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 $c = -\max x_i$)

Replacing x with x+c in the softmax equation, we get

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

Taking e^c common,

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

which gives us,

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

Using $c = -\max x_i$ is a good idea to prevent overflow issues. Raising the exponent to the powers of large numbers could potentially lead to extremely high values. When translating using $c = -\max x_i$ will ensure that the largest number possible would be equal to 0 (and $e^0 = 1$), this eliminating the potential for overflow issues.

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. The range of each element is [0,1].
2. The sum over all elements is 1. This is because, as part of the softmax function, all the elements are normalized by the sum over all elements.

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. We raise the element by the exponent. This ensures that each element is non-negative, (minumum is 0 when raised to the power of negative infinity). It also helps in capturing the difference between elements.
2. We store the sum of all the elements in S. This is to normalize the inputs
3. We divide each element by the sum S. This ensure that the vector is now interpretable as a probability distribution

---

Q1.3 (3 points)

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

Let's consider a 2-layer network

Without a non-linear activation function, the output of the hidden layer is:

$$Z = X \cdot W_{(1)} + b_{(1)}$$

and final output is:

$$Y = Z \cdot W_{(2)} + b_{(2)}$$

Expanding Z,

$$Y = (X \cdot W_{(1)} + b_{(1)}) \cdot W_{(2)} + b_{(2)}$$
$$Y = X \cdot W_{(1)}W_{(2)} + b_{(1)}W_{(2)} + b_{(2)}$$

This is equivalient to linear regression with weight = $W_{(1)}W_{(2)}$ and bias $b_{(1)}W_{(2)} + b_{(2)}$

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).

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

Calculating $\frac{d\sigma(x)}{dx}$,

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

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.

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

Attached at the end of the PDF

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, $\frac{d}{dx} f(g(x)) = f'(g(x))g'(x)$.

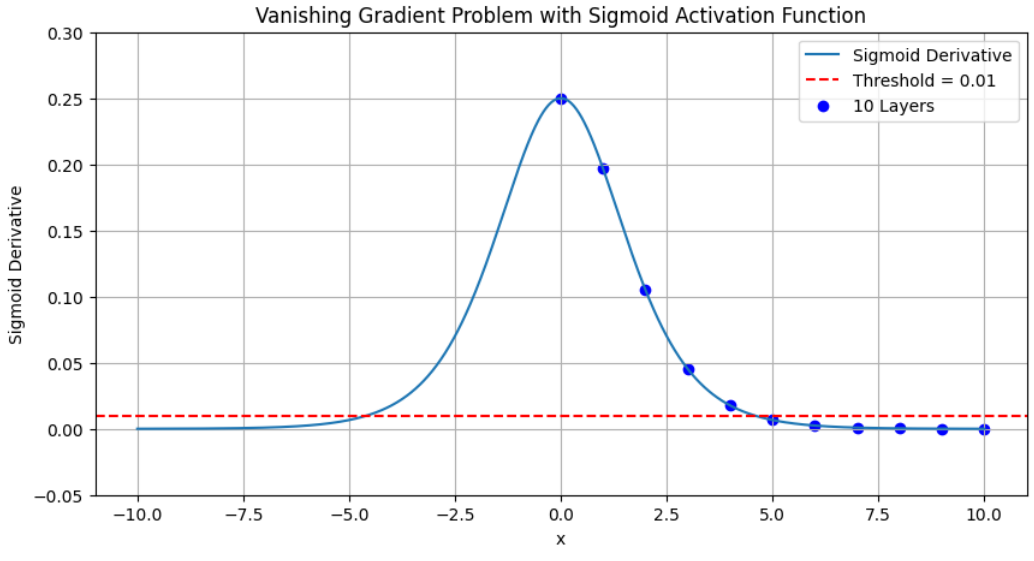
Q1.6.1 (1 point)

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)?

In the case of the sigmoid activation function, the derivative is given by $\sigma(x)(1-\sigma(x))$.

The range for this derivative is (0, 0.25]. If the neural network has many layers, the backpropagation of this gradient for the early layers will be a multiplication of many numbers less than 0.25. This results in a number that is very close to zero, effectively not propagating any of the learned change to the earlier layers. This is how the sigmoid activation function can lead to a "vanishing gradient".

```
import numpy as np
import matplotlib.pyplot as plt

# Sigmoid function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Derivative of sigmoid function
def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))

# Define x range
x = np.linspace(-10, 10, 1000)

# Plot sigmoid derivative
plt.figure(figsize=(10, 5))
plt.plot(x, sigmoid_derivative(x), label='Sigmoid Derivative')

# Set a threshold for the derivative
threshold = 0.01

# Plot threshold line
plt.axhline(y=threshold, color='r', linestyle='--', label='Threshold = 0.01')

# Plot vanishing gradient for 10-layer network
plt.scatter([0], [sigmoid_derivative(0)], color='b', label='10 Layers')

# Calculate vanishing gradient for 10-layer network
for i in range(1, 11):
    plt.scatter([i], [sigmoid_derivative(i)], color='b')

# Set plot attributes
plt.xlabel('x')
plt.ylabel('Sigmoid Derivative')
plt.title('Vanishing Gradient Problem with Sigmoid Activation Function')
plt.legend()
plt.grid(True)
plt.ylim(-0.05, 0.3)

# Show plot
plt.show()
```



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 $\tanh$?

The output range for sigmoid(x) is [0,1]

The output range for tanh(x) is [-1, 1]

We might prefer tanh as the outputs of this function are zero-centered. The outputs in sigmoid are not zero-centered. The gradient of the tanh is also larger than the gradient of the sigmoid function, making it less prone to the vanishing gradient problem. Thus, tanh is often preferred over sigmoid

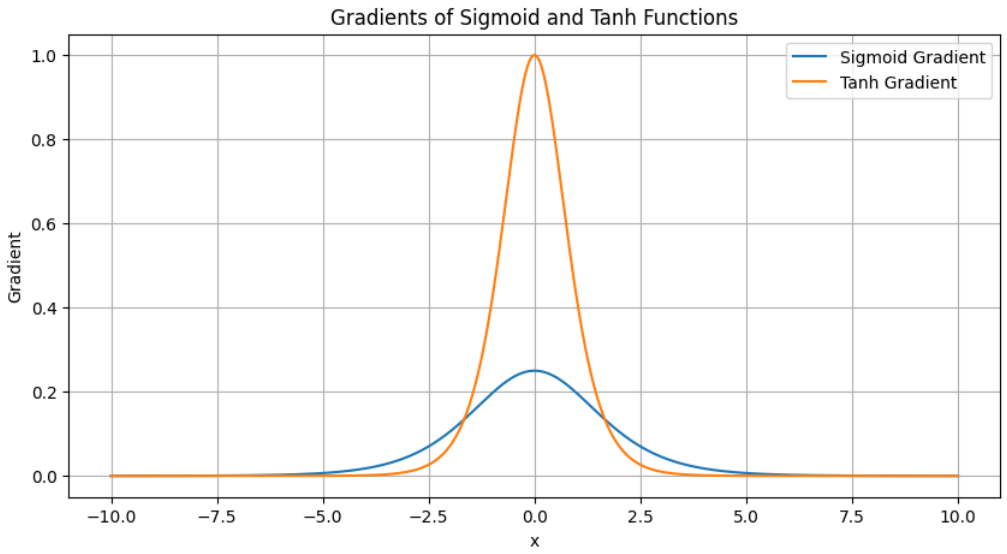
---

Q1.6.3 (1 point)

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

We see that the range of the tanh derivative seems to be higher than the values for sigmoid. This can help with the vanishing gradient problem. As the number of layers increases, ths gradient for sigmoid may be lost as the values are very close to zero. Since the tanh values are slightly higher, it can help mitigate this vanishing gradient problem as the gradient would not approach zero as fast as it would with the sigmoid function.

```
def tanh(x):
    return np.tanh(x)

# Derivative of tanh function
def tanh_derivative(x):
    return 1 - np.tanh(x)**2

x = np.linspace(-10, 10, 1000)

sigmoid_grad = sigmoid_derivative(x)
tanh_grad = tanh_derivative(x)


plt.figure(figsize=(10, 5))
plt.plot(x, sigmoid_grad, label='Sigmoid Gradient')
plt.plot(x, tanh_grad, label='Tanh Gradient')
plt.xlabel('x')
plt.ylabel('Gradient')
plt.title('Gradients of Sigmoid and Tanh Functions')
plt.legend()
plt.grid(True)
plt.show()
```



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 $\sigma(x)$.

Attached at the end of the PDF

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.

```
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 a network of weights with all zeros may not be a good idea as it would lead to identical gradients for all the layers. The output of each layer would essentially be the same, effectively making it a linear model.

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]).

```
############################## 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, 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

    ##########################

    variance = 2 / (in_size + out_size)
    W = np.random.uniform(-np.sqrt(6 * variance), np.sqrt(6 * variance), (in_size, out_size))
    b = np.random.uniform(-np.sqrt(6 * variance), np.sqrt(6 * variance), (out_size)) #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])?

