16-720 Computer Vision: Homework 5 (Spring 2023)
Neural Networks for Recognition

Instructor: Deva Ramanan
TAs: : Kangle Deng, Vidhi Jain, Xiaofeng Guo, Chung Hee Kim, Ingrid Navarro
Released on: April 12, 2023
Due: April 28, 2023

Instructions

1. **Integrity and collaboration:** Students are encouraged to work in groups but each student must submit their own work. Include the names of your collaborators in your write-up. Code should **NOT** be shared or copied. Please **DO NOT** use external code unless permitted. Plagiarism is prohibited and may lead to failure in this course.

2. Answer each question (for points) marked with a **Q** in the corresponding titled boxes.

3. **Start early!** This homework may take a long time to complete.

4. **Attempt to verify your implementation as you proceed.** If you don't verify that your implementation is correct on toy examples, you will risk having a huge mess when you put everything together.

5. **In your PDF, start a new page for each question and indicate the answer/page(s) correspondence carefully when submitting on Gradescope.** For some questions, this may leave a lot of blank space. If you skip a written question, just submit a blank page for it. This makes your work much easier to grade.

6. **Some questions will ask you to "Include your code in the writeup".** For those questions, you can either copy/paste the code into a `verbatim` environment or include screenshots of your code.

7. If you have any questions or need clarifications, please post in Slack or visit the TAs during office hours.

8. **Submission:** The submission is on Gradescope, **you will be submitting both your writeup and code zip file**. The zip file, `<andrew-id.zip>` contains your code and any results files we ask you to save. **Note: You have to submit your write-up separately to Gradescope, and include results (and code snippets, when requested) in the write-up**.

9. **Do not** submit anything from the `data/` folder in your submission.

10. For your code submission, **do not** use any libraries other than *numpy, scipy, scikit-image, matplotlib* and (in the appropriate section) *pytorch*. Including other libraries (for example, cv2, etc.) **may lead to loss of credit** on the assignment.

11. To get the data for Section 3 onwards, we have included some scripts in `scripts`. For those who are on operating systems like Windows and cannot run the .sh script, you can also manually download by clicking on the link and unzipping the data. Download and unzip

    `http://www.cs.cmu.edu/~lkeselma/16720a_data/data.zip`

    `http://www.cs.cmu.edu/~lkeselma/16720a_data/images.zip`

    and extract them to `data` and `image` folders.

12. For each coding question, refer to the comments inside the given Python scripts to see which function to implement. Insert your code into the designated place (where it says `your code here`), and **DO NOT** remove any of the given code elsewhere.

13. Feel free to use

    ```
    \codesection{name-of-file.py}
    ```

    to include the code for each question. Mark the pages for each question appropriately.

14. Increase the size of the answer blocks as required.

15. To include figures in the answer blocks, use only **'includegraphics'** but not inside of a **'figure'** block. This is because **'figure'** is a floating environment, but you want to place your graphics in a particular place. Or feel free to define the **figure** block outside the **your_solution** block.

Contents

1 Theory

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

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

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

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

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

Q1.1

$softmax(x_i + c) = \frac{e^{x_i+c}}{\sum_j e^{x_j+c}}$
$\implies softmax(x_i + c) = \frac{e^c e^{x_i}}{e^c \sum_j e^{x_j}}$
$\implies softmax(x_i + c) = \frac{e^{x_i}}{\sum_j e^{x_j}}$
$\implies softmax(x_i + c) = softmax(x_i)$
Therefore, in general, $softmax(x + c) = softmax(x)$

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

1. As $x \in \mathbb{R}^d$, what are the properties of $softmax(x)$, namely what is the range of each element of the $softmax(x)$? What is the sum of all elements in the $softmax(x)$?

Q1.2.1

The range of each element in $softmax(x)$ should be (0,1). All elements of the functions should sum to 1.

2. One could say that *"softmax takes an arbitrary real valued vector x and turns it into a ______________"*.

Q1.2.2

Probability distribution

3. What is the role of each step in the three-step process to compute $softmax(x)$? Explain.

Q1.2.3

Step 1 weights the data. Step 2 creates a sum which is averaged in Step 3 to normalize the data into a probability distribution.

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

Q1.3

Linear regression is the process of fitting data into a linear model. A multi-layer network with only linear activation functions will output a linear model. This means that the network would be equivalent to linear regression.
Taking a two layer network with only linear activation functions.
layer 1 $\implies y_1 = C_1(Ax+b)+C_2$
layer 2 $\implies y_2 = C_3(y_1)+C_4$
$\implies y_2 = C_3(C_1(Ax+b)+C_2)+C_4$
Layer 2 is still a linear model and will result in linear regression.

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

Q1.4

$\sigma' = \mathbf{\frac{1}{1+e^{-x}}}$
$\implies \sigma' = \frac{e^{-x}}{(1+e^{-x})^2}$
We can modify the sigmoid equation to get e^{-x} in terms of sigma.
$\implies \mathbf{e^{-x}} = \mathbf{\frac{1-\sigma}{\sigma}}$
We can also rewrite $\frac{1}{(1+e^{-x})^2}$ as σ^2
$\implies \sigma' = \sigma^{\mathbf{2}}(\mathbf{\frac{1-\sigma}{\sigma}}$
$\implies \sigma' = \sigma(\mathbf{1}-\sigma)$

+

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

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

We won't grade the derivative with respect to b but you should do it anyways, you will need it later in the assignment.

Q1.5

Given: $\mathbf{y} = \mathbf{Wx} + \mathbf{b}$
Taking the partial derivative using W, x and b:
$\implies \frac{\partial \mathbf{y}}{\partial \mathbf{W}} = \mathbf{x}, \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathbf{W}$ and $\frac{\partial \mathbf{y}}{\partial \mathbf{b}} = \mathbf{1}$
Using these equation we can determine the gradients.
$\frac{\partial \mathbf{J}}{\partial \mathbf{W}} = \frac{\partial \mathbf{J}}{\partial \mathbf{y}} . \frac{\partial \mathbf{y}}{\partial \mathbf{W}}$
Let $\frac{\partial \mathbf{J}}{\partial \mathbf{y}} = \delta$
$\implies \frac{\partial \mathbf{J}}{\partial \mathbf{W}} = \delta . x$
and
$\frac{\partial \mathbf{J}}{\partial \mathbf{x}} = \delta . W$, and $\frac{\partial \mathbf{J}}{\partial \mathbf{b}} = \delta$
Based on the shapes, the matrix multiplication would look like:
$\frac{\partial \mathbf{J}}{\partial \mathbf{W}} = (\delta^T . x)^T, \frac{\partial \mathbf{J}}{\partial \mathbf{x}} = \delta . W^T, \frac{\partial \mathbf{J}}{\partial \mathbf{b}} = \delta$

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

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

Q1.6.1

Because the sigmoid function is limited to a range of (0,1) if there are many layers in the neural network the ability of the function to recognize changes in gradient is reduced, resulting in what is known as "vanishing gradients".

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?

Q1.6.2

Tanh has a range from (-1, 1) while sigmoid is limited to range of (0, 1). Tanh's range would be preferable in networks with more layers as it would reduce the "vanishing gradient" effect.

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

Q1.6.3

The derivative of $tanh(x)$ is $1 - tanh(x)^2$. Since tanh has a range of (-1,1) this means that its derivative has a range of (0,2). This makes it better suited to tackle the vanishing gradient problem as it would be able to recognize a larger change in input.

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

Q1.6.4

$\mathbf{tanh}(\mathbf{x}) = \frac{\mathbf{1-e^{-2x}}}{\mathbf{1+e^{-2x}}} = \frac{1}{1+e^{-2x}} - \frac{e^{-2x}}{1+e^{-2x}}$
From the formulas established in 1.4, we get:
$\mathbf{tanh}(\mathbf{x}) = \sigma(\mathbf{2x}) - \sigma(\mathbf{2x})[\frac{\mathbf{1-\sigma(2x)}}{\boldsymbol{\sigma}\mathbf{(2x)}}] = 2\sigma(2x) - 1$

2 Implement a Fully Connected Network

When implementing the following functions, make sure you run `python/run_q2.py` along the way to check if your implemented functions work as expected.

2.1 Network Initialization

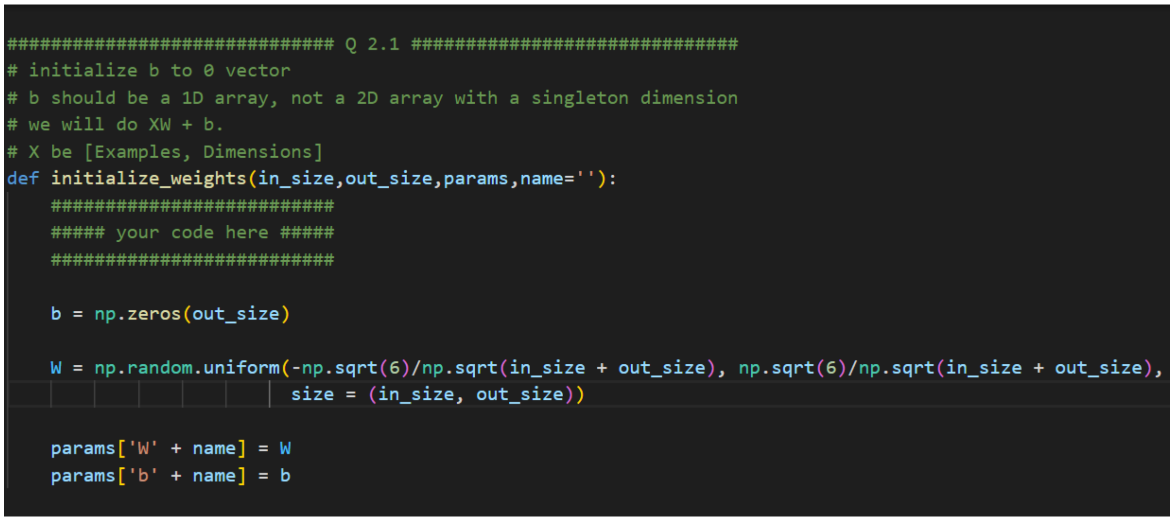
Q2.1.1 Theory [3 points] Why is it not a good idea to initialize a network with all zeros? If you imagine that every layer has weights and biases, what can a zero-initialized network output be after training?

Q2.1.1

Initializing a network with all zeros will cause it perform the same calculations in each iteration and produce the same outputs. A zero-initialized network's output after training will likely be as good as randomly guessing the solution.

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

Q2.1.2



```
############################## Q 2.1 ##############################
# initialize b to 0 vector
# b should be a 1D array, not a 2D array with a singleton dimension
# we will do XW + b.
# X be [Examples, Dimensions]
def initialize_weights(in_size,out_size,params,name=''):
    ##########################
    ##### your code here #####
    ##########################

    b = np.zeros(out_size)

    W = np.random.uniform(-np.sqrt(6)/np.sqrt(in_size + out_size), np.sqrt(6)/np.sqrt(in_size + out_size),
                      size = (in_size, out_size))

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

Q2.1.3 Theory [2 points] Why do we initialize with random numbers? Why do we scale the initialization depending on layer size (see Fig 6 in the [Glorot et al])?

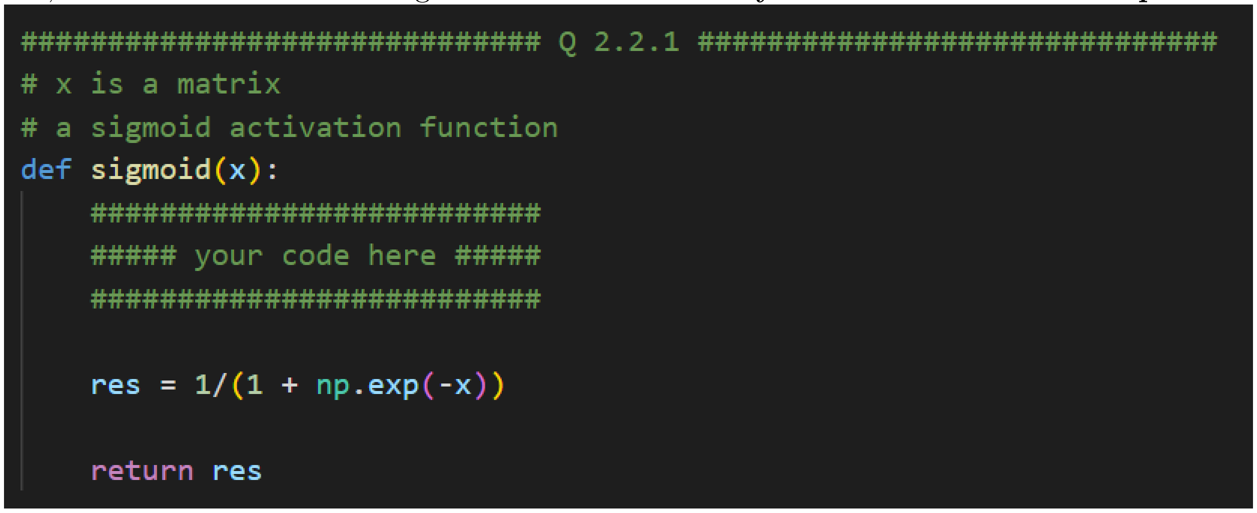
Q2.1.3

We initialize with random numbers to break any symmetry between neurons. If the initializations are symmetric, i.e., the same value, the resultant gradients after backpropagation would be the same resulting in a model that doesn't learn effectively. We scale initializations depending on the layer size to make sure that the outputs share statistical similarities, i.e., the activation value distributions look the same.

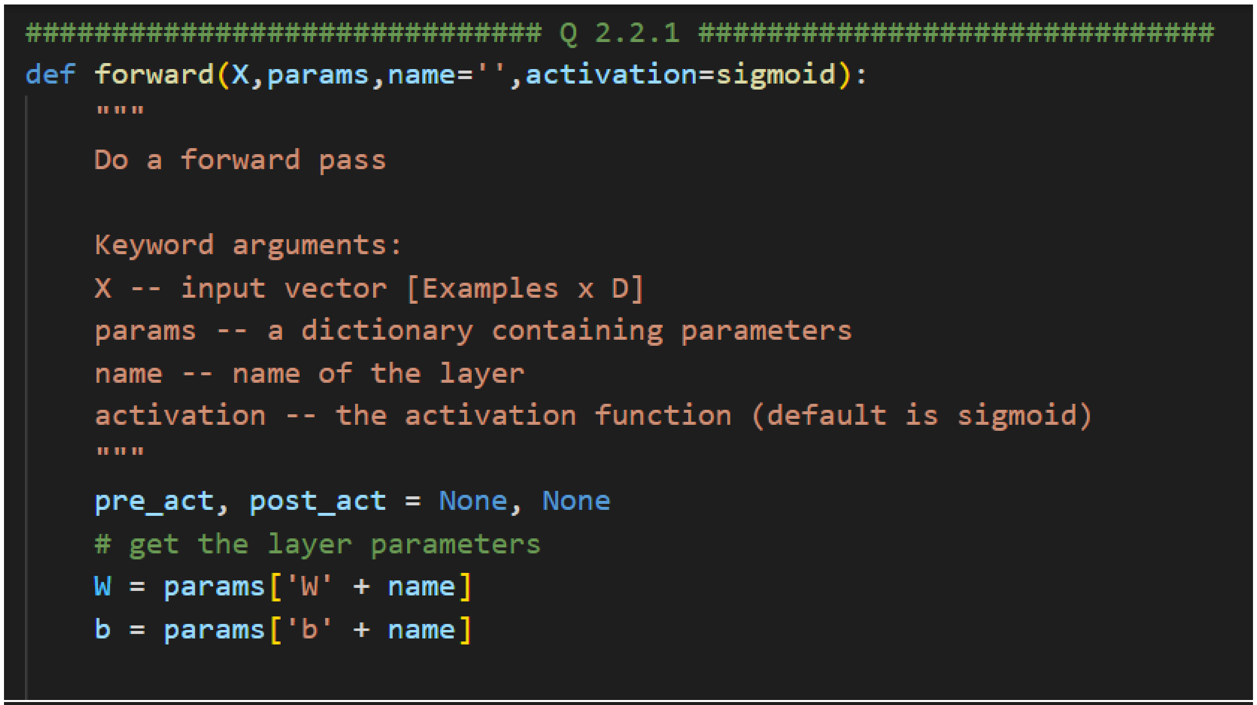
2.2 Forward Propagation

The appendix (sec 6.3) has the math for forward propagation, we will implement it here.

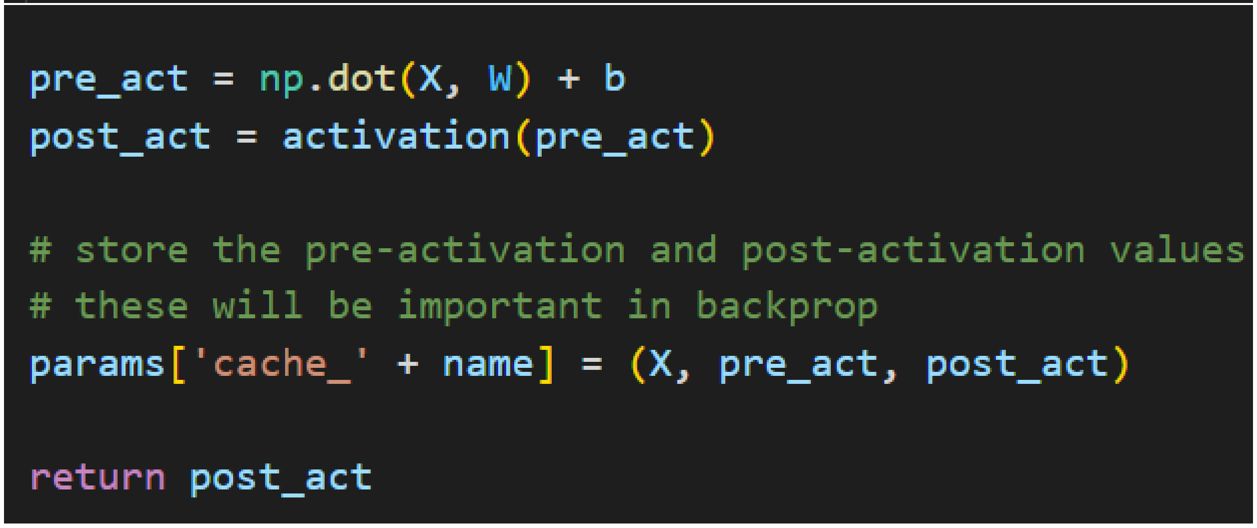
Q2.2.1 Code [4 points] In `python/nn.py`, implement sigmoid, along with forward propagation for a single layer with an activation function, namely $y = \sigma(XW + b)$, returning the output and intermediate results for an $N \times D$ dimension input X, with examples along the rows, and data dimensions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



```python
############################## Q 2.2.1 ##############################
# x is a matrix
# a sigmoid activation function
def sigmoid(x):
    ##########################
    ##### your code here #####
    ##########################

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

    return res
```



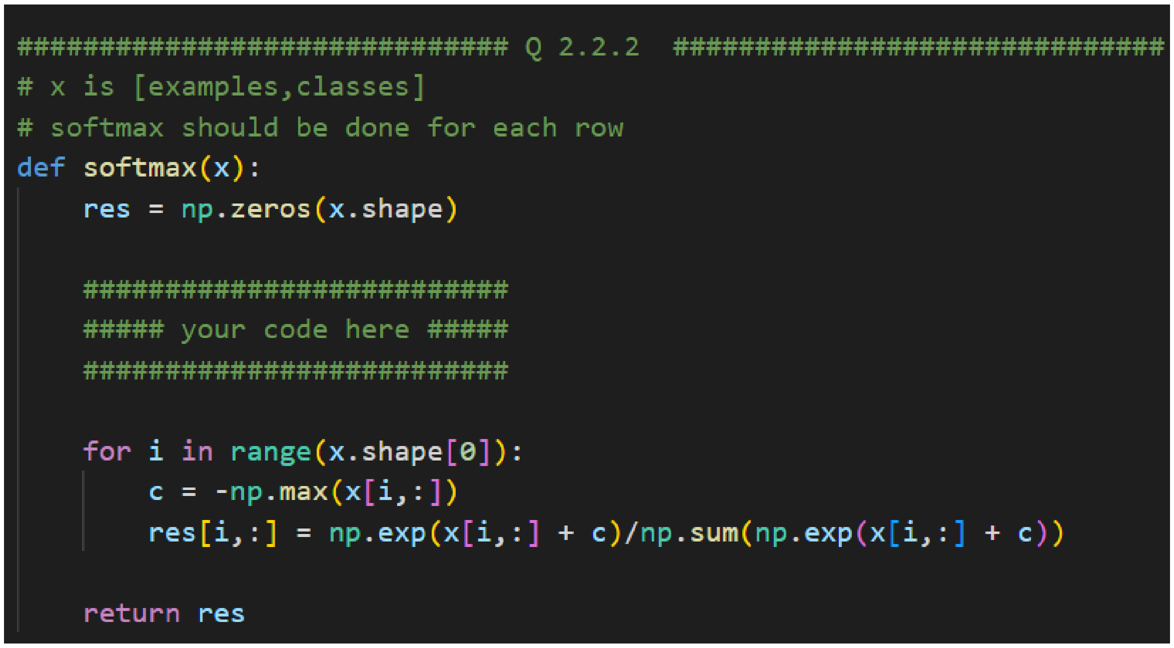
```python
############################## Q 2.2.1 ##############################
def forward(X,params,name='',activation=sigmoid):
    """
    Do a forward pass

