Homework 5: Neural Networks for Recognition

For each question please refer to the handout for more details.

Programming questions begin at **Q2**. **Remember to run all cells** and save the notebook to your local machine as a pdf for gradescope submission.

Collaborators

List your collaborators for all questions here:

Q1 Theory

Q1.1 (3 points)

Softmax is defined as below, for each index i in a vector $x \in \mathbb{R}^d$.

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

Prove that softmax is invariant to translation, that is

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

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

$$softmax(x + c)_i = \frac{e^{x_i + c}}{\sum_j e^{(x_j + c)}} = \frac{e^{x_i} e^c}{\sum_j e^{x_j} e^c} = \frac{e^c e^{x_i}}{e^c \sum_j e^{x_j}} = \frac{e^{x_i}}{\sum_j e^{x_j}} = softmax(x)_i$$
$$\forall c \in \mathbb{R}$$

We use $c = -\max x_i$ to avoid e^x getting too big when x is a very large value. By applying $c = -\max x_i$, the largest value of $e^{x_i + c}$ is 1, which will make the computation easier.

Q1.2

Softmax can be written as a three-step process, with $s_i = e^{x_i}$, $S = \sum s_i$ and $softmax(x)_i = \frac{1}{S} s_i$.

Q1.2.1 (1 point)

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

---

1. range of each element is $(0, 1]$.
2. sum over all elements is $\frac{1}{S}\sum_i s_i = 1$.

Q1.2.2 (1 point)

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

---

probability distribution

Q1.2.3 (1 point)

Now explain the role of each step in the multi-step process.

---

1. $s_i = e^{x_i}$, this exponential function is to map each element of the input to a positive number.
2. $S = \sum_i s_i$, this is to sum up all the exponentiated values, prepare to normalize constant for softmax function.
3. $softmax(x)_i = \frac{1}{S}s_i$, convert the vector of positie numbers into a probability distribution.

Q1.3 (3 points)

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

Suppose I have a NN with two layers, it takes the input $x \in \mathbb{R}^2$. \ The weight and bias of the first layer are $W_1 \in \mathbb{R}^{2\times 2}$ and $b_1 \in \mathbb{R}^2$. \ The weight and bias of the second layer are $W_2 \in \mathbb{R}^{2\times 2}$ and $b_2 \in \mathbb{R}^2$. \ For first layer, $z_1 = W_1x + b_1$. \ For second layer, $y = W_2z_1 + b_2$. \ If I use the z_1 from first layer, then I have:

$$y = W_2(W_1x + b_1) + b2 = (W_2W_1)x + (W_2b_1 + b_2) = Wx + b$$

Hence this is equivalent to linear regression.

Q1.4 (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).

$$
\begin{aligned}
\sigma(x) &= \frac{1}{1+e^{-x}} \\
\frac{\partial \sigma(x)}{\partial x} &= -\frac{1}{(1+e^{-x})^2}(-e^{-x}) \\
&= \frac{e^{-x}}{(1+e^{-x})^2} \\
&= \frac{1+e^{-x}}{(1+e^{-x})^2} - \frac{1}{(1+e^{-x})^2} \\
&= \sigma(x) - \sigma(x)^2
\end{aligned}
$$

Q1.5 (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 (a scalar) with respect to y, show how to get the gradients $\frac{\partial J}{\partial W}$, $\frac{\partial J}{\partial x}$ and $\frac{\partial J}{\partial b}$. Be sure to do the derivatives with scalars and re-form the matrix form afterwards. Here are some notional suggestions.

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

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

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

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

Q1.6

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

Q1.6.1 (1 point)

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 the gradient you derived in Q1.4)?

Since $\sigma(x)' = \sigma(x) - \sigma(x)^2 = \sigma(x)(1 - \sigma(x))$, that is to say that when $|x|$ gets large, $\sigma(x)$ approaches 0. So during the backpropagation, the derivative of the sigmoid function become smaller, the gradients of the weights become smaller.

Q1.6.2 (1 point)

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 $\tanh$?

$\sigma(x) \in (0, 1)$ \ $tanh(x) \in (-1, 1)$ \ The reason that $tanh(x)$ is preferred is that $tanh(x)$ is less likely to have vanishing gradient problem like sigmoid function and convergent faster.

Q1.6.3 (1 point)

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

Since $tanh(x)' = 1 - tanh(x)^2$, it covers larger range than the gradient of $\sigma(x)$.

Q1.6.4 (1 point)

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

$$
\begin{aligned}
tanh(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}} \\
&= \frac{1 - e^{-2x}}{1 + e^{-2x}} \\
&= 1 - \frac{2}{1 + e^{-2x}} \\
&= 2\sigma(2x) - 1
\end{aligned}
$$

Q2 Implement a Fully Connected Network

Run the following code to import the modules you'll need. When implementing the functions in Q2, make sure you run the test code (provided after Q2.3) along the way to check if your implemented functions work as expected.

In [2]:
```
import os
import numpy as np
import scipy.io
import matplotlib.pyplot as plt
```

```
import matplotlib.patches
from mpl_toolkits.axes_grid1 import ImageGrid

import skimage
import skimage.measure
import skimage.color
import skimage.restoration
import skimage.filters
import skimage.morphology
import skimage.segmentation
```

Q2.1 Network Initialization

Q2.1.1 (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?

Firstly, if all the weights are initialized to zero, every neuron in each layer will start with the same weight so that the learning follow the same gradient during training, which will make the NN depth unable to contribute. Secondly, the derivative w.r.t the loss function is the same for every weight, which means that there is no gradient descent can be taken.

Q2.1.2 (3 points)

Implement the initialize*weights() function to initialize the weights for a single layer with Xavier initialization, 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]).

In [3]:
```
############################## Q 2.1.2 ##############################
def initialize_weights(in_size,out_size,params,name=''):
    """
    we will do XW + b, with the size of the input data array X being [number of examples
    the weights W should be initialized as a 2D array
    the bias vector b should be initialized as a 1D array, not a 2D array with a singlet
    the output of this layer should be in size [number of examples, out_size]
    """
    W, b = None, None

    ##########################
    ##### your code here #####
    ##########################
    limit = np.sqrt(6 / (in_size + out_size))
    W = np.random.uniform(-limit, limit, (in_size, out_size))
    b = np.zeros(out_size)


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

Q2.1.3 (2 points)

Why do we scale the initialization depending on layer size (see Fig 6 in the [Glorot et al])?

Use the $\sqrt{\frac{6}{n_{in}+n_{out}}}$ as the normalization factor is to ensure the weights are neither not too small nor not too big so that won't vanish or explode the computation. The Fig 6 says that normalized initialization is able to get more stable activation values across the layers and prevent the 0-peak from increasing for higher layers.

Q2.2 Forward Propagation

Q2.2.1 (4 points)

Implement the sigmoid() function, which computes the elementwise sigmoid activation of entries in an input array. Then implement the forward() function which computes forward propagation for a single layer, namely $y = \sigma(XW + b)$.

In [36]:
```
############################## Q 2.2.1 ################################
def sigmoid(x):
    """
    Implement an elementwise sigmoid activation function on the input x,
    where x is a numpy array of size [number of examples, number of output dimensions]
    """
    res = None

    ###########################
    ##### your code here #####
    ###########################
    res = 1 / (1 + np.exp(-x))


    return res
    # return np.where(x >= 0, 1 / (1 + np.exp(-x)), np.exp(x) / (1 + np.exp(x)))
```

In [30]:
```
############################## Q 2.2.1 ################################
def forward(X,params,name='',activation=sigmoid):
    """
    Do a forward pass for a single layer that computes the output: activation(XW + b)

    Keyword arguments:
    X -- input numpy array of size [number of examples, number of input dimensions]
    params -- a dictionary containing parameters, as how you initialized in Q 2.1.2
    name -- name of the layer
    activation -- the activation function (default is sigmoid)
    """
    # compute the output values before and after the activation function
    pre_act, post_act = None, None
    # get the layer parameters
    W = params['W' + name]
    b = params['b' + name]


    ###########################
    ##### your code here #####
    ###########################
    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 backpropagation
```

```
    params['cache_' + name] = (X, pre_act, post_act)

    return post_act
```

Q2.2.2 (3 points)

Implement the softmax() function. Be sure to use the numerical stability trick you derived in Q1.1 softmax.

In [31]:
```
############################## Q 2.2.2  ##############################
def softmax(x):
    """
    x is a numpy array of size [number of examples, number of classes]
    softmax should be done for each row
    """
    res = None

    ###########################
    ##### your code here #####
    ###########################
    c = -np.max(x, axis = 1, keepdims = True)
    exp_x = np.exp(x + c)
    sum_exp_x = np.sum(exp_x, axis = 1, keepdims = True)
    res = exp_x / sum_exp_x


    return res
```

Q2.2.3 (3 points)

Implement the compute_loss_and_acc() function to compute the accuracy given a set of labels, along with the scalar loss across the data. The loss function generally used for classification is the cross-entropy loss.

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

Here $\mathbf{D}$ is the full training dataset of N data samples x (which are $D \times 1$ vectors, D is the dimensionality of data) and labels y (which are $C \times 1$ one-hot vectors, C is the number of classes), and $f : \mathbb{R}^D \to [0, 1]^C$ is the classifier which outputs the probabilities for the classes. The $\log$ is the natural $\log$.

In [32]:
```
############################## Q 2.2.3 ##############################
def compute_loss_and_acc(y, probs):
    """
    compute total loss and accuracy

    Keyword arguments:
    y -- the labels, which is a numpy array of size [number of examples, number of class
    probs -- the probabilities output by the classifier, i.e. f(x), which is a numpy arr
    """
    loss, acc = None, None

    ###########################
    ##### your code here #####
    ###########################
    loss = -np.sum(y * np.log(probs))
    pred_y = np.argmax(probs, axis=1)
    true_y = np.argmax(y, axis=1)

    acc = np.mean(pred_y == true_y)

    return loss, acc
```

Q2.3 Backwards Propagation

Q2.3 (7 points)

Implement the backwards() function to compute backpropagation for a single layer, given the original weights, the appropriate intermediate results, and the gradient with respect to the loss. You should return the gradient with respect to the inputs (grad_X) so that it can be used in the backpropagation for the previous layer. As a size check, your gradients should have the same dimensions as the original objects.

In [33]:
```
############################## Q 2.3 ################################
def sigmoid_deriv(post_act):
    """
    we give this to you, because you proved it in Q1.4
    it's a function of the post-activation values (post_act)
    """
    res = post_act*(1.0-post_act)
    return res

def backwards(delta,params,name='',activation_deriv=sigmoid_deriv):
    """
    Do a backpropagation pass for a single layer.

    Keyword arguments:
    delta -- gradients of the loss with respect to the outputs (errors to back propagate
    params -- a dictionary containing parameters, as how you initialized in Q 2.1.2
    name -- name of the layer
    activation_deriv -- the derivative of the activation function
    """
    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]

    # by the chain rule, do the derivative through activation first
    # (don't forget activation_deriv is a function of post_act)
    # then compute the gradients w.r.t W, b, and X
    ############################
    ##### your code here #####
    ############################
    # apply the activation function derivative
    delta = delta * activation_deriv(post_act)

    # gradients w.r.t W, b, X
    grad_W = np.dot(X.T, delta) / X.shape[0]
    grad_b = np.sum(delta, axis=0) / X.shape[0]
    grad_X = np.dot(delta, W.T)

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

Make sure you run below test code along the way to check if your implemented functions work as expected.