Xavier initalization ensures that the variance of activations across the layers is constant. This helps with the problem of vanishing/exploding gradients. In Figure 6, we see that the activation values for normalized initializations remains the same across the 5 layers as compared to the activation values for standard initialization.

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 $y = \sigma(XW + b)$.

```
############################## 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

    ##########################

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

    ##########################


    return res


############################## 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]

    ##########################
    pre_act = 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.

```
############################## 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

    ##########################
    x_max = np.max(x, axis=1, keepdims=True)
    x -= x_max

    # Compute the softmax function for each row
    exp_x = np.exp(x)
    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 $f : \mathbb{R}^D \rightarrow [0,1]^C$ is the classifier which outputs the probabilities for the classes. The $\log$ is the natural $\log$.

```
############################## 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 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

    loss = -np.sum(y * np.log(probs))

    # Compute accuracy
    predictions = np.argmax(probs, axis=1)
    true_labels = np.argmax(y, axis=1)
    correct = np.sum(predictions == true_labels)
    total = y.shape[0]
    acc = correct / total

    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.

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

    ##########################

    # 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.

```
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)


# 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))

# 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 two 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: -0.018938271364461287, 0.13
Q 2.1.2: 0.27828467719648486, 0.13
Q 2.2.1: sigmoid outputs should be zero and one  0.0 1.0
Q 2.2.2: 0.03677479354577559 0.9999999999999998 1.0
Q 2.2.3 loss: 71.4070390135011, 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-5-cc11817861bf>:11: 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.

```
############################ 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 = []

    ########################
    num_examples = x.shape[0]

    indices = np.arange(num_examples)
    np.random.shuffle(indices)

    for start_idx in range(0, num_examples, batch_size):
        end_idx = min(start_idx + batch_size, num_examples)
        batch_indices = indices[start_idx:end_idx]
        batch_x = x[batch_indices]
        batch_y = y[batch_indices]
        batches.append((batch_x, batch_y))

    ########################

    return batches


# 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
```

```
############################ 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:
        # 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
        for key in params:
            if 'grad_' in key:
                params[key.replace('grad_', '')] -= learning_rate * params[key]

    avg_acc /= len(batches)

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

```
itr: 00          loss: 69.59     acc : 0.25
itr: 100         loss: 44.99     acc : 0.60
itr: 200         loss: 37.74     acc : 0.72
itr: 300         loss: 33.41     acc : 0.72
itr: 400         loss: 30.30     acc : 0.80
```

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.

```
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")
```

```
--2024-04-10 20:55:02--  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  3.34MB/s    in 63s

2024-04-10 20:56:05 (3.28 MB/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
```

```
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)

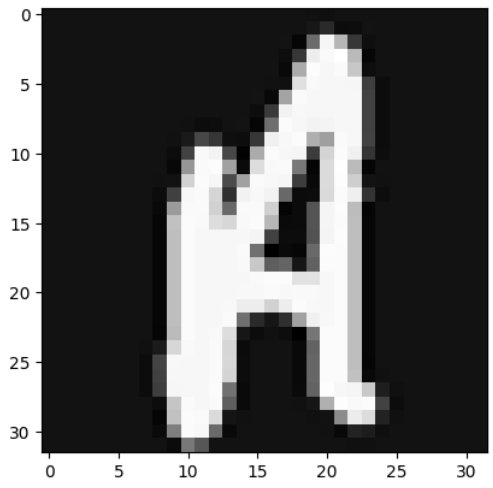
Train a network from scratch. Use a single hidden layer with 64 hidden units, and train for at least 50 epochs. The script will generate two plots:

(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.

```
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
```

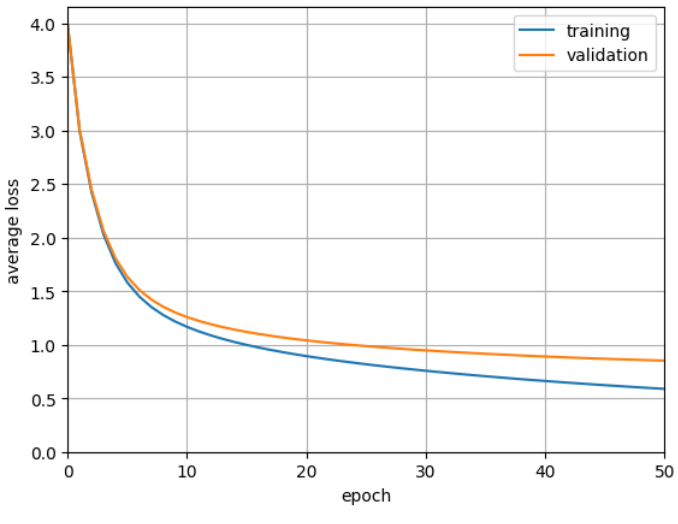


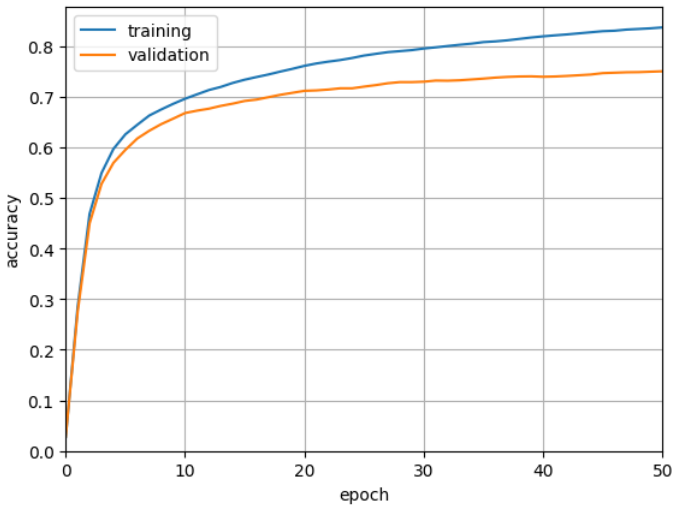
```
############################## Q 3.1 ##############################
max_iters = 50
# pick a batch size, learning rate
batch_size = 64
learning_rate =2e-3


hidden_size = 64
########################
np.random.seed(8)
########################

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
        for key in params:
            if 'grad_' in key:
                params[key.replace('grad_', '')] -= learning_rate * params[key]

    avg_acc /= len(batches)

    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: 36115.51   acc: 0.14
itr: 02   loss: 24137.43   acc: 0.49
itr: 04   loss: 18181.53   acc: 0.60
itr: 06   loss: 15258.47   acc: 0.65
itr: 08   loss: 13564.29   acc: 0.68
itr: 10   loss: 12431.29   acc: 0.70
itr: 12   loss: 11593.93   acc: 0.72
itr: 14   loss: 10931.59   acc: 0.73
itr: 16   loss: 10383.13   acc: 0.74
itr: 18   loss: 9913.99   acc: 0.76
itr: 20   loss: 9503.63   acc: 0.77
itr: 22   loss: 9138.58   acc: 0.78
itr: 24   loss: 8809.45   acc: 0.78
itr: 26   loss: 8509.26   acc: 0.79
itr: 28   loss: 8232.86   acc: 0.80
itr: 30   loss: 7976.42   acc: 0.80
itr: 32   loss: 7737.08   acc: 0.81
itr: 34   loss: 7512.66   acc: 0.82
itr: 36   loss: 7301.39   acc: 0.82
itr: 38   loss: 7101.82   acc: 0.83
itr: 40   loss: 6912.70   acc: 0.83
itr: 42   loss: 6732.98   acc: 0.83
itr: 44   loss: 6561.79   acc: 0.84
itr: 46   loss: 6398.37   acc: 0.84
itr: 48   loss: 6242.07   acc: 0.85
Validation accuracy:  0.7502777777777778
Test accuracy:  0.7583333333333333
```

```
# 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)
```

```
# 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()
```

