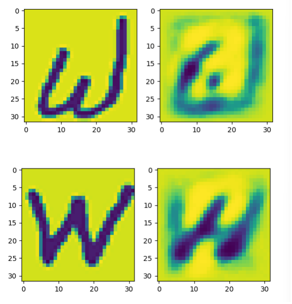


Figure 30: Class W Results

- Q:

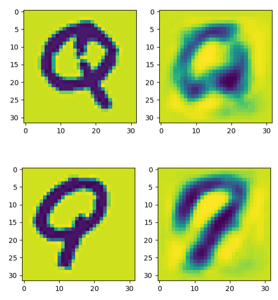


Figure 31: Class Q Results

- G:

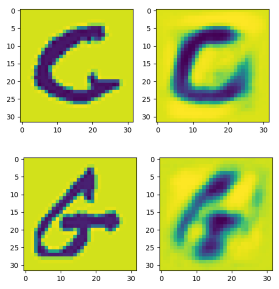
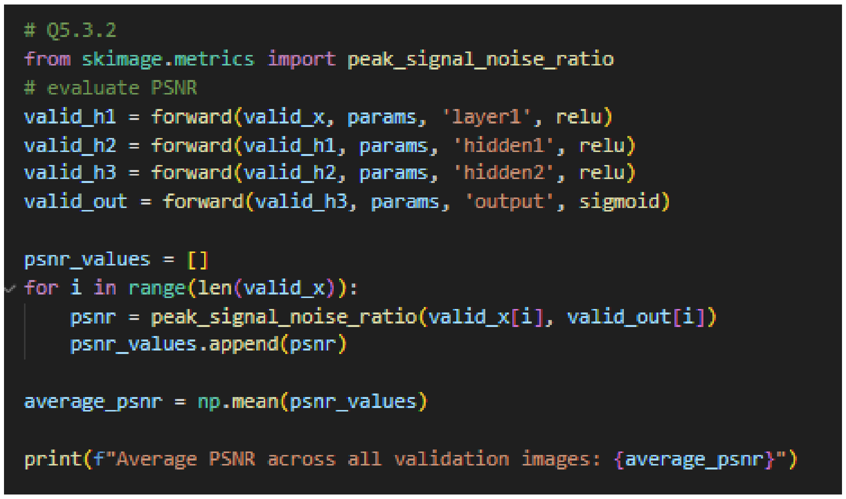


Figure 32: Class G Results

What we can see from the results of the autoencoder from Figures 28-32 are that the autoencoder outputs the blurred image of each class but also tries to reconstruct the image as well. However, the method is perfect as the reconstruction process so we see the blurry figure on the right for all images.

5.3.2 You may use skimage.metrics.peak signal noise ratio for convenience. Report the average PSNR you get from the autoencoder across all images in the validation set (it should be around 15).

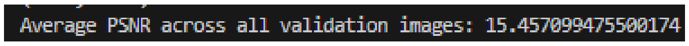
The implementation of the signal to noise ratio function can be found in Figure 33 below:



```python
# Q5.3.2
from skimage.metrics import peak_signal_noise_ratio
# evaluate PSNR
valid_h1 = forward(valid_x, params, 'layer1', relu)
valid_h2 = forward(valid_h1, params, 'hidden1', relu)
valid_h3 = forward(valid_h2, params, 'hidden2', relu)
valid_out = forward(valid_h3, params, 'output', sigmoid)

psnr_values = []
for i in range(len(valid_x)):
    psnr = peak_signal_noise_ratio(valid_x[i], valid_out[i])
    psnr_values.append(psnr)

average_psnr = np.mean(psnr_values)

print(f"Average PSNR across all validation images: {average_psnr}")
```

Figure 33: PSNR Function

The average PSNR value for this situation was found to be around 15.46 at the default settings.



```
Average PSNR across all validation images: 15.457099475500174
```