In [12]:
```
def linear(x):
    # Define a linear activation, which can be used to construct a "no activation" layer
    return x
```

```
def linear_deriv(post_act):
    return np.ones_like(post_act)
```

In [13]:

```
# test code
# generate some fake data
# feel free to plot it in 2D, what do you think these 4 classes are?
g0 = np.random.multivariate_normal([3.6,40],[[0.05,0],[0,10]],10)
g1 = np.random.multivariate_normal([3.9,10],[[0.01,0],[0,5]],10)
g2 = np.random.multivariate_normal([3.4,30],[[0.25,0],[0,5]],10)
g3 = np.random.multivariate_normal([2.0,10],[[0.5,0],[0,10]],10)
x = np.vstack([g0,g1,g2,g3])

# we will do XW + B in the forward pass
# this implies that the data X is in [number of examples, number of input dimensions]

# create labels
y_idx = np.array([0 for _ in range(10)] + [1 for _ in range(10)] + [2 for _ in range(10)
# turn to one-hot encoding, this implies that the labels y is in [number of examples, nu
y = np.zeros((y_idx.shape[0],y_idx.max()+1))
y[np.arange(y_idx.shape[0]),y_idx] = 1
print("data shape: {} labels shape: {}".format(x.shape, y.shape))

# parameters in a dictionary
params = {}

# Q 2.1.2
# we will build a two-layer neural network
# first, initialize the weights and biases for the two layers
# the first layer, in_size = 2 (the dimension of the input data), out_size = 25 (number
initialize_weights(2,25,params,'layer1')
# the output layer, in_size = 25 (number of neurons), out_size = 4 (number of classes)
initialize_weights(25,4,params,'output')
assert(params['Wlayer1'].shape == (2,25))
assert(params['blayer1'].shape == (25,))
assert(params['Woutput'].shape == (25,4))
assert(params['boutput'].shape == (4,))

# with Xavier initialization
# expect the means close to 0, variances in range [0.05 to 0.12]
print("Q 2.1.2: {}, {:.2f}".format(params['blayer1'].mean(),params['Wlayer1'].std()**2))
print("Q 2.1.2: {}, {:.2f}".format(params['boutput'].mean(),params['Woutput'].std()**2))

# Q 2.2.1
# implement sigmoid
# there might be an overflow warning due to exp(1000)
test = sigmoid(np.array([-1000,1000]))
print('Q 2.2.1: sigmoid outputs should be zero and one\t',test.min(),test.max())
# a forward pass on the first layer, with sigmoid activation
h1 = forward(x,params,'layer1',sigmoid)
assert(h1.shape == (40, 25))

# Q 2.2.2
# implement softmax
# a forward pass on the second layer (the output layer), with softmax so that the output
probs = forward(h1,params,'output',softmax)
# make sure you understand these values!
# should be positive, 1 (or very close to 1), 1 (or very close to 1)
print('Q 2.2.2:',probs.min(),min(probs.sum(1)),max(probs.sum(1)))
assert(probs.shape == (40,4))

# Q 2.2.3
# implement compute_loss_and_acc
loss, acc = compute_loss_and_acc(y, probs)
# should be around -np.log(0.25)*40 [~55] or higher, and 0.25
# if it is not, check softmax!
```

```
print("Q 2.2.3 loss: {}, acc:{:.2f}".format(loss,acc))

# Q 2.3
# here we cheat for you, you can use it in the training loop in Q2.4
# the derivative of cross-entropy(softmax(x)) is probs - 1[correct actions]
delta1 = probs - y

# backpropagation for the output layer
# we already did derivative through softmax when computing delta1 as above
# so we pass in a linear_deriv, which is just a vector of ones to make this a no-op
delta2 = backwards(delta1,params,'output',linear_deriv)
# backpropagation for the first layer
backwards(delta2,params,'layer1',sigmoid_deriv)

# the sizes of W and b should match the sizes of their gradients
for k,v in sorted(list(params.items())):
    if 'grad' in k:
        name = k.split('_')[1]
        # print the size of the gradient and the size of the parameter, the two sizes sh
        print('Q 2.3',name,v.shape, params[name].shape)
```

```
data shape: (40, 2) labels shape: (40, 4)
Q 2.1.2: 0.0, 0.06
Q 2.1.2: 0.0, 0.08
Q 2.2.1: sigmoid outputs should be zero and one  0.0 1.0
Q 2.2.2: 0.11454013178291919 0.9999999999999998 1.0000000000000002
Q 2.2.3 loss: 63.17037660633554, acc:0.25
Q 2.3 Wlayer1 (2, 25) (2, 25)
Q 2.3 Woutput (25, 4) (25, 4)
Q 2.3 blayer1 (25,) (25,)
Q 2.3 boutput (4,) (4,)
<ipython-input-4-2ccfa57689a0>:12: RuntimeWarning: overflow encountered in exp
  res = 1 / (1 + np.exp(-x))
```

Q2.4 Training Loop: Stochastic Gradient Descent

Q2.4 (5 points)

Implement the get_random_batches() function that takes the entire dataset (x and y) as input and splits it into random batches. 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.

In [34]:

```
################################## Q 2.4 ###################################
def get_random_batches(x,y,batch_size):
    """
    split x (data) and y (labels) into random batches
    return a list of [(batch1_x,batch1_y)...]
    """
    batches = []

    ############################
    ##### your code here #####
    ############################
    data_size = x.shape[0]
    indices = np.arange(data_size)
    np.random.shuffle(indices)

    for i in range(0, data_size, batch_size):
      batch_indices = indices[i: i+batch_size]
```

```
    batch_x = x[batch_indices]
    batch_y = y[batch_indices]
    batches.append((batch_x, batch_y))

return batches
```

In [15]:
```
# Q 2.4
batches = get_random_batches(x,y,5)
batch_num = len(batches)
# print batch sizes
print([_[0].shape[0] for _ in batches])
print(batch_num)
```

```
[5, 5, 5, 5, 5, 5, 5, 5]
8
```

In [15]:
```
############################## Q 2.4 ##############################
# WRITE A TRAINING LOOP HERE
max_iters = 500
learning_rate = 1e-3
# with default settings, you should get loss <= 35 and accuracy >= 75%
for itr in range(max_iters):
    total_loss = 0
    avg_acc = 0
    for xb,yb in batches:
        ##########################
        ##### your code here #####
        ##########################
        # forward
        h1 = forward(xb, params, 'layer1', sigmoid)
        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 to update the parameters
        params["Wlayer1"] -= learning_rate * params["grad_Wlayer1"]
        params["blayer1"] -= learning_rate * params["grad_blayer1"]
        params["Woutput"] -= learning_rate * params["grad_Woutput"]
        params["boutput"] -= learning_rate * params["grad_boutput"]

    avg_acc /= len(batches)

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

```
itr: 00          loss: 32.16     acc : 0.80
itr: 100         loss: 31.54     acc : 0.82
itr: 200         loss: 30.97     acc : 0.82
itr: 300         loss: 30.43     acc : 0.82
itr: 400         loss: 29.93     acc : 0.88
```

Q3 Training Models

Run below code to download and put the unzipped data in '/content/data' folder.

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.

In [16]:
```
if not os.path.exists('/content/data'):
  os.mkdir('/content/data')
  !wget http://www.cs.cmu.edu/~lkeselma/16720a_data/data.zip -O /content/data/data.zip
  !unzip "/content/data/data.zip" -d "/content/data"
  os.system("rm /content/data/data.zip")
```

```
--2024-04-11 01:57:15--  http://www.cs.cmu.edu/~lkeselma/16720a_data/data.zip
Resolving www.cs.cmu.edu (www.cs.cmu.edu)... 128.2.42.95
Connecting to www.cs.cmu.edu (www.cs.cmu.edu)|128.2.42.95|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 216305627 (206M) [application/zip]
Saving to: ‘/content/data/data.zip’

/content/data/data. 100%[===================>] 206.28M   311KB/s    in 8m 14s

2024-04-11 02:05:30 (427 KB/s) - ‘/content/data/data.zip’ saved [216305627/216305627]

Archive:  /content/data/data.zip
warning:  stripped absolute path spec from /
mapname:  conversion of  failed
  inflating: /content/data/nist26_valid.mat
  inflating: /content/data/nist26_model_60iters.mat
  inflating: /content/data/nist36_test.mat
  inflating: /content/data/nist26_test.mat
  inflating: /content/data/nist26_train.mat
  inflating: /content/data/nist36_train.mat
  inflating: /content/data/nist36_valid.mat
```

In [17]:
```
ls /content/data
```

```
nist26_model_60iters.mat*  nist26_train.mat*  nist36_test.mat*   nist36_valid.mat*
nist26_test.mat*           nist26_valid.mat*  nist36_train.mat*
```

Q3.1 (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:

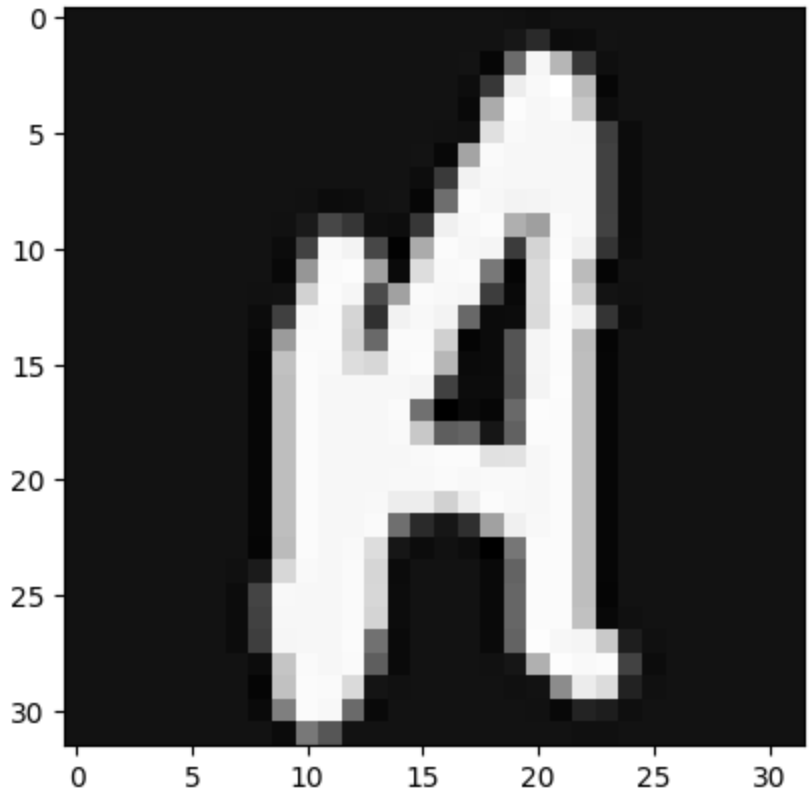
(1) the accuracy on both the training and validation set over the epochs, and

(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\%. Hint: Use fixed random seeds to improve reproducibility.

In [18]:
```
train_data = scipy.io.loadmat('/content/data/nist36_train.mat')
valid_data = scipy.io.loadmat('/content/data/nist36_valid.mat')
test_data = scipy.io.loadmat('/content/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 True: # view the data
    for crop in train_x:
        plt.imshow(crop.reshape(32,32).T, cmap="Greys")
        plt.show()
        break
```



In [20]:

```
################################ Q 3.1 ################################
max_iters = 50
# pick a batch size, learning rate
batch_size = 32
learning_rate = 1e-1
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',sigmoid)
    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',sigmoid)
    probs = forward(h1,params,'output',softmax)
    loss, acc = compute_loss_and_acc(valid_y, probs)
    valid_loss.append(loss/valid_x.shape[0])
    valid_acc.append(acc)

    total_loss = 0
    avg_acc = 0
    for xb,yb in batches:
        # training loop can be exactly the same as q2!
        ##########################
        ##### your code here #####
        ##########################
        # forward
        h1 = forward(xb, params, 'layer1', sigmoid)
        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 to update the parameters
        params["Wlayer1"] -= learning_rate * params["grad_Wlayer1"]
        params["blayer1"] -= learning_rate * params["grad_blayer1"]
        params["Woutput"] -= learning_rate * params["grad_Woutput"]
        params["boutput"] -= learning_rate * params["grad_boutput"]

    avg_acc /= batch_num

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

# record final training and validation accuracy and loss
h1 = forward(train_x,params,'layer1',sigmoid)
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',sigmoid)
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)

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

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

```
itr: 00   loss: 35506.63   acc: 0.16
itr: 02   loss: 21427.42   acc: 0.53
itr: 04   loss: 15789.02   acc: 0.63
```