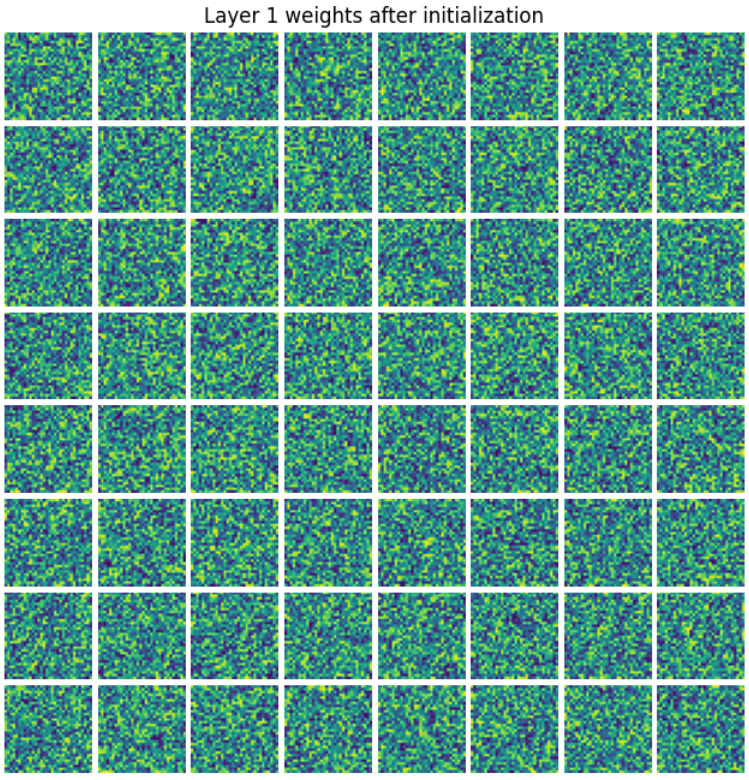


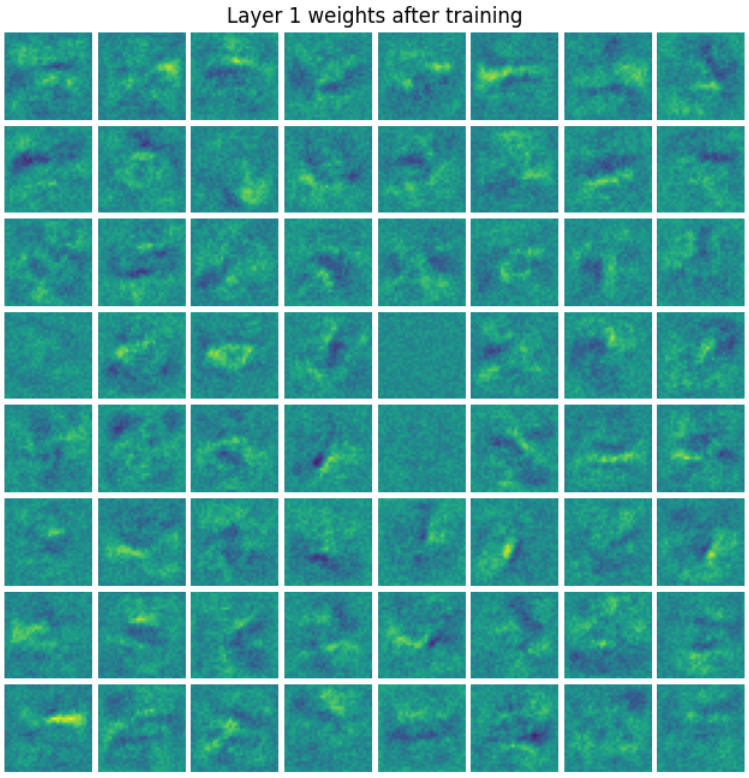


Q3.2 (3 points)

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?

```
############################## 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 initialized Weights seem to be random patterns. This is expected as the Weights were initialised using the Xavier distribution.

However, looking at the Weights after training, there seems to be some sort of patterns that can be observed. We see that some weights place more emphasis on different patterns in the data. For example, i see some weights that place heavy emphasis on straight horizontal lines in the center of the image, circular structures in the center of the image, blobs in the center .etc

---

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

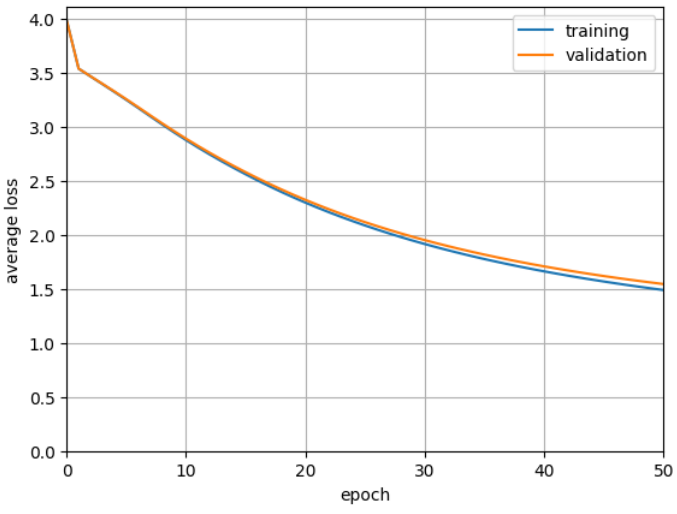
(3) one with your tuned learning rate.

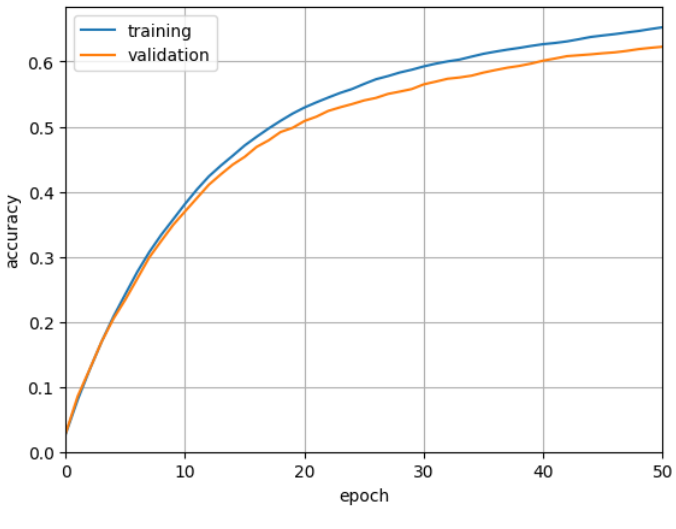
Include total of six plots (two will be the same from Q3.1). Comment on how the learning rates affect the training, and report the final accuracy of the best network on the test set. Hint: Use fixed random seeds to improve reproducibility.

```
############################## Q 3.3 ##############################
############################## Q 3.1 ##############################
max_iters = 50
# pick a batch size, learning rate
batch_size = 64
learning_rate =(2e-3)/10


hidden_size = 64
#########################
np.random.seed(8)
#########################

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
        for key in params:
            if 'grad_' in key:
                params[key.replace('grad_', '')] -= learning_rate * params[key]

    avg_acc /= len(batches)

    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: 39061.42   acc: 0.06
itr: 02   loss: 37040.54   acc: 0.14
itr: 04   loss: 35479.63   acc: 0.23
itr: 06   loss: 33855.41   acc: 0.30
itr: 08   loss: 32229.17   acc: 0.36
itr: 10   loss: 30697.44   acc: 0.40
itr: 12   loss: 29278.50   acc: 0.44
itr: 14   loss: 27964.81   acc: 0.47
itr: 16   loss: 26742.61   acc: 0.50
itr: 18   loss: 25612.38   acc: 0.52
itr: 20   loss: 24574.86   acc: 0.54
itr: 22   loss: 23622.81   acc: 0.55
itr: 24   loss: 22748.61   acc: 0.56
itr: 26   loss: 21945.57   acc: 0.57
itr: 28   loss: 21207.59   acc: 0.58
itr: 30   loss: 20529.00   acc: 0.59
itr: 32   loss: 19904.48   acc: 0.60
itr: 34   loss: 19329.10   acc: 0.61
itr: 36   loss: 18798.30   acc: 0.62
itr: 38   loss: 18307.88   acc: 0.62
itr: 40   loss: 17854.01   acc: 0.63
itr: 42   loss: 17433.22   acc: 0.63
itr: 44   loss: 17042.37   acc: 0.64
itr: 46   loss: 16678.61   acc: 0.64
itr: 48   loss: 16339.41   acc: 0.65
Validation accuracy:  0.6227777777777778
Test accuracy:  0.63
```

```
# 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()
```





Comparison of Losses and Accuracies

Original Learning Rate:

itr: 48 loss: 6242.07 acc: 0.85

Validation accuracy: 0.7502777777777778

Test accuracy: 0.7583333333333333

10 times Learning Rate:

itr: 48 loss: 10374.24 acc: 0.71

Validation accuracy: 0.6366666666666667

Test accuracy: 0.6605555555555556

One-tenth Learning Rate:

itr: 48 loss: 16339.41 acc: 0.65

Validation accuracy: 0.6227777777777778

Test accuracy: 0.63

Training Loss:

We see that for one-tenth the learning rate, the training loss decreases very slowly. For ten times the learning rate, the training loss is extremely jagged, and fails to go below 1. However, for our original tuned learning rate, we see that the loss decreases faster than for one-tenth lr, and is smoother as compared to 10*lr. Furthermore, the loss goes below 1 in this case.

Training Accuracy:

We see that for one-tenth the learning rate, the training accuracy increases very slowly, and doesn't go above 0.6. For ten times the learning rate, the training accuracy is extremely jagged, and fails to go above 0.8. However, for our original tuned learning rate, we see that the accuracy increases faster than for one-tenth lr, and is smoother as compared to 10*lr. Furthermore, the accuracy goes above 0.8 in this case.

We can conclude that one-tenth the learning rate fails to effectively move the model towards convergence in a fast manner. We also see that ten times the learning rate often overshoots the model beyond convergence, resulting in a jagged curve.

The best model accuracy is achieved through the original tuned learning rate.

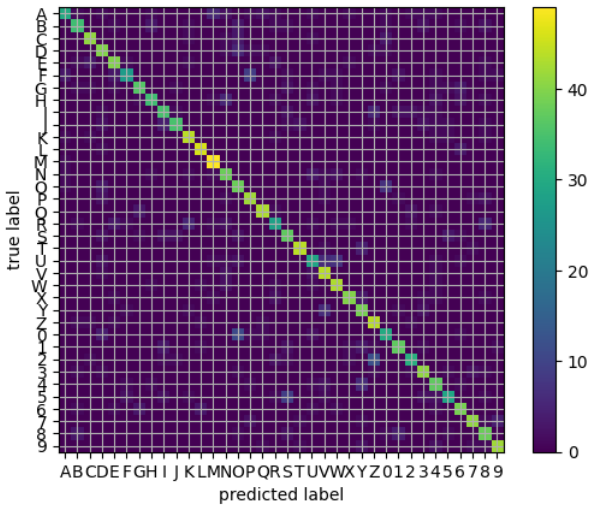
Validation accuracy: 0.7502777777777778

Test accuracy: 0.7583333333333333

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.