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



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

Q2.2.2

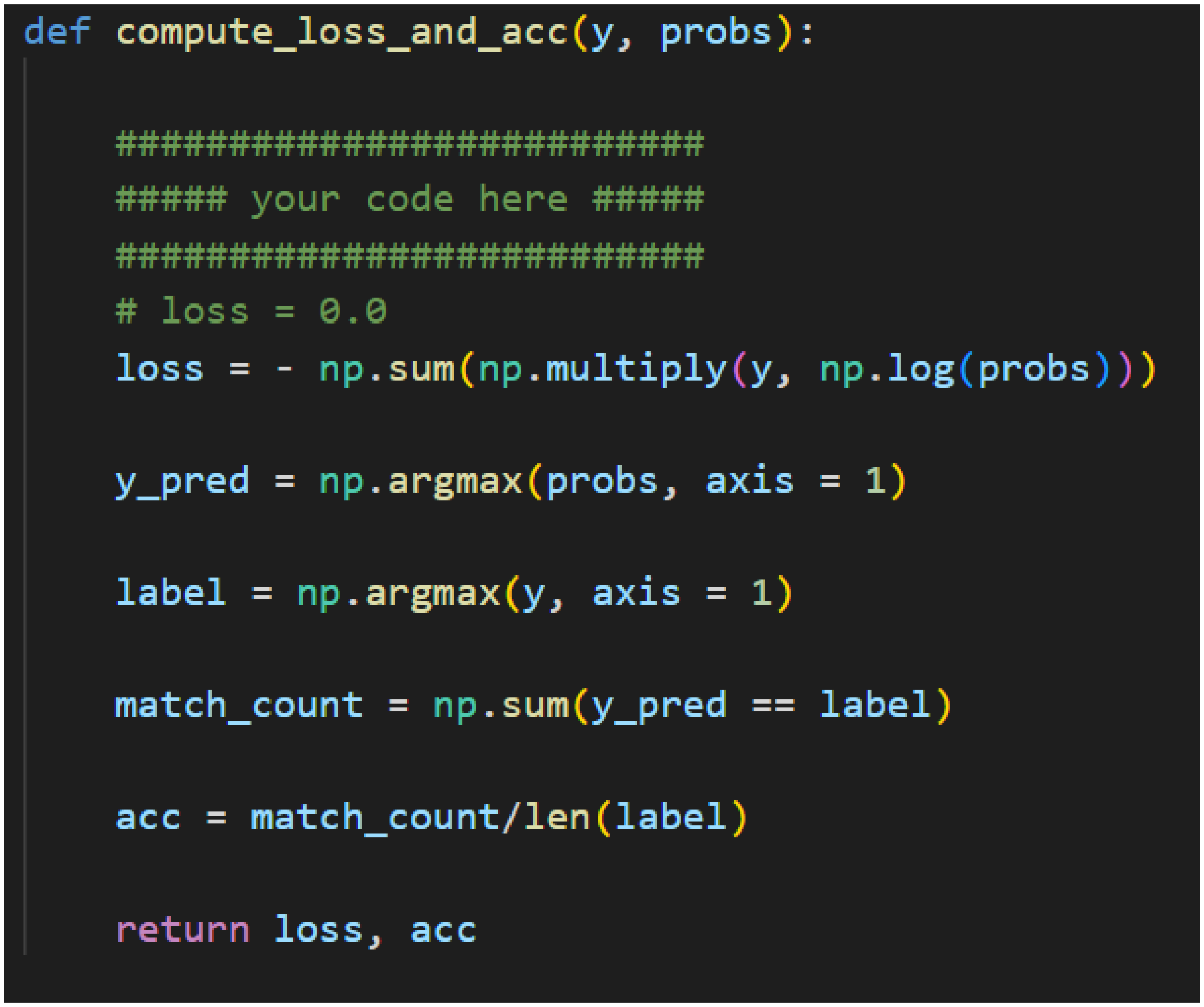


```
############################## Q 2.2.2  ##############################
# x is [examples,classes]
# softmax should be done for each row
def softmax(x):
    res = np.zeros(x.shape)

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

    for i in range(x.shape[0]):
        c = -np.max(x[i,:])
        res[i,:] = np.exp(x[i,:] + c)/np.sum(np.exp(x[i,:] + c))

    return res
```

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

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

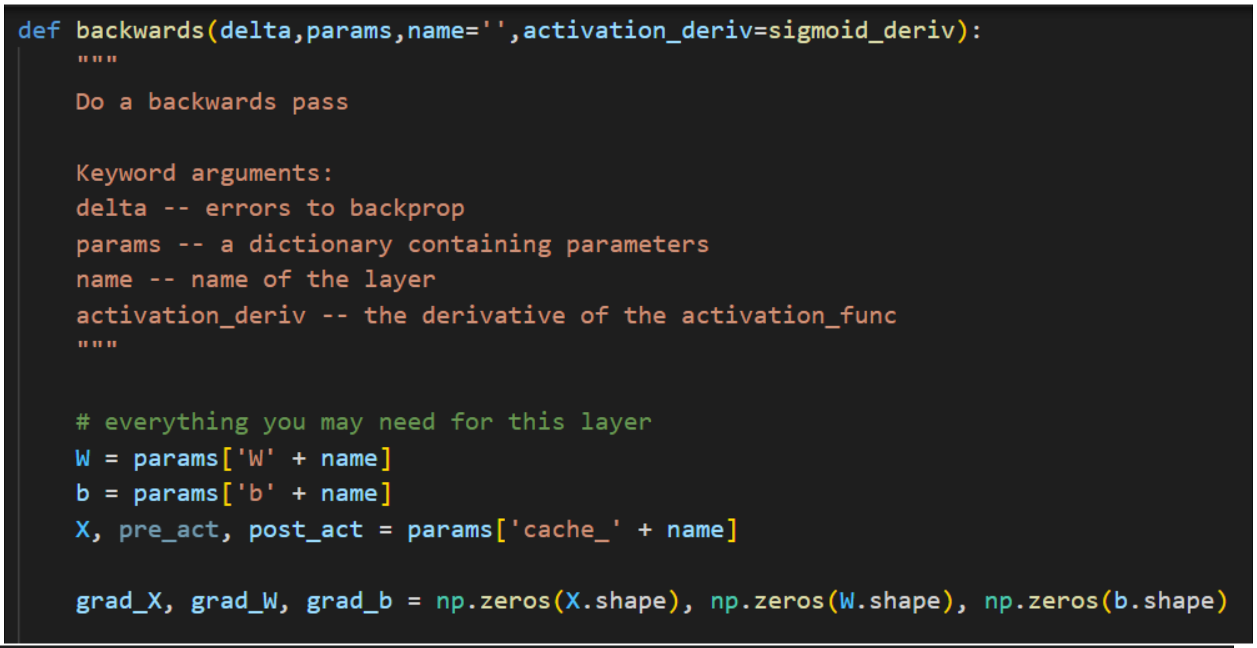
Here $\mathbf{D}$ is the full training dataset of data samples $\mathbf{x}$ ($N \times 1$ vectors, N = dimensionality of data) and labels $\mathbf{y}$ ($C \times 1$ one-hot vectors, C = number of classes), and $\mathbf{f} : \mathbb{R}^N \to [0,1]^C$ is the classifier. The log is the natural log. **Include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



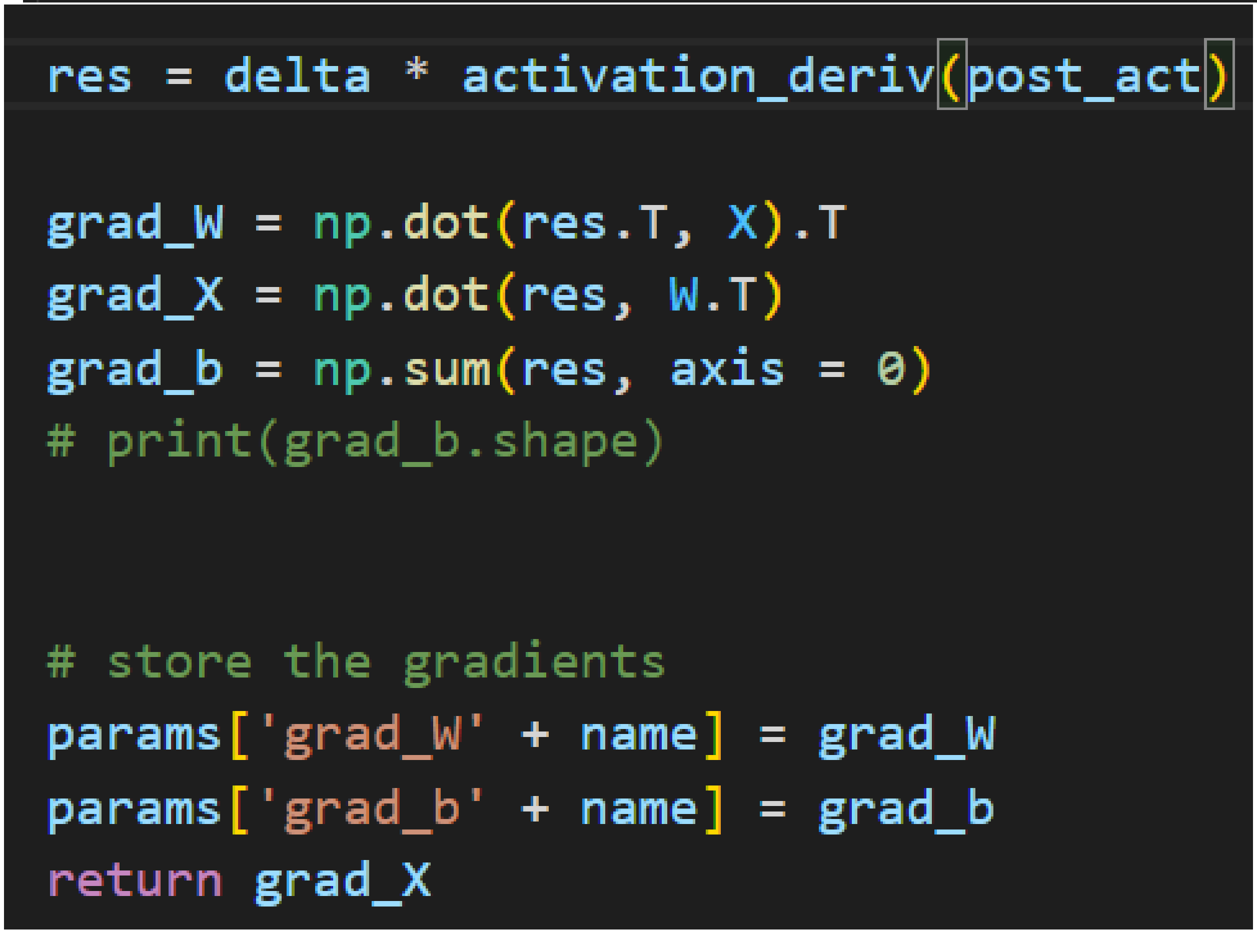
```python
def compute_loss_and_acc(y, probs):

    ############################
    ##### your code here #####
    ############################
    # loss = 0.0
    loss = - np.sum(np.multiply(y, np.log(probs)))

    y_pred = np.argmax(probs, axis = 1)

    label = np.argmax(y, axis = 1)

    match_count = np.sum(y_pred == label)

    acc = match_count/len(label)

    return loss, acc
```

2.3 Backwards Propagation

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



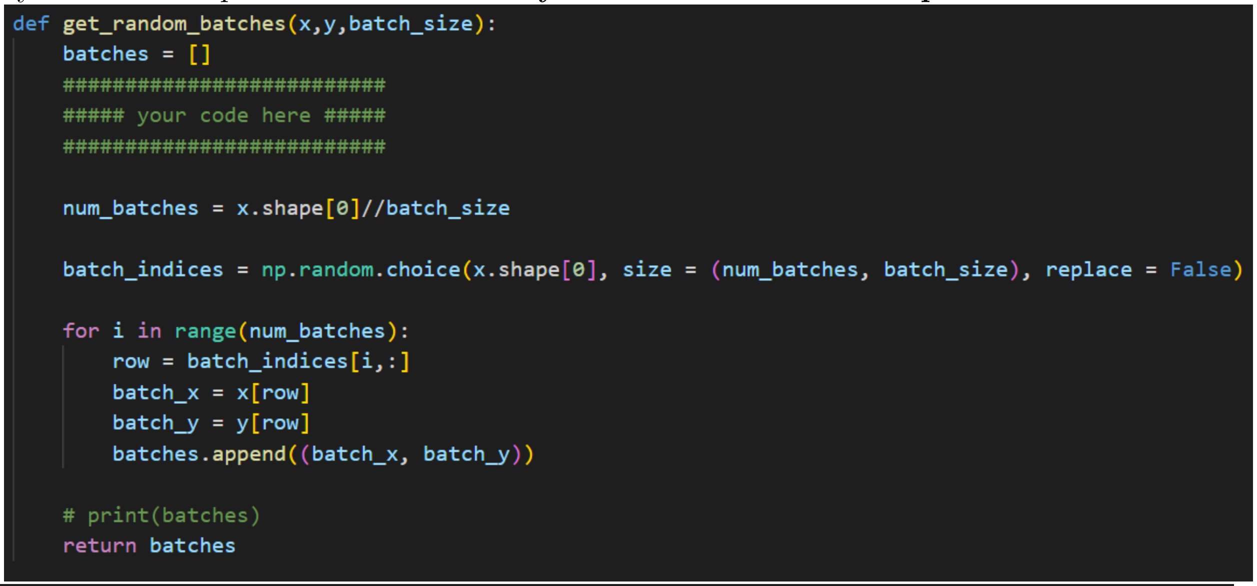
```
def backwards(delta,params,name='',activation_deriv=sigmoid_deriv):
    """
    Do a backwards pass

    Keyword arguments:
    delta -- errors to backprop
    params -- a dictionary containing parameters
    name -- name of the layer
    activation_deriv -- the derivative of the activation_func
    """

    # everything you may need for this layer
    W = params['W' + name]
    b = params['b' + name]
    X, pre_act, post_act = params['cache_' + name]

    grad_X, grad_W, grad_b = np.zeros(X.shape), np.zeros(W.shape), np.zeros(b.shape)
```



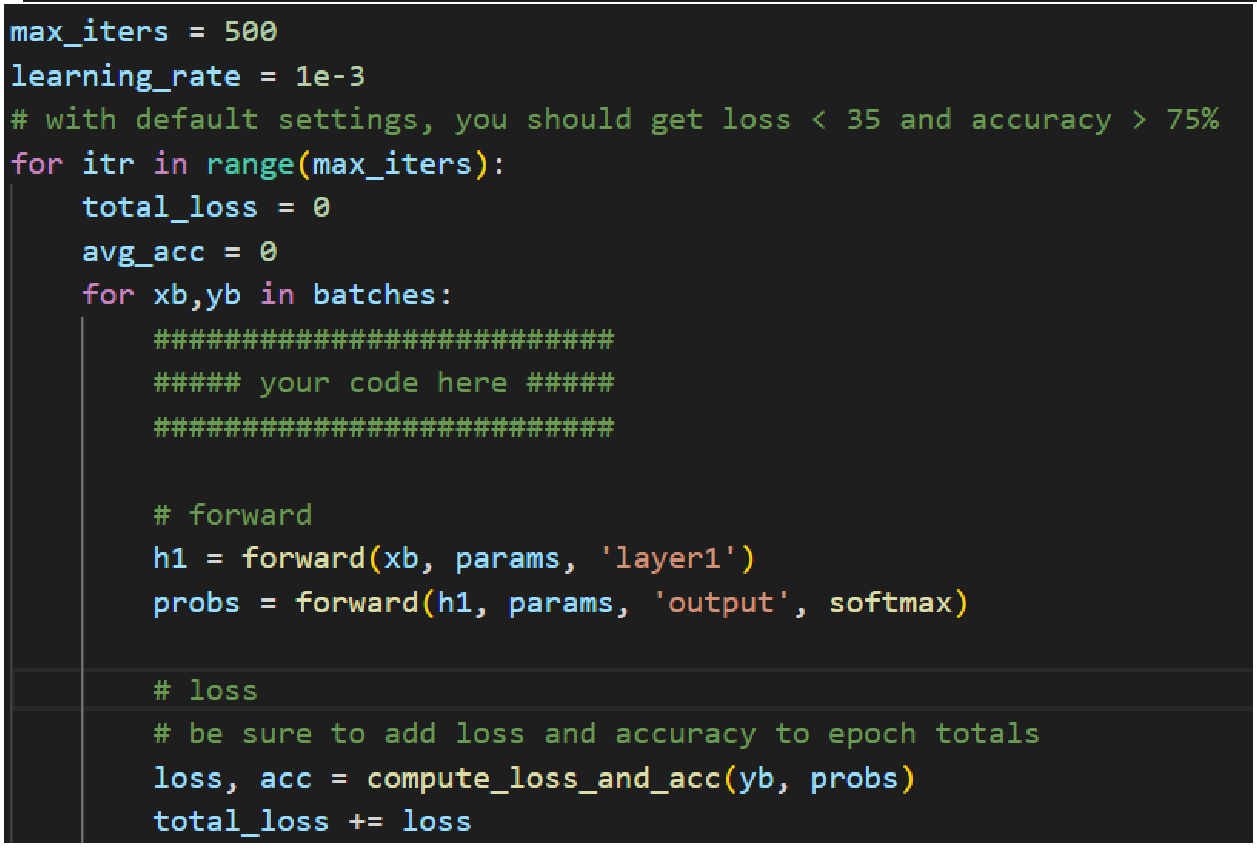
```
res = delta * activation_deriv(post_act)

grad_W = np.dot(res.T, X).T
grad_X = np.dot(res, W.T)
grad_b = np.sum(res, axis = 0)
# print(grad_b.shape)


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

2.4 Training Loop: Stochastic Gradient Descent

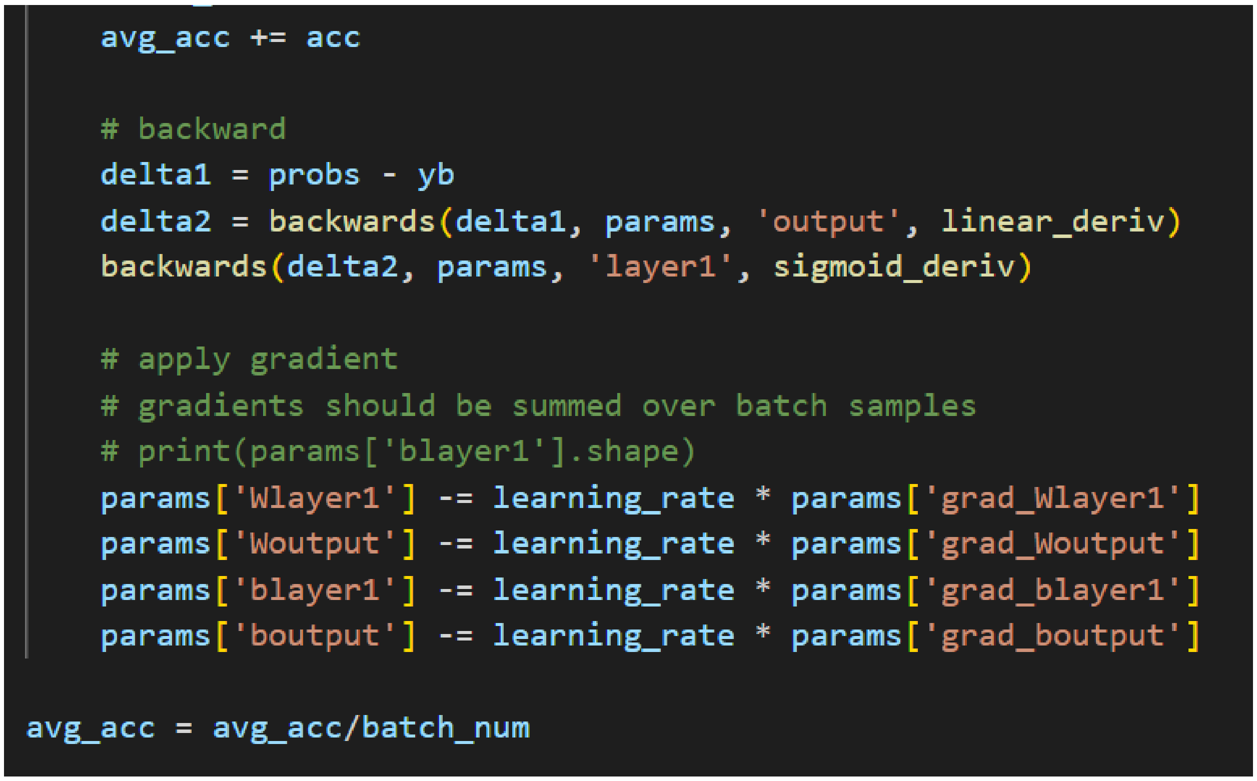
Q2.4 Code [5 points] In `python/nn.py`, write a function that takes the entire dataset (`x` and `y`) as input and splits it into random batches. In `python/run_q2.py`, write a training loop that iterates over the batches, does forward and backward propagation, and applies a gradient update. The provided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



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

    num_batches = x.shape[0]//batch_size

    batch_indices = np.random.choice(x.shape[0], size = (num_batches, batch_size), replace = False)

    for i in range(num_batches):
        row = batch_indices[i,:]
        batch_x = x[row]
        batch_y = y[row]
        batches.append((batch_x, batch_y))

    # print(batches)
    return batches
```