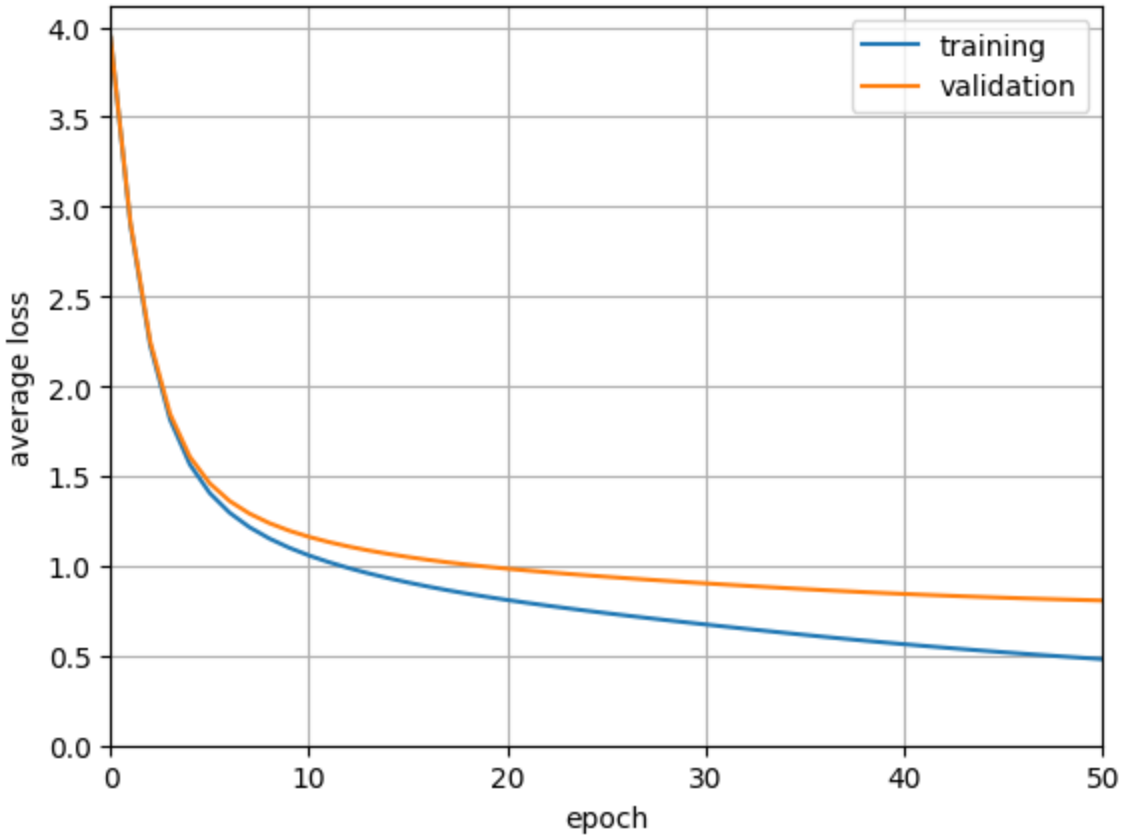
```
itr: 06   loss: 13391.83   acc: 0.67
itr: 08   loss: 12015.87   acc: 0.70
itr: 10   loss: 11057.63   acc: 0.72
itr: 12   loss: 10316.11   acc: 0.74
itr: 14   loss: 9706.31   acc: 0.75
itr: 16   loss: 9185.08   acc: 0.77
itr: 18   loss: 8727.63   acc: 0.78
itr: 20   loss: 8318.87   acc: 0.79
itr: 22   loss: 7949.10   acc: 0.80
itr: 24   loss: 7611.68   acc: 0.81
itr: 26   loss: 7301.56   acc: 0.82
itr: 28   loss: 7014.77   acc: 0.82
itr: 30   loss: 6748.27   acc: 0.83
itr: 32   loss: 6499.76   acc: 0.84
itr: 34   loss: 6267.38   acc: 0.84
itr: 36   loss: 6049.54   acc: 0.85
itr: 38   loss: 5844.80   acc: 0.86
itr: 40   loss: 5651.83   acc: 0.86
itr: 42   loss: 5469.38   acc: 0.87
itr: 44   loss: 5296.36   acc: 0.87
itr: 46   loss: 5131.83   acc: 0.88
itr: 48   loss: 4975.00   acc: 0.88
Validation accuracy:  0.7738888888888888
Test accuracy:  0.7844444444444445
```

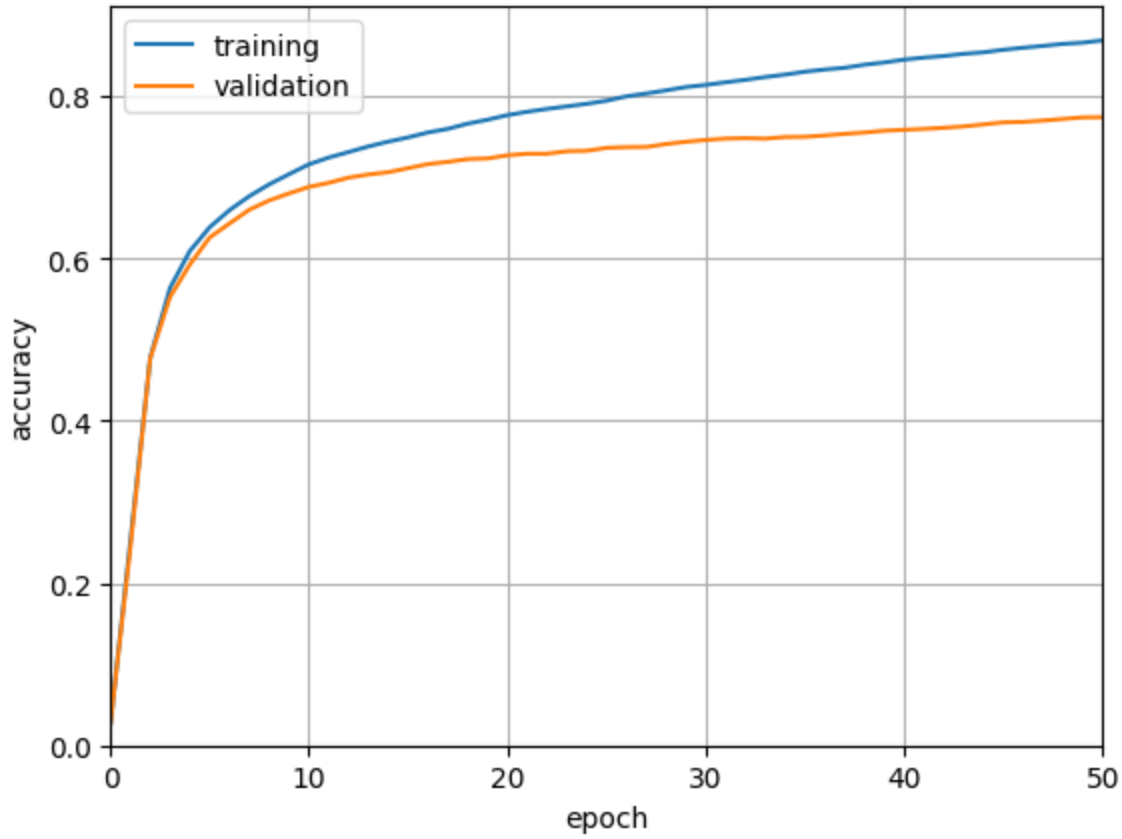
In [21]:

```python
# save the final network
import pickle

saved_params = {k:v for k,v in params.items() if '_' not in k}
with open('/content/q3_weights.pickle', 'wb') as handle:
  pickle.dump(saved_params, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

In [22]:

```python
# 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")
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()
```



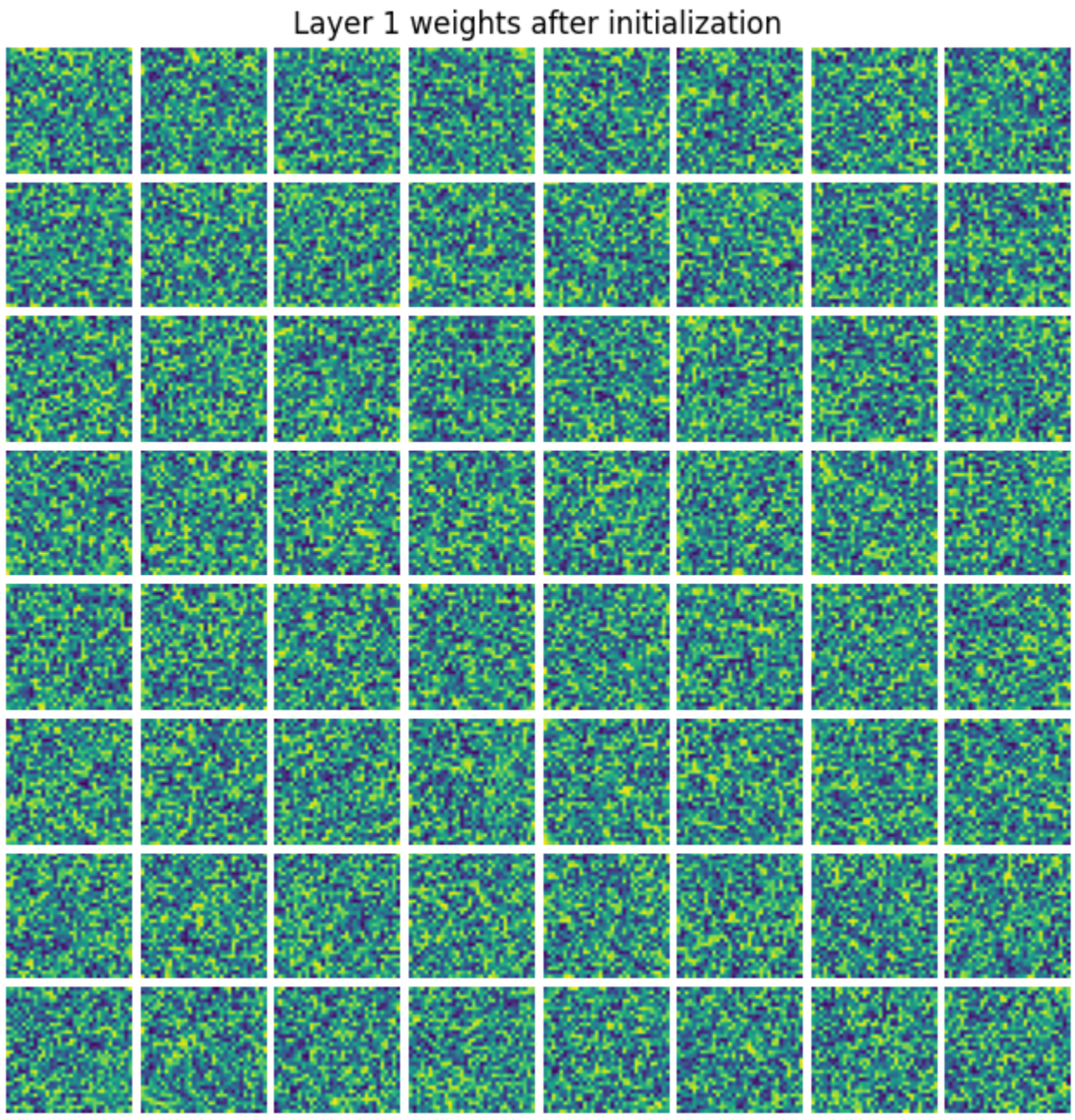


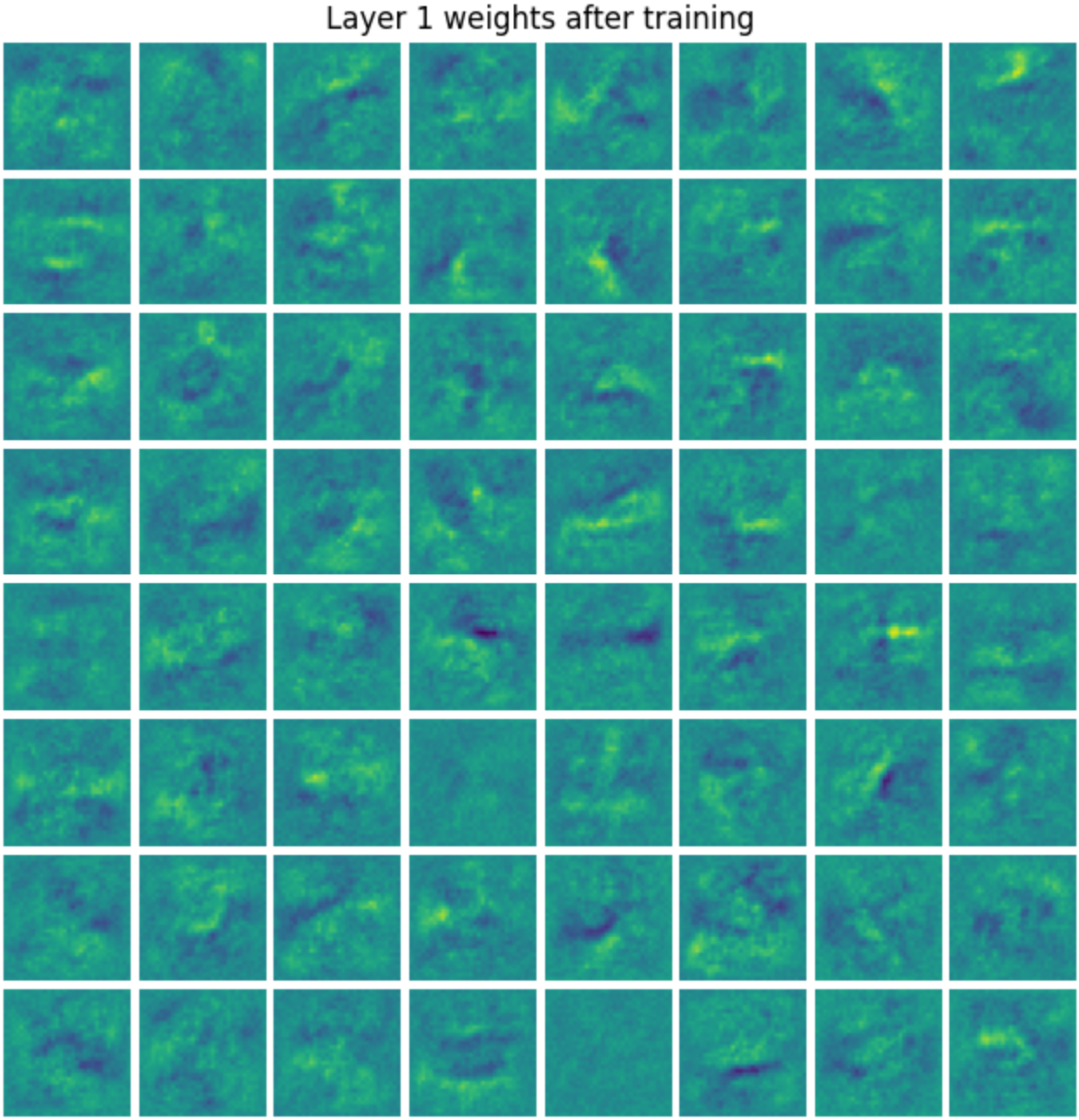
Q3.2 (3 points)

The provided code will visualize the first layer weights as 64 32x32 images, both immediately after initialization and after full training. Generate both visualizations. Comment on the learned weights and compare them to the initialized weights. Do you notice any patterns?

In [23]:
```
################################ Q 3.2 ################################
```