```
from sklearn.metrics import confusion_matrix as cm

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

# compute confusion matrix
########################
predicted_labels = np.argmax(test_probs, axis=1)
correct_labels = np.argmax(test_y, axis=1)
confusion_matrix = cm(correct_labels, predicted_labels)
########################



# 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()
```



Commonly confused classes:

- Z and 2: They have similar shapes
- 0 and 0: They have exactly the same shapes, so it makes sense to get confused
- Y and 4: They have similar shapes, especially depending on the way the 4 is written
- S and 5: They have almost exactly the same shapes
- R and K: In cases where the top part of the R may not be connected properly, it may appear as a K

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)

```
# 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
input_size = 1024
hidden1_size = 32  # Hidden layer size

# Layer 1: 1024 to 32 dimensions with ReLU activation
initialize_weights(input_size, hidden1_size, params, 'layer1')
# Layer 2: 32 to 32 dimensions with ReLU activation
initialize_weights(hidden1_size, hidden1_size, params,"layer2")
# Layer 3: 32 to 32 dimensions with ReLU activation
initialize_weights(hidden1_size, hidden1_size,params, "layer3")
# Layer 4: 32 to 1024 dimensions with sigmoid activation
initialize_weights(hidden1_size, input_size, 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.

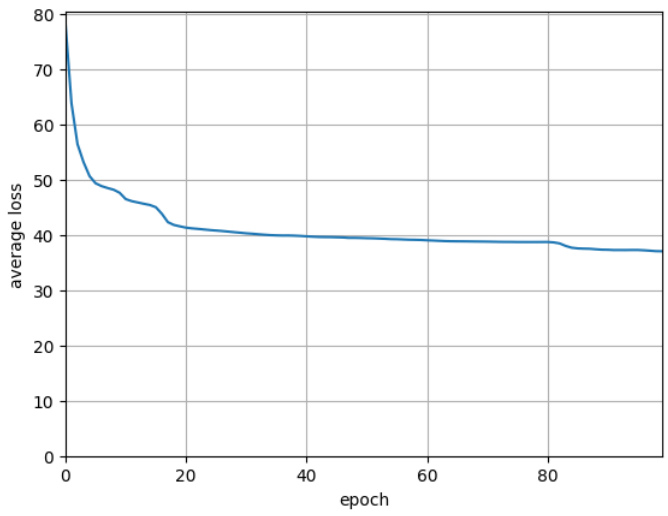
```
momentum_params = {}
for k, v in params.items():
    momentum_params["M_"+k] = np.zeros_like(v)

combined_params = {}
for k, v in params.items():
    combined_params[k] = v

for k, v in momentum_params.items():
    combined_params[k] = v
```

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?

```
######################### 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:
        # forward
        h1 = forward(xb,combined_params,'layer1',relu)
        h2 = forward(h1,combined_params,'layer2',relu)
        h3 = forward(h2,combined_params,'layer3',relu)
        probs = forward(h3,combined_params,'output',sigmoid)

        # loss
        loss = np.sum((xb - probs) ** 2, axis=1)
        total_loss += np.sum(loss)

        # backward
        delta1 = 2*(probs - xb)
        delta2 = backwards(delta1,combined_params,'output',sigmoid_deriv)
        delta3 = backwards(delta2,combined_params,'layer3',relu_deriv)
        delta4 = backwards(delta3,combined_params,'layer2',relu_deriv)
        backwards(delta4,combined_params,'layer1',relu_deriv)

        # update the parameters
        for key in combined_params:
          if 'M_' in key:
            #print(key)
            combined_params[key] = combined_params[key]*0.9- learning_rate * combined_params[key.replace('M_', 'grad_')]
            combined_params[key.replace('M_','')] += combined_params[key]


    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: 845831.82
itr: 02          loss: 609420.15
itr: 04          loss: 546964.98
itr: 06          loss: 527355.19
itr: 08          loss: 520325.41
itr: 10          loss: 502017.73
itr: 12          loss: 495431.27
itr: 14          loss: 490521.81
itr: 16          loss: 473313.05
itr: 18          loss: 451463.06
itr: 20          loss: 446191.56
itr: 22          loss: 443798.69
itr: 24          loss: 441491.26
itr: 26          loss: 439635.11
itr: 28          loss: 437334.44
itr: 30          loss: 435123.18
itr: 32          loss: 433297.79
itr: 34          loss: 431594.39
itr: 36          loss: 430860.58
itr: 38          loss: 430539.09
itr: 40          loss: 429285.62
itr: 42          loss: 428053.09
itr: 44          loss: 427740.81
itr: 46          loss: 427060.81
itr: 48          loss: 426271.03
itr: 50          loss: 425636.74
itr: 52          loss: 424937.68
itr: 54          loss: 423809.15
itr: 56          loss: 423135.91
itr: 58          loss: 422457.61
itr: 60          loss: 421504.12
itr: 62          loss: 420432.59
itr: 64          loss: 419651.52
itr: 66          loss: 419424.44
itr: 68          loss: 419126.36
itr: 70          loss: 418882.53
itr: 72          loss: 418450.91
itr: 74          loss: 418301.47
itr: 76          loss: 418126.08
itr: 78          loss: 418100.73
itr: 80          loss: 418201.53
itr: 82          loss: 415409.24
itr: 84          loss: 407005.53
itr: 86          loss: 405197.80
itr: 88          loss: 403987.03
itr: 90          loss: 402960.71
itr: 92          loss: 402465.25
itr: 94          loss: 402567.60
itr: 96          loss: 401934.82
itr: 98          loss: 400251.34
```



It seems that the training loss drastically decreases initially, with the decrease in loss slowing down as the number of epochs increases.

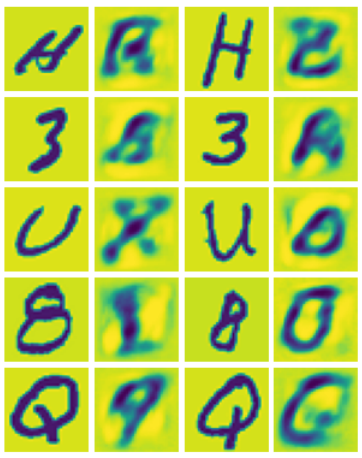
As momentum initially builds up rapidly, we see a fast decrease in average loss. Then, again as the number of epochs increases, the Autoencoder seems to learn the essential features of our input data, leading to a very gradual decrease in our loss.

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?

```
############################## 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=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
# TODO: name the output reconstructed_x
#########################
reconstructed_x = forward(visualize_x,combined_params,'layer1',relu)
h2 = forward(h1,combined_params,'layer2',relu)
h3 = forward(h2,combined_params,'layer3',relu)
reconstructed_x = forward(h3,combined_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()
```



We see that the reconstructed validation images vaguely resemble the overall prominent patterns/structures of the digit present in the original images. However, there are certain differences that can be observed. The reconstructed images are blurrier as compared to the original images, possibly to preserve a sense of overall structure.

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).

```
############################## Q 4.3.2 ##############################
from skimage.metrics import peak_signal_noise_ratio

psnr_values = []
for i in range(valid_x.shape[0]):
    # Run validation image through the network
    h1 = forward(valid_x[i],combined_params,'layer1',relu)
    h2 = forward(h1,combined_params,'layer2',relu)
    h3 = forward(h2,combined_params,'layer3',relu)
    reconstructed_image = forward(h3,combined_params,'output',sigmoid)

    # Calculate PSNR (assuming max_pixel_value is 255 for 8-bit grayscale images)
    try:
        psnr = peak_signal_noise_ratio(valid_x[i].reshape(32,32), reconstructed_image.reshape(32, 32))
        psnr_values.append(psnr)
    except ValueError:  # Handle potential errors (e.g., different image shapes)
        print(f"Error calculating PSNR for image {i}. Skipping.")

# Calculate and report average PSNR
if psnr_values:
    average_psnr = np.mean(psnr_values)
    print(f"Average PSNR across validation set: {average_psnr:.2f}")
else:
    print("No valid PSNR values calculated.")
```

```
Average PSNR across validation set: 14.54
```

Average PSNR across validation set: 14.54

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.

```
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.zip
  !unzip "/content/images/images.zip" -d "/content/images"
  os.system("rm /content/images/images.zip")
```

```
--2024-04-10 23:58:16--  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  2.07MB/s    in 1.5s

2024-04-10 23:58:18 (2.07 MB/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
```

```
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. In case the characters are touching eachother, the connected components assumption woudl fail. The model would incorrectly consider the characters to be a single character. This could possibly lead to erroneous results.
2. The model assumes the text to be on a flat surface for the 3rd step to work efficiently. In case of curved or distorted surfaces, the sorting would result in erroneous results.

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.