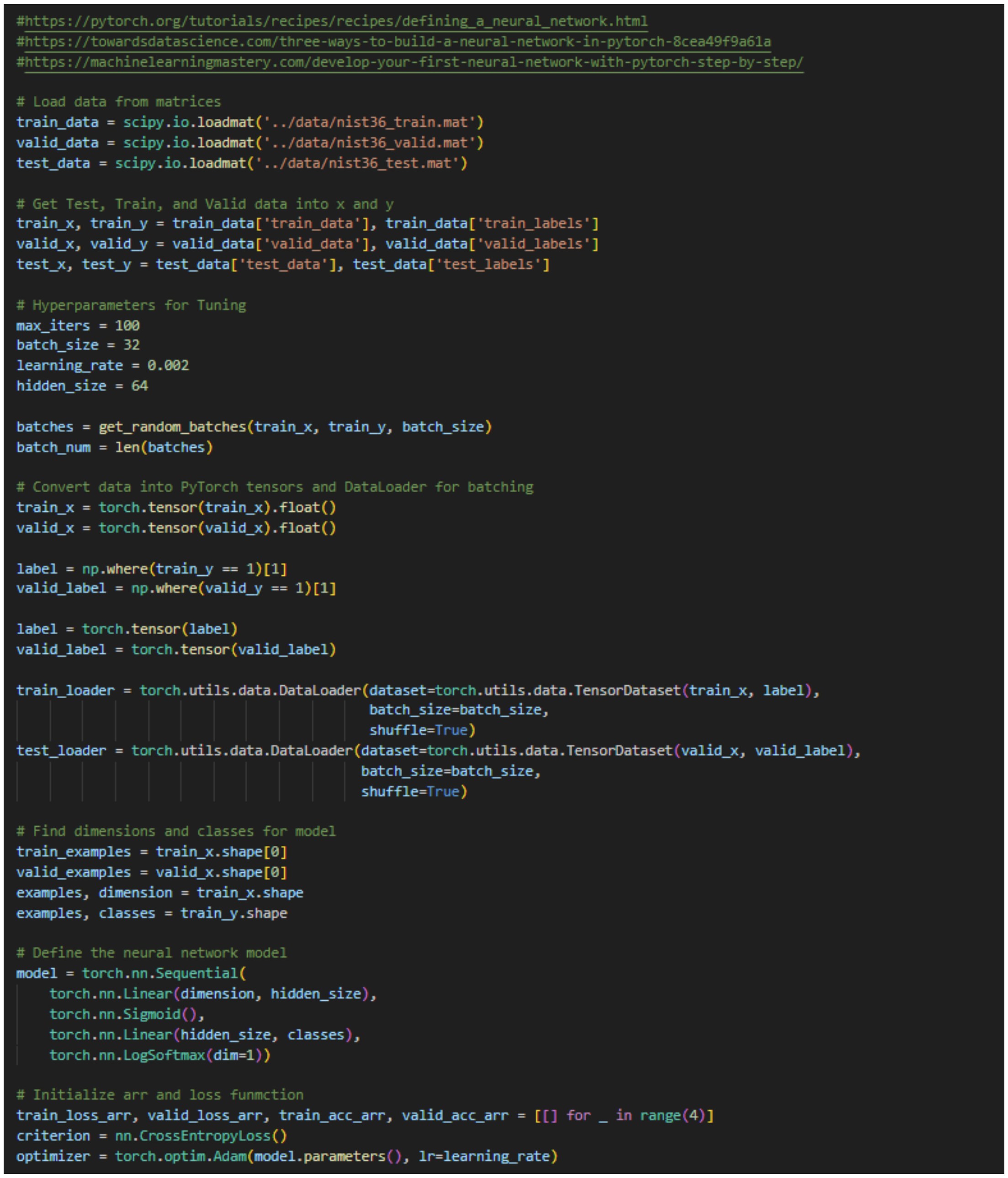
Figure 34: PSNR Average Value

6 PyTorch

6.1 Train a Neural Network in PyTorch

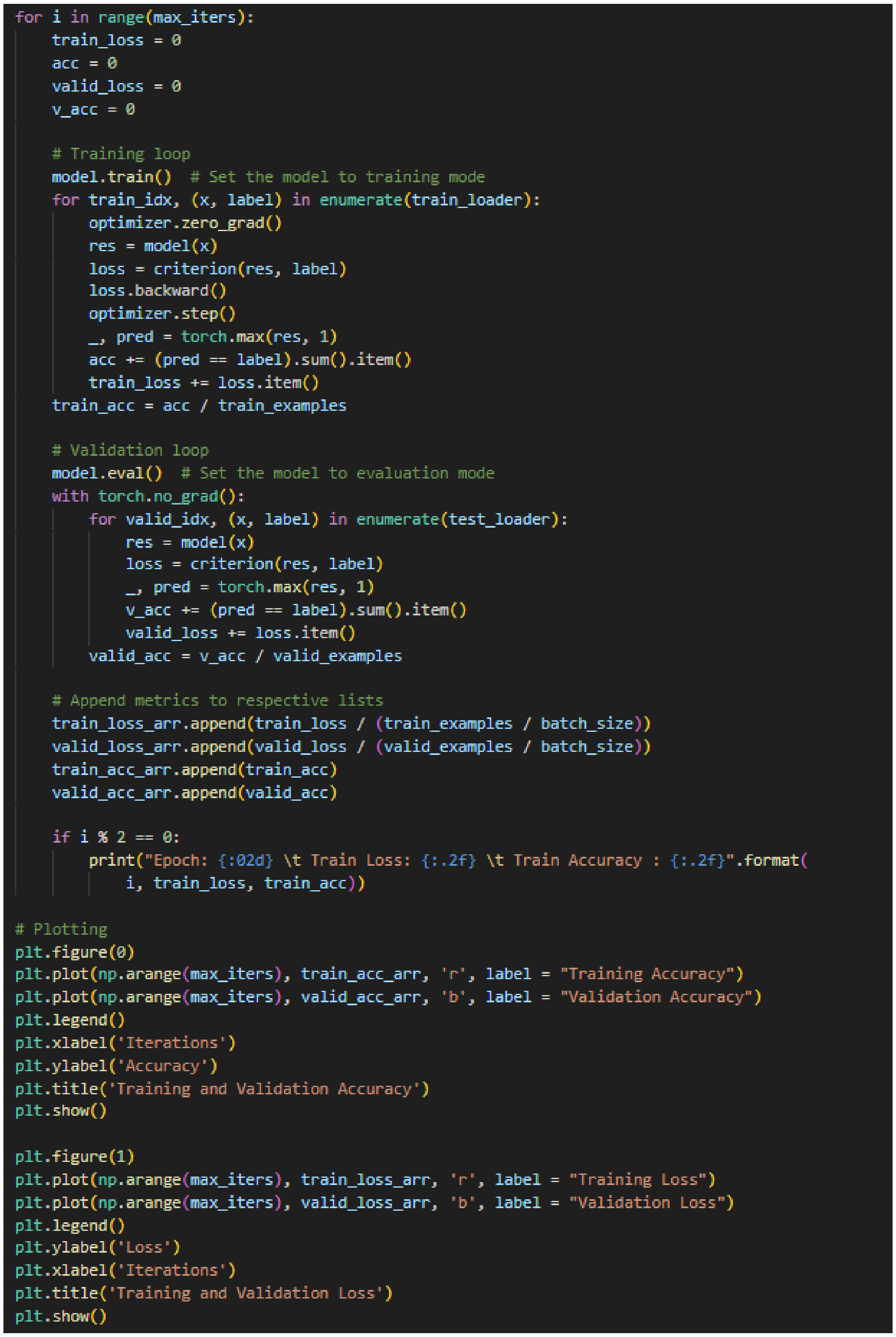
6.1.1 Re-write and re-train your fully-connected network on the included NIST36 in PyTorch. Plot training accuracy and loss over time.

The implementation for this function using PyTorch can be found in Figure 35 and 36 below.