```
# visualize weights
fig = plt.figure(figsize=(8,8))
plt.title("Layer 1 weights after initialization")
plt.axis("off")
grid = ImageGrid(fig, 111, nrows_ncols=(8, 8), axes_pad=0.05)
for i, ax in enumerate(grid):
    ax.imshow(layer1_W_initial[:,i].reshape((32, 32)).T)
    ax.set_axis_off()
plt.show()

v = np.max(np.abs(params['Wlayer1']))
fig = plt.figure(figsize=(8,8))
plt.title("Layer 1 weights after training")
plt.axis("off")
grid = ImageGrid(fig, 111, nrows_ncols=(8, 8), axes_pad=0.05)
for i, ax in enumerate(grid):
    ax.imshow(params['Wlayer1'][:,i].reshape((32, 32)).T, vmin=-v, vmax=v)
    ax.set_axis_off()
plt.show()
```





The weights after initialization before training have randomly distributed noise and unable to show anything. The trained weights start to show some recognizable pattern which means the training is successful.

Q3.3 (3 points)

Use the code in Q3.1 to train and generate accuracy and loss plots for each of these three networks:

(1) one with 10 times your tuned learning rate,

(2) one with one-tenth your tuned learning rate, and

(3) one with your tuned learning rate.

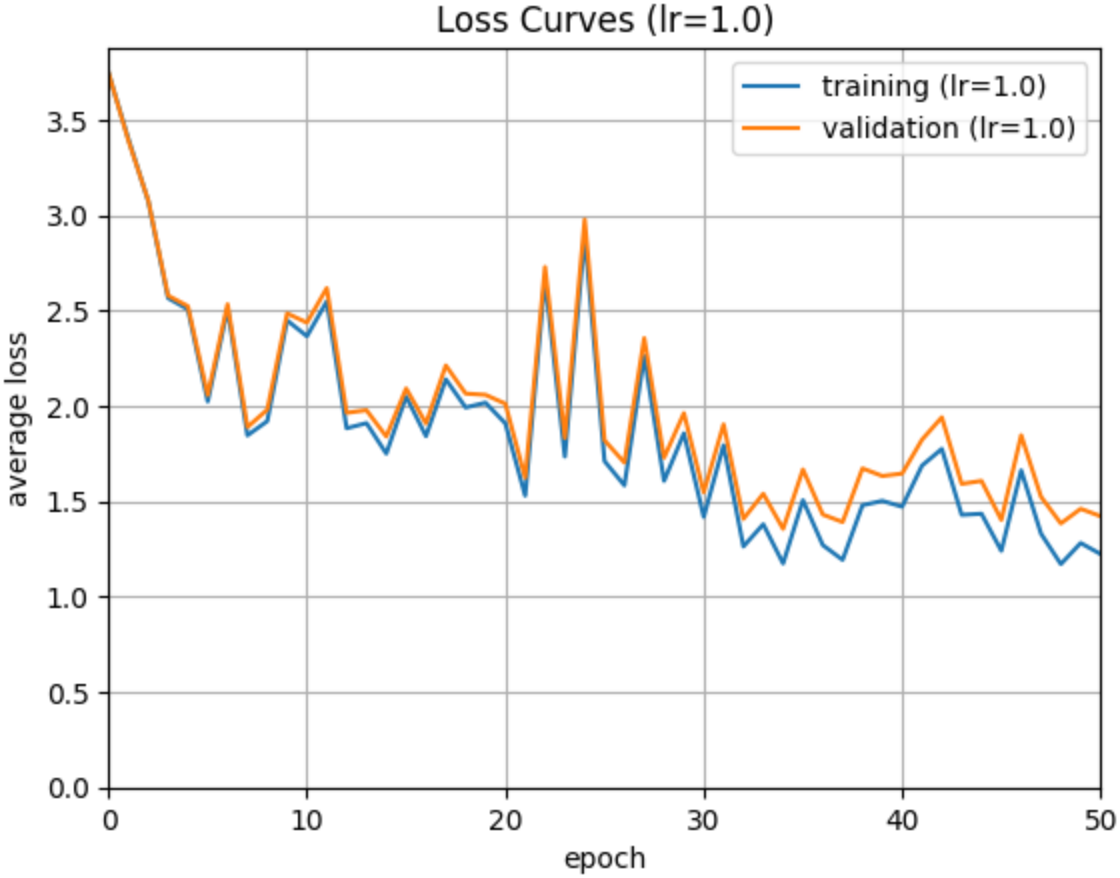
Include total of six plots (two will be the same from Q3.1). Comment on how the learning rates affect the training, and report the final accuracy of the best network on the test set. Hint: Use fixed random seeds to improve reproducibility.

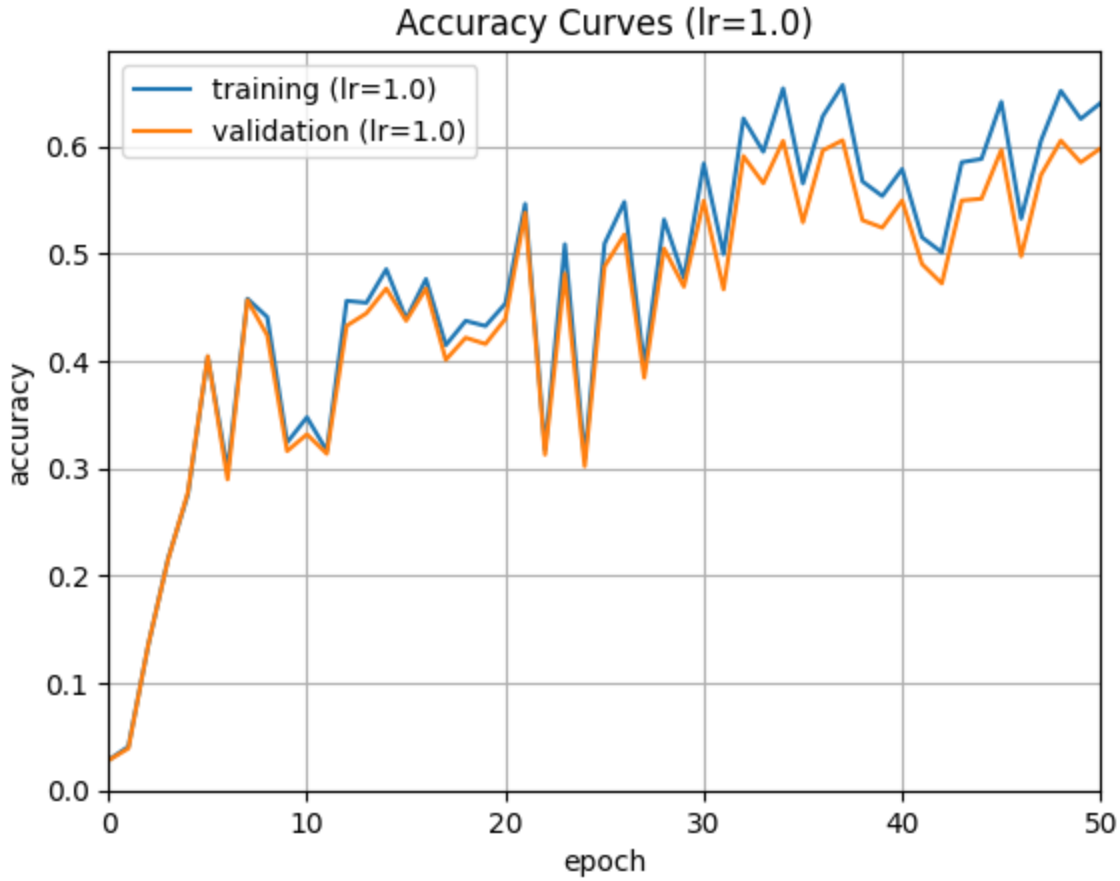
In [24]:

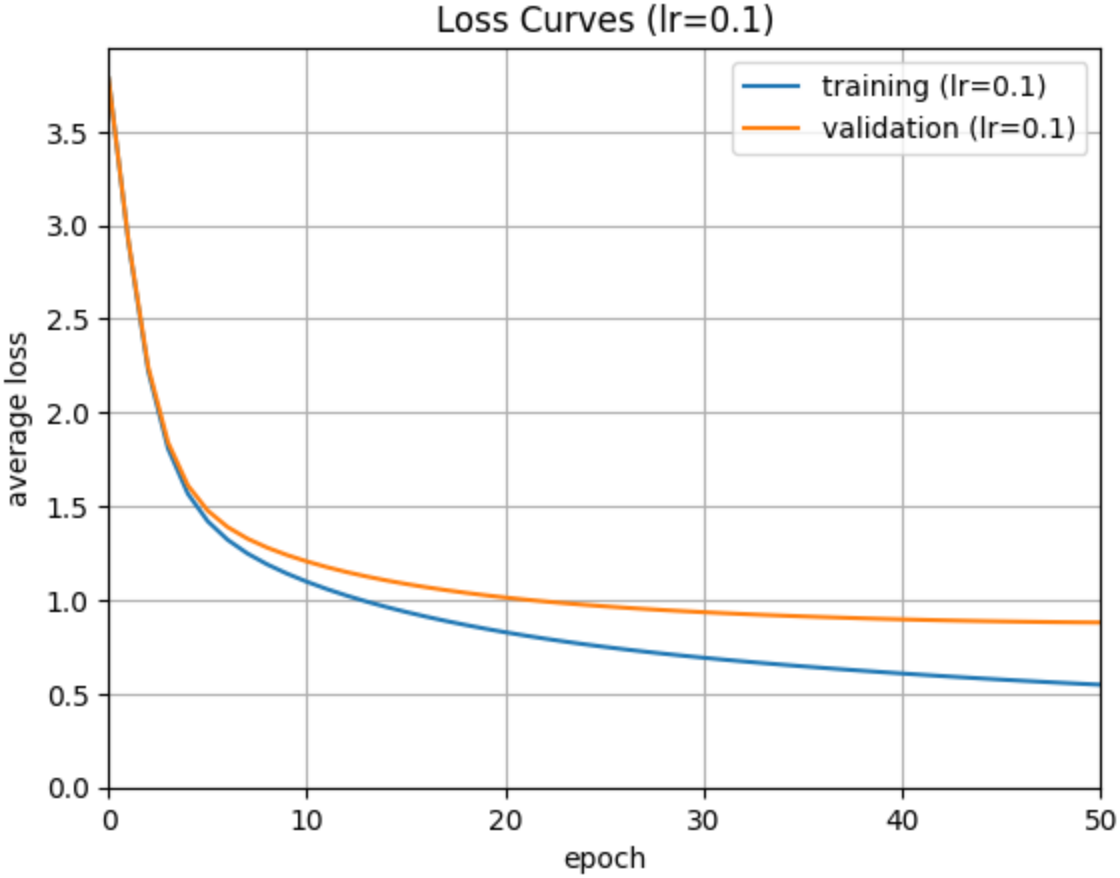
```
############################## Q 3.3 ##############################
#########################
##### your code here #####
#########################
def train_network(learning_rate):
  max_iters = 50
  # pick a batch size, learning rate
  batch_size = 32
  hidden_size = 64