```
from skimage import color, filters, feature, morphology, measure

############################## Q 5.2 ##############################

def findLetters(image):
  """
  takes a color image
  returns a list of bounding boxes and black_and_white image
  """
  # insert processing in here
  # one idea estimate noise -> denoise -> greyscale -> threshold -> morphology -> label -> skip small boxes
  # this can be 10 to 15 lines of code using skimage functions

  bboxes = []
  bw = None

  #grayscale
  gray = color.rgb2gray(image)

  #blur
  gray = filters.gaussian(1-gray, sigma=1)

  #thresholding
  thresh = filters.threshold_otsu(gray)
  bw = gray > thresh

  # morphological operations
  bw1 = morphology.opening(bw, morphology.disk(2))
  bw1 = morphology.closing(bw1, morphology.disk(1))

  # find connected components
  labeled_image, num_objects = measure.label(bw1, connectivity=2, return_num=True)

  # filter small and large components
  for label in range(1, num_objects + 1):
    mask = labeled_image == label

    # Extract bounding box coordinates
    y1, x1, y2, x2 = measure.regionprops(labeled_image*mask)[0].bbox

    bboxes.append([y1, x1, y2, x2])
  return bboxes, bw
```

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.

```python
from skimage import color, filters, feature, morphology, measure

############################ Q 5.2 ############################

def findLetters(image):
  """
  takes a color image
  returns a list of bounding boxes and black_and_white image
  """
  # insert processing in here
  # one idea estimate noise -> denoise -> greyscale -> threshold -> morphology -> label -> skip small boxes
  # this can be 10 to 15 lines of code using skimage functions

  bboxes = []
  bw = None

  #grayscale
  gray = color.rgb2gray(image)

  #blur
  gray = filters.gaussian(1-gray, sigma=1)

  #thresholding
  thresh = filters.threshold_otsu(gray)
  bw = gray > thresh

  # morphological operations
  bw1 = morphology.opening(bw, morphology.disk(2))
  bw1 = morphology.closing(bw1, morphology.disk(1))

  # find connected components
  labeled_image, num_objects = measure.label(bw1, connectivity=2, return_num=True)

  # filter small and large components
  for label in range(1, num_objects + 1):
    mask = labeled_image == label

    # Extract bounding box coordinates
    y1, x1, y2, x2 = measure.regionprops(labeled_image*mask)[0].bbox

    bboxes.append([y1, x1, y2, x2])
  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.

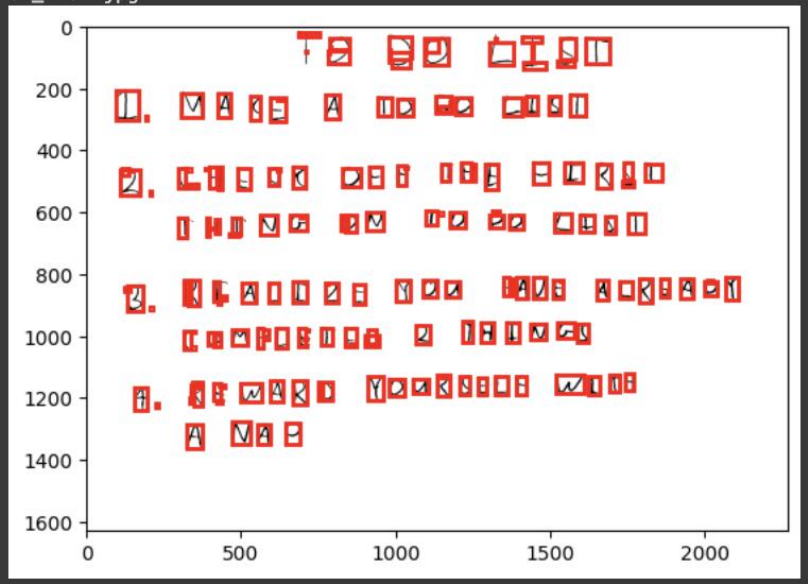
Q5.3 (Extra Credit) (3 points)

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.

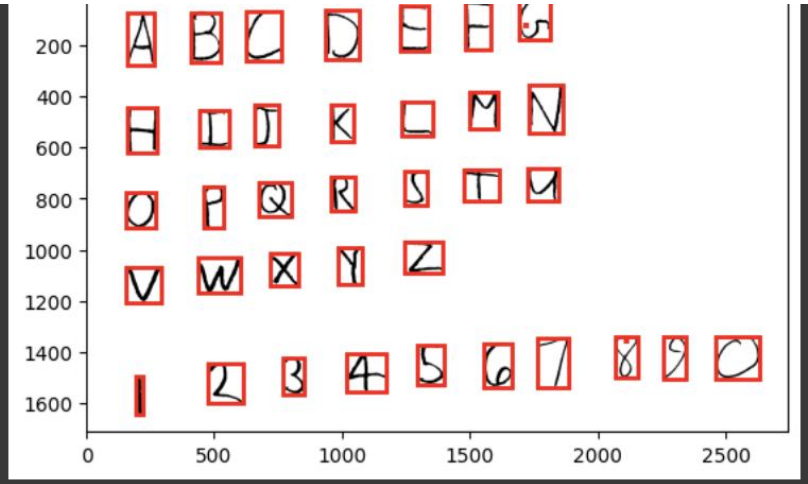
20s

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

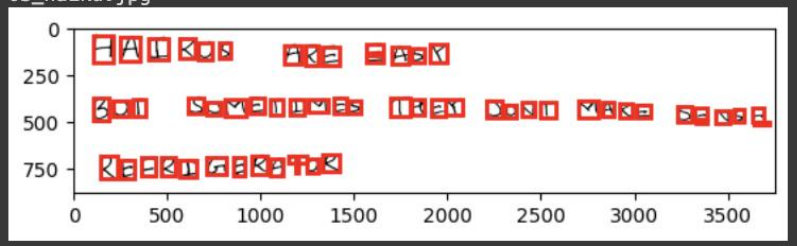
01_list.jpg



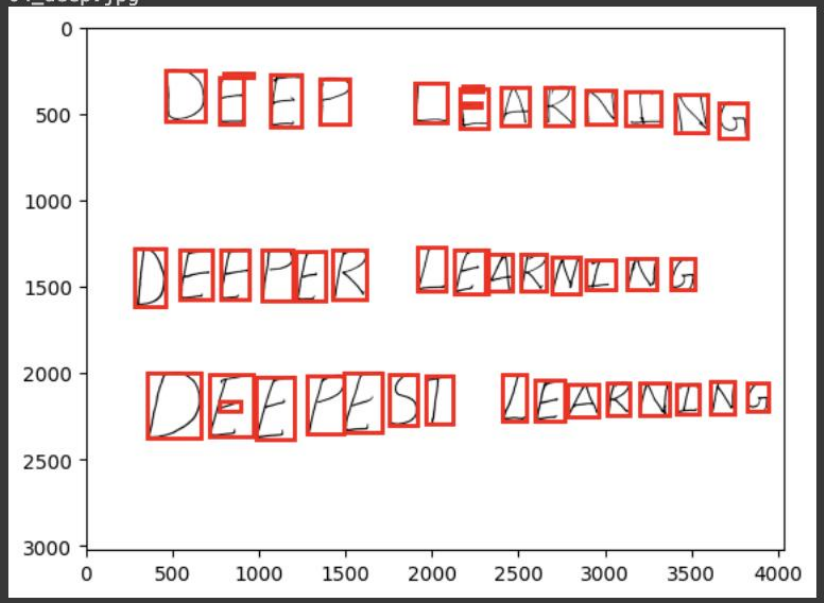
02_letters.jpg



03_haiku.jpg



04_deep.jpg



Q5.4 (Extra Credit) (8 points)

$y = Wx + b$

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

- $\frac{\partial J}{\partial W} \Rightarrow \frac{\partial J}{\partial y} \cdot \frac{\partial y}{\partial W}$

  $\Rightarrow \delta \frac{\partial (Wx + b)}{\partial W}$

  $\Rightarrow \delta \cdot x$

- $\frac{\partial J}{\partial x} \Rightarrow \frac{\partial J}{\partial y} \frac{\partial y}{\partial x}$ (chain rule)