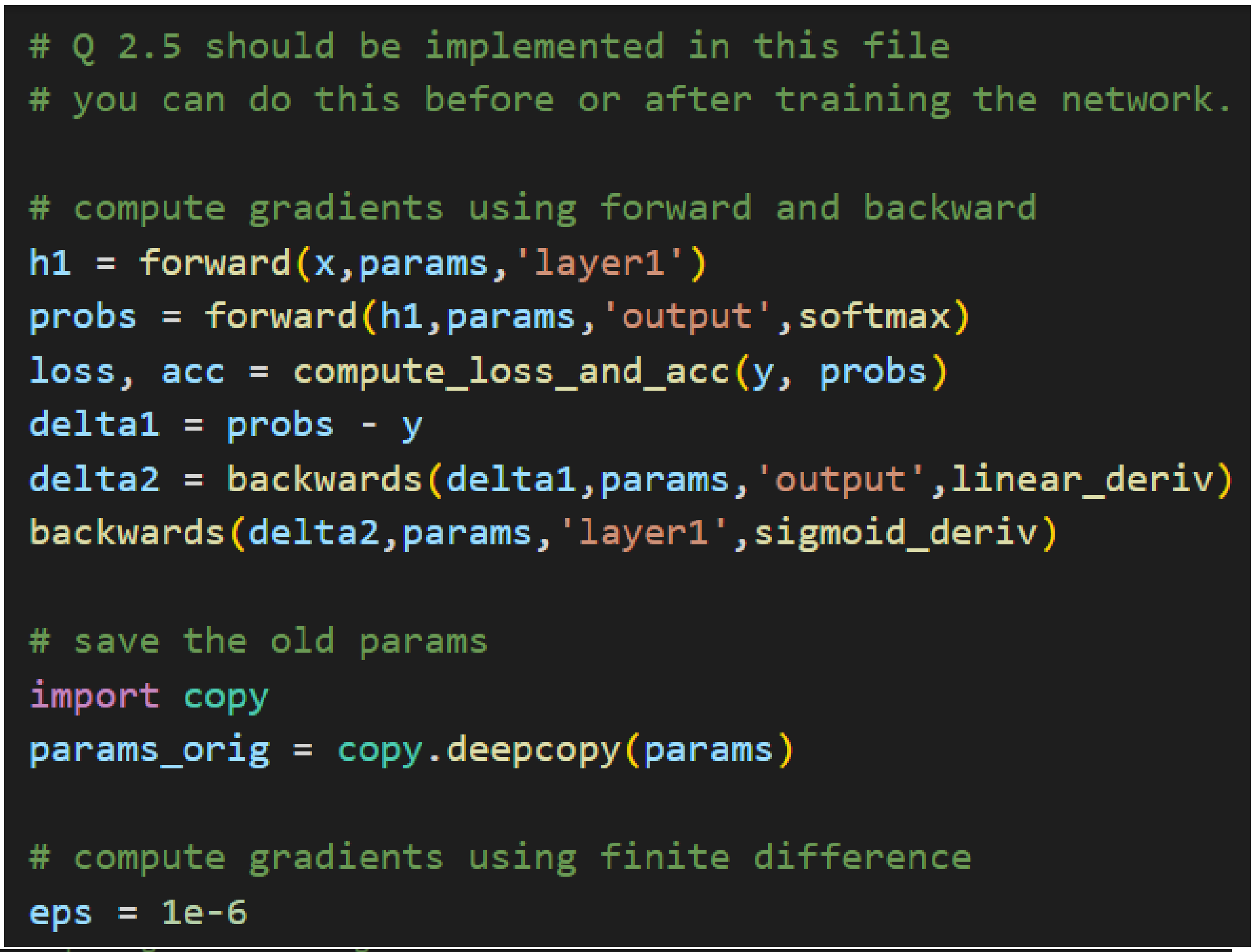
```python
max_iters = 500
learning_rate = 1e-3
# with default settings, you should get loss < 35 and accuracy > 75%
for itr in range(max_iters):
    total_loss = 0
    avg_acc = 0
    for xb,yb in batches:
        ##########################
        ##### your code here #####
        ##########################

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

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



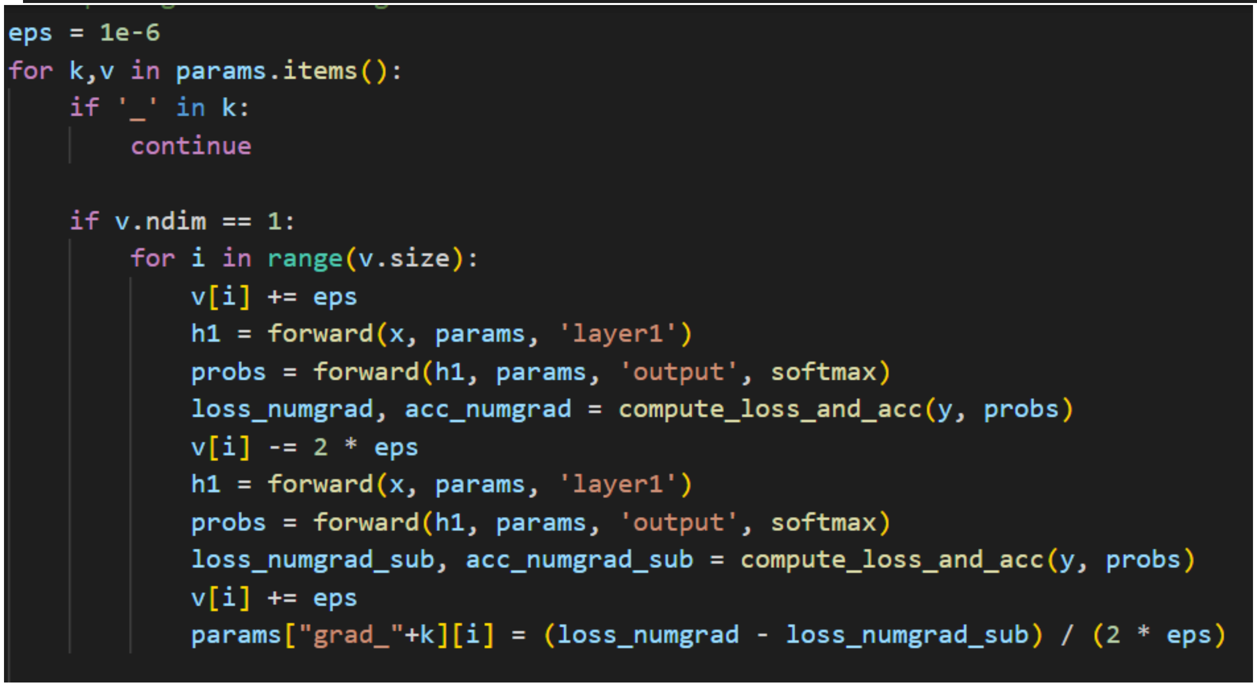
```
    avg_acc += acc

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

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

avg_acc = avg_acc/batch_num
```

2.5 Numerical Gradient Checker

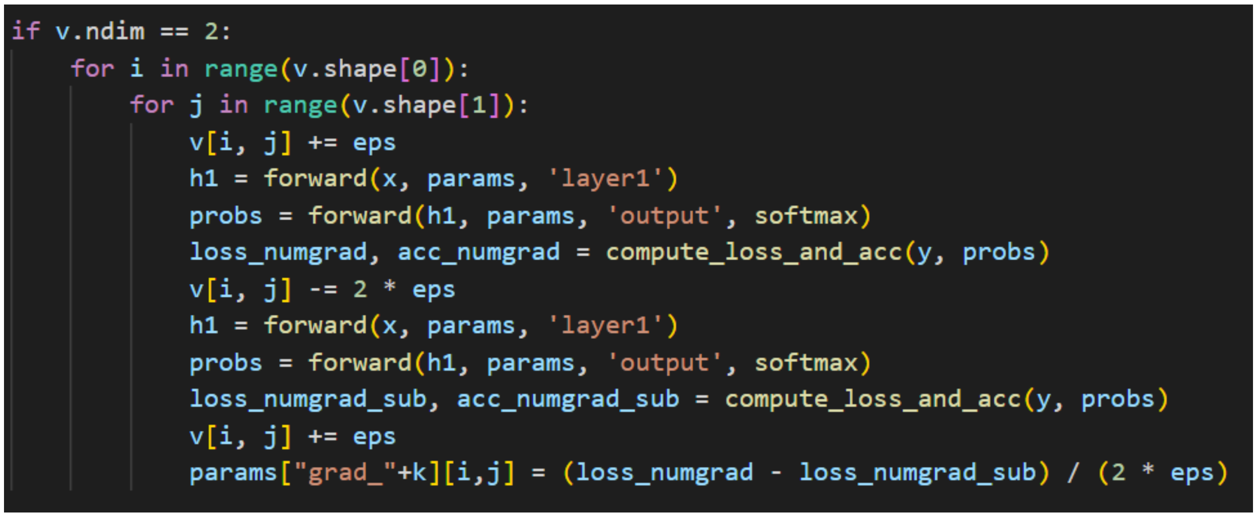
Q2.5 Code [5 points] In `python/run_q2.py`, implement a numerical gradient checker. Instead of using the analytical gradients computed from the chain rule, add ϵ offset to each element in the weights, and compute the numerical gradient of the loss with central differences. Central differences is just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



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

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

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

# compute gradients using finite difference
eps = 1e-6
```



```python
eps = 1e-6
for k,v in params.items():
    if '_' in k:
        continue

    if v.ndim == 1:
        for i in range(v.size):
            v[i] += eps
            h1 = forward(x, params, 'layer1')
            probs = forward(h1, params, 'output', softmax)
            loss_numgrad, acc_numgrad = compute_loss_and_acc(y, probs)
            v[i] -= 2 * eps
            h1 = forward(x, params, 'layer1')
            probs = forward(h1, params, 'output', softmax)
            loss_numgrad_sub, acc_numgrad_sub = compute_loss_and_acc(y, probs)
            v[i] += eps
            params["grad_"+k][i] = (loss_numgrad - loss_numgrad_sub) / (2 * eps)
```



```
if v.ndim == 2:
    for i in range(v.shape[0]):
        for j in range(v.shape[1]):
            v[i, j] += eps
            h1 = forward(x, params, 'layer1')
            probs = forward(h1, params, 'output', softmax)
            loss_numgrad, acc_numgrad = compute_loss_and_acc(y, probs)
            v[i, j] -= 2 * eps
            h1 = forward(x, params, 'layer1')
            probs = forward(h1, params, 'output', softmax)
            loss_numgrad_sub, acc_numgrad_sub = compute_loss_and_acc(y, probs)
            v[i, j] += eps
            params["grad_"+k][i,j] = (loss_numgrad - loss_numgrad_sub) / (2 * eps)
```

3 Training Models

Follow instructions to download the data in `data` and `image` folders.

Since our input images are 32×32 images, unrolled into one 1024-dimensional vector, that gets multiplied by $\mathbf{W}^{(1)}$, each row of $\mathbf{W}^{(1)}$ can be seen as a weight image. Reshaping each row into a 32×32 image can give us an idea of what types of images each unit in the hidden layer has a high response to.

We have provided you three data `.mat` files to use for this section.

The training data in `nist36_train.mat` contains samples for each of the 26 upper-case letters of the alphabet and the 10 digits. This is the set you should use for training your network.

The cross-validation set in `nist36_valid.mat` contains samples from each class, and should be used in the training loop to see how the network is performing on data that it is not training on. This will help to spot overfitting.

Finally, the test data in `nist36_test.mat` contains testing data, and should be used for the final evaluation of your best model to see how well it will generalize to new unseen data.

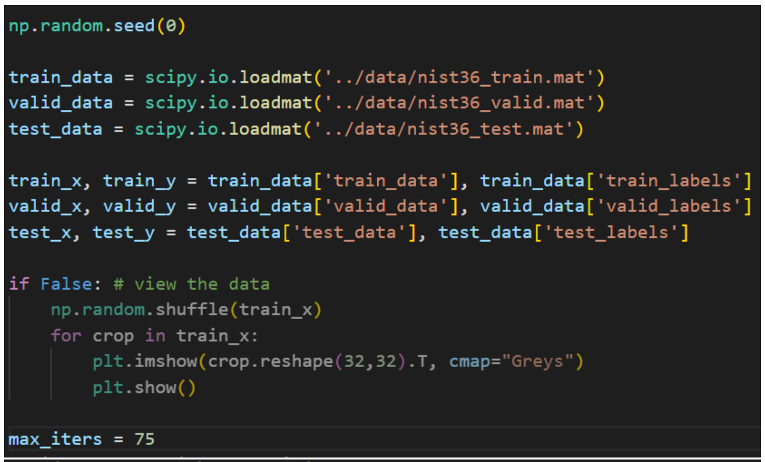
Use `python/run_q3.py` for this question, and refer to the comments for what to implement.

Q3.1 Code [5 points] Train a network from scratch. Use a single hidden layer with 64 hidden units, and train for at least 50 epochs. The script will generate two plots:

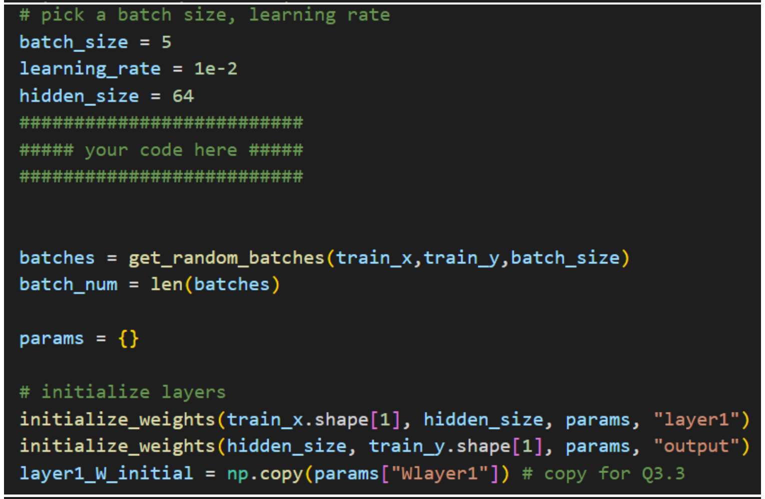
(1) the accuracy on both the training and validation set over the epochs, and

(2) the cross-entropy loss averaged over the data.

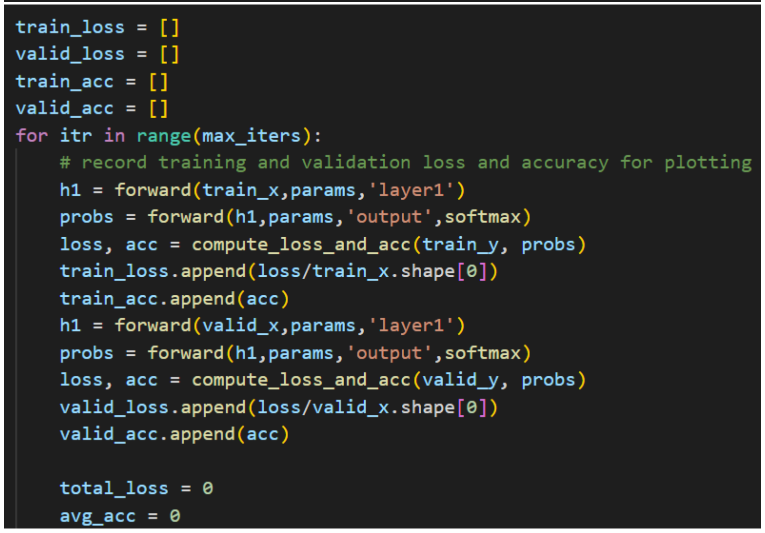
Tune the batch size and learning rate for accuracy on the validation set of at least 75%. Include the plots in your write-up. Hint: Use fixed random seeds to improve reproducibility.



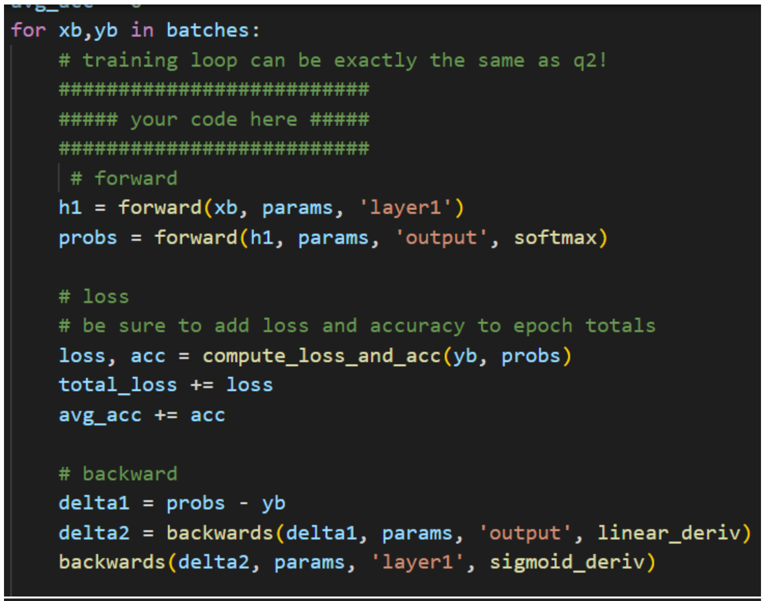
```
np.random.seed(0)

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 = 75
```



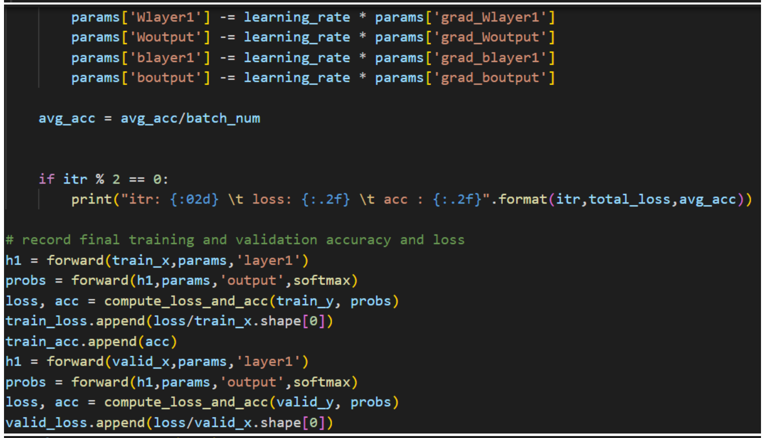
```
# pick a batch size, learning rate
batch_size = 5
learning_rate = 1e-2
hidden_size = 64
##########################
##### your code here #####
##########################


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
```



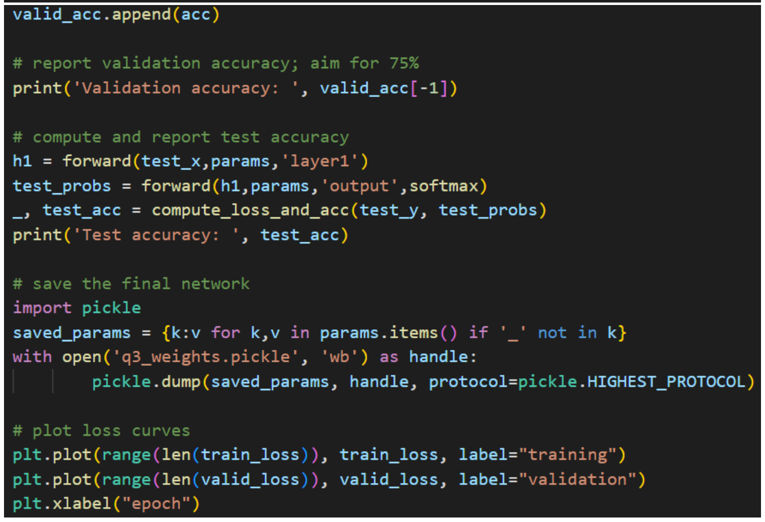
```
train_loss = []
valid_loss = []
train_acc = []
valid_acc = []
for itr in range(max_iters):
    # record training and validation loss and accuracy for plotting
    h1 = forward(train_x,params,'layer1')
    probs = forward(h1,params,'output',softmax)
    loss, acc = compute_loss_and_acc(train_y, probs)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



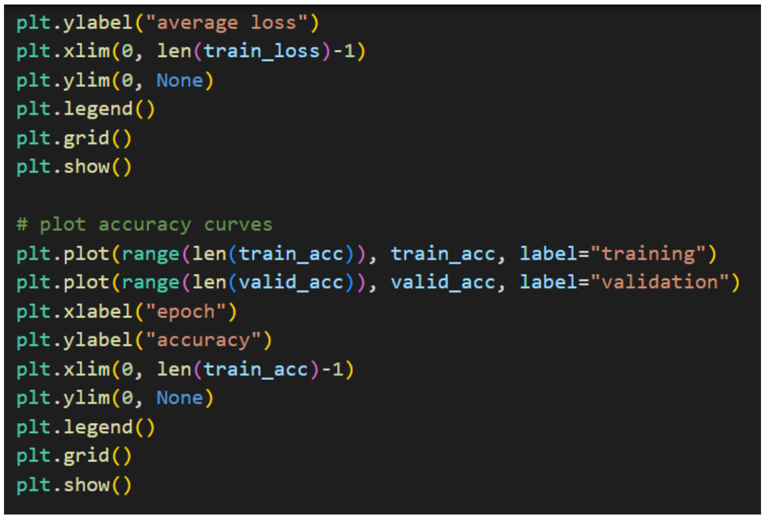
```python
for xb,yb in batches:
    # training loop can be exactly the same as q2!
    ##########################
    ##### your code here #####
    ##########################
    # forward
    h1 = forward(xb, params, 'layer1')
    probs = forward(h1, params, 'output', softmax)

    # 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)
```



```python
        params['Wlayer1'] -= learning_rate * params['grad_Wlayer1']
        params['Woutput'] -= learning_rate * params['grad_Woutput']
        params['blayer1'] -= learning_rate * params['grad_blayer1']
        params['boutput'] -= learning_rate * params['grad_boutput']

    avg_acc = avg_acc/batch_num


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

# record final training and validation accuracy and loss
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)
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])
```

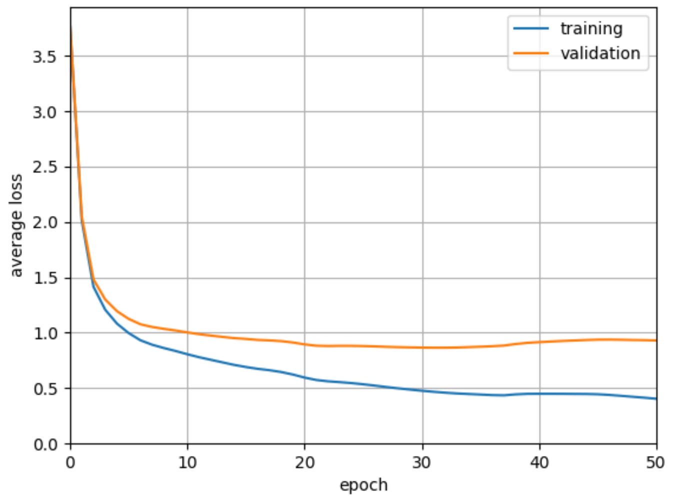


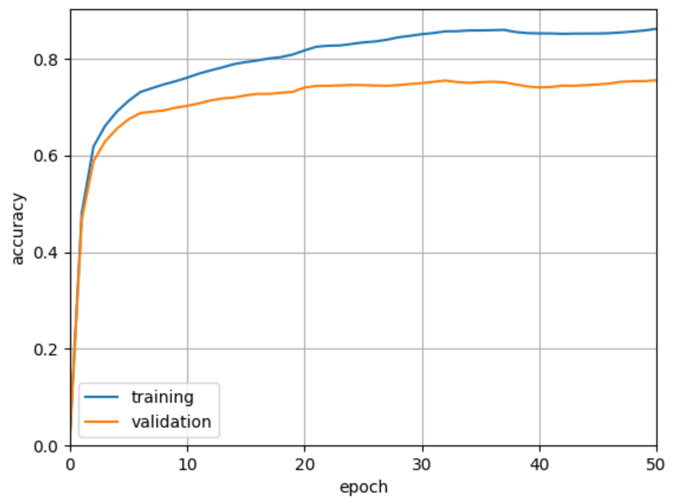
```python
valid_acc.append(acc)