  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',sigmoid)
      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',sigmoid)
      probs = forward(h1,params,'output',softmax)
      loss, acc = compute_loss_and_acc(valid_y, probs)
      valid_loss.append(loss/valid_x.shape[0])
      valid_acc.append(acc)

      total_loss = 0
      avg_acc = 0
      for xb,yb in batches:
          # forward
          h1 = forward(xb, params, 'layer1', sigmoid)
          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 to update the parameters
          params["Wlayer1"] -= learning_rate * params["grad_Wlayer1"]
          params["blayer1"] -= learning_rate * params["grad_blayer1"]
          params["Woutput"] -= learning_rate * params["grad_Woutput"]
          params["boutput"] -= learning_rate * params["grad_boutput"]

      avg_acc /= batch_num
  # record final training and validation accuracy and loss
  h1 = forward(train_x,params,'layer1',sigmoid)
```

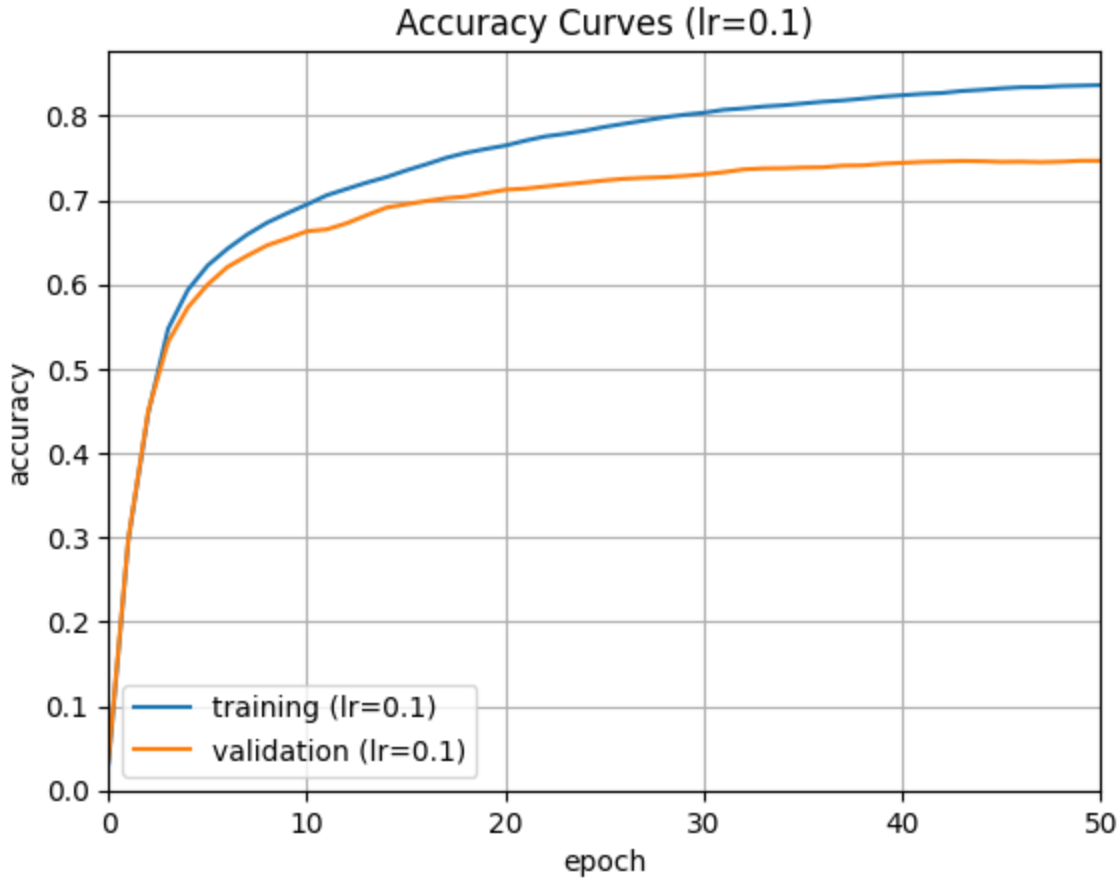
```
  probs = forward(h1,params,'output',softmax)
  loss, acc = compute_loss_and_acc(train_y, probs)
  train_loss.append(loss/train_x.shape[0])
  train_acc.append(acc)

  h1 = forward(valid_x,params,'layer1',sigmoid)
  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)

  # report validation accuracy; aim for 75%
  print(f"Validation accuracy(lr={learning_rate}): ", valid_acc[-1])

  # compute and report test accuracy
  h1 = forward(test_x,params,'layer1',sigmoid)
  test_probs = forward(h1,params,'output',softmax)
  _, test_acc = compute_loss_and_acc(test_y, test_probs)
  print(f"Test accuracy(lr={learning_rate}): ", test_acc)
  return train_loss, valid_loss, train_acc, valid_acc

# call train_network three times with assigned learning rates
tuned_lr = 1e-1
results = {}
for lr in [10*tuned_lr, tuned_lr, tuned_lr/10]:
    results[lr] = train_network(learning_rate=lr)

