# 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**: Rayna Hata, Ethan Holand

---

# 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 $c = -\max x_i$)

---

$$\begin{aligned} softmax(x+c)_i &= \frac{e^{x_i} e^c}{\sum_j e^{x_j} e^c} \\ &= \frac{e^{x_i}}{\sum_j e^{x_j}} \\ &= softmax(x)_i \end{aligned}$$

Using $c = -max\ x_i$ will regularize values being exponentiate to negative, which will limit the range of output to between 0 and 1. This will improve numerical stability when compared to inserting raw values.

---

## 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?

---

$$softmax(x)_i \in (0,1)$$
$$\sum softmax(x)_i = 1$$

---

### Q1.2.2 (1 point)

One could say that

$$\text{softmax takes an arbitrary real valued vector } x \text{ and turns it into a } \textcolor{red}{\backslash\text{rule}}3cm0.1mm$$

---

probability distribution

---

### Q1.2.3 (1 point)

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

---

$s_i$: turn all elements strictly positive

$S$: find the sum of the transformed values

$\frac{1}{S} s_i$: normalizes all elements by limiting range to $(0, 1)$

---

## Q1.3 (3 points)

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

---

each layer:

$$y = wx + b$$

n-layers:

$$\begin{aligned} y_n &= w_n y_{n-1} + b_n \\ y_n &= w_n(w_{n-1}y_{n-2} + b_{n-1}) + b_n \\ y_n &= w_n w_{n-1} y_{n-2} + w_n b_{n-1} + b_n \\ y_n &= \left(\prod_i^n w_i\right) y_1 + \sum_i^n w_i b_{i-1} + b_n \\ y_n &= W y_1 + B \end{aligned}$$

---

## 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).

---

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

---

## 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.

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

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

## 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)?

gradients approach zero because the derivative is only between 0 and 0.25, so back propagation over multiple layers of small outputs will lead to even smaller gradient.

### 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 $\tanh$?

$tanh$ outputs between $(-1, 1)$ , and sigmoid outputs range from $(0, 1)$. We might prefer it beasue it's symetric and centered at zero, providing more stable gradients and decreases output sensitivity.

### Q1.6.3 (1 point)

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

---

$tanh$ has a maximum gradient of 1, which doesn't reduce the gradient as aggressively over multiple layers.

---

### 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 $\sigma(x)$.

---

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

---

# 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.

```python
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?

---

Initializing the network to zero will produce ill defined gradient and can prevent backpropagation.

---

### 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 [2]:
```
################################ 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 o
f examples, in_size]
    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 singleton dimension
    the output of this layer should be in size [number of examples, out_size]
    """
    W, b = None, None

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

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

---

We want to normalize depending on layer size to prevent extreme gradients. Thye need to have a reasonable distribution so they will be less affected by repeated forward an back propagation.

---

## 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 [3]: ############################### 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 dim
ensions]
    """
    res = None

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

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

```
In [4]: ############################### 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 dimens
ions]
    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 [5]:
```
################################ 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 #####
    ###########################

    res = np.exp(x - np.max(x, axis=1, keepdims=True))
    res /= np.sum(res, axis=1, keepdims=True)

    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 [6]:
```
################################ 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, numbe
r of classes]
    probs -- the probabilities output by the classifier, i.e. f(x), which is a
numpy array of size [number of examples, number of classes]
    """
    loss, acc = None, None

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

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


    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 [7]: ############################## 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), in [number of examples, number of output dimensions]
    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 #####
    ##########################

    delta = delta * activation_deriv(post_act)
    grad_W = np.dot(X.T, delta)
    grad_b = np.sum(delta, axis=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 [8]: def linear(x):
    # Define a linear activation, which can be used to construct a "no activat
ion" layer
    return x

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

In [9]:

```
# 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)] + [3 for _ in range(10)])
# turn to one-hot encoding, this implies that the labels y is in [number of examples, number of classes]
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 of neurons)
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))
```

```python
# Q 2.2.2
# implement softmax
# a forward pass on the second layer (the output layer), with softmax so that
the outputs are class probabilities
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 tw
o sizes should be the same
        print('Q 2.3',name,v.shape, params[name].shape)
```

```
data shape: (40, 2) labels shape: (40, 4)
Q 2.1.2: 0.0, 0.09
Q 2.1.2: 0.0, 0.06
Q 2.2.1: sigmoid outputs should be zero and one  0.0 1.0
Q 2.2.2: 0.060371132089046924 0.9999999999999998 1.0000000000000002
Q 2.2.3 loss: 76.5828414453866, 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,)

/tmp/ipykernel_16927/2312559038.py:13: RuntimeWarning: overflow encountered i
n 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 [10]:
```
############################## 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 #####
    ##########################

    indices = np.arange(x.shape[0])
    np.random.shuffle(indices)
    x = x[indices]
    y = y[indices]
    for i in range(0, x.shape[0], batch_size):
        batches.append((x[i:i+batch_size], y[i:i+batch_size]))

    return batches
```

In [11]:
```
# 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 [12]:
```python
############################## 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')

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

```
itr: 00          loss: 69.76     acc : 0.25
itr: 100         loss: 40.89     acc : 0.58
itr: 200         loss: 33.61     acc : 0.82
itr: 300         loss: 29.54     acc : 0.85
itr: 400         loss: 26.89     acc : 0.87
```

# 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 [13]:
```
if not os.path.exists('./content/data'):
  !mkdir ./content
  !mkdir ./content/data
  !wget http://www.cs.cmu.edu/~lkeselma/16720a_data/data.zip -O ./content/dat
a/data.zip
  !unzip "./content/data/data.zip" -d "./content/data"
  os.system("rm ./content/data/data.zip")
```

```
--2024-04-10 02:06:42--  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   615KB/s    in 7m 29s

2024-04-10 02:14:11 (471 KB/s) - ‘./content/data/data.zip’ saved [216305627/2
16305627]

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 [14]:
```
!ls ./content/data
```