# report validation accuracy; aim for 75%
print('Validation accuracy: ', valid_acc[-1])

# compute and report test accuracy
h1 = forward(test_x,params,'layer1')
test_probs = forward(h1,params,'output',softmax)
_, test_acc = compute_loss_and_acc(test_y, test_probs)
print('Test accuracy: ', test_acc)

# save the final network
import pickle
saved_params = {k:v for k,v in params.items() if '_' not in k}
with open('q3_weights.pickle', 'wb') as handle:
        pickle.dump(saved_params, handle, protocol=pickle.HIGHEST_PROTOCOL)

# plot loss curves
plt.plot(range(len(train_loss)), train_loss, label="training")
plt.plot(range(len(valid_loss)), valid_loss, label="validation")
plt.xlabel("epoch")
```



```python
plt.ylabel("average loss")
plt.xlim(0, len(train_loss)-1)
plt.ylim(0, None)
plt.legend()
plt.grid()
plt.show()

# plot accuracy curves
plt.plot(range(len(train_acc)), train_acc, label="training")
plt.plot(range(len(valid_acc)), valid_acc, label="validation")
plt.xlabel("epoch")
plt.ylabel("accuracy")
plt.xlim(0, len(train_acc)-1)
plt.ylim(0, None)
plt.legend()
plt.grid()
plt.show()
```

Q3.1





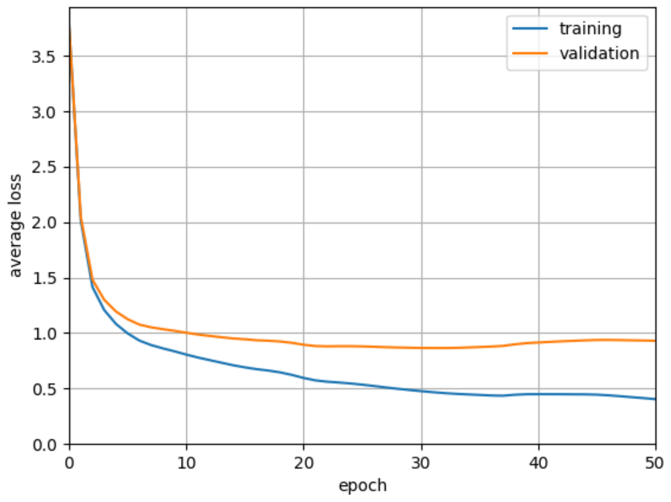
Q3.2 Tune the Learning rate - Writeup [3 points] Use the script in Q3.1 to train and generate accuracy and loss plots for each of these three networks:

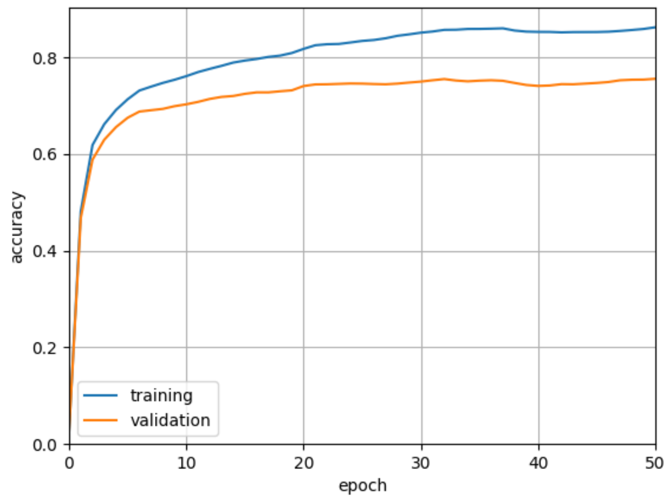
(1) one with your tuned learning rate,
(2) one with 10 times that learning rate, and
(3) one with one-tenth that learning rate.

Include total of six plots in your write-up. Comment on how the learning rates affect the training, and report the final accuracy of the best network on the test set. *Hint: Use fixed random seeds to improve reproducibility.*

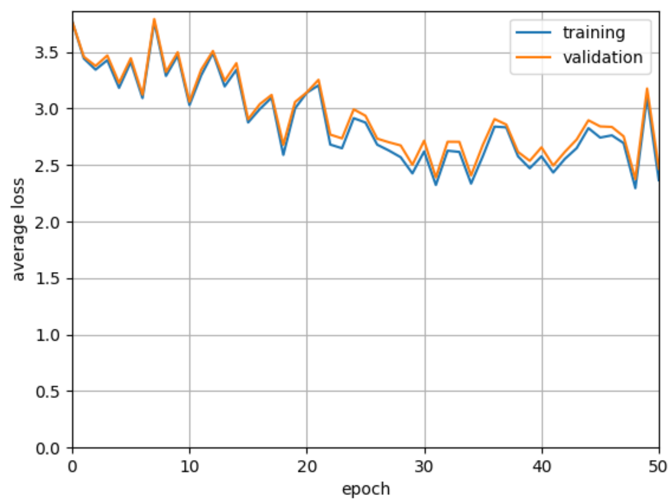
Plots:

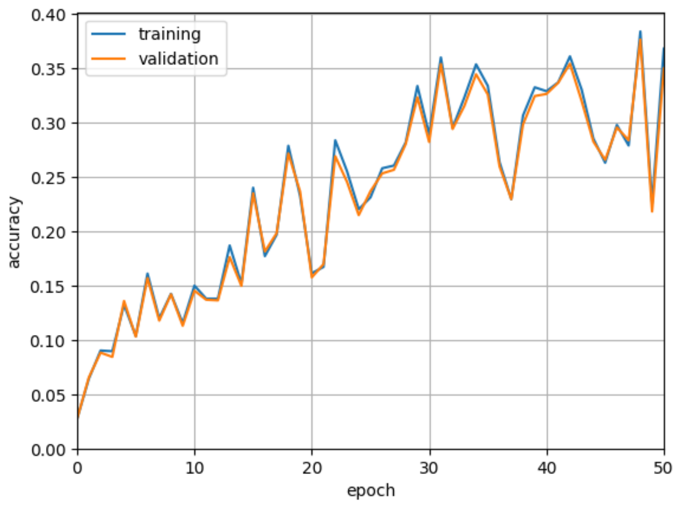
(1) Tuned learning rate:



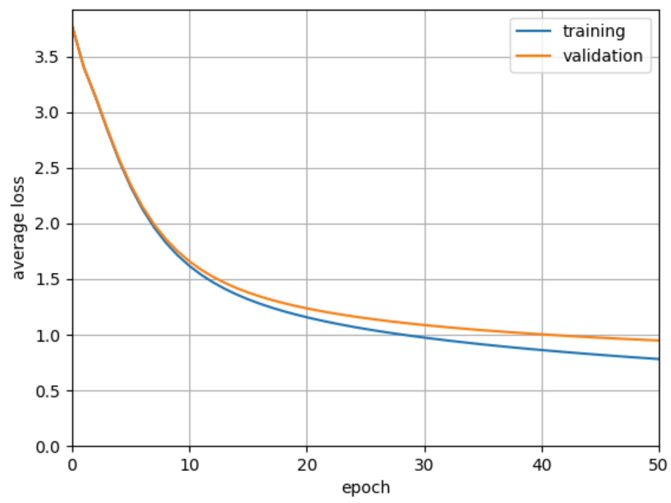


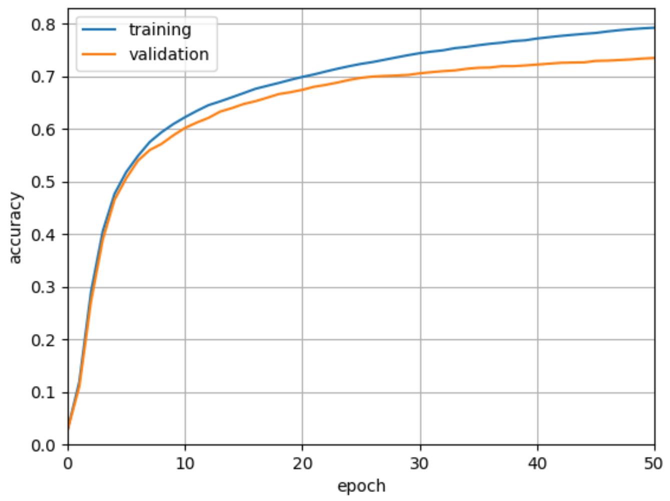
(2) Ten times the learning rate:





(3) One-Tenth the learning rate:

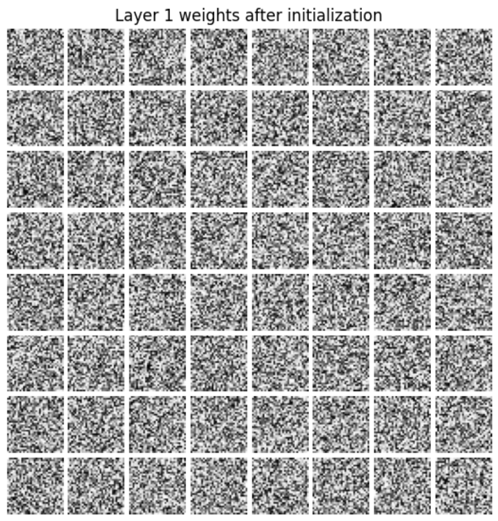


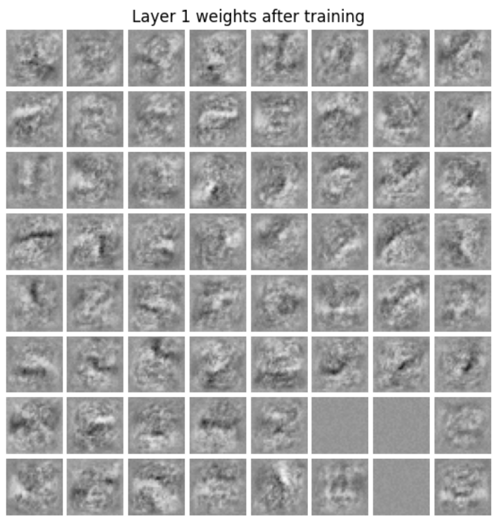


Q3.2

The plots show that increasing the learning rate drastically reduces performance. Decreasing the learning rate shows a slightly lower accuracy than the tuned learning rate. However, the gap between the training and validation curves are much closer than the curves in the tuned plots.

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

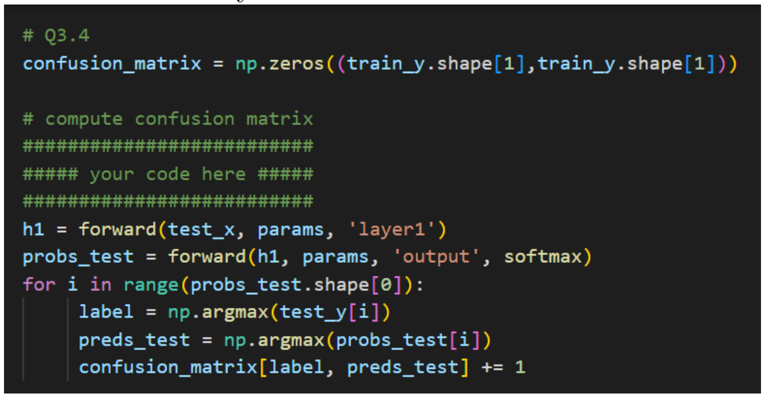




Q3.3

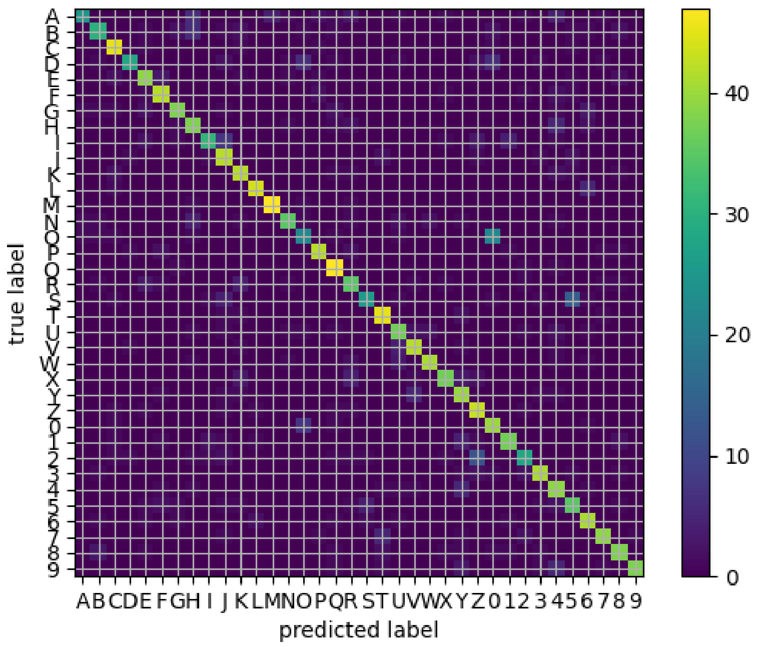
The initialized weights are completely random and show not pattern of any kind. The learned weights, on the other hand, show distinct patterns showing that certain weights are correlated.

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



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

# compute confusion matrix
##########################
##### your code here #####
##########################
h1 = forward(test_x, params, 'layer1')
probs_test = forward(h1, params, 'output', softmax)
for i in range(probs_test.shape[0]):
    label = np.argmax(test_y[i])
    preds_test = np.argmax(probs_test[i])
    confusion_matrix[label, preds_test] += 1
```

Q3.4



The matrix shows us that the most commonly confused classes are the letter 'O' and the number 0 as well 5 and 'S'. Another example is 'Z' and 2. This is all due to the fact that they share similar shapes.

4 Extract Text from Images

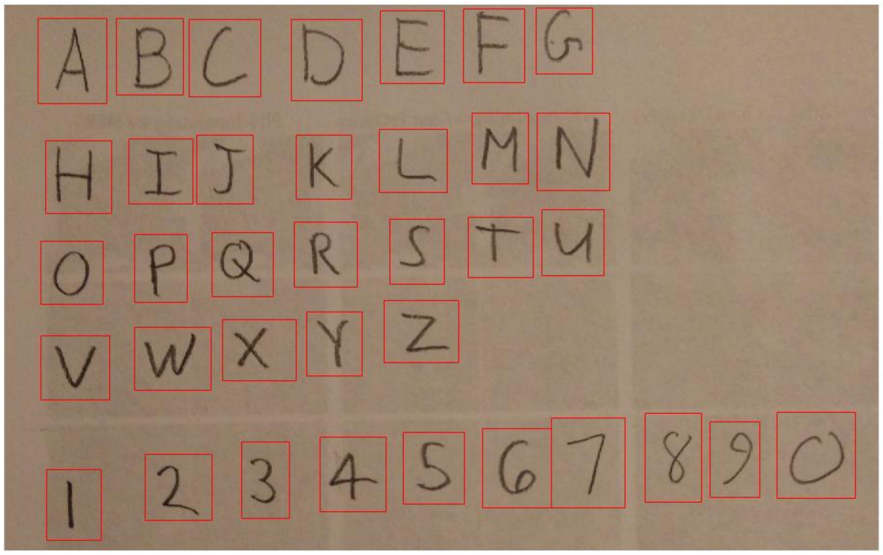


Figure 1: Sample image with handwritten characters annotated with boxes around each character.

Now that you have a network that can recognize handwritten letters with reasonable accuracy, you can now use it to parse text in an image. Given an image with some text on it, our goal is to have a function that returns the actual text in the image. However, since your neural network expects a binary image with a single character, you will need to process the input image to extract each character. There are various approaches that can be done so feel free to use any strategy you like.

Here we outline one possible method, another is given in a tutorial

1. Process the image (blur, threshold, opening morphology, etc., perhaps in that order) to classify all pixels as being part of a character or background.
2. Find connected groups of character pixels (see skimage.measure.label). Place a bounding box around each connected component.
3. Group the letters based on which line of the text they are a part of, and sort each group so that the letters are in the order they appear on the page.
4. Take each bounding box one at a time and resize it to 32×32, classify it with your network, and report the characters in order (inserting spaces when it makes sense).

Since the network you trained likely does not have perfect accuracy, you can expect there to be some errors in your final text parsing. Whichever method you choose to implement for character detection, you should be able to place a box on most of the characters in the image. We have provided you with `01_list.jpg`, `02_letters.jpg`, `03_haiku.jpg` and `04_deep.jpg` to test your implementation on.

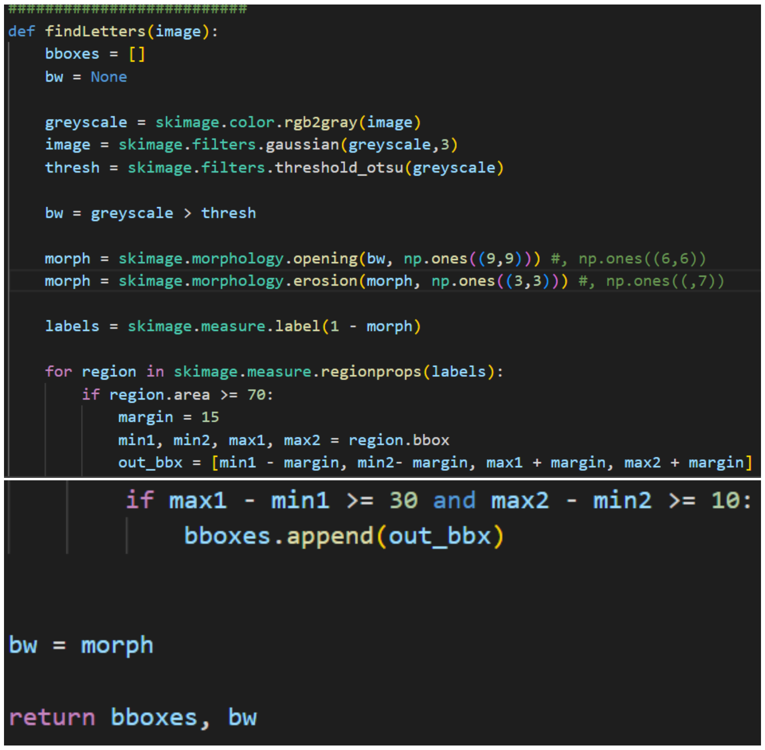
Q4.1 Failures due to Assumptions - Theory [4 points] The method outlined above is pretty simplistic, and while it works for the given text samples, it makes several assumptions. What are two big assumptions that the sample method makes? In your writeup, include two example images where you expect the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

Q4.1

The first major assumption this model makes is that it assumes all characters are distinct and don't share similar features. For example, when written out, the number 0 and the letter 'o' are indistinguishable. This results in a very common misclassifications of the text.
The model also assumes that each character is separated. If the letters are connected and don't have any space between them (like in cursive writing), the bounding boxes are not likely to be able to distinguish between separate letters.

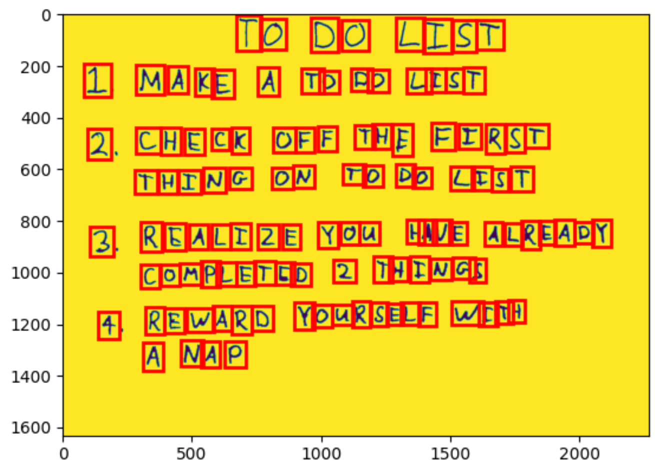
Q4.2 Find Letters - Code [10 points] In `python/q4.py`, implement the function to find letters in the image. Given an RGB image, this function should return bounding boxes for all of the located handwritten characters in the image, as well as a binary black-and-white version of the image `im`. Each row of the matrix should contain `[y1,x1,y2,x2]` the positions of the top-left and bottom-right corners of the box. The black-and-white image should be between 0.0 to 1.0, **with the characters in black and the background in white**. Hint: Since we read text left to right, top to bottom, we can use this to cluster the coordinates. **Include your code in the write-up.**

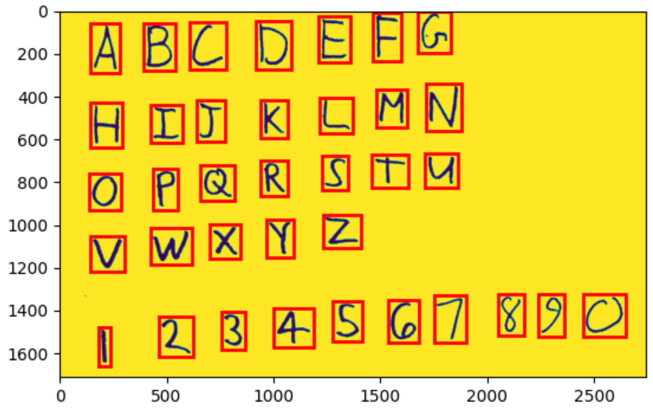
Q4.2



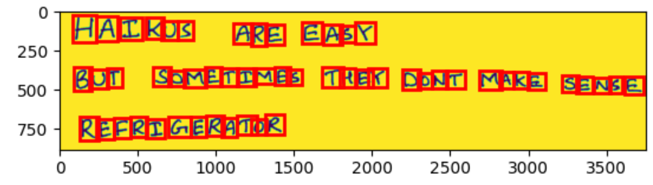
```
##########################
def findLetters(image):
    bboxes = []
    bw = None

    greyscale = skimage.color.rgb2gray(image)
    image = skimage.filters.gaussian(greyscale,3)
    thresh = skimage.filters.threshold_otsu(greyscale)

    bw = greyscale > thresh

    morph = skimage.morphology.opening(bw, np.ones((9,9))) #, np.ones((6,6))
    morph = skimage.morphology.erosion(morph, np.ones((3,3))) #, np.ones((,7))

    labels = skimage.measure.label(1 - morph)

    for region in skimage.measure.regionprops(labels):
        if region.area >= 70:
            margin = 15
            min1, min2, max1, max2 = region.bbox
            out_bbx = [min1 - margin, min2- margin, max1 + margin, max2 + margin]
            if max1 - min1 >= 30 and max2 - min2 >= 10:
                bboxes.append(out_bbx)


    bw = morph

    return bboxes, bw
```

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

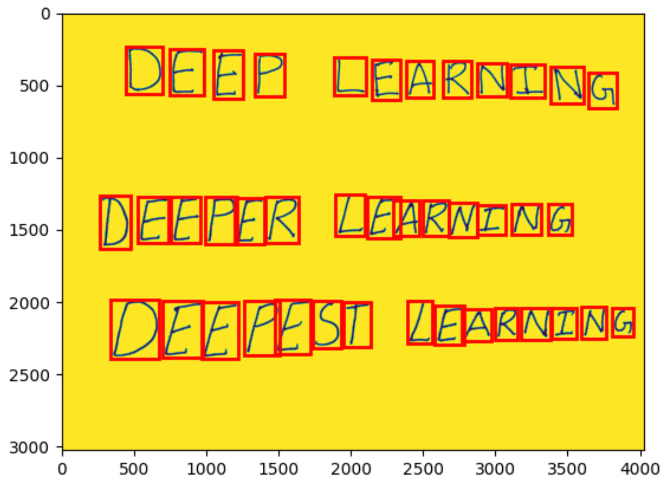




A B C D E F G
H I J K L M N
O P Q R S T U
V W X Y Z
1 2 3 4 5 6 7 8 9 0



HAIKUS ARE EASY
BUT SOMETIMES THEY DONT MAKE SENSE
REFRIGERATOR



DEEP LEARNING
DEEPER LEARNING
DEEPEST LEARNING

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

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

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

Q4.4

```
Image 1:
T0 D0 LIST
I HAXE A TO D0 LIST
2 CHECK 0FF THE FIR5T THING 0H T0 D0 LI5T
3 REALIZE Y0U BHVE ALREADT C0MPLETED Z THINGG
4 REWARD Y0UR6ELF WIT4 R NAP
Image 2:
2 B L D E F Y
H I J K L M N
Q P Q R 5 T U
V W X Y Z
1 Z 3 G 5 G 7 8 9
Image 3:
HAIKUS ARE BASY
BUT SQMETIMES TREX DOHT MAKE SENQE
R8FRIGBRATOR
Image 4:
JEEF LBARMING
DFBPEK LEAR4ING
UEBR8ST LEARNING
Total accuracy: 78.5%
```

5 (Extra Credit) Image Compression with Autoencoders

An autoencoder is a neural network that is trained to attempt to copy its input to its output, but it usually allows copying only approximately. This is typically achieved by restricting the number of hidden nodes inside the autoencoder; in other words, the autoencoder would be forced to learn to *represent* data with this limited number of hidden nodes. This is a useful way of learning compressed representations. In this section, we will continue using the NIST36 dataset you have from the previous questions. Use `python/run_q5.py` for this question.