for lr, (train_loss, valid_loss, train_acc, valid_acc) in results.items():
  # plot loss curves
  plt.plot(range(len(train_loss)), train_loss, label=f"training (lr={lr})")
  plt.plot(range(len(valid_loss)), valid_loss, label=f"validation (lr={lr})")
  plt.title(f"Loss Curves (lr={lr})")
  plt.xlabel("epoch")
  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=f"training (lr={lr})")
  plt.plot(range(len(valid_acc)), valid_acc, label=f"validation (lr={lr})")
  plt.title(f"Accuracy Curves (lr={lr})")
  plt.xlabel("epoch")
  plt.ylabel("accuracy")
  plt.xlim(0, len(train_acc)-1)
  plt.ylim(0, None)
  plt.legend()
  plt.grid()
  plt.show()
```

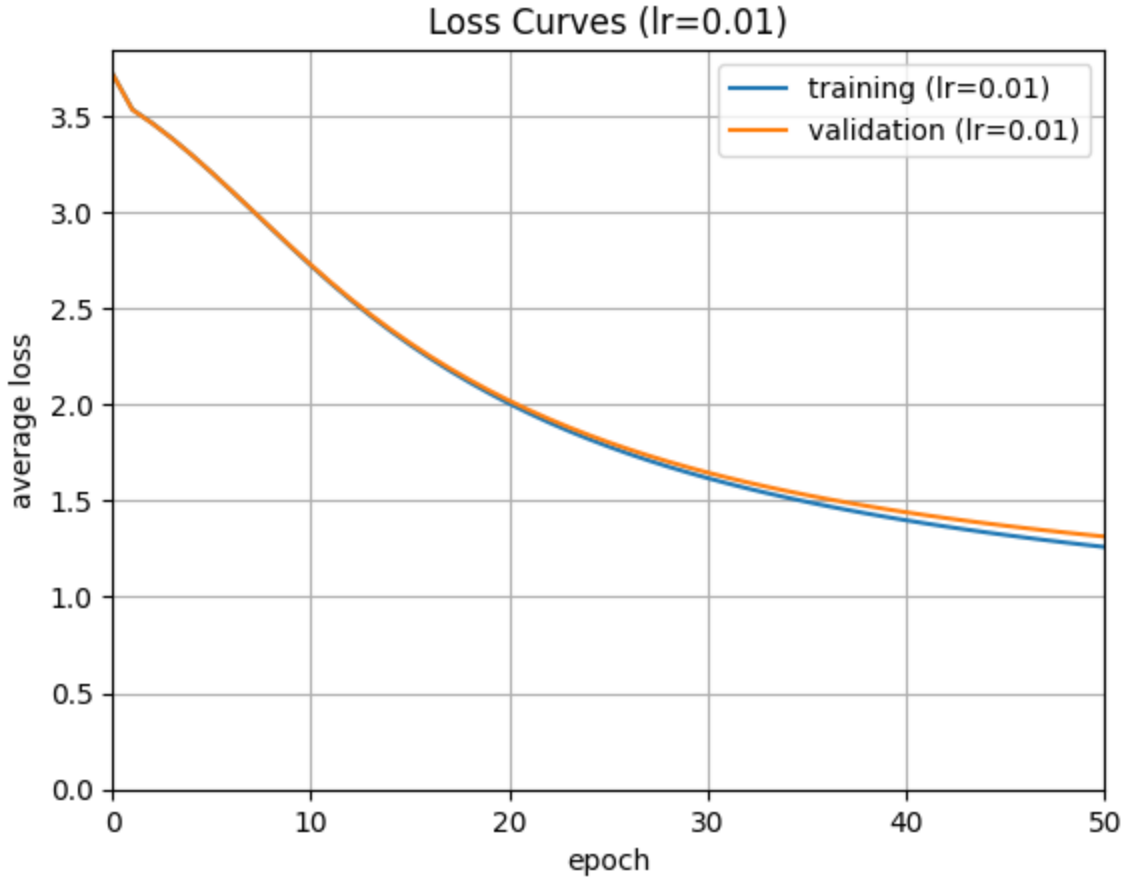
```
Validation accuracy(lr=1.0):  0.5980555555555556
Test accuracy(lr=1.0):  0.6061111111111112
Validation accuracy(lr=0.1):  0.7466666666666667
Test accuracy(lr=0.1):  0.7605555555555555
Validation accuracy(lr=0.01):  0.6619444444444444
Test accuracy(lr=0.01):  0.6627777777777778
```

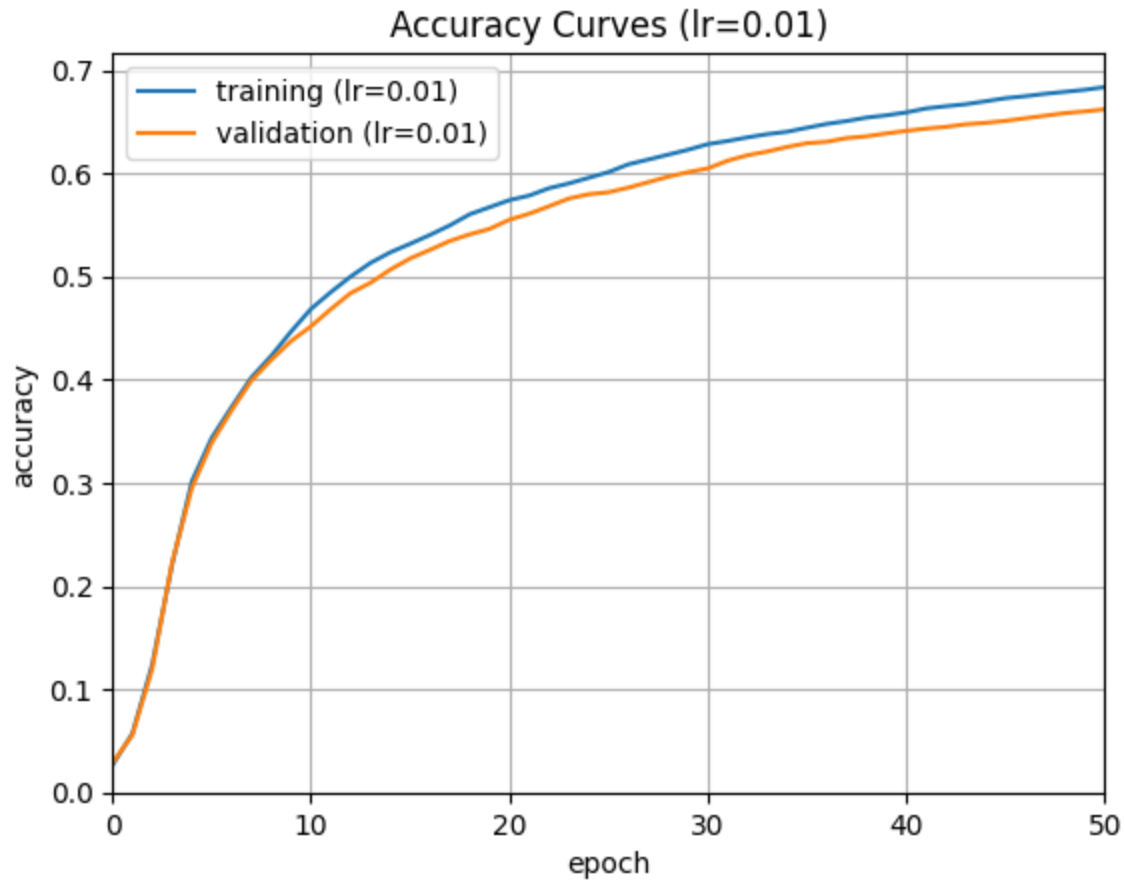

Loss Curves (lr=1.0)
training (lr=1.0)
validation (lr=1.0)
average loss
epoch
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
10
20
30
40
50


Accuracy Curves (lr=1.0)
training (lr=1.0)
validation (lr=1.0)
accuracy
epoch
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0
10
20
30
40
50


Loss Curves (lr=0.1)
training (lr=0.1)
validation (lr=0.1)
average loss
epoch
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
10
20
30
40
50


Accuracy Curves (lr=0.1)
training (lr=0.1)
validation (lr=0.1)
accuracy
epoch
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0
10
20
30
40
50





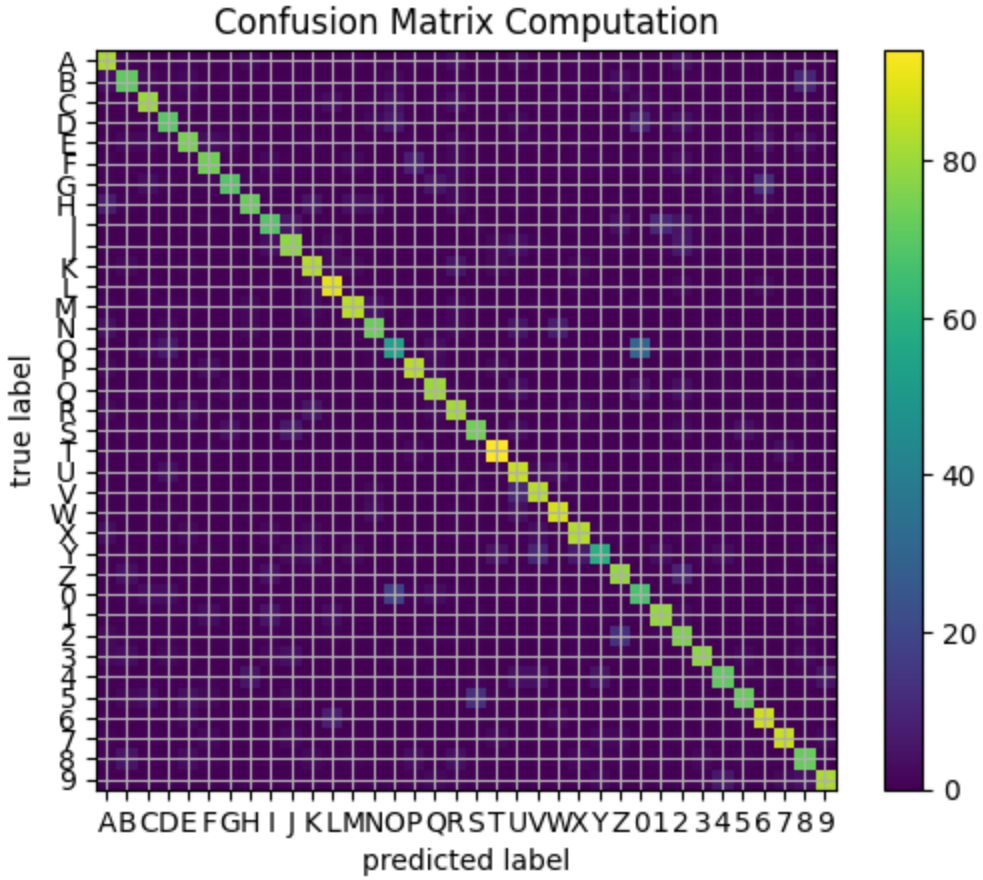
So the best accuracy happens when $\text{learning rate} = 0.1$ and is 0.752. When lr is set to ten times of the tuned lr which is 1, the loss curve and accuracy curve goes crazy during the epoch while the accuracy is decreased. When lr is set to one-tenth of the tuned lr which is 0.01, the validation acc and training acc

becomes more close both in loss curve and accuracy curve, but the final accuray is still not as good as the tuned learning rate.

Q3.4 (3 points)

Compute and 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.

In [25]:
```
############################## Q 3.4 ##############################
# confusion_matrix = np.zeros((train_y.shape[1],train_y.shape[1]))

# compute confusion matrix
#########################
##### your code here #####
#########################
valid_acc = None
h1 = forward(valid_x, params, "layer1", sigmoid)
probs = forward(h1, params, "output", softmax)
valid_loss, valid_acc = compute_loss_and_acc(valid_y, probs)
confusion_matrix = np.zeros((train_y.shape[1],train_y.shape[1]))
valid_pred_y = np.argmax(probs, axis = 1)
for i in range(valid_pred_y.shape[0]):
    pred = valid_pred_y[i]
    label = np.argmax(valid_y[i])
    confusion_matrix[label][pred] += 1


# visualize confusion matrix
import string
plt.imshow(confusion_matrix,interpolation='nearest')
plt.grid()
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.title("Confusion Matrix Computation")
plt.xlabel("predicted label")
plt.ylabel("true label")
plt.colorbar()
plt.show()
```



The most apparent confusing pair is (0, O), and others includes (5, S) and (2, Z), which is difficult to learn since even human eyes are unable to recognize them correctly everytime.

Q4 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.

Q4.1 Building the Autoencoder

Q4.1 (4 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. We will build an autoencoder with the layers listed below. Initialize the layers with the initialize_weights() function you wrote in Q2.1.2.

- 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)

In [38]:
```
# here we provide the relu activation and its derivative for you
from collections import Counter

def relu(x):
    return np.maximum(x,0)

def relu_deriv(x):
    return (x > 0).astype(float)

############################## Q 4.1 ##############################
params = Counter()

# initialize layers here
#########################
##### your code here #####
#########################
initialize_weights(1024, 32, params, name='layer1')
initialize_weights(32, 32, params, name='layer2')
initialize_weights(32, 32, params, name='layer3')
initialize_weights(32, 1024, params, name='output')
```

Q4.2 Training the Autoencoder

Q4.2.1 (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 momentum, populate the parameters dictionary with zero-initialized momentum accumulators M, one for each parameter. Then simply perform both update equations for every batch.

Q4.2.2 (6 points)

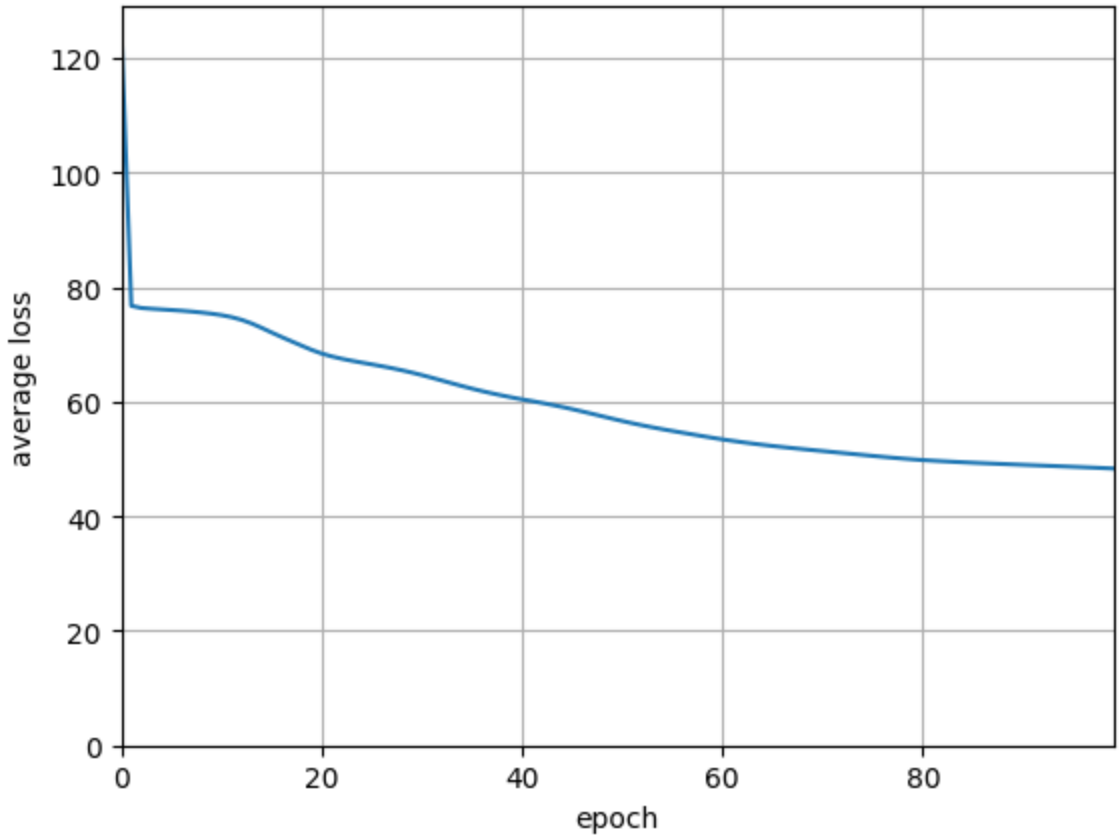
Using the provided default settings, train the network for 100 epochs. The loss function that you will use is the total squared error for the output image compared to the input image (they should be the same!). Plot the training loss curve. What do you observe?

In [39]:
```
########################## Q 4.2.1 & Q 4.2.2 ########################
# the NIST36 dataset
train_data = scipy.io.loadmat('/content/data/nist36_train.mat')
valid_data = scipy.io.loadmat('/content/data/nist36_valid.mat')

# we don't need labels now!
train_x = train_data['train_data']
valid_x = valid_data['valid_data']

max_iters = 100
# pick a batch size, learning rate
batch_size = 36
learning_rate =  3e-5
hidden_size = 32
lr_rate = 20
batches = get_random_batches(train_x,np.ones((train_x.shape[0],1)),batch_size)
batch_num = len(batches)

# should look like your previous training loops
losses = []
```

```
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 the total squared error, i.e. the sum of (x-y)^2
        # delta is the d/dx of (x-y)^2
        # to implement momentum
        #   just use 'M_'+name variables as momentum accumulators to keep a saved value
        #   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

        ############################
        ##### your code here #####
        ############################
        # forward
        h1 = forward(xb, params, 'layer1', relu)
        h2 = forward(h1, params, 'layer2', relu)
        h3 = forward(h2, params, 'layer3', relu)
        y_pred = forward(h3, params, 'output', sigmoid)