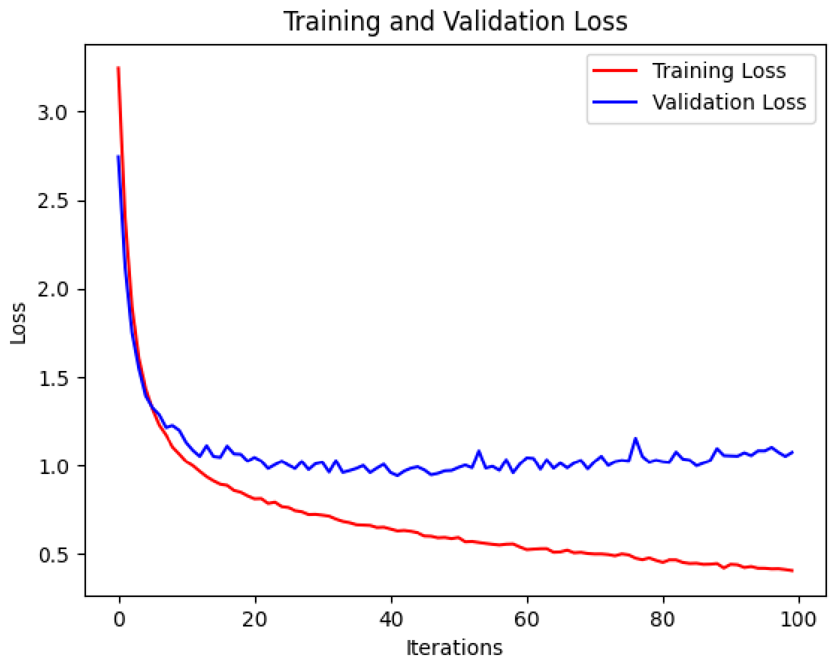
```python
#https://pytorch.org/tutorials/recipes/recipes/defining_a_neural_network.html
#https://towardsdatascience.com/three-ways-to-build-a-neural-network-in-pytorch-8cea49f9a61a
#https://machinelearningmastery.com/develop-your-first-neural-network-with-pytorch-step-by-step/

# Load data from matrices
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')

# Get Test, Train, and Valid data into x and y
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']

# Hyperparameters for Tuning
max_iters = 100
batch_size = 32
learning_rate = 0.002
hidden_size = 64

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

# Convert data into PyTorch tensors and DataLoader for batching
train_x = torch.tensor(train_x).float()
valid_x = torch.tensor(valid_x).float()

label = np.where(train_y == 1)[1]
valid_label = np.where(valid_y == 1)[1]

label = torch.tensor(label)
valid_label = torch.tensor(valid_label)

train_loader = torch.utils.data.DataLoader(dataset=torch.utils.data.TensorDataset(train_x, label),
                                           batch_size=batch_size,
                                           shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=torch.utils.data.TensorDataset(valid_x, valid_label),
                                          batch_size=batch_size,
                                          shuffle=True)

# Find dimensions and classes for model
train_examples = train_x.shape[0]
valid_examples = valid_x.shape[0]
examples, dimension = train_x.shape
examples, classes = train_y.shape

# Define the neural network model
model = torch.nn.Sequential(
    torch.nn.Linear(dimension, hidden_size),
    torch.nn.Sigmoid(),
    torch.nn.Linear(hidden_size, classes),
    torch.nn.LogSoftmax(dim=1))

# Initialize arr and loss funmction
train_loss_arr, valid_loss_arr, train_acc_arr, valid_acc_arr = [[] for _ in range(4)]
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

Figure 35: PyTorch Implementation I

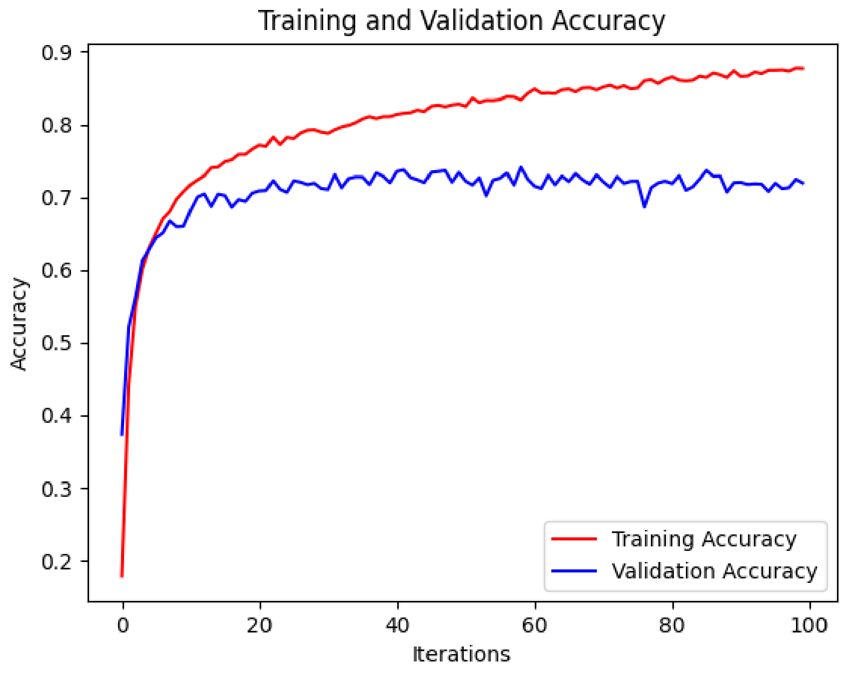