5.1 Building the Autoencoder

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

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

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

Q5.1.1

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

Q5.1.2

5.2 Training the Autoencoder

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

Q5.2

5.3 Evaluating the Autoencoder

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

Q5.3.1

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

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

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

Q5.3.2

6 PyTorch

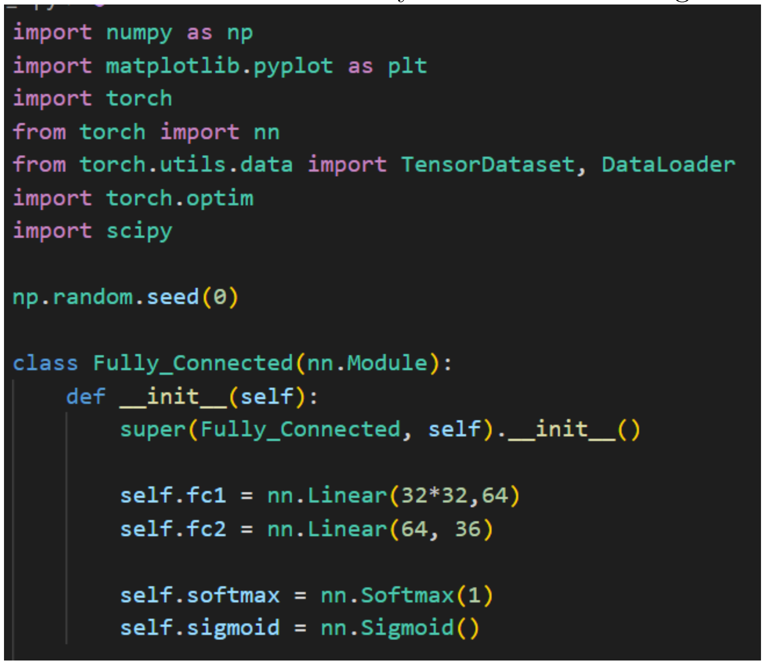
While you were able to derive manual backpropagation rules for sigmoid and fully-connected layers, wouldn't it be nice if someone did that for lots of useful primitives and made it fast and easy to use for general computation? Meet automatic differentiation. Since we have high-dimensional inputs (images) and low-dimensional outputs (a scalar loss), it turns out **forward mode AD** is very efficient. Popular autodiff packages include pytorch (Facebook), tensorflow (Google), autograd (Boston-area academics). Autograd provides its own replacement for numpy operators and is a drop-in replacement for numpy, except you can ask for gradients now. The other two are able to utilize GPUs to perform highly optimized and parallel computations, and are very popular for researchers who train large networks. Tensorflow asks you to build a computational graph using its API, and then is able to pass data through that graph. PyTorch builds a dynamic graph and allows you to mix autograd functions with normal python code much more smoothly, so it is currently more popular in academia.

We will use PyTorch as a framework. Many computer vision projects use neural networks as a basic building block, so familiarity with one of these frameworks is a good skill to develop. Here, we basically replicate and slightly expand our handwritten character recognition networks, but do it in PyTorch instead of doing it ourselves. Feel free to use any tutorial you like, but we like the official one or this tutorial (in a jupyter notebook) or these slides (starting from number 35).

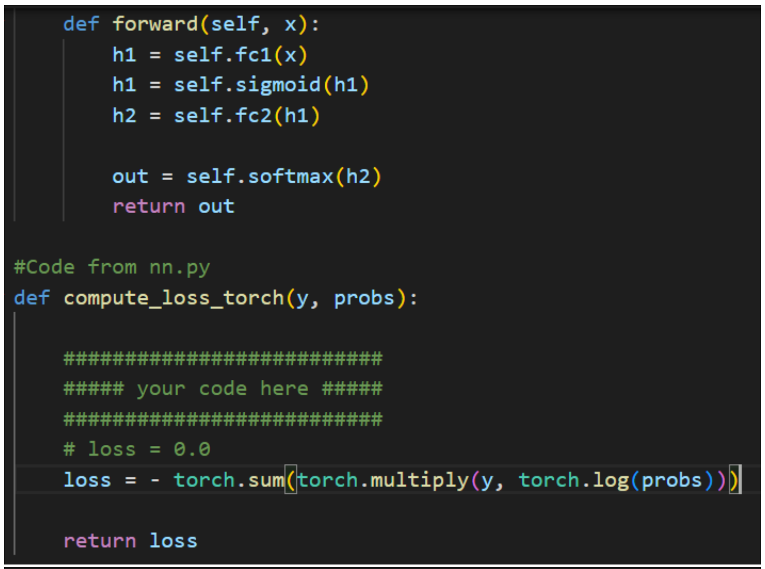
For this section, you're free to implement these however you like. All of the tasks required here are fairly small and don't require a GPU if you use small networks. Include the neural network code for each part in the writeup.

6.1 Train a neural network in PyTorch

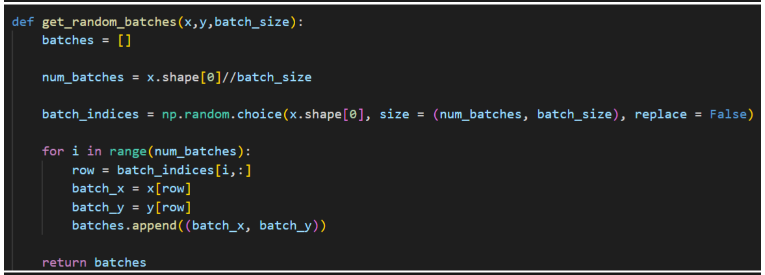
Q6.1.1 Code/Writeup [**5 points**] Re-write and re-train your **fully-connected** network on the included **NIST36** in PyTorch. Plot training accuracy and loss over time.



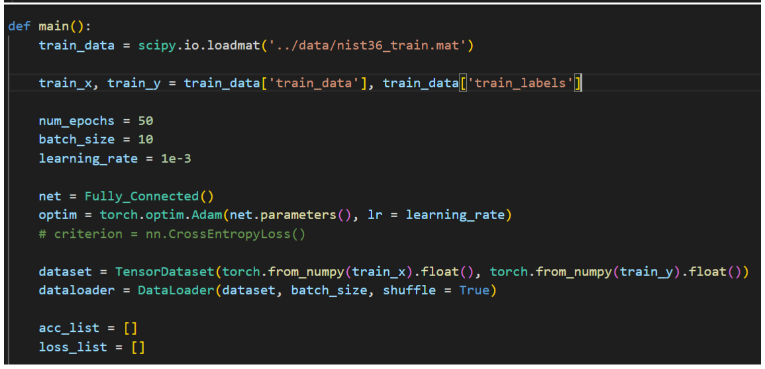
```
import numpy as np
import matplotlib.pyplot as plt
import torch
from torch import nn
from torch.utils.data import TensorDataset, DataLoader
import torch.optim
import scipy

np.random.seed(0)

class Fully_Connected(nn.Module):
    def __init__(self):
        super(Fully_Connected, self).__init__()

        self.fc1 = nn.Linear(32*32,64)
        self.fc2 = nn.Linear(64, 36)

        self.softmax = nn.Softmax(1)
        self.sigmoid = nn.Sigmoid()
```



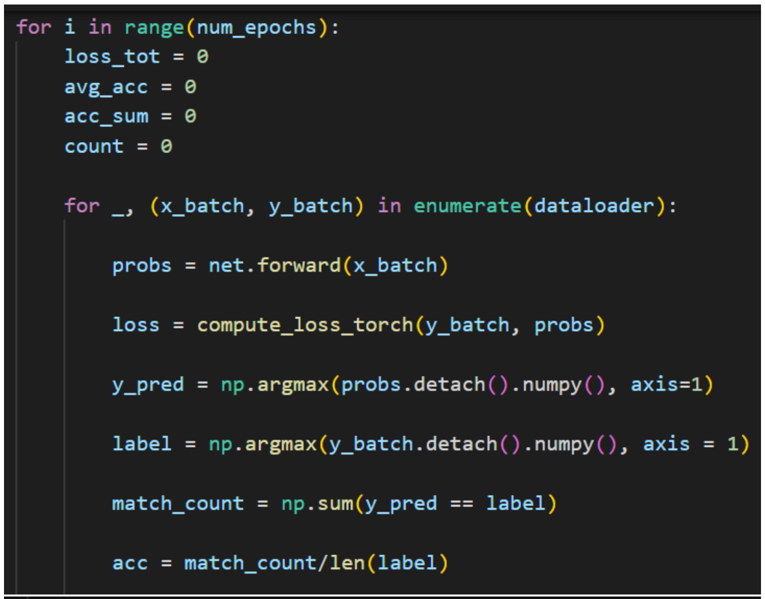
```python
    def forward(self, x):
        h1 = self.fc1(x)
        h1 = self.sigmoid(h1)
        h2 = self.fc2(h1)

        out = self.softmax(h2)
        return out

#Code from nn.py
def compute_loss_torch(y, probs):

    ##########################
    ##### your code here #####
    ##########################
    # loss = 0.0
    loss = - torch.sum(torch.multiply(y, torch.log(probs)))

    return loss
```



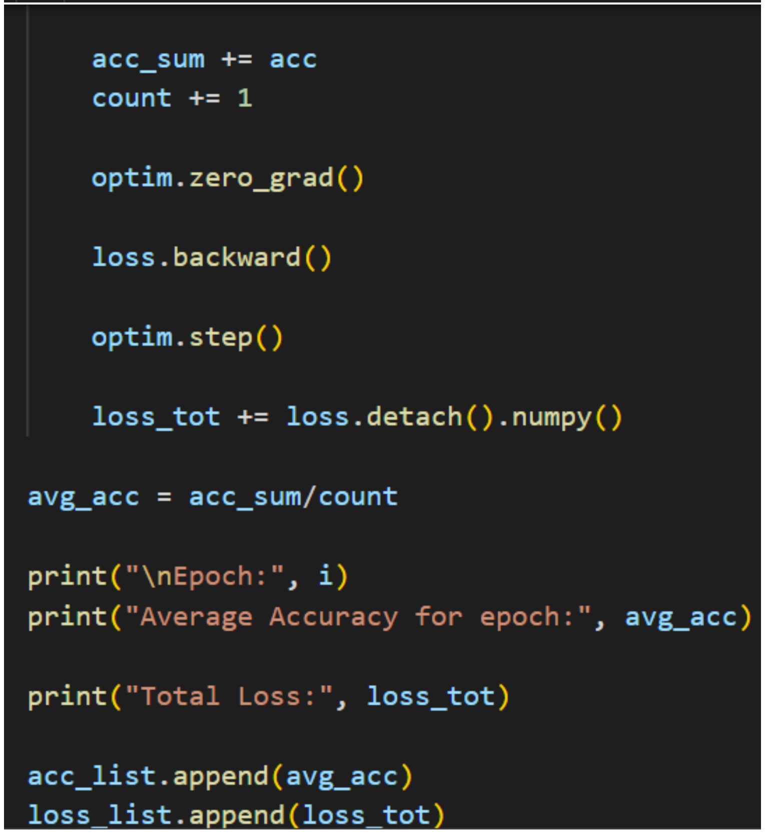
```python
def get_random_batches(x,y,batch_size):
    batches = []

    num_batches = x.shape[0]//batch_size

    batch_indices = np.random.choice(x.shape[0], size = (num_batches, batch_size), replace = False)

    for i in range(num_batches):
        row = batch_indices[i,:]
        batch_x = x[row]
        batch_y = y[row]
        batches.append((batch_x, batch_y))

    return batches
```



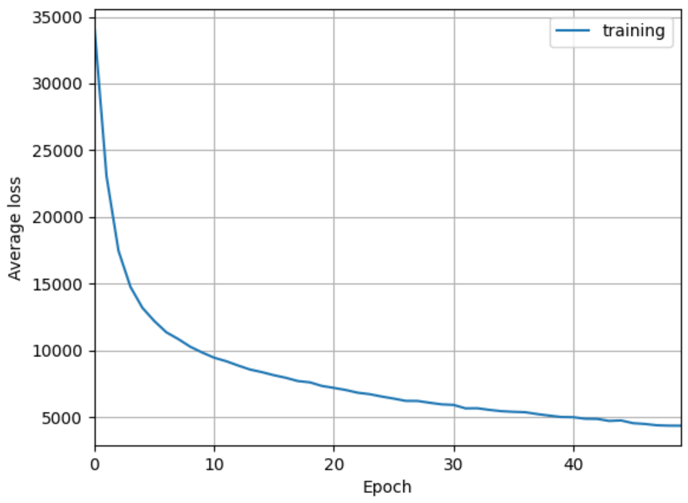
```python
def main():
    train_data = scipy.io.loadmat('../data/nist36_train.mat')

    train_x, train_y = train_data['train_data'], train_data['train_labels']

    num_epochs = 50
    batch_size = 10
    learning_rate = 1e-3

    net = Fully_Connected()
    optim = torch.optim.Adam(net.parameters(), lr = learning_rate)
    # criterion = nn.CrossEntropyLoss()

    dataset = TensorDataset(torch.from_numpy(train_x).float(), torch.from_numpy(train_y).float())
    dataloader = DataLoader(dataset, batch_size, shuffle = True)

    acc_list = []
    loss_list = []
```

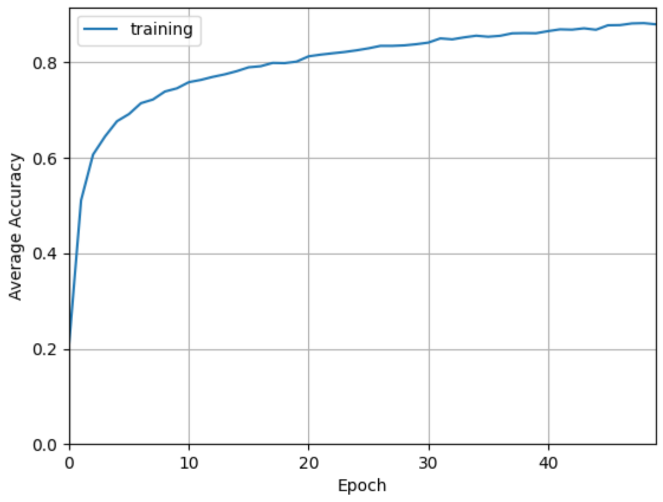


```
for i in range(num_epochs):
    loss_tot = 0
    avg_acc = 0
    acc_sum = 0
    count = 0

    for _, (x_batch, y_batch) in enumerate(dataloader):

        probs = net.forward(x_batch)

        loss = compute_loss_torch(y_batch, probs)

        y_pred = np.argmax(probs.detach().numpy(), axis=1)

        label = np.argmax(y_batch.detach().numpy(), axis = 1)

        match_count = np.sum(y_pred == label)

        acc = match_count/len(label)
```



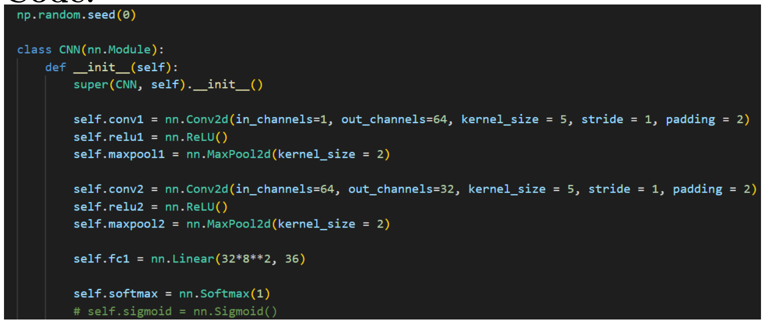
```
        acc_sum += acc
        count += 1

        optim.zero_grad()

        loss.backward()

        optim.step()

        loss_tot += loss.detach().numpy()

    avg_acc = acc_sum/count

    print("\nEpoch:", i)
    print("Average Accuracy for epoch:", avg_acc)

    print("Total Loss:", loss_tot)

    acc_list.append(avg_acc)
    loss_list.append(loss_tot)
```



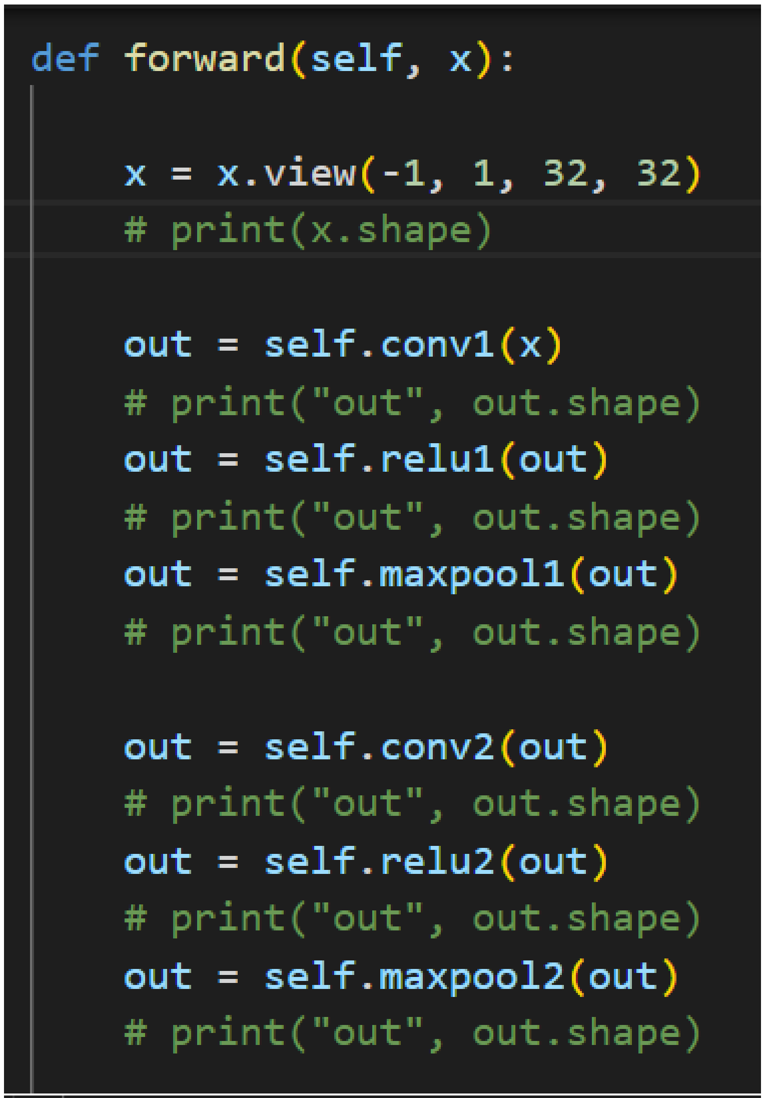


Q6.1.2 Code/Writeup [5 points] Train a **convolutional** neural network with PyTorch on the included **NIST36** dataset. Compare its performance with the previous fully-connected network.