```
nist26_model_60iters.mat  nist26_train.mat  nist36_test.mat   nist36_valid.ma
t
nist26_test.mat           nist26_valid.mat  nist36_train.mat
```

## Q3.1 (5 points)

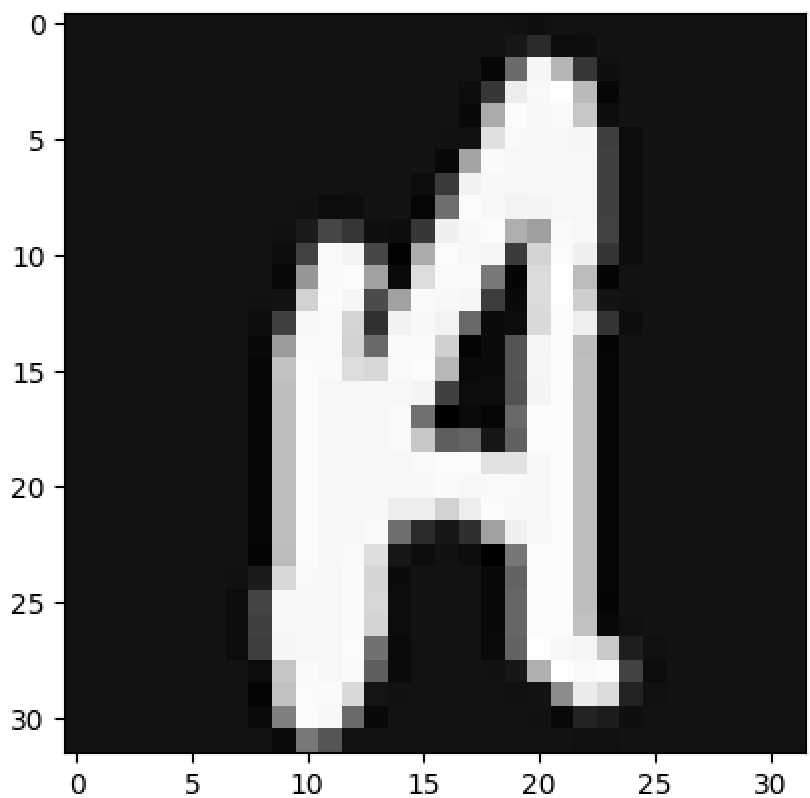
Train a network from scratch. Use a single hidden layer with 64 hidden units, and train for at least 50 epochs. The script will generate two plots:

(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 [15]:
```python
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 [16]:

```
############################### Q 3.1 ###############################
np.random.seed(69420)
max_iters = 50
# pick a batch size, learning rate
batch_size = None
learning_rate = None
hidden_size = 64
##########################
##### your code here #####
##########################
batch_size = 35
learning_rate = 2e-3

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

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

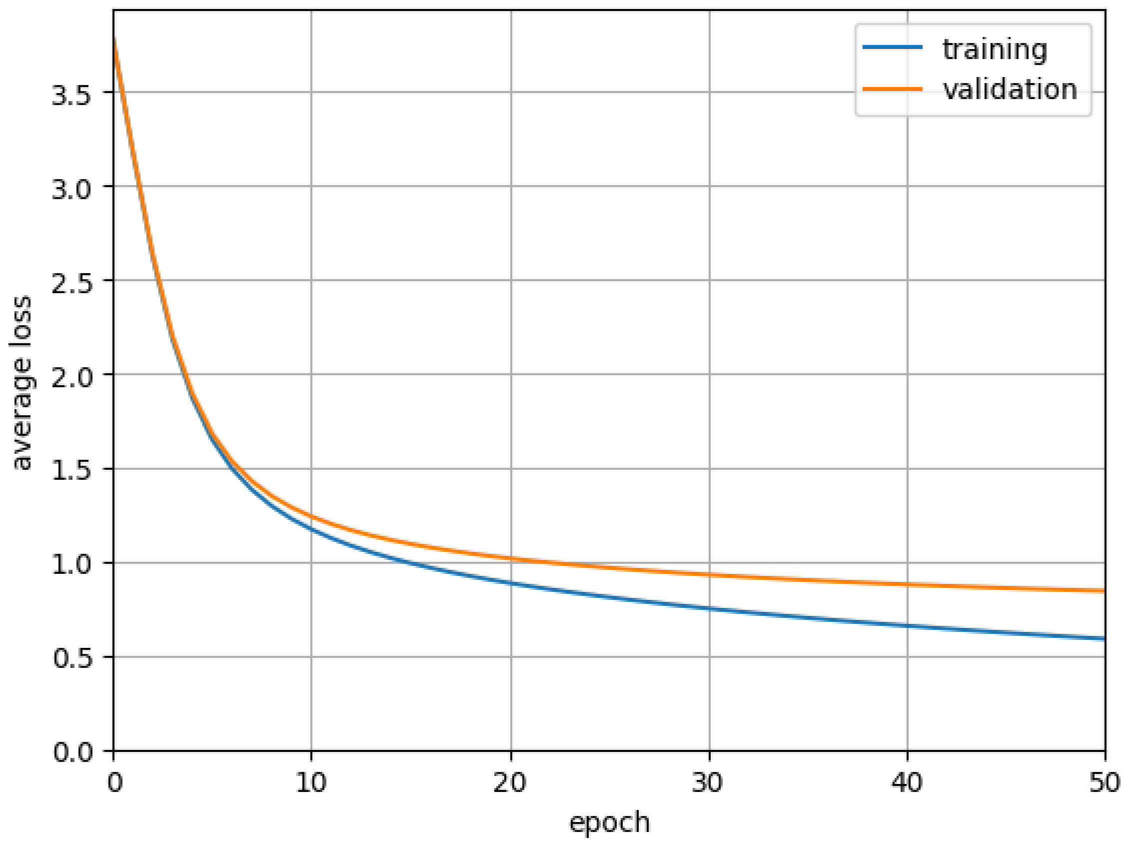
```
        # backward
        delta1 = probs - yb
        delta2 = backwards(delta1, params, 'output', linear_deriv)
        backwards(delta2, params, 'layer1')