  $\Rightarrow \delta \frac{\partial (wx + b)}{\partial x}$

  $\Rightarrow \delta . W$

- $\frac{\partial J}{\partial b} \Rightarrow \frac{\partial J}{\partial y} \cdot \frac{\partial y}{\partial b}$ (chain rule)

  $\Rightarrow \delta . \frac{\partial (wx + b)}{\partial b}$

  $\Rightarrow \delta$

In Matrix Form,

$\frac{\partial J}{\partial W} \Rightarrow \delta . x \Rightarrow \delta . x^T$

$_{kx1}\ _{dx1}$

$\frac{\partial J}{\partial x} \Rightarrow \delta . W \Rightarrow W^T . \delta$

$_{kx1}\ _{kxd}$

$\frac{\partial J}{\partial b} \Rightarrow \delta$

sigmoid

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

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

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

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

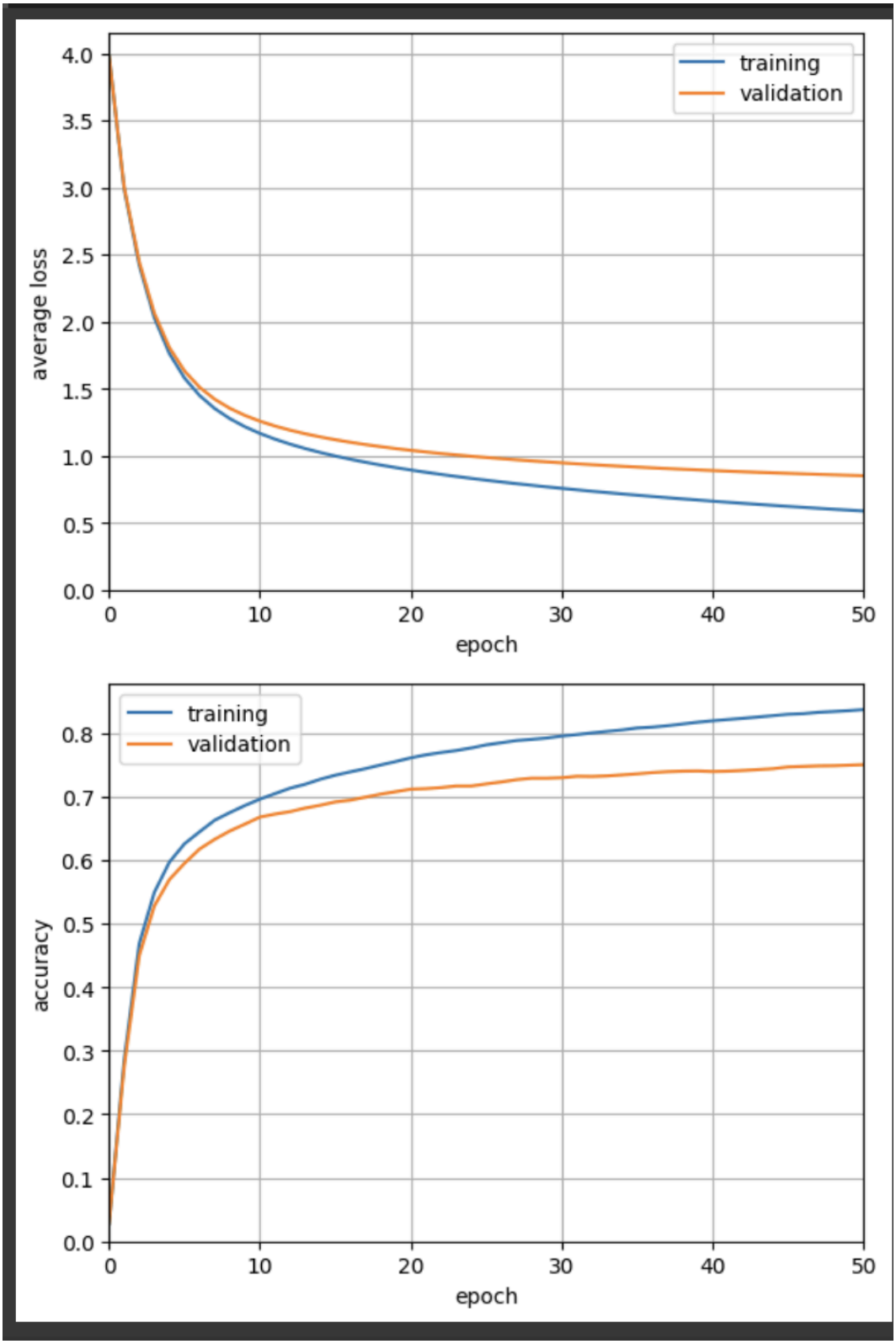
$$\Rightarrow \frac{1-\left(\frac{1-\sigma(x)}{\sigma(x)}\right)^2}{1+\left(\frac{1-\sigma(x)}{\sigma(x)}\right)^2}$$

$$\Rightarrow \frac{\sigma^2(x) - [1+\sigma^2(x)-2\sigma(x)]}{\sigma^2(x)+1+\sigma^2(x)-2\sigma(x)}$$

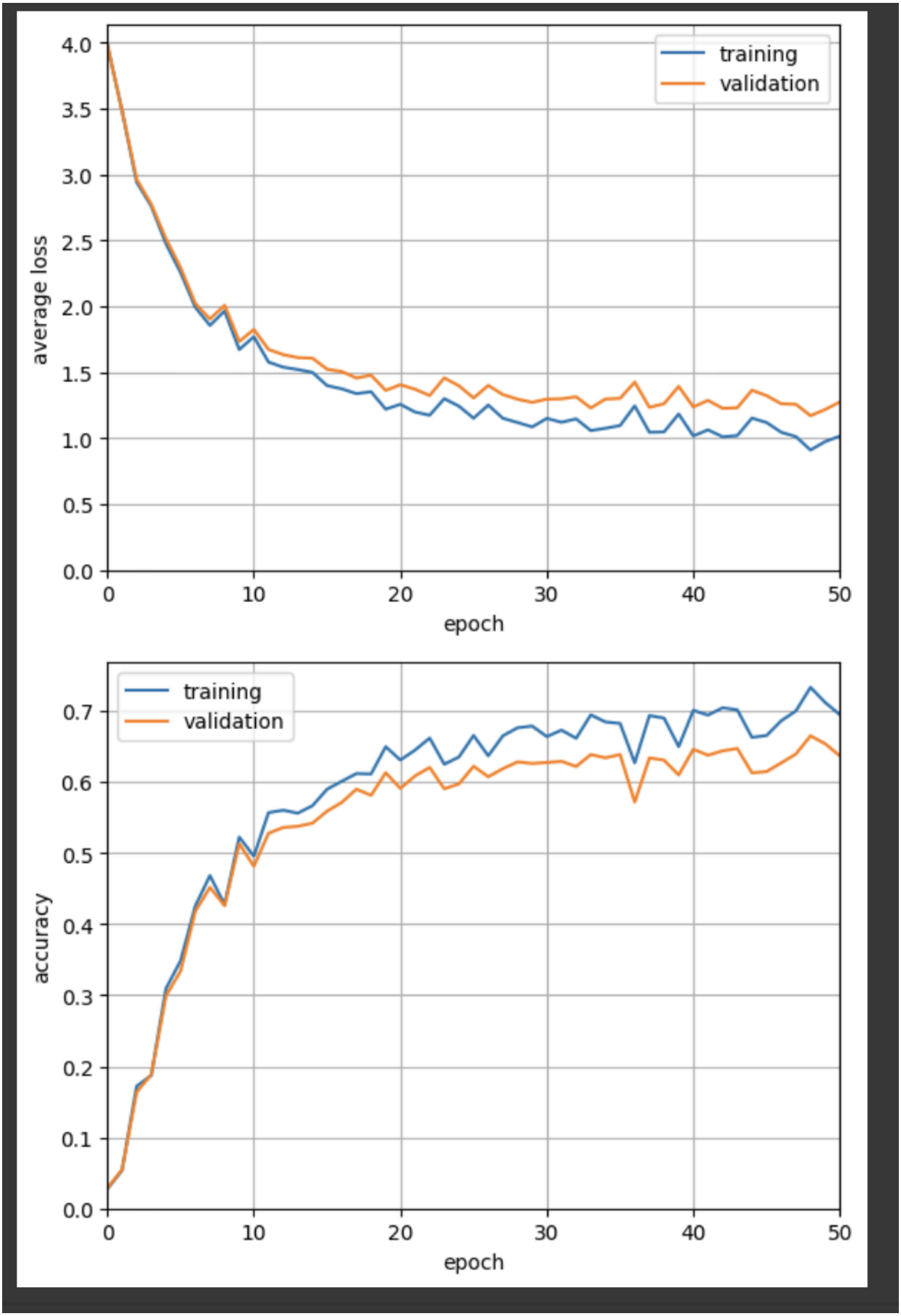
$$\Rightarrow \frac{\sigma^2(x) - 1 - \sigma^2(x) + 2\sigma(x)}{1+2\sigma^2(x)-2\sigma(x)}$$