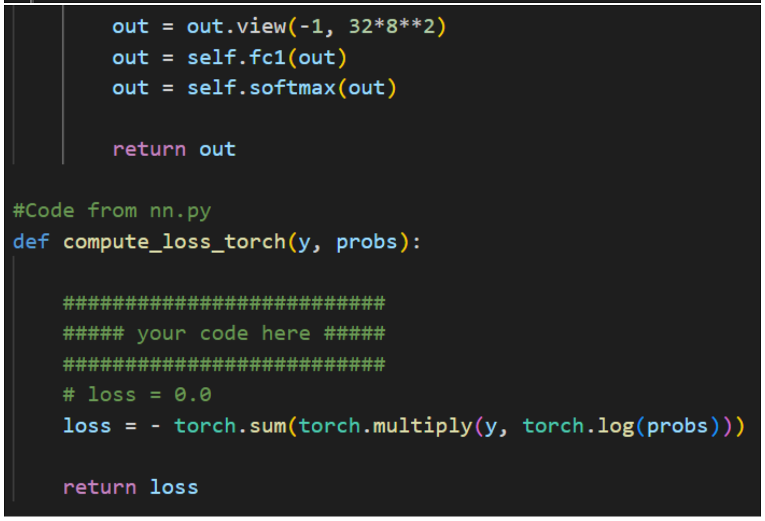
Code:



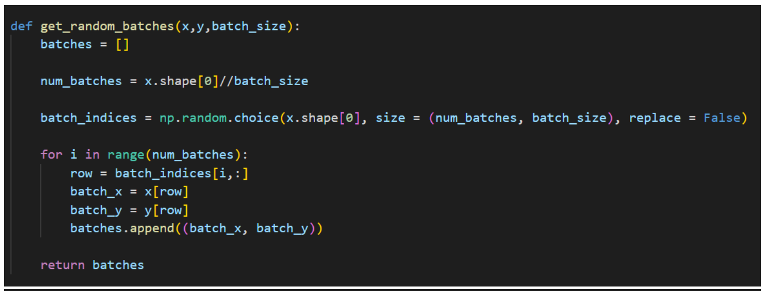
```
np.random.seed(0)

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=1, out_channels=64, kernel_size = 5, stride = 1, padding = 2)
        self.relu1 = nn.ReLU()
        self.maxpool1 = nn.MaxPool2d(kernel_size = 2)

        self.conv2 = nn.Conv2d(in_channels=64, out_channels=32, kernel_size = 5, stride = 1, padding = 2)
        self.relu2 = nn.ReLU()
        self.maxpool2 = nn.MaxPool2d(kernel_size = 2)

        self.fc1 = nn.Linear(32*8**2, 36)

        self.softmax = nn.Softmax(1)
        # self.sigmoid = nn.Sigmoid()
```



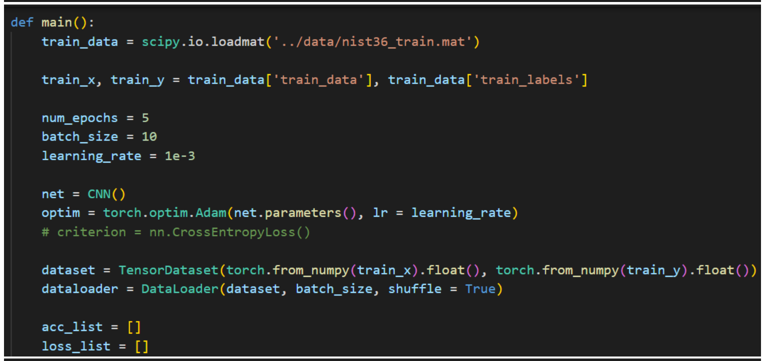
```python
def forward(self, x):

    x = x.view(-1, 1, 32, 32)
    # print(x.shape)

    out = self.conv1(x)
    # print("out", out.shape)
    out = self.relu1(out)
    # print("out", out.shape)
    out = self.maxpool1(out)
    # print("out", out.shape)

    out = self.conv2(out)
    # print("out", out.shape)
    out = self.relu2(out)
    # print("out", out.shape)
    out = self.maxpool2(out)
    # print("out", out.shape)
```



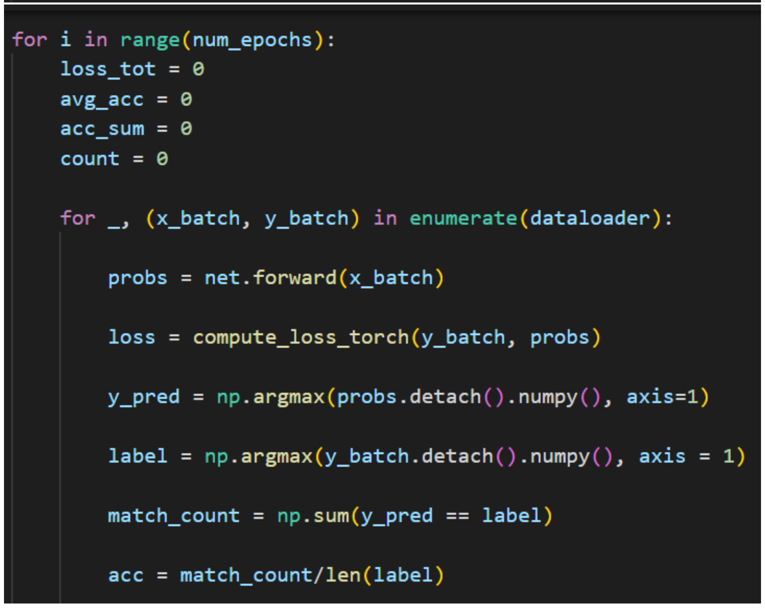
```python
        out = out.view(-1, 32*8**2)
        out = self.fc1(out)
        out = self.softmax(out)

        return out

#Code from nn.py
def compute_loss_torch(y, probs):

    ##########################
    ##### your code here #####
    ##########################
    # loss = 0.0
    loss = - torch.sum(torch.multiply(y, torch.log(probs)))

    return loss
```



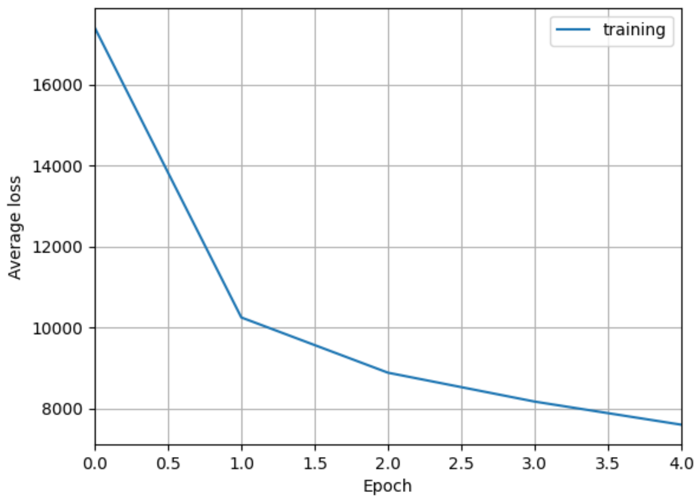
```python
def get_random_batches(x,y,batch_size):
    batches = []

    num_batches = x.shape[0]//batch_size

    batch_indices = np.random.choice(x.shape[0], size = (num_batches, batch_size), replace = False)

    for i in range(num_batches):
        row = batch_indices[i,:]
        batch_x = x[row]
        batch_y = y[row]
        batches.append((batch_x, batch_y))

    return batches
```

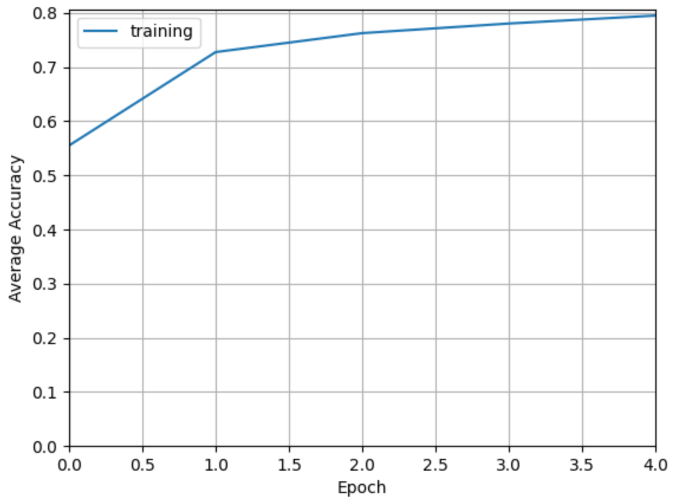


```python
def main():
    train_data = scipy.io.loadmat('../data/nist36_train.mat')

    train_x, train_y = train_data['train_data'], train_data['train_labels']

    num_epochs = 5
    batch_size = 10
    learning_rate = 1e-3

    net = CNN()
    optim = torch.optim.Adam(net.parameters(), lr = learning_rate)
    # criterion = nn.CrossEntropyLoss()

    dataset = TensorDataset(torch.from_numpy(train_x).float(), torch.from_numpy(train_y).float())
    dataloader = DataLoader(dataset, batch_size, shuffle = True)

    acc_list = []
    loss_list = []
```

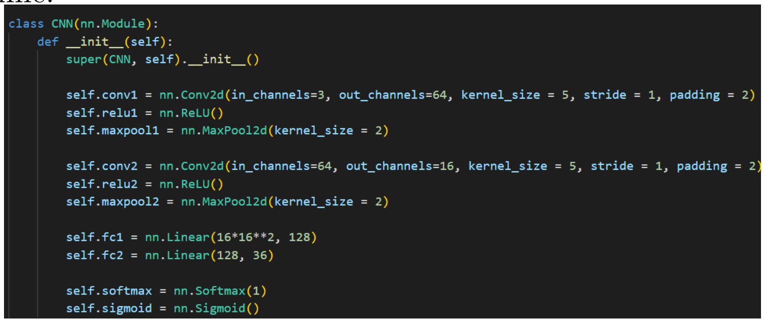


```python
for i in range(num_epochs):
    loss_tot = 0
    avg_acc = 0
    acc_sum = 0
    count = 0

    for _, (x_batch, y_batch) in enumerate(dataloader):

        probs = net.forward(x_batch)

        loss = compute_loss_torch(y_batch, probs)

        y_pred = np.argmax(probs.detach().numpy(), axis=1)

        label = np.argmax(y_batch.detach().numpy(), axis = 1)

        match_count = np.sum(y_pred == label)

        acc = match_count/len(label)
```

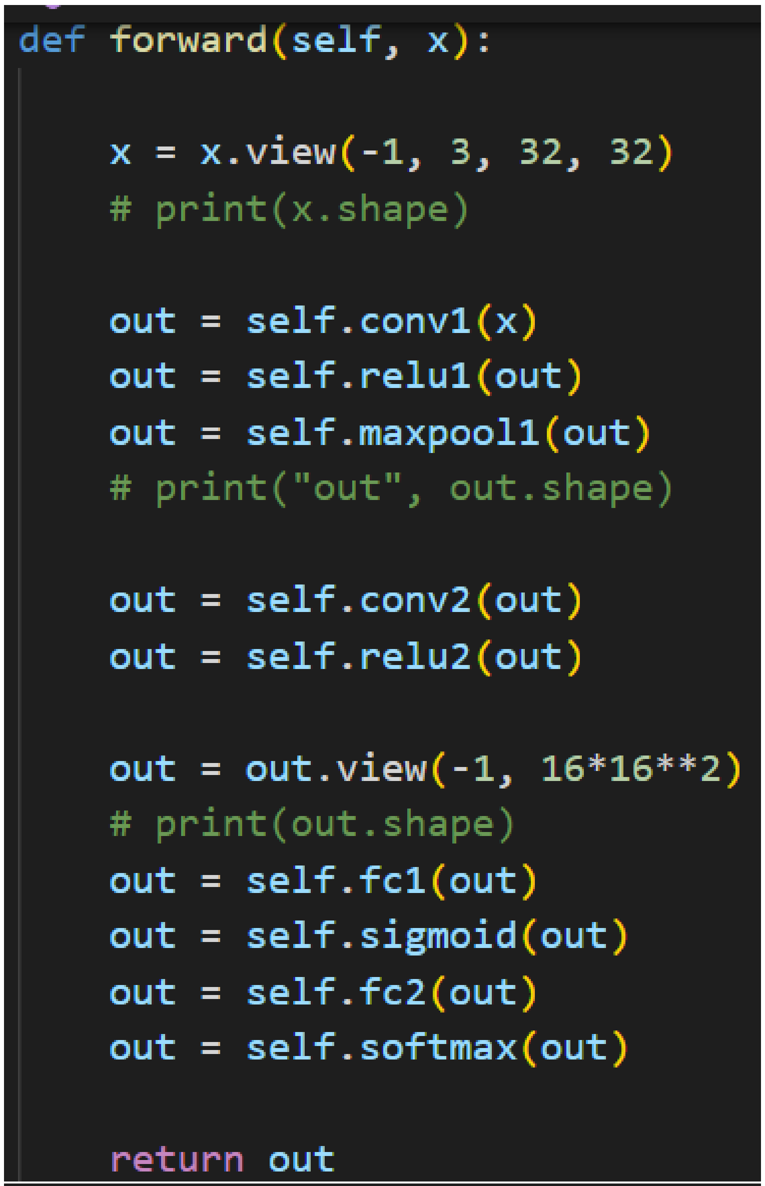




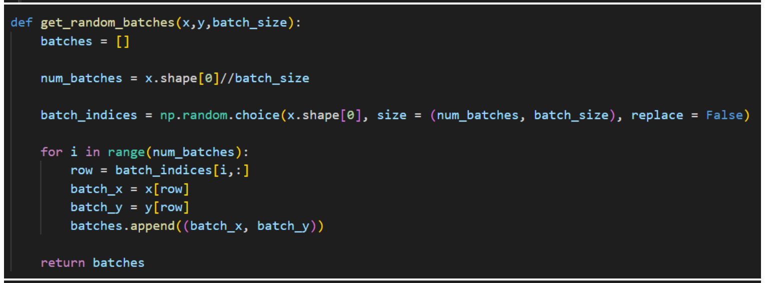
Q6.1.3 Code/Writeup [5 points] Train a **convolutional** neural network with PyTorch on **CIFAR-10** (`torchvision.datasets.CIFAR10`). Plot training accuracy and loss over time.



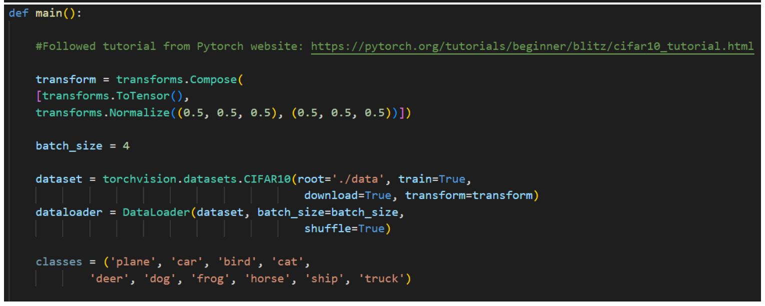
```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size = 5, stride = 1, padding = 2)
        self.relu1 = nn.ReLU()
        self.maxpool1 = nn.MaxPool2d(kernel_size = 2)

        self.conv2 = nn.Conv2d(in_channels=64, out_channels=16, kernel_size = 5, stride = 1, padding = 2)
        self.relu2 = nn.ReLU()
        self.maxpool2 = nn.MaxPool2d(kernel_size = 2)

        self.fc1 = nn.Linear(16*16**2, 128)
        self.fc2 = nn.Linear(128, 36)

        self.softmax = nn.Softmax(1)
        self.sigmoid = nn.Sigmoid()
```



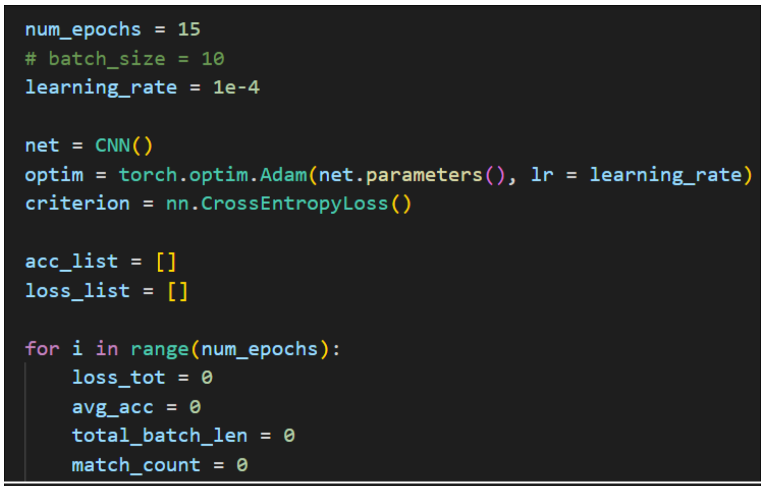
```python
def forward(self, x):

    x = x.view(-1, 3, 32, 32)
    # print(x.shape)

    out = self.conv1(x)
    out = self.relu1(out)
    out = self.maxpool1(out)
    # print("out", out.shape)

    out = self.conv2(out)
    out = self.relu2(out)

    out = out.view(-1, 16*16**2)
    # print(out.shape)
    out = self.fc1(out)
    out = self.sigmoid(out)
    out = self.fc2(out)
    out = self.softmax(out)

    return out
```



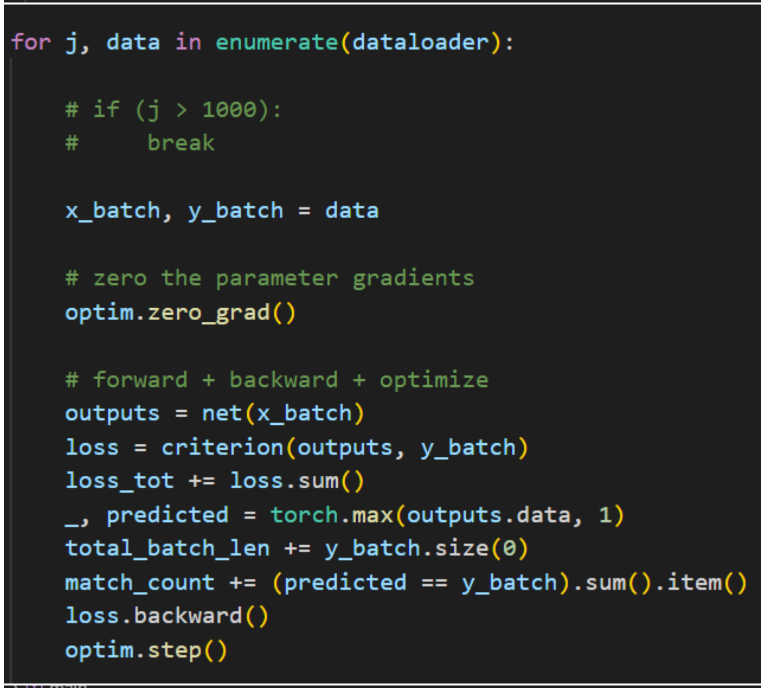
```python
def get_random_batches(x,y,batch_size):
    batches = []

    num_batches = x.shape[0]//batch_size

    batch_indices = np.random.choice(x.shape[0], size = (num_batches, batch_size), replace = False)

    for i in range(num_batches):
        row = batch_indices[i,:]
        batch_x = x[row]
        batch_y = y[row]
        batches.append((batch_x, batch_y))

    return batches
```



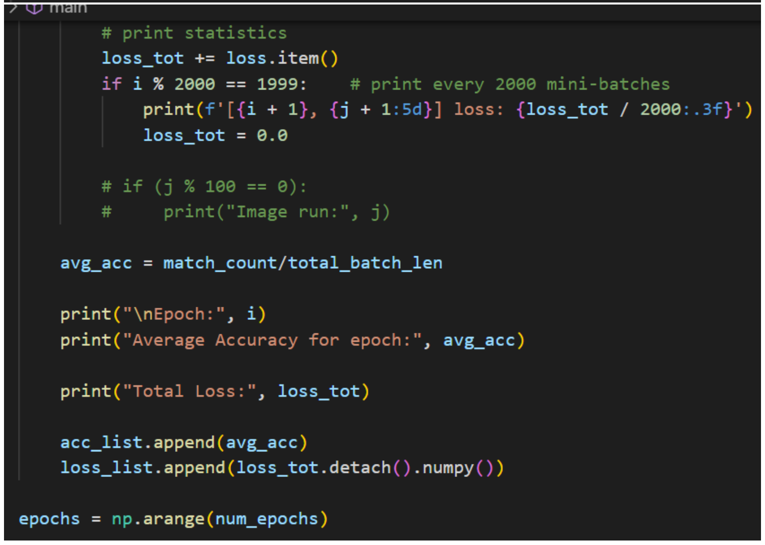
```python
def main():

    #Followed tutorial from Pytorch website: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

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

    batch_size = 4

    dataset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                            download=True, transform=transform)
    dataloader = DataLoader(dataset, batch_size=batch_size,
                                            shuffle=True)

    classes = ('plane', 'car', 'bird', 'cat',
            'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```



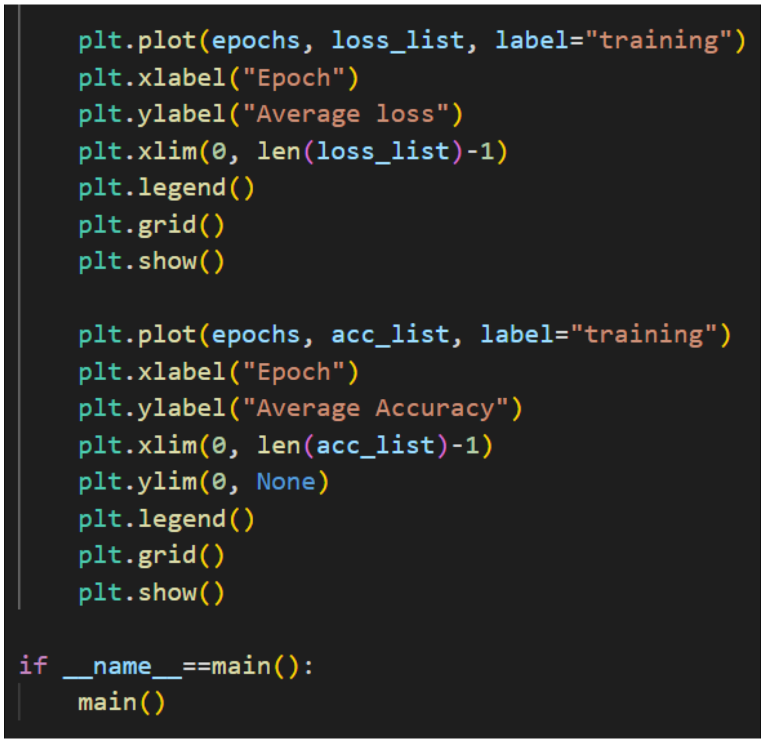
```
num_epochs = 15
# batch_size = 10
learning_rate = 1e-4

net = CNN()
optim = torch.optim.Adam(net.parameters(), lr = learning_rate)
criterion = nn.CrossEntropyLoss()

acc_list = []
loss_list = []

for i in range(num_epochs):
    loss_tot = 0
    avg_acc = 0
    total_batch_len = 0
    match_count = 0
```



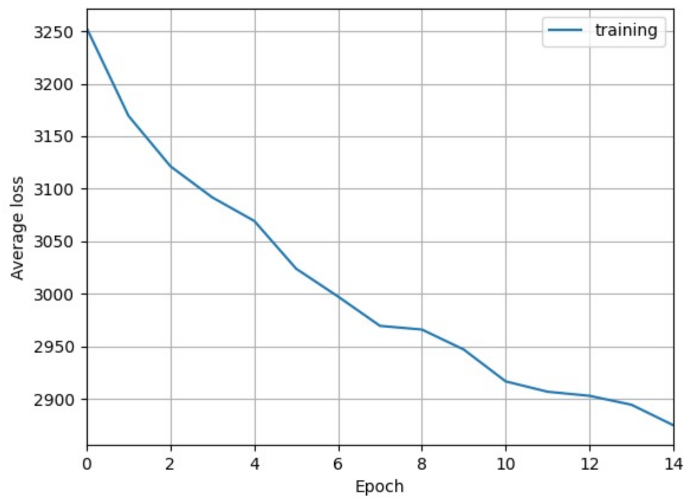
```
for j, data in enumerate(dataloader):