        # 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_los
s,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)
```

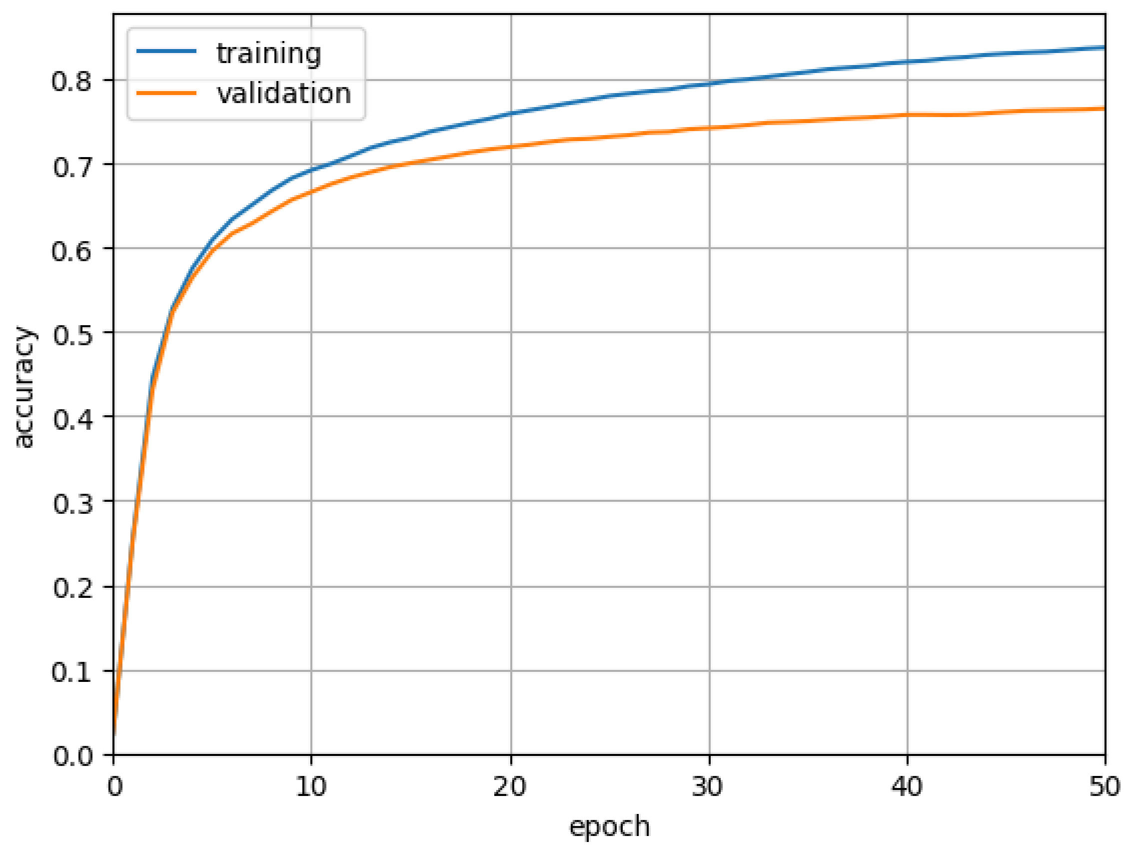
```
itr: 00   loss: 37019.81   acc: 0.12
itr: 02   loss: 25931.06   acc: 0.48
itr: 04   loss: 19113.00   acc: 0.59
itr: 06   loss: 15745.02   acc: 0.64
itr: 08   loss: 13878.42   acc: 0.67
itr: 10   loss: 12673.30   acc: 0.69
itr: 12   loss: 11805.91   acc: 0.71
itr: 14   loss: 11134.70   acc: 0.73
itr: 16   loss: 10587.62   acc: 0.74
itr: 18   loss: 10124.17   acc: 0.75
itr: 20   loss: 9719.97   acc: 0.76
itr: 22   loss: 9359.57   acc: 0.77
itr: 24   loss: 9032.87   acc: 0.77
itr: 26   loss: 8733.10   acc: 0.78
itr: 28   loss: 8455.61   acc: 0.79
itr: 30   loss: 8197.08   acc: 0.80
itr: 32   loss: 7954.96   acc: 0.80
itr: 34   loss: 7727.27   acc: 0.81
itr: 36   loss: 7512.40   acc: 0.81
itr: 38   loss: 7308.99   acc: 0.82
itr: 40   loss: 7115.94   acc: 0.82
itr: 42   loss: 6932.27   acc: 0.83
itr: 44   loss: 6757.15   acc: 0.83
itr: 46   loss: 6589.89   acc: 0.84
itr: 48   loss: 6429.86   acc: 0.84
Validation accuracy:  0.7652777777777777
Test accuracy:  0.7688888888888888
```

In [17]:

```
# 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 [18]:
```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()
```


training
validation
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
average loss
0
10
20
30
40
50
epoch


training
validation
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
accuracy
0
10
20
30
40
50
epoch

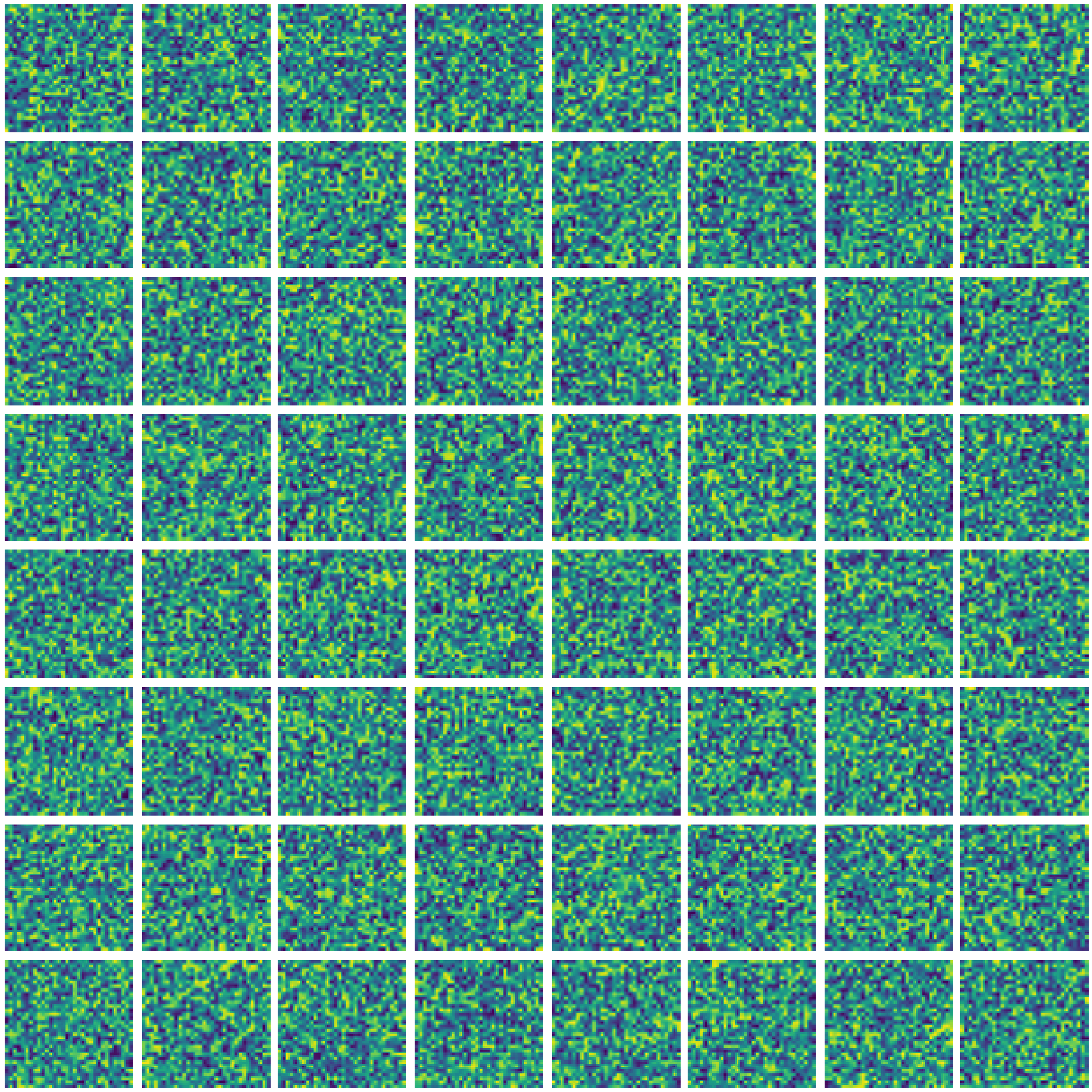
## Q3.2 (3 points)