```python
for i in range(max_iters):
    train_loss = 0
    acc = 0
    valid_loss = 0
    v_acc = 0

    # Training loop
    model.train()  # Set the model to training mode
    for train_idx, (x, label) in enumerate(train_loader):
        optimizer.zero_grad()
        res = model(x)
        loss = criterion(res, label)
        loss.backward()
        optimizer.step()
        _, pred = torch.max(res, 1)
        acc += (pred == label).sum().item()
        train_loss += loss.item()
    train_acc = acc / train_examples

    # Validation loop
    model.eval()  # Set the model to evaluation mode
    with torch.no_grad():
        for valid_idx, (x, label) in enumerate(test_loader):
            res = model(x)
            loss = criterion(res, label)
            _, pred = torch.max(res, 1)
            v_acc += (pred == label).sum().item()
            valid_loss += loss.item()
        valid_acc = v_acc / valid_examples

    # Append metrics to respective lists
    train_loss_arr.append(train_loss / (train_examples / batch_size))
    valid_loss_arr.append(valid_loss / (valid_examples / batch_size))
    train_acc_arr.append(train_acc)
    valid_acc_arr.append(valid_acc)

    if i % 2 == 0:
        print("Epoch: {:02d} \t Train Loss: {:.2f} \t Train Accuracy : {:.2f}".format(
            i, train_loss, train_acc))

# Plotting
plt.figure(0)
plt.plot(np.arange(max_iters), train_acc_arr, 'r', label = "Training Accuracy")
plt.plot(np.arange(max_iters), valid_acc_arr, 'b', label = "Validation Accuracy")
plt.legend()
plt.xlabel('Iterations')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy')
plt.show()

plt.figure(1)
plt.plot(np.arange(max_iters), train_loss_arr, 'r', label = "Training Loss")
plt.plot(np.arange(max_iters), valid_loss_arr, 'b', label = "Validation Loss")
plt.legend()
plt.ylabel('Loss')
plt.xlabel('Iterations')
plt.title('Training and Validation Loss')
plt.show()
```

Figure 36: PyTorch Implementation II



(a) Loss for Network

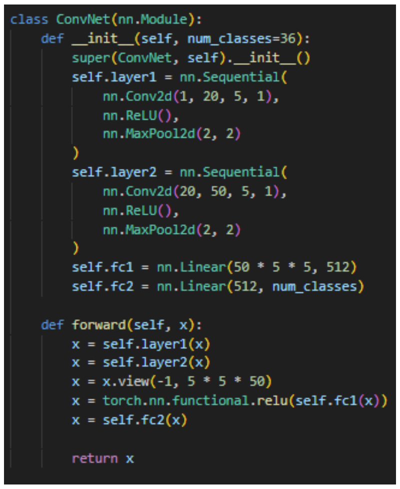


(b) Accuracy for Network

Figure 37: Loss and Accuracy Figures for PyTorch Model

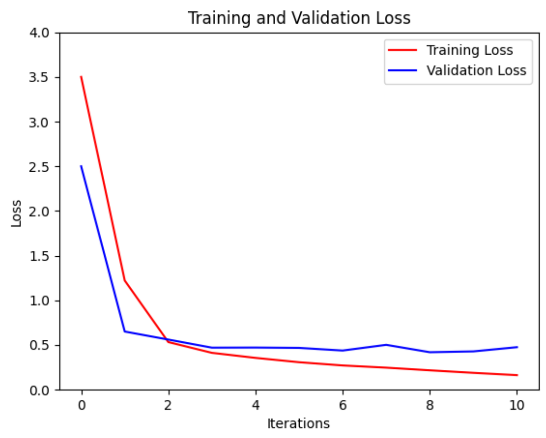
6.1.2 Train a convolutional neural network with PyTorch on the included NIST36 dataset. Compare its performance with the previous fully-connected network.

The implementation for this convolution neural network using PyTorch can be found in Figure 38 below.

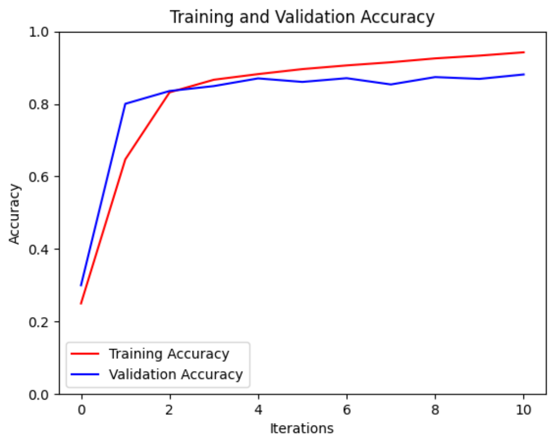