$$\tanh(x) \Rightarrow \frac{2\cdot\sigma(x)-1}{1+2\sigma^2(x)-2\sigma(x)}$$

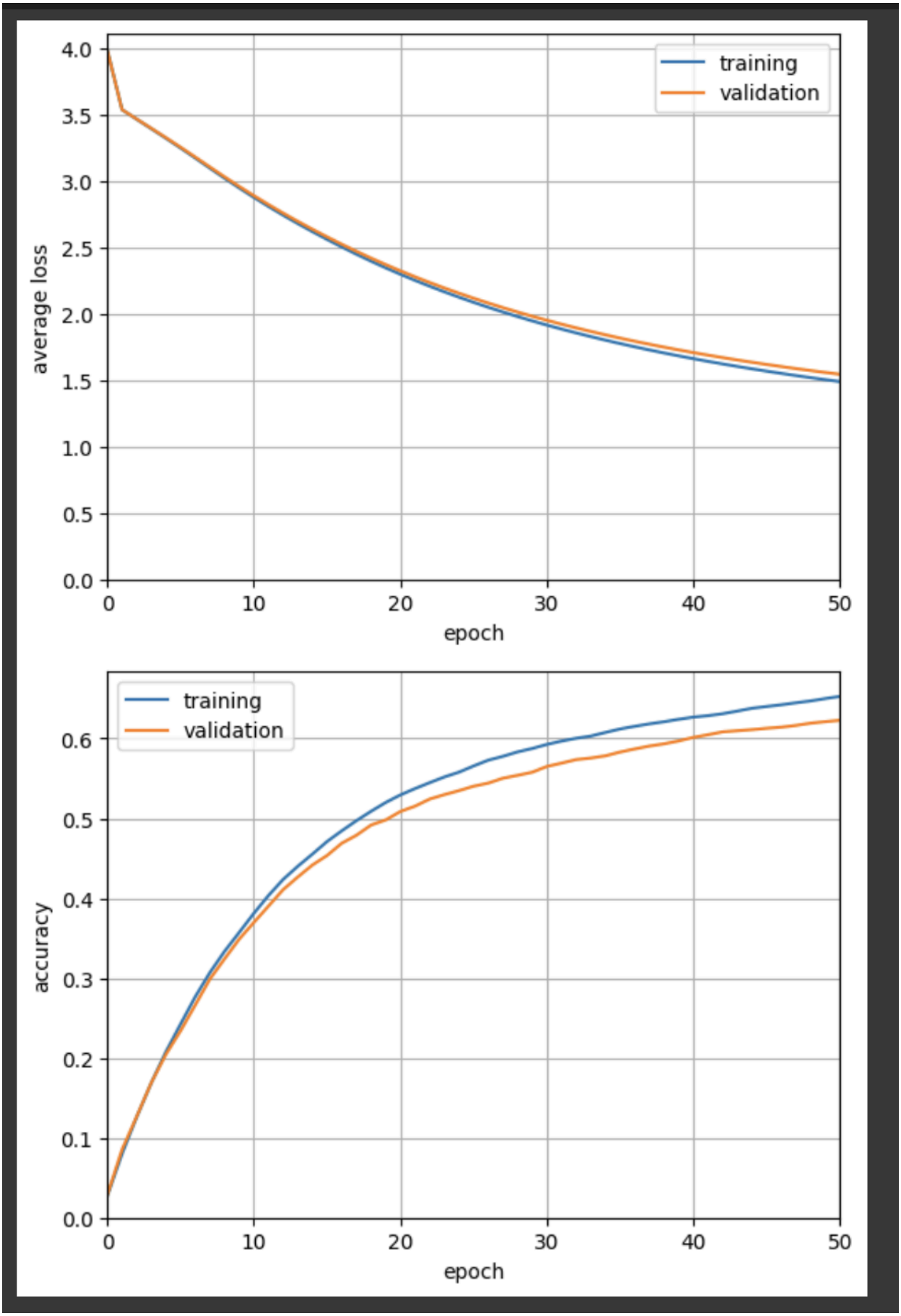
Original Learning Rate

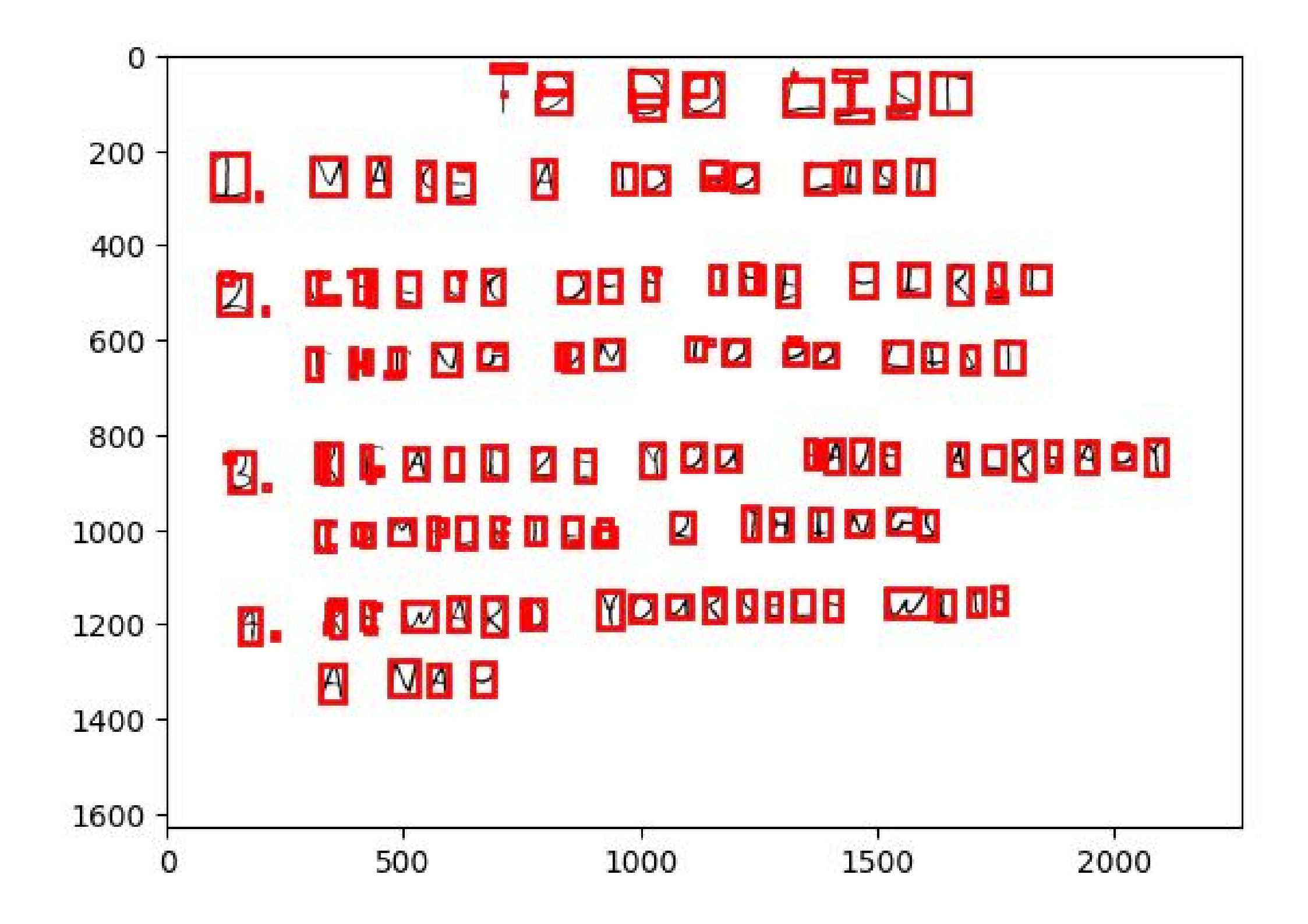


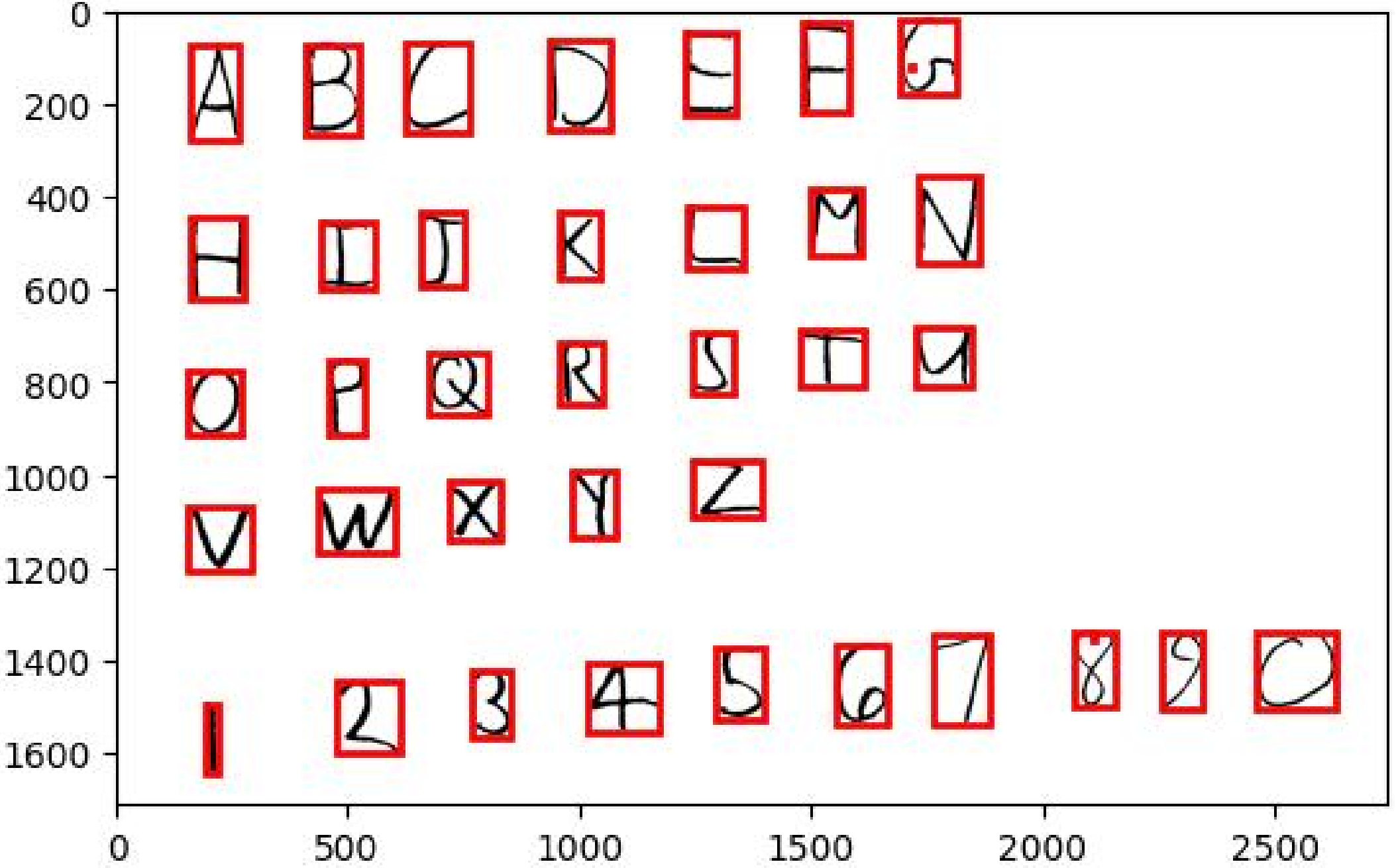
Learning Rate*10

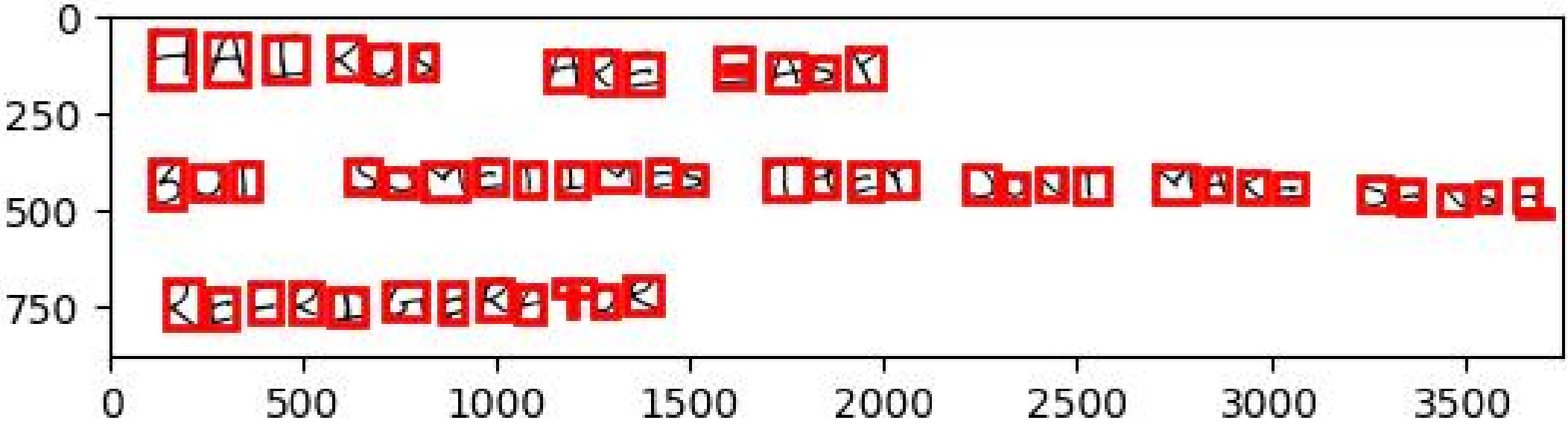