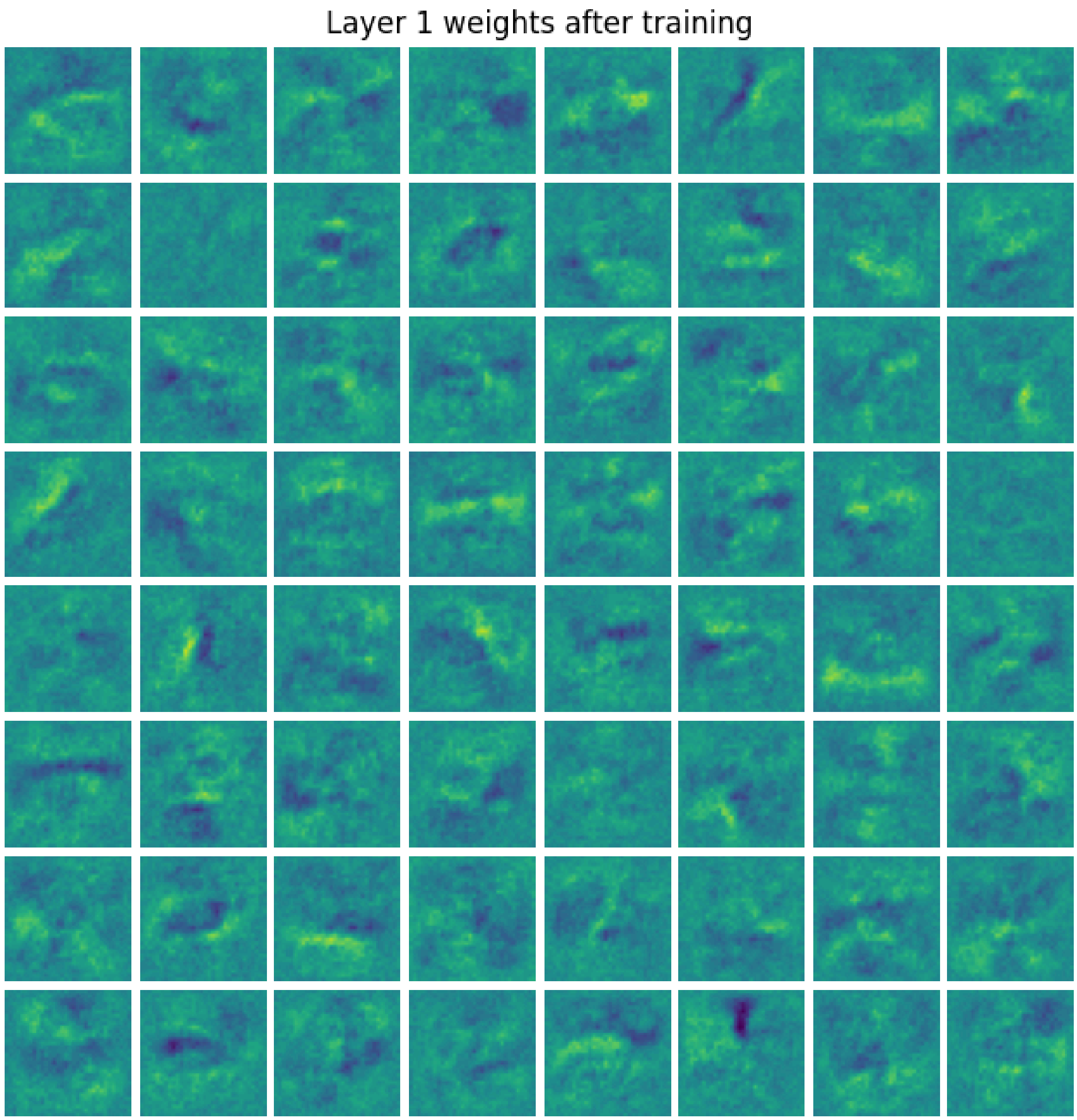
The provided code will visualize the first layer weights as 64 32x32 images, both immediately after initialization and after full training. Generate both visualizations. Comment on the learned weights and compare them to the initialized weights. Do you notice any patterns?

In [19]:

```
############################## 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()
```

Layer 1 weights after initialization






The initialized weights look very random but the learned weights seem to highlight specific portions of the image more than others, forming a somewhat smooth-ish surface. A parttern is that a very bright spot is usually neighbor to a really dark spot, which means these can act like filters where it's looking for a very specific feature in a patch of the image.

## 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 [20]:
```
############################## Q 3.3 ##############################
##########################
##### your code here #####
##########################

def trainer(train_x,
            train_y,
            valid_x,
            valid_y,
            batch_size,
            learning_rate,
            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:
            # training loop can be exactly the same as q2!
            ##########################
            ##### your code here #####
            ##########################
            h1 = forward(xb, params, 'layer1', sigmoid)
            probs = forward(h1, params, 'output', softmax)

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

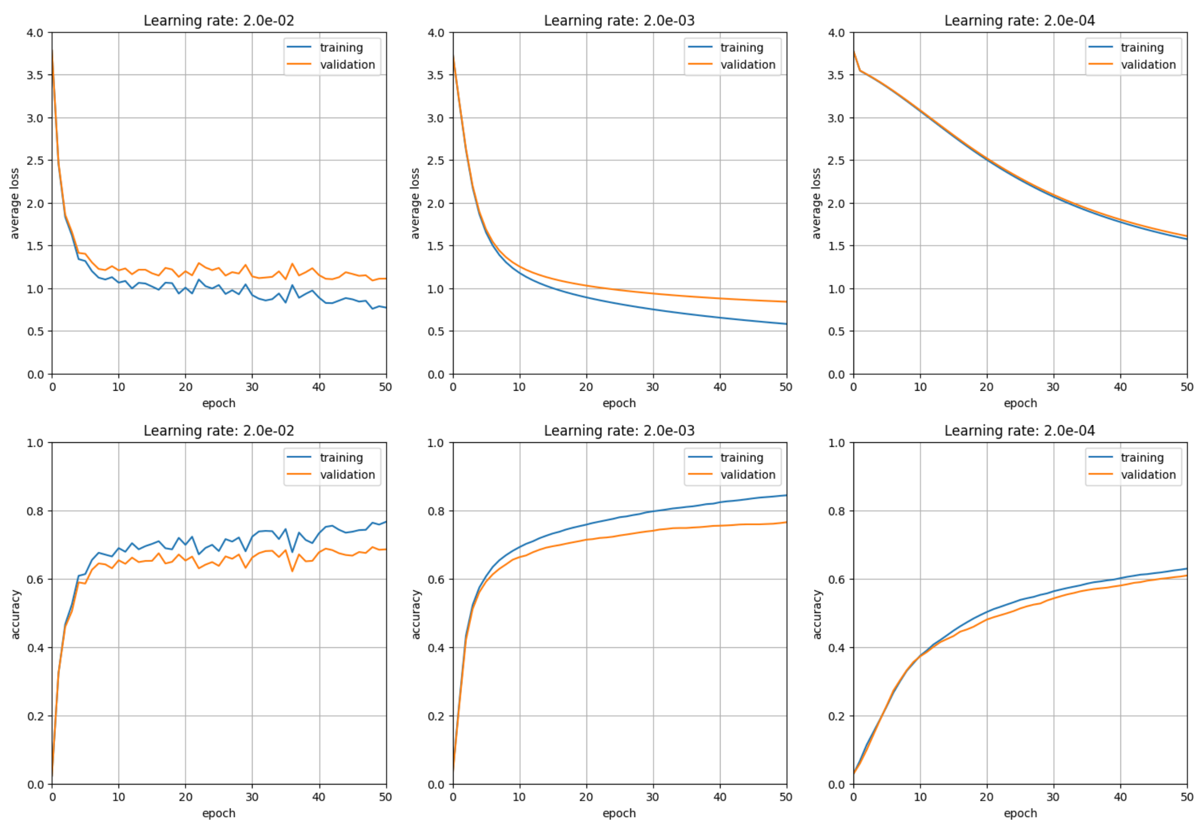
```
            # backward
            delta1 = probs - yb
            delta2 = backwards(delta1, params, 'output', linear_deriv)
            backwards(delta2, params, 'layer1')