        # updated loss: sum squared error
        loss = np.sum((xb - y_pred) ** 2)
        total_loss += loss

        # updated delta: derivative of loss w.r.t the output
        delta = 2 * (y_pred - xb)

        # backward
        delta = backwards(delta, params, 'output', sigmoid_deriv)
        delta = backwards(delta, params, 'layer3', relu_deriv)
        delta = backwards(delta, params, 'layer2', relu_deriv)
        backwards(delta, params, 'layer1', relu_deriv)

        # update weights with momentum rate
        for layer in ['layer1', 'layer2', 'layer3', 'output']:
            params['M_W' + layer] = 0.9 * params['M_W' + layer] - learning_rate * params
            params['M_b' + layer] = 0.9 * params['M_b' + layer] - learning_rate * params
            params['W' + layer] += params['M_W' + layer]
            params['b' + layer] += params['M_b' + layer]


    losses.append(total_loss/train_x.shape[0])
    if itr % 2 == 0:
        print("itr: {:02d} \t loss: {:.2f}".format(itr,total_loss))
    if itr % lr_rate == lr_rate-1:
        learning_rate *= 0.9

# plot loss curve
plt.plot(range(len(losses)), losses)
plt.xlabel("epoch")
plt.ylabel("average loss")
plt.xlim(0, len(losses)-1)
plt.ylim(0, None)
plt.grid()
plt.show()
```

```
itr: 00         loss: 1352484.22
itr: 02         loss: 824974.94
itr: 04         loss: 822319.21
itr: 06         loss: 819836.45
itr: 08         loss: 816422.91
itr: 10         loss: 811441.74
itr: 12         loss: 802651.41
itr: 14         loss: 787559.56
itr: 16         loss: 770211.67
itr: 18         loss: 753858.96
```

```
itr: 20          loss: 739035.70
itr: 22          loss: 729224.45
itr: 24          loss: 721967.51
itr: 26          loss: 715152.23
itr: 28          loss: 707495.73
itr: 30          loss: 698392.25
itr: 32          loss: 688146.46
itr: 34          loss: 677774.03
itr: 36          loss: 668238.94
itr: 38          loss: 659942.96
itr: 40          loss: 652623.10
itr: 42          loss: 645838.41
itr: 44          loss: 638323.84
itr: 46          loss: 629829.92
itr: 48          loss: 620709.07
itr: 50          loss: 611786.63
itr: 52          loss: 603652.45
itr: 54          loss: 596408.15
itr: 56          loss: 589763.46
itr: 58          loss: 583339.68
itr: 60          loss: 577131.81
itr: 62          loss: 571753.06
itr: 64          loss: 566934.70
itr: 66          loss: 562685.00
itr: 68          loss: 558823.86
itr: 70          loss: 555138.62
itr: 72          loss: 551479.48
itr: 74          loss: 547817.46
itr: 76          loss: 544252.37
itr: 78          loss: 540977.74
itr: 80          loss: 538150.52
itr: 82          loss: 535937.09
itr: 84          loss: 534009.83
itr: 86          loss: 532287.41
itr: 88          loss: 530700.86
itr: 90          loss: 529190.66
itr: 92          loss: 527711.95
itr: 94          loss: 526223.08
itr: 96          loss: 524689.60
itr: 98          loss: 523081.95
```



At first, the average loss drops quickly. Then with the epoch goes, the average loss curve tend to convergent to a stable value.

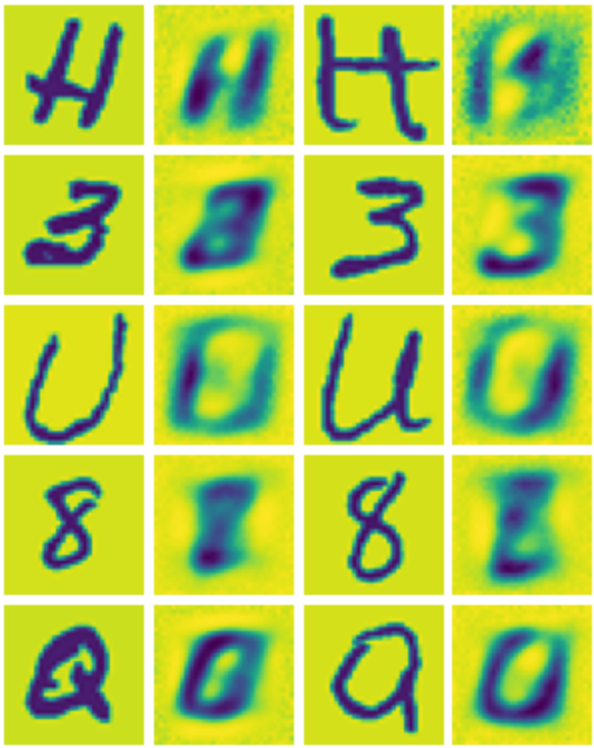
Q4.3 Evaluating the Autoencoder

Q4.3.1 (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 show 2 validation images and their reconstruction. What differences do you observe in the reconstructed validation images compared to the original ones?

In [41]:
```
############################## Q 4.3.1 ##############################
# choose 5 classes (change if you want)
visualize_labels = ["H", "3", "U", "8", "Q"]

# get 2 validation images from each label to visualize
visualize_x = np.zeros((2*len(visualize_labels), valid_x.shape[1]))
for i, label in enumerate(visualize_labels):
    idx = 26+int(label) if label.isnumeric() else string.ascii_lowercase.index(label.low
    choices = np.random.choice(np.arange(100*idx, 100*(idx+1)), 2, replace=False)
    visualize_x[2*i:2*i+2] = valid_x[choices]

# run visualize_x through your network
# using the forward() function you wrote in Q2.2.1
reconstructed_x = visualize_x
# TODO: name the output reconstructed_x
###########################
##### your code here #####
###########################
h1 = forward(visualize_x, params, 'layer1', relu)
```

```
h2 = forward(h1, params, 'layer2', relu)
h3 = forward(h2, params, 'layer3', relu)
reconstructed_x = forward(h3, params, 'output', sigmoid)

# visualize
fig = plt.figure()
plt.axis("off")
grid = ImageGrid(fig, 111, nrows_ncols=(len(visualize_labels), 4), axes_pad=0.05)
for i, ax in enumerate(grid):
    if i % 2 == 0:
        ax.imshow(visualize_x[i//2].reshape((32, 32)).T)
    else:
        ax.imshow(reconstructed_x[i//2].reshape((32, 32)).T)
    ax.set_axis_off()
plt.show()
```



The reconstructed images seems like the Gaussian blur effect of the original ones. There are observable detail loss in each class. While in 'H' and '8' much of the detail is lost, some key features are still remained. For the noise, the reconstructed images seems more clean than the original ones.

Q4.3.2 (5 points)

Let's evaluate the reconstruction quality using Peak Signal-to-noise Ratio (PSNR). PSNR is defined as

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

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).

In [49]:
```
############################## Q 4.3.2 ##############################
```

```
from skimage.metrics import peak_signal_noise_ratio
# evaluate PSNR
##########################
##### your code here #####
##########################
psnrs = []
for original, reconstructed in zip(visualize_x, reconstructed_x):
    # Calculate the PSNR
    psnr = peak_signal_noise_ratio(original, reconstructed, data_range=original.max() -
    psnrs.append(psnr)

average_psnr = np.mean(psnrs)
print(f"The average PSNR is: {average_psnr}")
```

```
The average PSNR is: 13.542136495460209
```

The average PSNR is: 13.542136495460209

Q5 (Extra Credit) Extract Text from Images

Run below code to download and put the unzipped data in '/content/images' folder. We have provided you with 01_list.jpg, 02_letters.jpg, 03_haiku.jpg and 04_deep.jpg to test your implementation on.

In [50]:
```
if not os.path.exists('/content/images'):
  os.mkdir('/content/images')
  !wget http://www.cs.cmu.edu/~lkeselma/16720a_data/images.zip -O /content/images/images
  !unzip "/content/images/images.zip" -d "/content/images"
  os.system("rm /content/images/images.zip")
```

```
--2024-04-11 02:37:27--  http://www.cs.cmu.edu/~lkeselma/16720a_data/images.zip
Resolving www.cs.cmu.edu (www.cs.cmu.edu)... 128.2.42.95
Connecting to www.cs.cmu.edu (www.cs.cmu.edu)|128.2.42.95|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 3248168 (3.1M) [application/zip]
Saving to: ‘/content/images/images.zip’

/content/images/ima 100%[===================>]   3.10M   408KB/s    in 15s

2024-04-11 02:37:43 (216 KB/s) - ‘/content/images/images.zip’ saved [3248168/3248168]

Archive:  /content/images/images.zip
warning:  stripped absolute path spec from /
mapname:  conversion of  failed
  inflating: /content/images/03_haiku.jpg
  inflating: /content/images/01_list.jpg
  inflating: /content/images/02_letters.jpg
  inflating: /content/images/04_deep.jpg
```

In [ ]:
```
ls /content/images
```

```
01_list.jpg*  02_letters.jpg*  03_haiku.jpg*  04_deep.jpg*
```

Q5.1 (Extra Credit) (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?

---

1. The method assumes that the characters do not overlap or touch each other. And there is a clear distinction between the character pixels and the background. It also assumes that the characters are perfectly separated so that the bounding boxes will not contain multiple characters.
2. Grouping letters assume that the text lines are aligned horizontally and spaced so that they can be separated into specific lines.

Q5.2 (Extra Credit) (10 points)

Implement the findLetters() 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 white and the background in black (consistent with the images in nist36). Hint: Since we read text left to right, top to bottom, we can use this to cluster the coordinates.

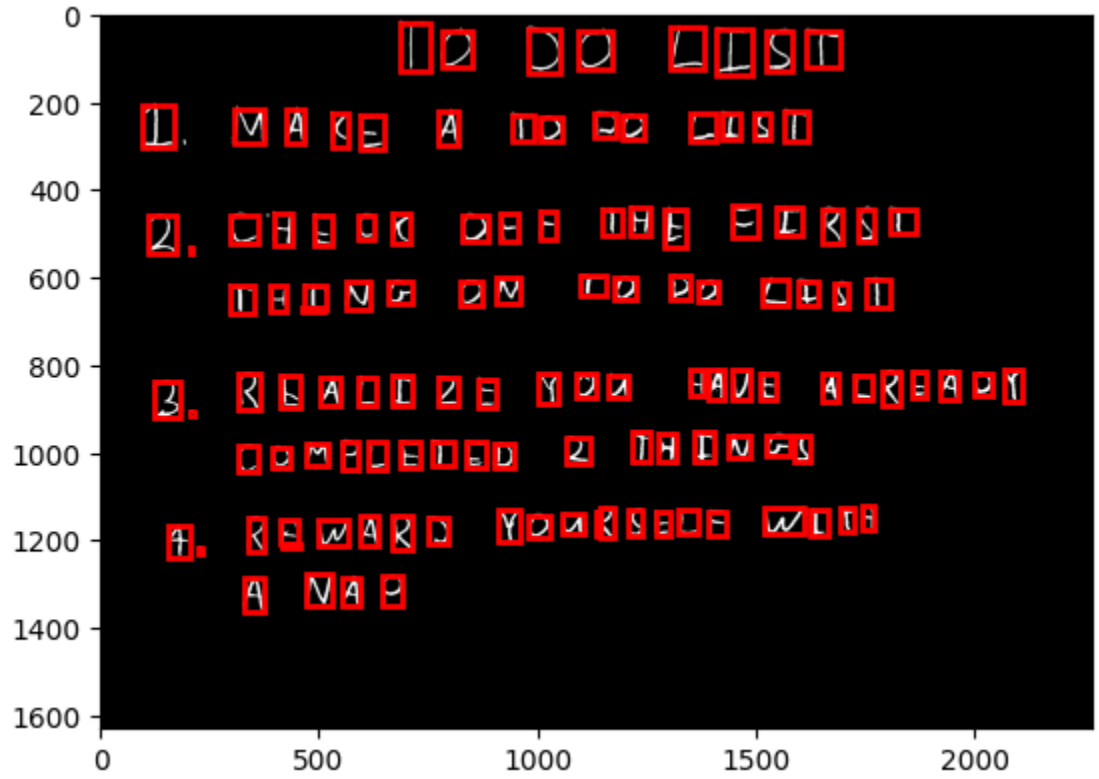
In [59]:
```
############################## Q 5.2 ################################
def findLetters(image):
    """
    takes a color image
    returns a list of bounding boxes and black_and_white image
    """
    bboxes = []
    bw = None
    # insert processing in here
    # one idea estimate noise -> denoise -> greyscale -> threshold -> morphology -> labe
    # this can be 10 to 15 lines of code using skimage functions