```
class ConvNet(nn.Module):
    def __init__(self, num_classes=36):
        super(ConvNet, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 20, 5, 1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2)
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(20, 50, 5, 1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2)
        )
        self.fc1 = nn.Linear(50 * 5 * 5, 512)
        self.fc2 = nn.Linear(512, num_classes)

    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = x.view(-1, 5 * 5 * 50)
        x = torch.nn.functional.relu(self.fc1(x))
        x = self.fc2(x)

        return x
```

Figure 38: ConvNet Implementation



(a) Loss for Network



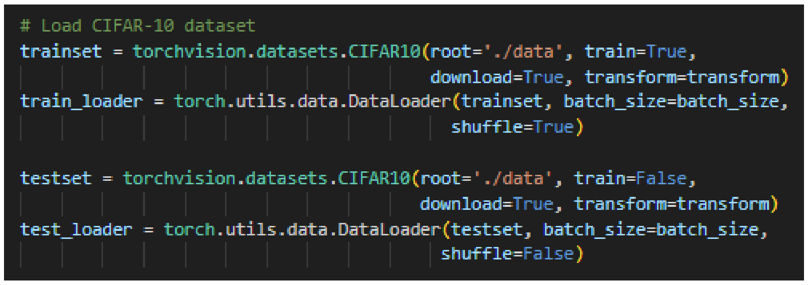
(b) Accuracy for Network

Figure 39: Loss and Accuracy Figures for PyTorch ConvNet Model

If we compare the previous neural network to the current convolution neural network, what we can see here is that the new ConvNet converged to steady state much faster than the previous one. Along with that, we see that the accuracy increased from around 0.75 to 0.9 for the validation set. In terms of the shape, both the loss and accuracy, experience a sharp drop/increase for the first two epochs and then slows down much more.

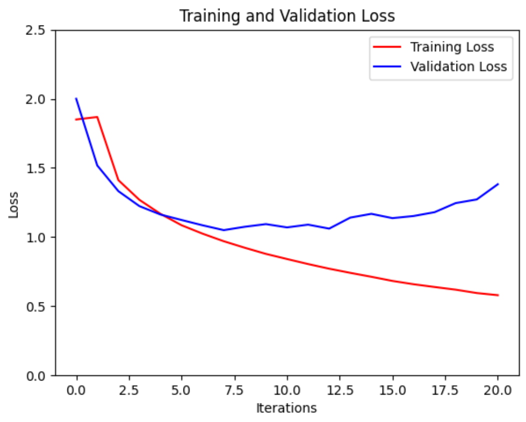
6.1.3 Train a convolutional neural network with PyTorch on CIFAR-10 (torchvision.datasets.CIFAR10). Plot training accuracy and loss over time.

The implementation for this convolution neural network using PyTorch on CIFAR-10 can be found in Figure 40 below.

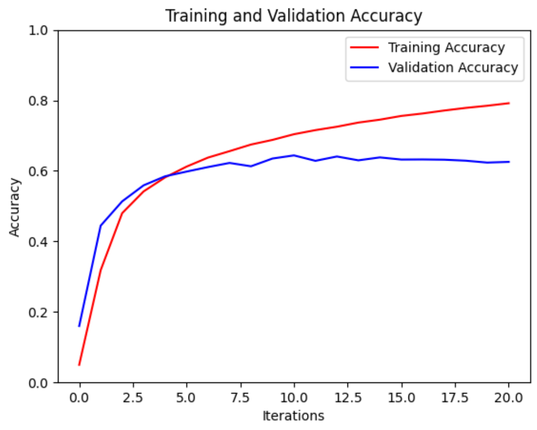


```
# Load CIFAR-10 dataset
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
train_loader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                           shuffle=True)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
test_loader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                          shuffle=False)
```

Figure 40: CIFAR-10 Implementation



(a) Loss for Network



(b) Accuracy for Network

Figure 41: Loss and Accuracy Figures for PyTorch on CIFAR-10