            # 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)

    return train_loss, valid_loss, train_acc, valid_acc
```

In [21]:
```
np.random.seed(69420)

learning_rates = [10*learning_rate, learning_rate, learning_rate/10]

train_loss = np.zeros((len(learning_rates), max_iters+1))
valid_loss = train_loss.copy()
train_acc = train_loss.copy()
valid_acc = train_loss.copy()

for i, lr in enumerate(learning_rates):
    train_loss[i], valid_loss[i], train_acc[i], valid_acc[i] = trainer(train_
x, train_y, valid_x, valid_y, batch_size, lr)
```

```
itr: 00   loss: 34986.49   acc: 0.11
itr: 02   loss: 20414.31   acc: 0.45
itr: 04   loss: 15707.15   acc: 0.57
itr: 06   loss: 14308.37   acc: 0.62
itr: 08   loss: 13577.29   acc: 0.63
itr: 10   loss: 13191.65   acc: 0.64
itr: 12   loss: 12027.27   acc: 0.67
itr: 14   loss: 12416.65   acc: 0.66
itr: 16   loss: 11768.08   acc: 0.68
itr: 18   loss: 12346.57   acc: 0.66
itr: 20   loss: 12164.17   acc: 0.66
itr: 22   loss: 11686.97   acc: 0.68
itr: 24   loss: 11515.25   acc: 0.68
itr: 26   loss: 11405.37   acc: 0.68
itr: 28   loss: 10721.77   acc: 0.70
itr: 30   loss: 10749.74   acc: 0.70
itr: 32   loss: 10360.27   acc: 0.71
itr: 34   loss: 9813.09   acc: 0.73
itr: 36   loss: 10413.21   acc: 0.71
itr: 38   loss: 10450.59   acc: 0.71
itr: 40   loss: 9940.38   acc: 0.72
itr: 42   loss: 9548.62   acc: 0.73
itr: 44   loss: 9558.15   acc: 0.73
itr: 46   loss: 9221.33   acc: 0.74
itr: 48   loss: 8812.76   acc: 0.75
Validation accuracy:  0.6855555555555556
Test accuracy:  0.7144444444444444
itr: 00   loss: 37047.91   acc: 0.11
itr: 02   loss: 25919.20   acc: 0.48
itr: 04   loss: 19138.94   acc: 0.59
itr: 06   loss: 15800.88   acc: 0.64
itr: 08   loss: 13932.83   acc: 0.67
itr: 10   loss: 12718.23   acc: 0.69
itr: 12   loss: 11836.06   acc: 0.71
itr: 14   loss: 11146.46   acc: 0.72
itr: 16   loss: 10580.37   acc: 0.74
itr: 18   loss: 10099.45   acc: 0.75
itr: 20   loss: 9680.36   acc: 0.76
itr: 22   loss: 9308.01   acc: 0.77
itr: 24   loss: 8972.16   acc: 0.78
itr: 26   loss: 8665.64   acc: 0.79
itr: 28   loss: 8383.24   acc: 0.79
itr: 30   loss: 8121.12   acc: 0.80
itr: 32   loss: 7876.32   acc: 0.81
itr: 34   loss: 7646.53   acc: 0.81
itr: 36   loss: 7429.89   acc: 0.82
itr: 38   loss: 7224.91   acc: 0.82
itr: 40   loss: 7030.35   acc: 0.83
itr: 42   loss: 6845.17   acc: 0.83
itr: 44   loss: 6668.51   acc: 0.84
itr: 46   loss: 6499.63   acc: 0.84
itr: 48   loss: 6337.88   acc: 0.85
Validation accuracy:  0.765
Test accuracy:  0.7677777777777778
itr: 00   loss: 38845.70   acc: 0.04
itr: 02   loss: 37592.28   acc: 0.15
itr: 04   loss: 36547.10   acc: 0.23
```