    ###########################
    ##### your code here #####
    ###########################
    # denoise
    denoise_image = skimage.restoration.denoise_bilateral(image, multichannel = True)
    # convert to grayscale
    gray_image = skimage.color.rgb2gray(image)
    # threshold
    threshold = skimage.filters.threshold_otsu(gray_image)
    bw = gray_image < threshold
    bw = skimage.morphology.binary_closing(bw, skimage.morphology.square(3))
    # label connected regions
    labeled_bw, num_features = skimage.measure.label(bw, return_num=True, connectivity=2
    # bounding boxes
    regions = skimage.measure.regionprops(labeled_bw)

    # get rid of small regions and create bounding boxes
    for region in regions:
        if region.area > 50:
            minr, minc, maxr, maxc = region.bbox
            bboxes.append([minr, minc, maxr, maxc])

    # bw image in range [0, 1]
    bw = bw.astype(float)

    return bboxes, bw
```

Q5.3 (Extra Credit) (3 points)

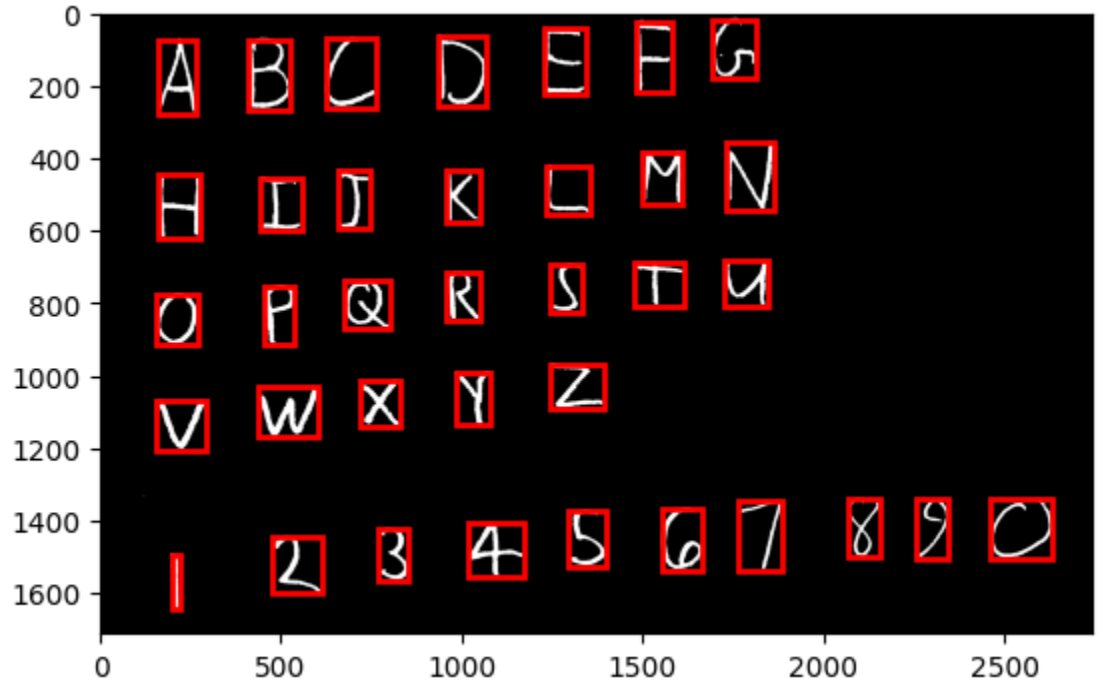
Using the provided code below, visualize all of the located boxes on top of the binary image to show the accuracy of your findLetters() function. Include all the provided sample images with the boxes.

In [60]:
```
############################### Q 5.3 ################################
# do not include any more libraries here!
# no opencv, no sklearn, etc!
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=UserWarning)

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

    print('\n' + img)
    plt.imshow(1-bw, cmap="Greys") # reverse the colors of the characters and the backgr
    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()
```

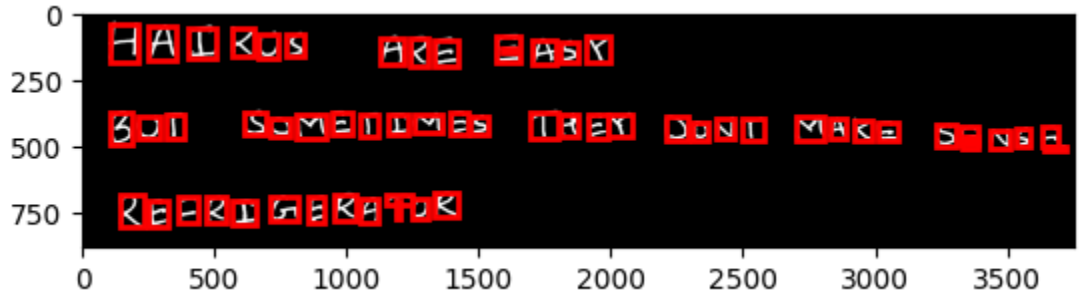
01_list.jpg



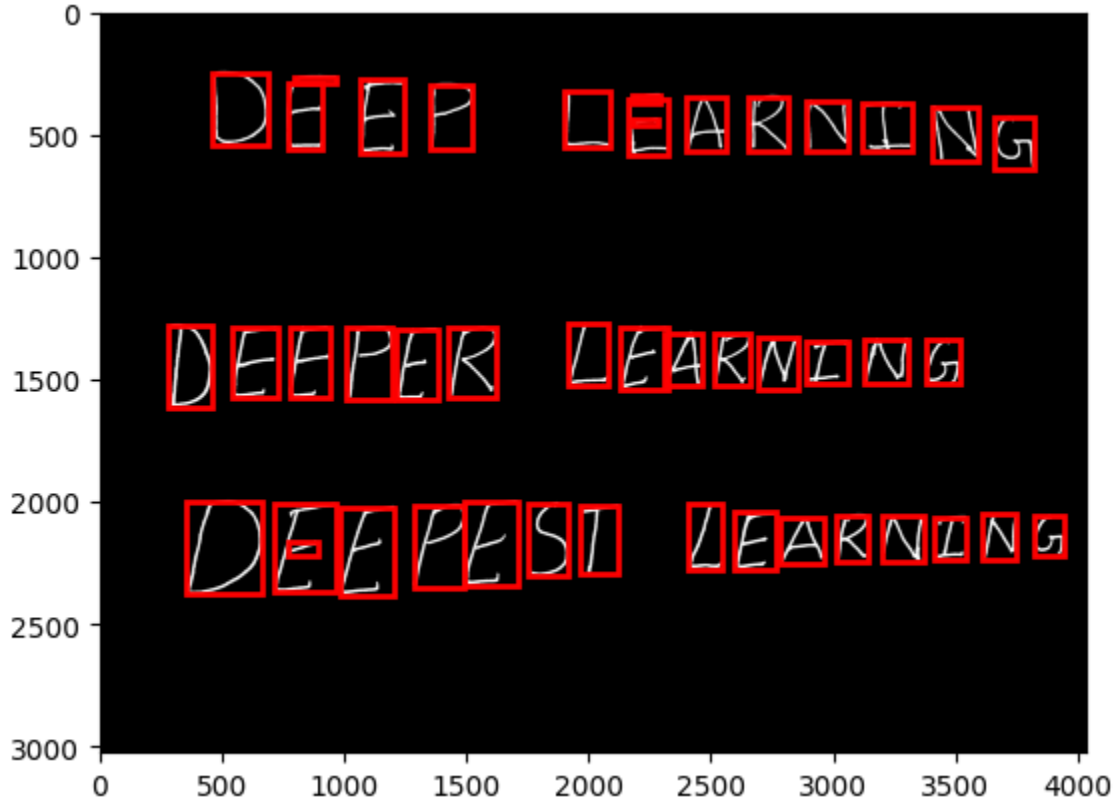
02_letters.jpg



03_haiku.jpg



04_deep.jpg



Q5.4 (Extra Credit) (8 points)

You will now load the image, find the character locations, classify each one with the network you trained in Q3.1, and return the text contained in the image. Be sure you try to make your detected images look like the images from the training set. Visualize them and act accordingly. If you find that your classifier performs poorly, consider dilation under skimage morphology to make the letters thicker.

Your solution is correct if you can correctly detect most of the letters and classify approximately 70\% of the letters in each of the sample images.

Run your code on all the provided sample images in '/content/images'. Show the extracted text. It is fine if your code ignores spaces, but if so, please provide a written answer with manually added spaces.

In [63]:
```
############################### Q 5.4 ################################
for imgno, img in enumerate(sorted(os.listdir('/content/images'))):
    im1 = skimage.img_as_float(skimage.io.imread(os.path.join('/content/images',img)))
    bboxes, bw = findLetters(im1)
    print('\n' + img)

    # find the rows using..RANSAC, counting, clustering, etc.
    ##########################
    ##### your code here #####
    ##########################
    # cluster bounding boxes into rows
    points = [(bbox, ((bbox[0] + bbox[2]) // 2, (bbox[1] + bbox[3]) // 2)) for bbox in b
    rows = []
    for bbox, center in sorted(points, key=lambda x: x[1]):
        match_found = False
        for row in rows:
            if abs(center[0] - np.mean([pt[1][0] for pt in row])) < np.mean([pt[0][2] -
                row.append((bbox, center))
                match_found = True
                break
        if not match_found:
            rows.append([(bbox, center)])
    rows = [sorted(row, key=lambda x: x[1][1]) for row in rows]


    # crop the bounding boxes
    # note.. before you flatten, transpose the image (that's how the dataset is!)
    # consider doing a square crop, and even using np.pad() to get your images looking m
    ##########################
    ##### your code here #####
    ##########################
    # use bounding boxes to create a dataset
    dataset = []
    for row in rows:
        dataset_row = []
        for bbox, _ in row:
            minr, minc, maxr, maxc = bbox
            # Crop and pad the image to make it square
            image = bw[minr:maxr, minc:maxc]
            H, W = image.shape
            padding = ((max(H, W) - min(H, W)) // 2 + 1, max(H, W) // 20)
            pad_width = (padding, padding) if H > W else ((padding[1], padding[1]), (pad
            image = np.pad(image, pad_width, "constant", constant_values=1)
            image = skimage.transform.resize(image, (32, 32))
            image = skimage.morphology.erosion(image, skimage.morphology.square(3))
            dataset_row.append(image.T.flatten())
        dataset.append(np.vstack(dataset_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
    params = pickle.load(open('/content/q3_weights.pickle','rb'))
    ############################
    ##### your code here #####
    ############################
    for row in dataset:
        out = forward(row, params, 'layer1', sigmoid)
        probs = forward(out, params, 'output', softmax)
        preds = np.argmax(probs, axis=1)
        print(' '.join(letters[pred] for pred in preds))
    print("-" * 50)
```

```
01_list.jpg
D B B B B N B N
B Q D D B B N N Q R Q N D N
B V R D D D D R B B B B D B Q D D W
U B R U Q B X O Q H Q X Q N D N
B T D B B B B N Q B B 8 S D B Q B B D Q D M D
B B 6 B B Q N Q N R D 3 B 1 R D
B L D B 2 B B B B B B 8 D D D D Q D B N B 3
D Q B B
--------------------------------------------------

02_letters.jpg
D B B B B D B
B N B Q B Q B
B D B B B N G
R B Q B Q
D N D H B B B D D R
--------------------------------------------------

03_haiku.jpg
B B M Q B B B D D B 2 Y N Q
D R U N A B N U U B N J U D D 2 B A 3 U B Q B B Q B S Q B D S 2
B N B B V R D Q B U B R D
--------------------------------------------------

04_deep.jpg
B
B
B Q B B B Q B B B B N B B
D B B B B B B B B B B N Q B
B Q B B B B D D Q B B B B B B B
--------------------------------------------------
```

YOUR ANSWER HERE... (if your code ignores spaces)