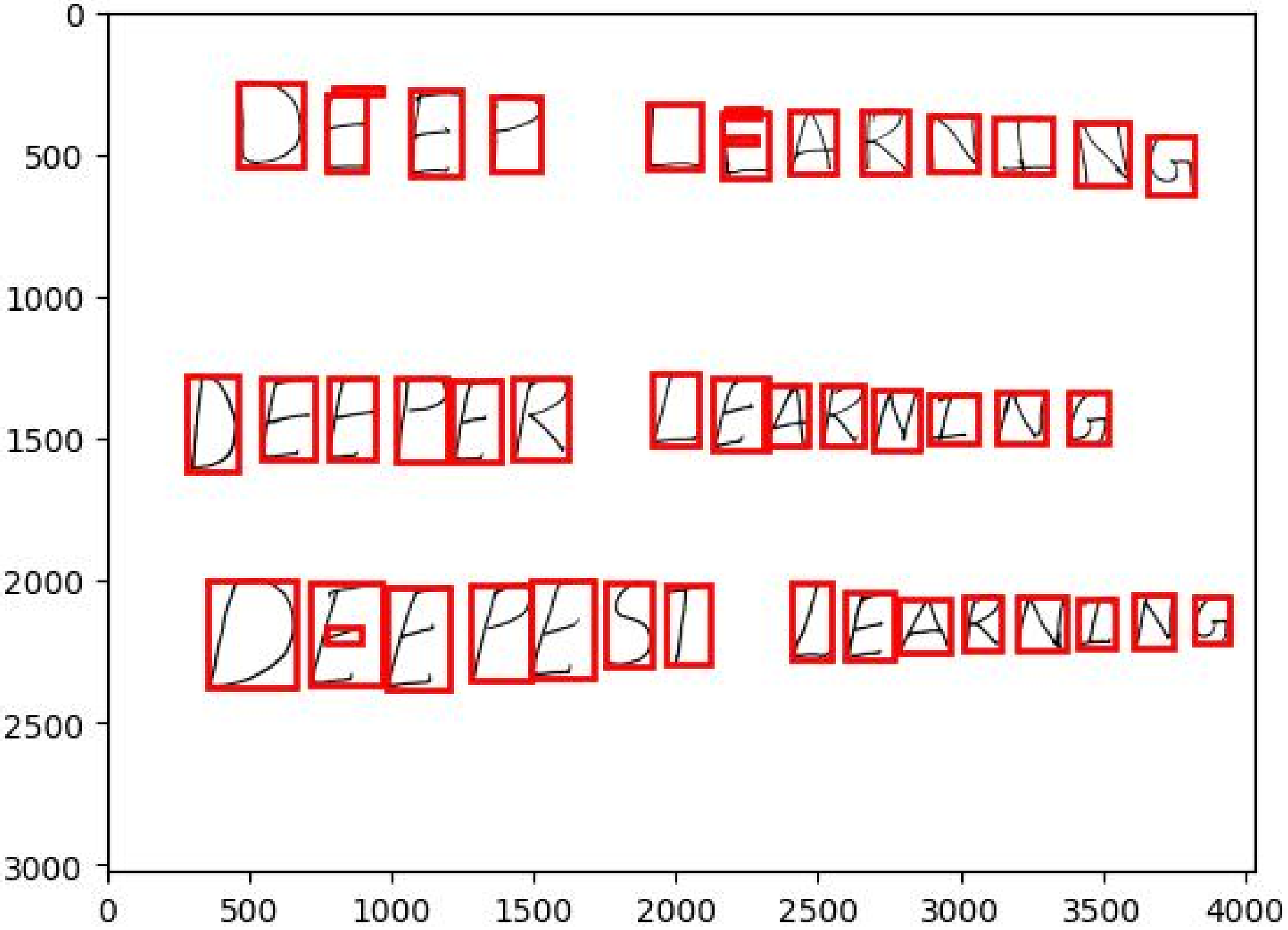
Learning Rate/10




0
200
400
600
800
1000
1200
1400
1600
0
500
1000
1500
2000


0
200
400
600
800
1000
1200
1400
1600
0
500
1000
1500
2000
2500


0
250
500
750
0
500
1000
1500
2000
2500
3000
3500


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