```
itr: 06   loss: 35392.44   acc: 0.30
itr: 08   loss: 34161.62   acc: 0.36
itr: 10   loss: 32890.45   acc: 0.39
itr: 12   loss: 31605.81   acc: 0.42
itr: 14   loss: 30328.16   acc: 0.45
itr: 16   loss: 29084.67   acc: 0.47
itr: 18   loss: 27899.14   acc: 0.49
itr: 20   loss: 26778.91   acc: 0.51
itr: 22   loss: 25726.28   acc: 0.53
itr: 24   loss: 24741.65   acc: 0.54
itr: 26   loss: 23824.13   acc: 0.55
itr: 28   loss: 22971.82   acc: 0.56
itr: 30   loss: 22182.01   acc: 0.57
itr: 32   loss: 21451.44   acc: 0.58
itr: 34   loss: 20776.48   acc: 0.59
itr: 36   loss: 20153.29   acc: 0.59
itr: 38   loss: 19577.94   acc: 0.60
itr: 40   loss: 19046.53   acc: 0.61
itr: 42   loss: 18555.28   acc: 0.61
itr: 44   loss: 18100.61   acc: 0.62
itr: 46   loss: 17679.15   acc: 0.62
itr: 48   loss: 17287.79   acc: 0.63
Validation accuracy:  0.6091666666666666
Test accuracy:  0.615
```

In [22]:
```
# plot the results in a 2x3 tiled grid
fig, axs = plt.subplots(2, 3, figsize=(18, 12))
for i, ax in enumerate(axs.flatten()):
    if i < 3:
        ax.plot(range(len(train_loss[i])), train_loss[i], label="training")
        ax.plot(range(len(valid_loss[i])), valid_loss[i], label="validation")
        ax.set_title("Learning rate: {:.1e}".format(learning_rates[i]))
        ax.set_xlabel("epoch")
        ax.set_ylabel("average loss")
        ax.set_xlim(0, len(train_loss[i])-1)
        ax.set_ylim(0, 4)
        ax.legend()
        ax.grid()
    else:
        ax.plot(range(len(train_acc[i-3])), train_acc[i-3], label="training")
        ax.plot(range(len(valid_acc[i-3])), valid_acc[i-3], label="validatio
n")
        ax.set_title("Learning rate: {:.1e}".format(learning_rates[i-3]))
        ax.set_xlabel("epoch")
        ax.set_ylabel("accuracy")
        ax.set_xlim(0, len(train_acc[i-3])-1)
        ax.set_ylim(0, 1)
        ax.legend()
        ax.grid()
```



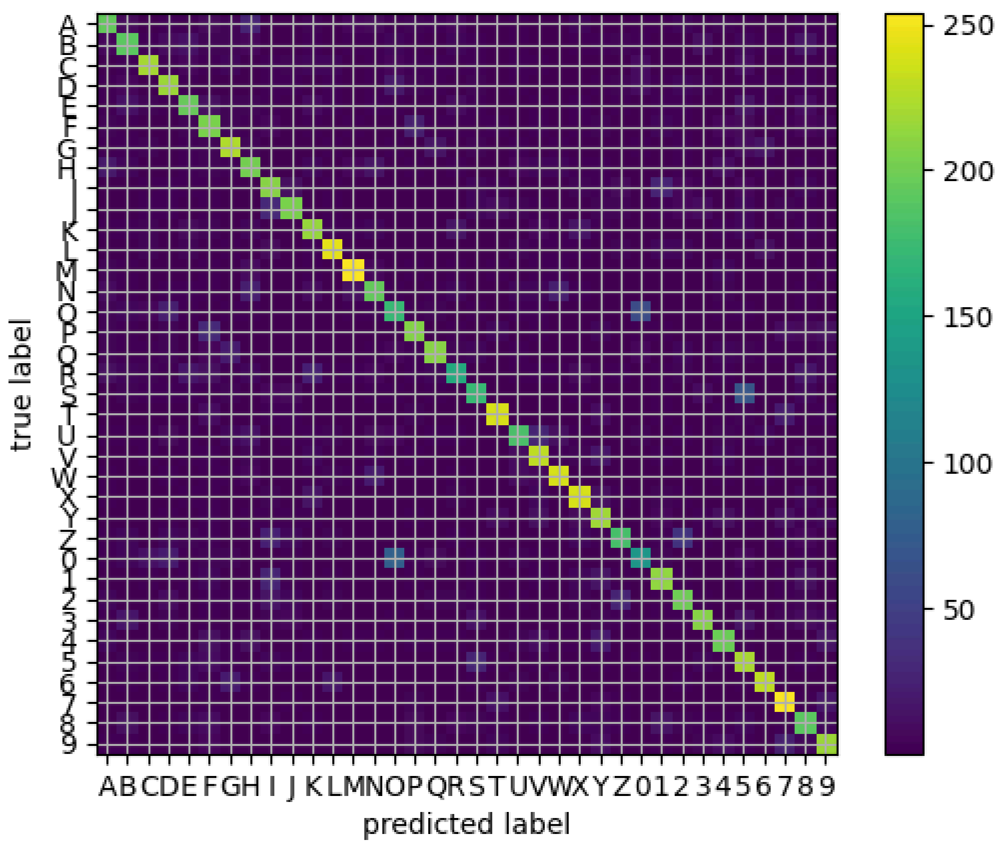
Higher learning rate leads to less smooth evolution of both accuracy and loss. Lower learning rate lead to very slow improvement in performance. Best final accuracy is 76.7%

## 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 [23]:

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

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

for i in range(0, train_x.shape[0], batch_size):
    h1 = forward(train_x[i:i+batch_size], params, 'layer1', sigmoid)
    probs = forward(h1, params, 'output', softmax)
    confusion_matrix += np.dot(train_y[i:i+batch_size].T, probs)



# 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.xlabel("predicted label")
plt.ylabel("true label")
plt.colorbar()
plt.show()
```



top few:

- O and 0
- S and 5
- Z and 2
- P and F

# 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 [24]:
```
# 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 #####
##########################


# - 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 ou
tput for us)

initialize_weights(1024, 32, params, "layer1")
initialize_weights(32, 32, params, "layer2")
initialize_weights(32, 32, params, "layer3")
initialize_weights(32, 1024, params, "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.

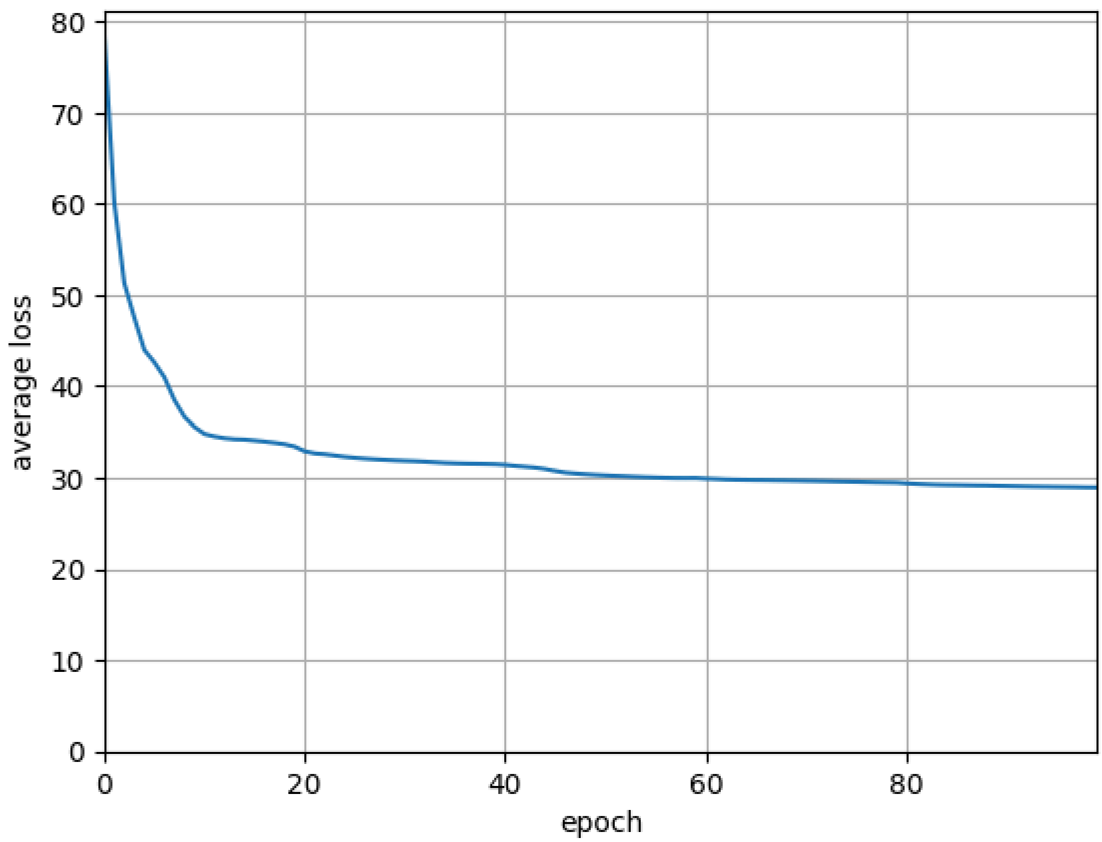
### Q4.2.2 (6 points)

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 [25]:
```
########################### 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)

for k in params.copy().keys():
    if 'W' or 'b' in k:
        params['M_' + k] = np.zeros_like(params[k])

# should look like your previous training loops
losses = []
for itr in range(max_iters):
    tot_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 sa
ved value over steps
        #   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 condi
tions