    # if (j > 1000):
    #     break

    x_batch, y_batch = data

    # zero the parameter gradients
    optim.zero_grad()

    # forward + backward + optimize
    outputs = net(x_batch)
    loss = criterion(outputs, y_batch)
    loss_tot += loss.sum()
    _, predicted = torch.max(outputs.data, 1)
    total_batch_len += y_batch.size(0)
    match_count += (predicted == y_batch).sum().item()
    loss.backward()
    optim.step()
```

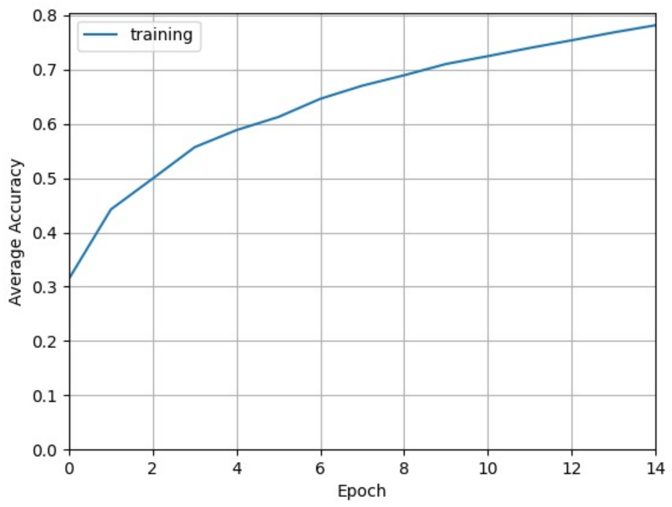


```
        # print statistics
        loss_tot += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print(f'[{i + 1}, {j + 1:5d}] loss: {loss_tot / 2000:.3f}')
            loss_tot = 0.0

        # if (j % 100 == 0):
        #     print("Image run:", j)

    avg_acc = match_count/total_batch_len

    print("\nEpoch:", i)
    print("Average Accuracy for epoch:", avg_acc)

    print("Total Loss:", loss_tot)

    acc_list.append(avg_acc)
    loss_list.append(loss_tot.detach().numpy())

epochs = np.arange(num_epochs)
```



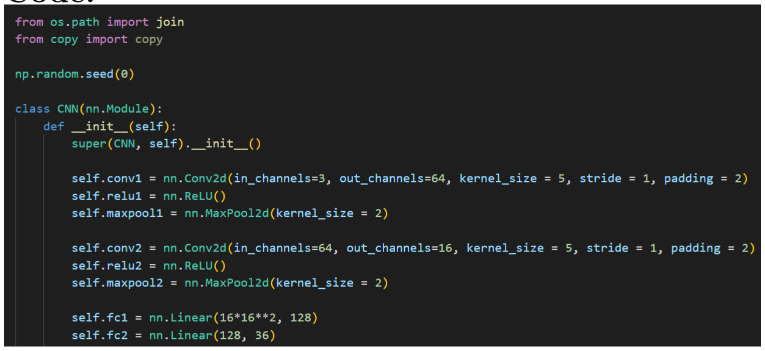
```
    plt.plot(epochs, loss_list, label="training")
    plt.xlabel("Epoch")
    plt.ylabel("Average loss")
    plt.xlim(0, len(loss_list)-1)
    plt.legend()
    plt.grid()
    plt.show()

    plt.plot(epochs, acc_list, label="training")
    plt.xlabel("Epoch")
    plt.ylabel("Average Accuracy")
    plt.xlim(0, len(acc_list)-1)
    plt.ylim(0, None)
    plt.legend()
    plt.grid()
    plt.show()

if __name__==main():
    main()
```



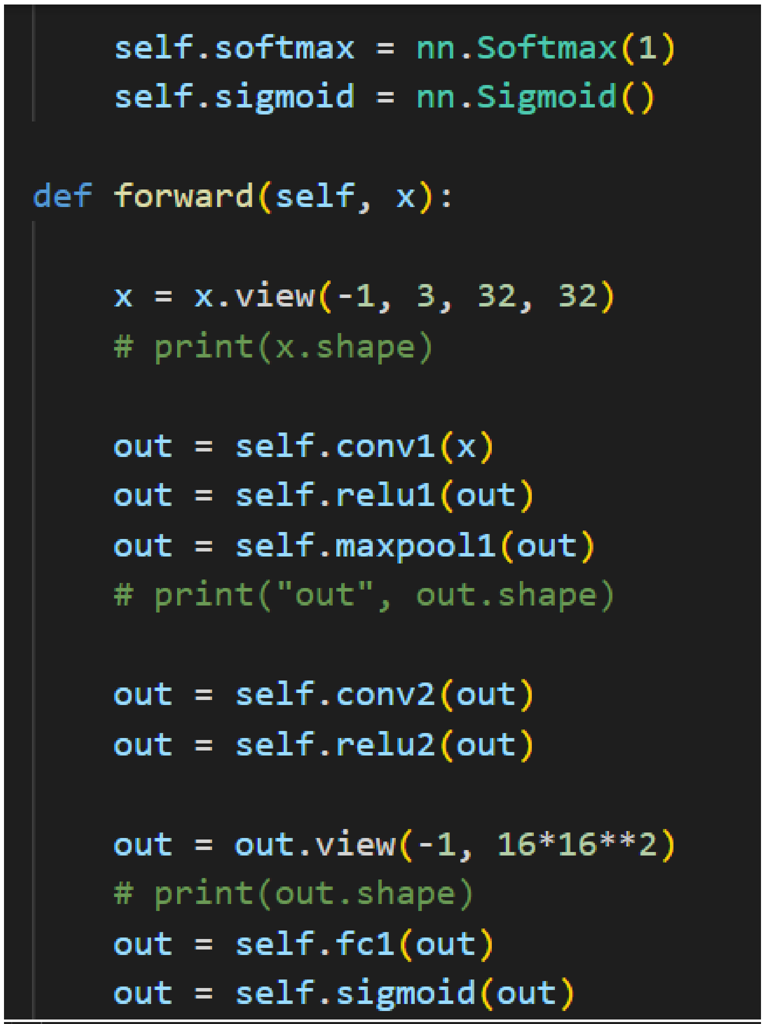


Q6.1.4 Code/Writeup [10 points] In Homework 1, we tried scene classification with the bag-of-words (BoW) approach on a subset of the **SUN database**. Use the same dataset in HW1, and implement a **convolutional** neural network with PyTorch for **scene** classification. Compare your result with the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

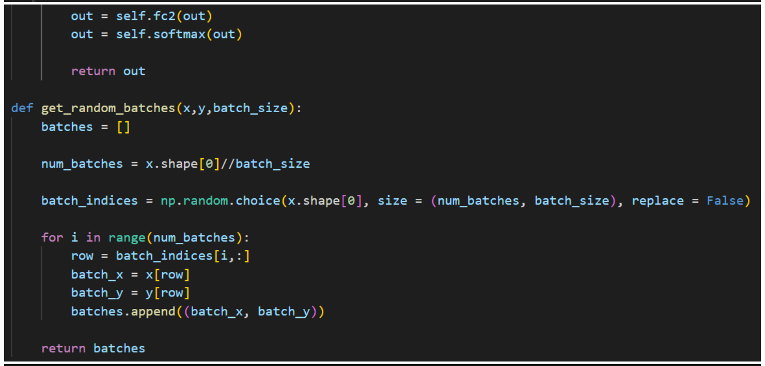
Code:



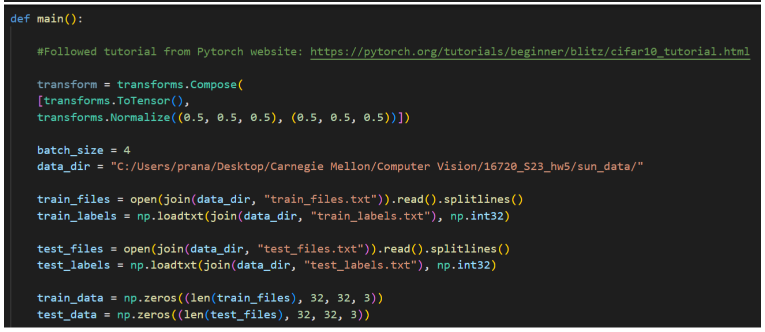
```python
from os.path import join
from copy import copy

np.random.seed(0)

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size = 5, stride = 1, padding = 2)
        self.relu1 = nn.ReLU()
        self.maxpool1 = nn.MaxPool2d(kernel_size = 2)

        self.conv2 = nn.Conv2d(in_channels=64, out_channels=16, kernel_size = 5, stride = 1, padding = 2)
        self.relu2 = nn.ReLU()
        self.maxpool2 = nn.MaxPool2d(kernel_size = 2)

        self.fc1 = nn.Linear(16*16**2, 128)
        self.fc2 = nn.Linear(128, 36)
```



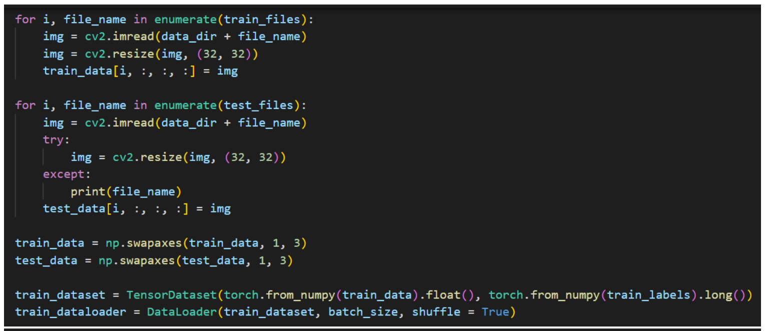
```python
        self.softmax = nn.Softmax(1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):

        x = x.view(-1, 3, 32, 32)
        # print(x.shape)

        out = self.conv1(x)
        out = self.relu1(out)
        out = self.maxpool1(out)
        # print("out", out.shape)

        out = self.conv2(out)
        out = self.relu2(out)

        out = out.view(-1, 16*16**2)
        # print(out.shape)
        out = self.fc1(out)
        out = self.sigmoid(out)
```



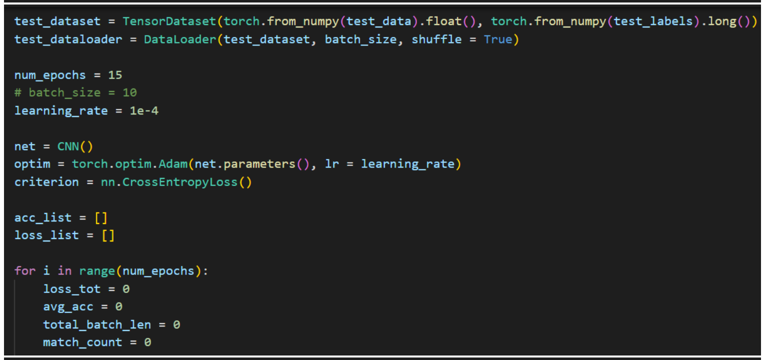
```python
        out = self.fc2(out)
        out = self.softmax(out)

        return out

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

    num_batches = x.shape[0]//batch_size

    batch_indices = np.random.choice(x.shape[0], size = (num_batches, batch_size), replace = False)

    for i in range(num_batches):
        row = batch_indices[i,:]
        batch_x = x[row]
        batch_y = y[row]
        batches.append((batch_x, batch_y))

    return batches
```



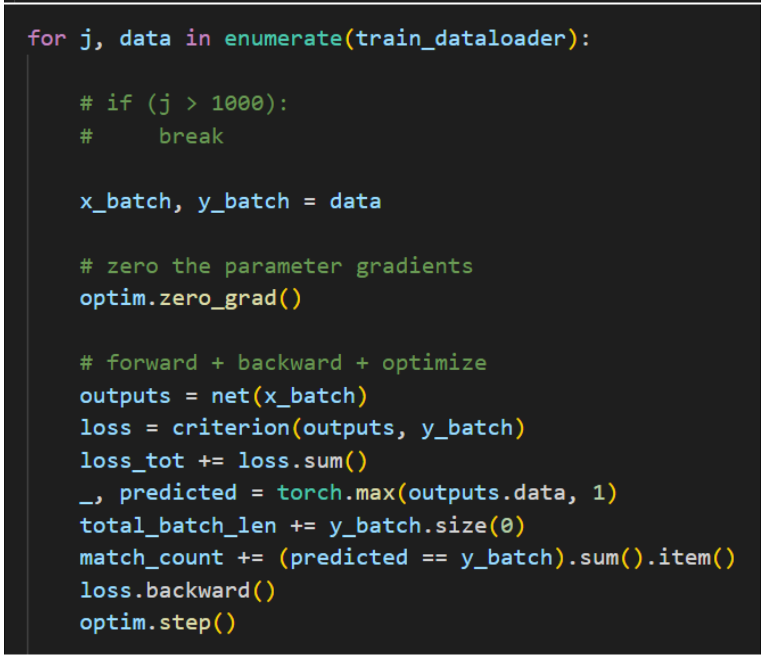
```python
def main():

    #Followed tutorial from Pytorch website: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

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

    batch_size = 4
    data_dir = "C:/Users/prana/Desktop/Carnegie Mellon/Computer Vision/16720_S23_hw5/sun_data/"

    train_files = open(join(data_dir, "train_files.txt")).read().splitlines()
    train_labels = np.loadtxt(join(data_dir, "train_labels.txt"), np.int32)

    test_files = open(join(data_dir, "test_files.txt")).read().splitlines()
    test_labels = np.loadtxt(join(data_dir, "test_labels.txt"), np.int32)

    train_data = np.zeros((len(train_files), 32, 32, 3))
    test_data = np.zeros((len(test_files), 32, 32, 3))
```



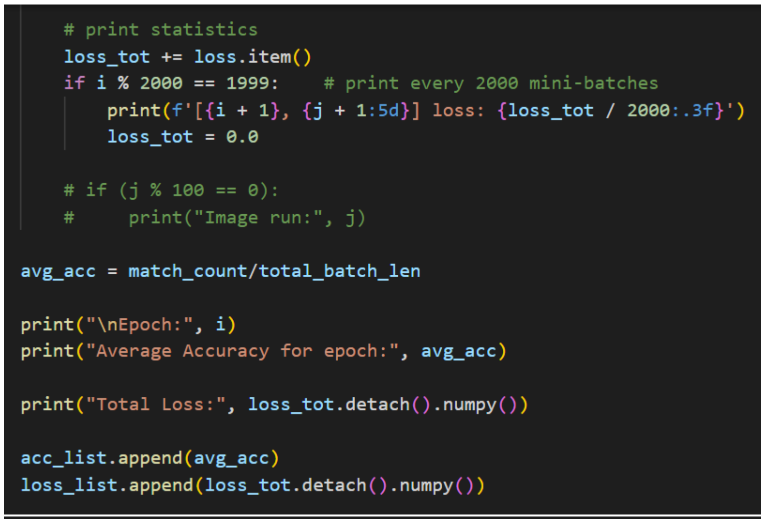
```
for i, file_name in enumerate(train_files):
    img = cv2.imread(data_dir + file_name)
    img = cv2.resize(img, (32, 32))
    train_data[i, :, :, :] = img

for i, file_name in enumerate(test_files):
    img = cv2.imread(data_dir + file_name)
    try:
        img = cv2.resize(img, (32, 32))
    except:
        print(file_name)
    test_data[i, :, :, :] = img

train_data = np.swapaxes(train_data, 1, 3)
test_data = np.swapaxes(test_data, 1, 3)

train_dataset = TensorDataset(torch.from_numpy(train_data).float(), torch.from_numpy(train_labels).long())
train_dataloader = DataLoader(train_dataset, batch_size, shuffle = True)
```



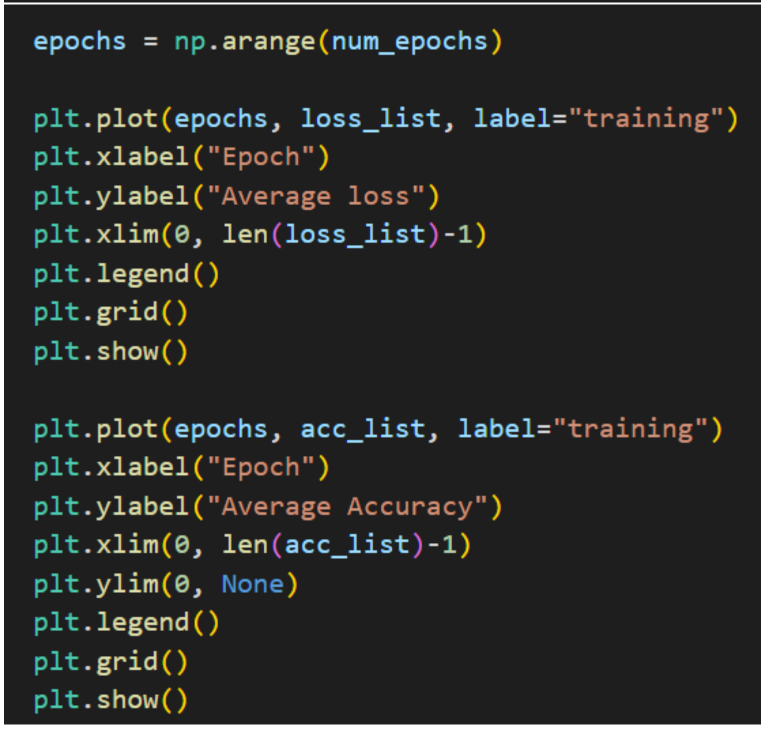
```
test_dataset = TensorDataset(torch.from_numpy(test_data).float(), torch.from_numpy(test_labels).long())
test_dataloader = DataLoader(test_dataset, batch_size, shuffle = True)

num_epochs = 15
# batch_size = 10
learning_rate = 1e-4

net = CNN()
optim = torch.optim.Adam(net.parameters(), lr = learning_rate)
criterion = nn.CrossEntropyLoss()

acc_list = []
loss_list = []

for i in range(num_epochs):
    loss_tot = 0
    avg_acc = 0
    total_batch_len = 0
    match_count = 0
```



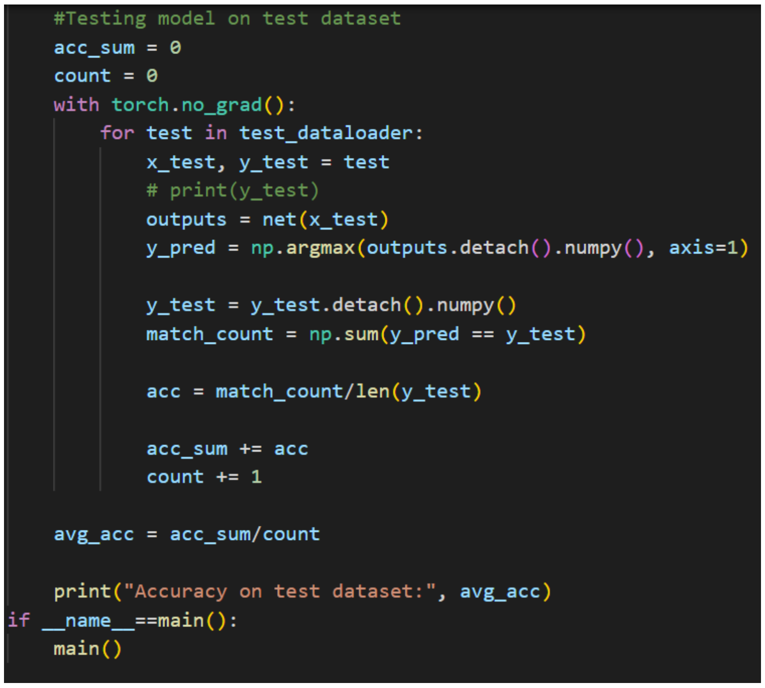
```
for j, data in enumerate(train_dataloader):