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

        h1 = forward(xb, params, 'layer1', relu)
        h2 = forward(h1, params, 'layer2', relu)
        h3 = forward(h2, params, 'layer3', relu)
        probs = forward(h3, params, 'output', sigmoid)

        tot_loss += np.sum((xb - probs)**2)

        delta = 2 * (probs - xb)
        gradient = backwards(delta, params, 'output', sigmoid_deriv)
        gradient = backwards(gradient, params, 'layer3', relu_deriv)
        gradient = backwards(gradient, params, 'layer2', relu_deriv)
        gradient = backwards(gradient, params, 'layer1', relu_deriv)

        for k in params.keys():
            if 'grad' in k:
```

```
                name = k.split('_')[1]
                params['M_' + name] = 0.9 * params['M_' + name] - learning_rat
e * params[k]
                params[name] += params['M_' + name]


    losses.append(tot_loss/train_x.shape[0])
    if itr % 2 == 0:
        print("itr: {:02d} \t loss: {:.2f}".format(itr,tot_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: 849344.70
itr: 02          loss: 554203.86
itr: 04          loss: 475016.63
itr: 06          loss: 442698.63
itr: 08          loss: 396081.16
itr: 10          loss: 375863.83
itr: 12          loss: 370864.69
itr: 14          loss: 369264.77
itr: 16          loss: 366882.12
itr: 18          loss: 363780.79
itr: 20          loss: 355511.89
itr: 22          loss: 351984.35
itr: 24          loss: 348918.50
itr: 26          loss: 346782.61
itr: 28          loss: 345347.63
itr: 30          loss: 344192.86
itr: 32          loss: 343191.94
itr: 34          loss: 341706.59
itr: 36          loss: 340931.57
itr: 38          loss: 340542.58
itr: 40          loss: 339485.28
itr: 42          loss: 337243.18
itr: 44          loss: 334564.71
itr: 46          loss: 330167.57
itr: 48          loss: 328237.38
itr: 50          loss: 326859.73
itr: 52          loss: 325820.00
itr: 54          loss: 324787.81
itr: 56          loss: 324055.44
itr: 58          loss: 323783.53
itr: 60          loss: 323135.47
itr: 62          loss: 322126.38
itr: 64          loss: 321636.06
itr: 66          loss: 321365.58
itr: 68          loss: 320991.98
itr: 70          loss: 320563.79
itr: 72          loss: 320098.89
itr: 74          loss: 319628.26
itr: 76          loss: 319107.66
itr: 78          loss: 318604.68
itr: 80          loss: 317542.95
itr: 82          loss: 316328.45
itr: 84          loss: 315622.60
itr: 86          loss: 315256.02
itr: 88          loss: 314939.16
itr: 90          loss: 314381.31
itr: 92          loss: 313880.40
itr: 94          loss: 313511.56
itr: 96          loss: 313170.00
itr: 98          loss: 312755.93
```




---

The loss function seems to plateau after 80 iters.

---

## 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 [26]:

```
############################### 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.lower())
    choices = np.random.choice(np.arange(100*idx, 100*(idx+1)), 2, replace=Fal
se)
    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(reconstructed_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()
```

Reconstructed images los fine details on the edge. Some even have extra detail which are not present in the orginal image (8 left). Some are missing features (both Q's look like O's).

## 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 $\text{MAX}_I$ is the maximum possible pixel value of the image, and $\text{MSE}$ (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 [28]:
```
############################### Q 4.3.2 ###############################
from skimage.metrics import peak_signal_noise_ratio
# evaluate PSNR
##########################
##### your code here #####
##########################

psnr_vals = []
for i in range(0, valid_x.shape[0], batch_size):
    h1 = forward(valid_x[i:i+batch_size], params, 'layer1', relu)
    h2 = forward(h1, params, 'layer2', relu)
    h3 = forward(h2, params, 'layer3', relu)
    probs = forward(h3, params, 'output', sigmoid)
    psnr_vals.append(peak_signal_noise_ratio(valid_x[i:i+batch_size], probs))

psnr_avg = np.mean(psnr_vals)
print("Average PSNR: {:.5f}".format(psnr_avg))
```

```
Average PSNR: 15.47045
```

I got a PSNR val of about 15.47.

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

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?

YOUR ANSWER HERE...

## 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 [ ]:

```
############################## 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 -> morpholo
gy -> label -> skip small boxes
    # this can be 10 to 15 lines of code using skimage functions

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


    return bboxes, bw
```

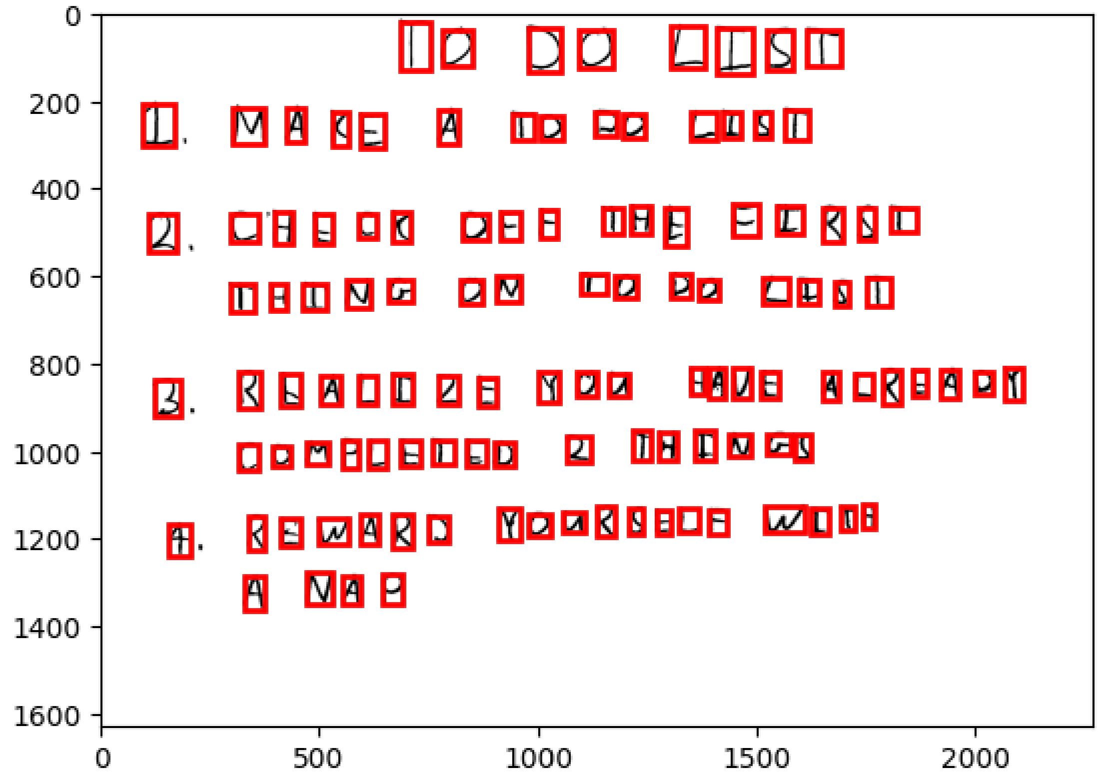
## Q5.3 (Extra Credit) (3 points)

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.

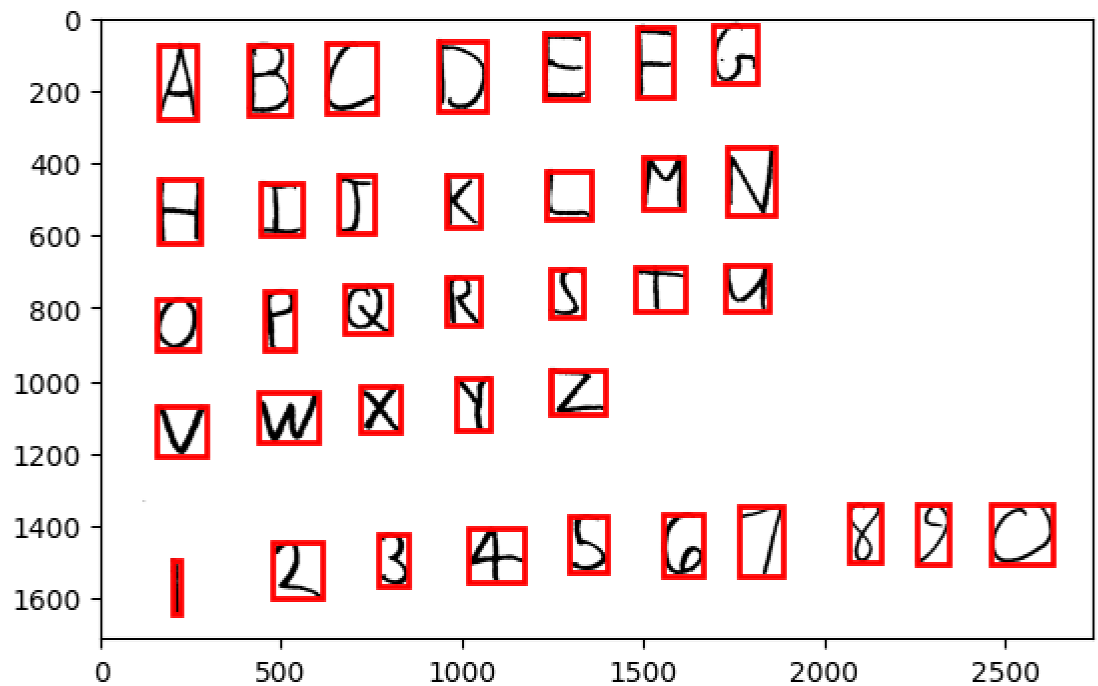
In [ ]:

```
############################## 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/image
s',img)))
    bboxes, bw = findLetters(im1)

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

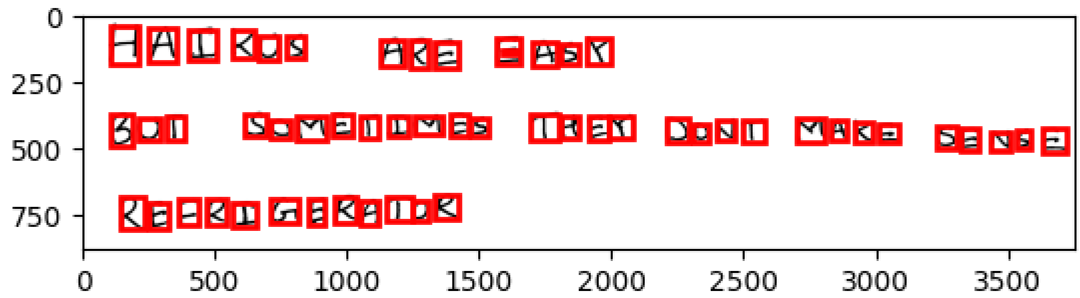
01_list.jpg



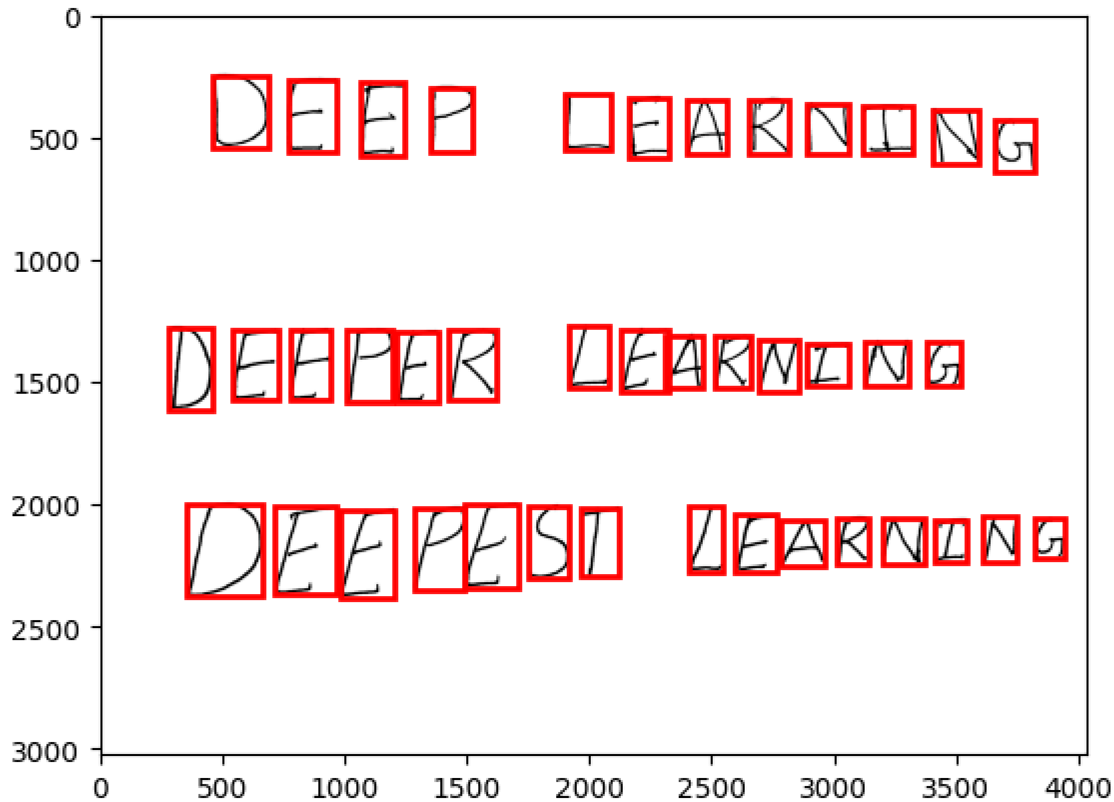

02_letters.jpg




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 [ ]:
```
############################### 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/image
s',img)))
    bboxes, bw = findLetters(im1)
    print('\n' + img)

    # find the rows using..RANSAC, counting, clustering, etc.
    ##########################
    ##### your code here #####
    ##########################



    # crop the bounding boxes
    # note.. before you flatten, transpose the image (that's how the dataset i
s!)
    # consider doing a square crop, and even using np.pad() to get your images
looking more like the dataset
    ##########################
    ##### your code here #####
    ##########################



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

---

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

---