    # if (j > 1000):
    #     break

    x_batch, y_batch = data

    # zero the parameter gradients
    optim.zero_grad()

    # forward + backward + optimize
    outputs = net(x_batch)
    loss = criterion(outputs, y_batch)
    loss_tot += loss.sum()
    _, predicted = torch.max(outputs.data, 1)
    total_batch_len += y_batch.size(0)
    match_count += (predicted == y_batch).sum().item()
    loss.backward()
    optim.step()
```



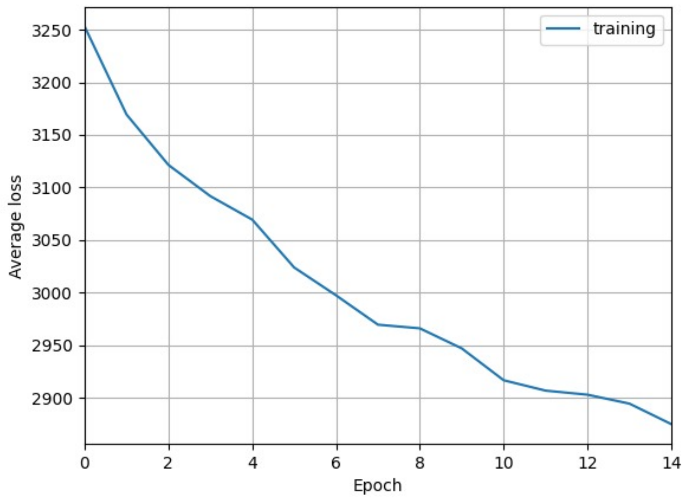
```python
    # print statistics
    loss_tot += loss.item()
    if i % 2000 == 1999:    # print every 2000 mini-batches
        print(f'[{i + 1}, {j + 1:5d}] loss: {loss_tot / 2000:.3f}')
        loss_tot = 0.0

    # if (j % 100 == 0):
    #     print("Image run:", j)

avg_acc = match_count/total_batch_len

print("\nEpoch:", i)
print("Average Accuracy for epoch:", avg_acc)

print("Total Loss:", loss_tot.detach().numpy())

acc_list.append(avg_acc)
loss_list.append(loss_tot.detach().numpy())
```

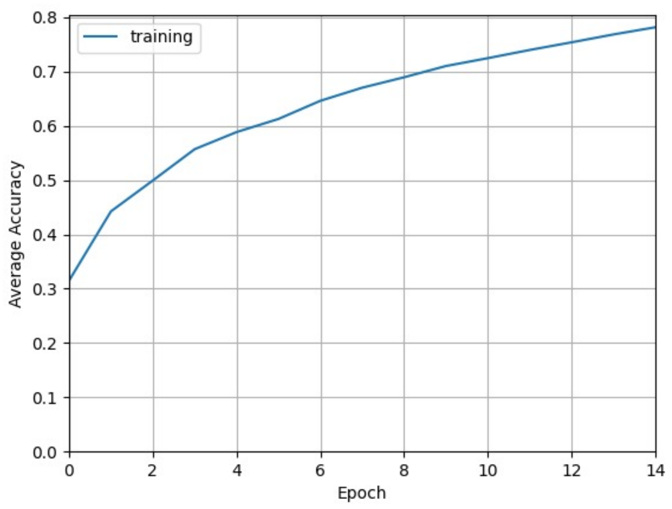


```python
epochs = np.arange(num_epochs)

plt.plot(epochs, loss_list, label="training")
plt.xlabel("Epoch")
plt.ylabel("Average loss")
plt.xlim(0, len(loss_list)-1)
plt.legend()
plt.grid()
plt.show()

plt.plot(epochs, acc_list, label="training")
plt.xlabel("Epoch")
plt.ylabel("Average Accuracy")
plt.xlim(0, len(acc_list)-1)
plt.ylim(0, None)
plt.legend()
plt.grid()
plt.show()
```



```
    #Testing model on test dataset
    acc_sum = 0
    count = 0
    with torch.no_grad():
        for test in test_dataloader:
            x_test, y_test = test
            # print(y_test)
            outputs = net(x_test)
            y_pred = np.argmax(outputs.detach().numpy(), axis=1)

            y_test = y_test.detach().numpy()
            match_count = np.sum(y_pred == y_test)

            acc = match_count/len(y_test)

            acc_sum += acc
            count += 1

    avg_acc = acc_sum/count

    print("Accuracy on test dataset:", avg_acc)
if __name__==main():
    main()
```





Q6.1.4

The final accuracy on the test dataset was 72% with the training dataset being accurate 85% of the time. This was a better result than the bag of words which was approximately 65%. The CNN was far faster than the BoW method while only taking 15 epochs to reach the result. However, the result is still overfitting and is likely due to the images having features that are difficult to extract.

6.2 Fine Tuning

When training from scratch, a lot of epochs and data are often needed to learn anything meaningful. One way to avoid this is to instead initialize the weights more intelligently.

These days, it is most common to initialize a network with weights from another deep network that was trained for a different purpose. This is because, whether we are doing image classification, segmentation, recognition etc..., most real images share common properties. Simply copying the weights from the other network to yours gives your network a head start, so your network does not need to learn these common weights from scratch all over again. This is also referred to as fine tuning.

Q6.2 Code/Writeup [5 points] Fine-tune a single layer classifier using pytorch on the flowers 17 (or flowers 102!) dataset using squeezenet1_1, as well as an architecture you've designed yourself (for example 3 convolutional layers followed 2 fully connected layers, it's standard slide 6) and trained from scratch. How do they compare?

We include a script in `scripts/` to fetch the flowers dataset and extract it in a way that torchvision.datasets.ImageFolder can consume it, see an example, from `data/oxford-flowers17`. You should look at how SqueezeNet is defined, and just replace the classifier layer. There exists a pretty good example for fine-tuning in PyTorch.

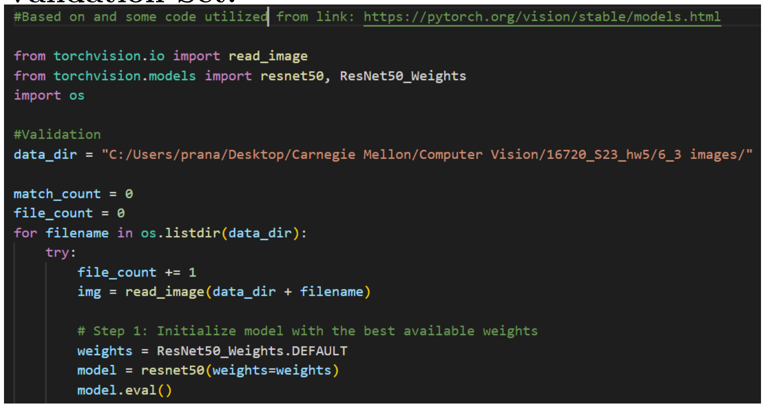
Q6.2

6.3 Neural Networks in the Real World

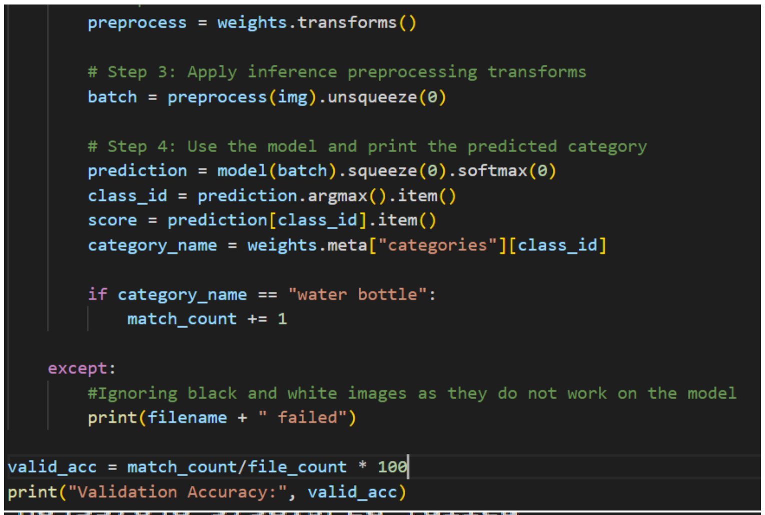
Often, we train neural networks on standard datasets and evaluate on held out validation data. How would a model trained on ImageNet perform in the wild?

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

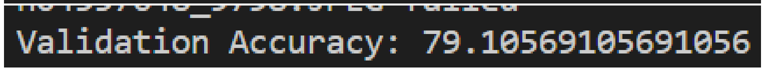
Validation Set:



```
#Based on and some code utilized from link: https://pytorch.org/vision/stable/models.html

from torchvision.io import read_image
from torchvision.models import resnet50, ResNet50_Weights
import os

#Validation
data_dir = "C:/Users/prana/Desktop/Carnegie Mellon/Computer Vision/16720_S23_hw5/6_3 images/"

match_count = 0
file_count = 0
for filename in os.listdir(data_dir):
    try:
        file_count += 1
        img = read_image(data_dir + filename)

        # Step 1: Initialize model with the best available weights
        weights = ResNet50_Weights.DEFAULT
        model = resnet50(weights=weights)
        model.eval()
```

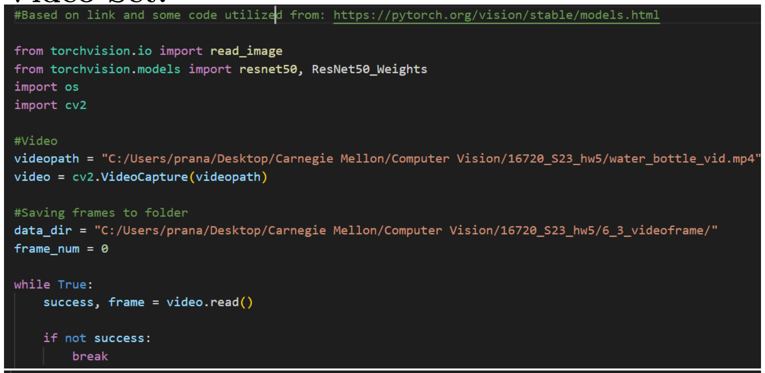


```
        preprocess = weights.transforms()

        # Step 3: Apply inference preprocessing transforms
        batch = preprocess(img).unsqueeze(0)

        # Step 4: Use the model and print the predicted category
        prediction = model(batch).squeeze(0).softmax(0)
        class_id = prediction.argmax().item()
        score = prediction[class_id].item()
        category_name = weights.meta["categories"][class_id]

        if category_name == "water bottle":
            match_count += 1

    except:
        #Ignoring black and white images as they do not work on the model
        print(filename + " failed")

valid_acc = match_count/file_count * 100
print("Validation Accuracy:", valid_acc)
```

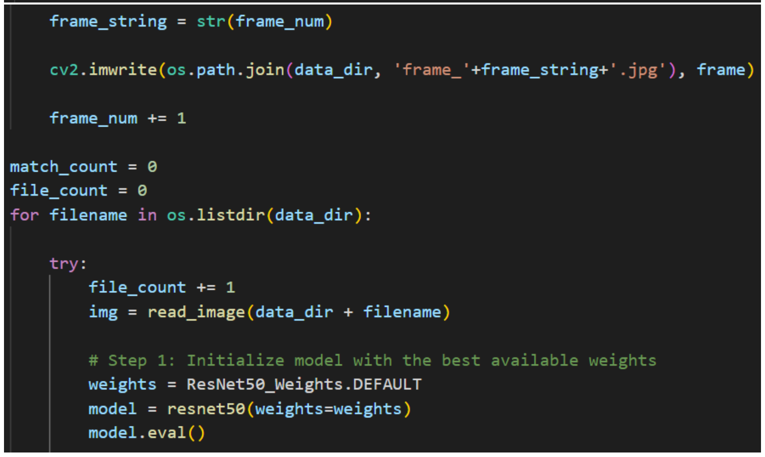


```
Validation Accuracy: 79.10569105691056
```

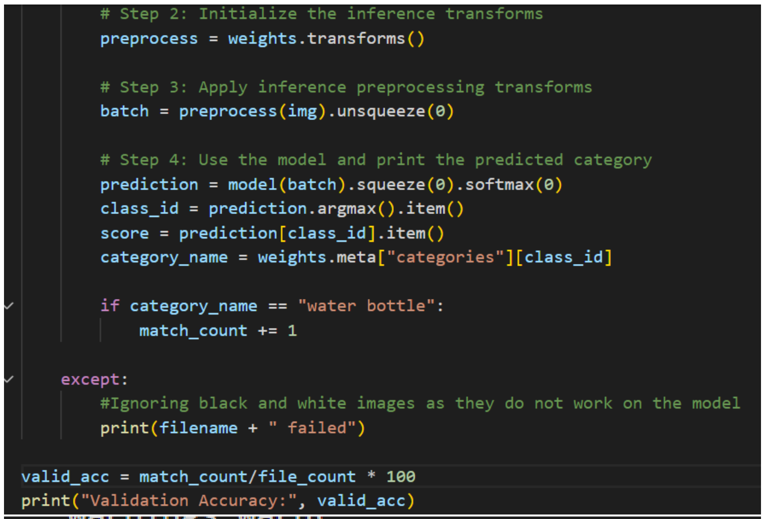
Video Set:



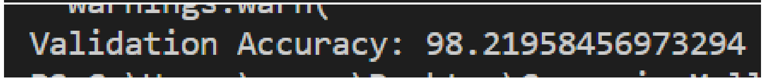
```
#Based on link and some code utilized from: https://pytorch.org/vision/stable/models.html

from torchvision.io import read_image
from torchvision.models import resnet50, ResNet50_Weights
import os
import cv2

#Video
videopath = "C:/Users/prana/Desktop/Carnegie Mellon/Computer Vision/16720_S23_hw5/water_bottle_vid.mp4"
video = cv2.VideoCapture(videopath)

#Saving frames to folder
data_dir = "C:/Users/prana/Desktop/Carnegie Mellon/Computer Vision/16720_S23_hw5/6_3_videoframe/"
frame_num = 0

while True:
    success, frame = video.read()

    if not success:
        break
```



```
    frame_string = str(frame_num)

    cv2.imwrite(os.path.join(data_dir, 'frame_'+frame_string+'.jpg'), frame)

    frame_num += 1

match_count = 0
file_count = 0
for filename in os.listdir(data_dir):

    try:
        file_count += 1
        img = read_image(data_dir + filename)

        # Step 1: Initialize model with the best available weights
        weights = ResNet50_Weights.DEFAULT
        model = resnet50(weights=weights)
        model.eval()
```



```
        # Step 2: Initialize the inference transforms
        preprocess = weights.transforms()

        # Step 3: Apply inference preprocessing transforms
        batch = preprocess(img).unsqueeze(0)

        # Step 4: Use the model and print the predicted category
        prediction = model(batch).squeeze(0).softmax(0)
        class_id = prediction.argmax().item()
        score = prediction[class_id].item()
        category_name = weights.meta["categories"][class_id]

        if category_name == "water bottle":
            match_count += 1

    except:
        #Ignoring black and white images as they do not work on the model
        print(filename + " failed")

valid_acc = match_count/file_count * 100
print("Validation Accuracy:", valid_acc)
```



```
  warnings.warn(
Validation Accuracy: 98.21958456973294
```

Q6.3

The video does much better than the validation set (98 % vs 76 % accuracy), likely because the water bottle used is prominent in the dataset is prominently weighted with this water bottle type. The model could be made more robust by adding more datapoints and water bottle shapes (like Aquafina bottle because it resembles a soda bottle).

Deliverables

The assignment should be submitted to Gradescope. The writeup should be submitted as a pdf named <AndrewId>.pdf. The code should be submitted as a zip named <AndrewId>.zip. The zip when uncompressed should produce the following files.

- `nn.py`
- `q4.py`
- `run_q2.py`
- `run_q3.py`
- `run_q4.py`
- `run_q5.py` (extra credit)
- any `.py` files for Q6
- `util.py`

References

[1] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. 2010. http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf.

[2] P. J. Grother. Nist special database 19 – handprinted forms and characters database. https://www.nist.gov/srd/nist-special-database-19, 1995.

Appendix: Neural Network Overview

Deep learning has quickly become one of the most applied machine learning techniques in computer vision. Convolutional neural networks have been applied to many different computer vision problems such as image classification, recognition, and segmentation with great success. In this assignment, you will first implement a fully connected feed-forward neural network for handwritten character classification. Then in the second part, you will implement a system to locate characters in an image, which you can then classify with your deep network. The end result will be a system that, given an image of handwritten text, will output the text contained in the image.

Mathematical overview

Here we will give a brief overview of the math for a single hidden layer feed-forward network. For a more detailed look at the math and derivation, please see the class slides.

A fully-connected network $\mathbf{f}$, for classification, applies a series of linear and non-linear functions to an input data vector $\mathbf{x}$ of size $N \times 1$ to produce an output vector $\mathbf{f}(\mathbf{x})$ of size $C \times 1$, where each element i of the output vector represents the probability of $\mathbf{x}$ belonging to the class i. Since the data samples are of dimensionality N, this means the input layer has N input units. To compute the value of the output units, we must first compute the values of all the hidden layers. The first hidden layer *pre-activation* $\mathbf{a}^{(1)}(\mathbf{x})$ is given by

$$\mathbf{a}^{(1)}(\mathbf{x}) = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$

Then the *post-activation* values of the first hidden layer $\mathbf{h}^{(1)}(\mathbf{x})$ are computed by applying a non-linear activation function $\mathbf{g}$ to the *pre-activation* values

$$\mathbf{h}^{(1)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(1)}(\mathbf{x})) = \mathbf{g}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$$

Subsequent hidden layer ($1 < t \leq T$) pre- and post activations are given by:

$$\begin{aligned}\mathbf{a}^{(t)}(\mathbf{x}) &= \mathbf{W}^{(t)}\mathbf{h}^{(t-1)} + \mathbf{b}^{(t)} \\ \mathbf{h}^{(t)}(\mathbf{x}) &= \mathbf{g}(\mathbf{a}^{(t)}(\mathbf{x}))\end{aligned}$$

The output layer *pre-activations* $\mathbf{a}^{(T)}(\mathbf{x})$ are computed in a similar way

$$\mathbf{a}^{(T)}(\mathbf{x}) = \mathbf{W}^{(T)}\mathbf{h}^{(T-1)}(\mathbf{x}) + \mathbf{b}^{(T)}$$

and finally the *post-activation* values of the output layer are computed with

$$\mathbf{f}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(T)}(\mathbf{x})) = \mathbf{o}(\mathbf{W}^{(T)}\mathbf{h}^{(T-1)}(\mathbf{x}) + \mathbf{b}^{(T)})$$

where $\mathbf{o}$ is the output activation function. Please note the difference between $\mathbf{g}$ and $\mathbf{o}$! For this assignment, we will be using the sigmoid activation function for the hidden layer, so:

$$\mathbf{g}(y) = \frac{1}{1 + \exp(-y)}$$

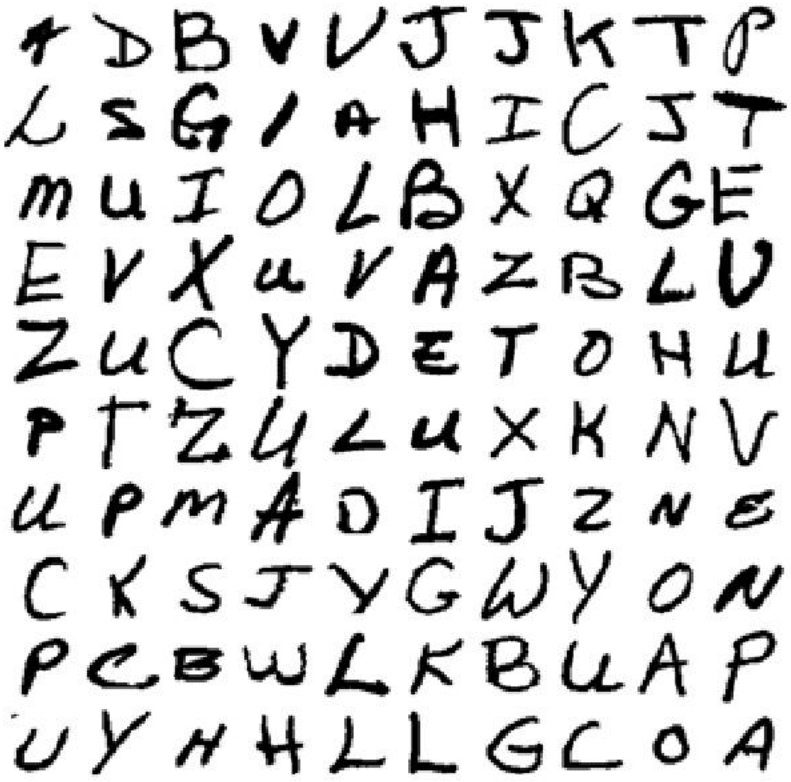


Figure 2: Samples from NIST Special 19 dataset [2]

where when $\mathbf{g}$ is applied to a vector, it is applied element wise across the vector.

Since we are using this deep network for classification, a common output activation function to use is the softmax function. This will allow us to turn the real value, possibly negative values of $\mathbf{a}^{(T)}(\mathbf{x})$ into a set of probabilities (vector of positive numbers that sum to 1). Letting $\mathbf{x}_i$ denote the i^{th} element of the vector $\mathbf{x}$, the softmax function is defined as:

$$\mathbf{o}_i(\mathbf{y}) = \frac{\exp(\mathbf{y}_i)}{\sum_j \exp(\mathbf{y}_j)}$$

Gradient descent is an iterative optimisation algorithm, used to find the local optima. To find the local minima, we start at a point on the function and move in the direction of negative gradient (steepest descent) till some stopping criteria is met.

Backprop

The update equation for a general weight $W_{ij}^{(t)}$ and bias $b_i^{(t)}$ is

$$W_{ij}^{(t)} = W_{ij}^{(t)} - \alpha * \frac{\partial L_{\mathbf{f}}}{\partial W_{ij}^{(t)}}(\mathbf{x}) \qquad b_i^{(t)} = b_i^{(t)} - \alpha * \frac{\partial L_{\mathbf{f}}}{\partial b_i^{(t)}}(\mathbf{x})$$

α is the learning rate. Please refer to the backpropagation slides for more details on how to derive the gradients. Note that here we are using softmax loss (which is different from the least square loss in the slides).

Debugging Checklist for Training Neural Networks

- Input-output pairs should make sense. Inspect them!
- Data loads correctly. Visualize it!
- Data is transformed correctly.
- Model and Data input-target dimensions must match
- Know what the output should look like. Is the result reasonable?
- Batch size is > 1, else the gradient will be too noisy.
- Learning rate is tuned – not too small, not too large.
- Save the best checkpoint. Check on the validation set for overfitting!
- Run inference is in eval mode.
- Log everything!
- Fix the random seed in PyTorch / NumPy / OS when debugging
- Complex Error Logs are often Simple Bugs. Localize your errors in Tracebacks!
- Establish the invariants. What must be true? (Assert Statement)
- Make small changes between experiments to localize errors.
- Search Google/stack overflow with the error messages.