Homework 4: Neural Networks for Recognition, Detection, and Tracking

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 $c = -\max x_i$)

Let $c \in \mathbb{R}$, then for all index i, we have

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

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

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

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

$$softmax(x + c)_i = softmax(x)_i$$

Since this equality hols for all index i, we have

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

Using $c = 0$, the numerator exponential can take on values between 0 to infinity. However, using $c = -\max x_i$, the numerator exponential can only take on values between 0 to 1. Translating the vector x by $c = -\max x_i$ is a good idea since we get better numerical stability during division calculations and prevent overflows caused by large exponential terms, while we are also guaranteed mathematically that the solution is invariant from the translation from the calculation above.

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)

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?

The range of each element is $(0, 1]$.

The sum over all elements is 1.

Q1.2.2 (1 point)

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

Probability mass function (i.e. discrete probability distribution function) of d possible outcomes (d is the dimension of x).

Q1.2.3 (1 point)

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

Step 1: Exponentiate each entry in the vector x to convert it into a positive frequency.

Step 2: Calculate the total frequency by summing over each frequency in the vector x.

Step 3: Normalize the frequencies by total frequency so that it becomes a probability distribution.

Q1.3 (3 points)

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

A n-layer neural network without a non-linear activation function will update according to as follows:

$$y = W_n x_n + b_n$$
$$x_i = W_{i-1} x_{i-1} + b_{i-1}, \forall i \in \mathbb{N}_{[1,n]}$$

Where W, x, y, b are defined in Q1.5. Also let x_1 denote the original input. Applying the above equations recursively, we obtain:

$$y = W_n(W_{n-1} x_{n-1} + b_{n-1}) + b_n$$
$$y = (W_n W_{n-1}) x_{n-1} + (W_n b_{n-1} + b_n)$$
$$y = \prod_{i=1}^{n} W_i x + \sum_{i=1}^{n} (\prod_{j=i+1}^{n} W_j) b_i$$

Denoting $W' = \prod_{i=1}^{n} W_i$ and $b' = \sum_{i=1}^{n} (\prod_{j=i+1}^{n} W_j)$, we obtain

$$y = W' x + b'$$

Which is the equation for a linear regression. So the problem of optimizing the W_i's and b_i's for a n-layer neural network is equivalent to solving W' and b' for a linear regression when non-linear activation functions are missing.

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).

The gradient of the sigmoid function is given as:

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

$$\frac{dx}{dx} \qquad \frac{dx}{dx}$$

Applying the chain rule:

$$\frac{d\sigma(x)}{dx} = (-1)(1+e^{-x})^{-2}(e^{-x})(-1)$$
$$\frac{d\sigma(x)}{dx} = \frac{1}{1+e^{-x}}\frac{e^{-x}}{1+e^{-x}}$$
$$\frac{d\sigma(x)}{dx} = \frac{1}{1+e^{-x}}(1-\frac{1}{1+e^{-x}})$$
$$\therefore \frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x))$$

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.

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

Given $y_i = \sum_{j=1}^{d} x_j W_{ij} + b_i$, we have

$$\frac{\partial y_i}{\partial W_{jk}} = x_k I(i=j)$$

Where I is the indicator function (i.e. 1 if $i = j$, 0 otherwise). So (using numerator layout/Jacobian notation):

$$\frac{\partial y_i}{\partial W} = \begin{bmatrix} \frac{\partial y_i}{\partial W_{11}} & \frac{\partial y_i}{\partial W_{21}} & \cdots & \frac{\partial y_i}{\partial W_{k1}} \\ \frac{\partial y_i}{\partial W_{12}} & \frac{\partial y_i}{\partial W_{22}} & \cdots & \frac{\partial y_i}{\partial W_{k2}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_i}{\partial W_{1d}} & \frac{\partial y_i}{\partial W_{2d}} & \cdots & \frac{\partial y_i}{\partial W_{kd}} \end{bmatrix}$$

$$\frac{\partial y_i}{\partial W} = \begin{bmatrix} 0 & x_1 & \dots & 0 \\ 0 & x_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & x_d & \dots & 0 \end{bmatrix}$$

$$\frac{\partial y_i}{\partial W} = \begin{bmatrix} 0 & \dots & x & \dots & 0 \end{bmatrix}$$

Where only the i'th column contains the vector x. So:

$$\frac{\partial J}{\partial W} = \sum_{i=1}^{k} \frac{\partial J}{\partial y_i}\frac{\partial y_i}{\partial W}$$

$$\frac{\partial J}{\partial W} = \begin{bmatrix} \frac{\partial J}{\partial y_1}x_1 & \frac{\partial J}{\partial y_2}x_1 & \dots & \frac{\partial J}{\partial y_k}x_1 \\ \frac{\partial J}{\partial y_1}x_2 & \frac{\partial J}{\partial y_2}x_2 & \dots & \frac{\partial J}{\partial y_k}x_2 \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial J}{\partial y_1}x_d & \frac{\partial J}{\partial y_2}x_d & \dots & \frac{\partial J}{\partial y_k}x_d \end{bmatrix}$$

$$\therefore \frac{\partial J}{\partial W} = x\delta^T \in \mathbb{R}^{d\times k}$$

Similarly,

$$\frac{\partial y_i}{\partial x_j} = W_{ij}$$

So

$$\frac{\partial y_i}{\partial x} = \begin{bmatrix} W_{i1} & W_{i2} & \dots & W_{id} \end{bmatrix}$$

$$\frac{\partial \boldsymbol{J}}{\partial x} = \sum_{i=1}^{k} \frac{\partial \boldsymbol{J}}{\partial y_i} \frac{\partial y_i}{\partial x}$$

$$\frac{\partial \boldsymbol{J}}{\partial x} = \sum_{i=1}^{k} \frac{\partial \boldsymbol{J}}{\partial y_i} \begin{bmatrix} W_{i1} & W_{i2} & \dots & W_{id} \end{bmatrix}$$

$$\therefore \frac{\partial \boldsymbol{J}}{\partial x} = \delta^T W \in \mathbb{R}^{1 \times d}$$

Finally,

$$\frac{\partial y_i}{\partial b_j} = I(i = j)$$

So

$$\frac{\partial y_i}{\partial b} = \begin{bmatrix} 0 & \dots & 1 & \dots & 0 \end{bmatrix}$$

Where only the i'th column contains the element 1. So:

$$\frac{\partial \boldsymbol{J}}{\partial b} = \sum_{i=1}^{k} \frac{\partial \boldsymbol{J}}{\partial y_i} \frac{\partial y_i}{\partial b}$$

$$\frac{\partial \boldsymbol{J}}{\partial b} = \begin{bmatrix} \frac{\partial J}{\partial y_1} & \dots & \frac{\partial J}{\partial y_i} & \dots & \frac{\partial J}{\partial y_k} \end{bmatrix}$$

$$\therefore \frac{\partial \boldsymbol{J}}{\partial b} = \delta^T \in \mathbb{R}^{1 \times k}$$

Note that we are using numerator layout/Jacobian notation. If using denominator notation, the results are transposed and we obtain:

$$\frac{\partial \boldsymbol{J}}{\partial W} = \delta x^T \in \mathbb{R}^{k \times d}$$

$$\frac{\partial \boldsymbol{J}}{\partial x} = W^T \delta \in \mathbb{R}^{d \times 1}$$

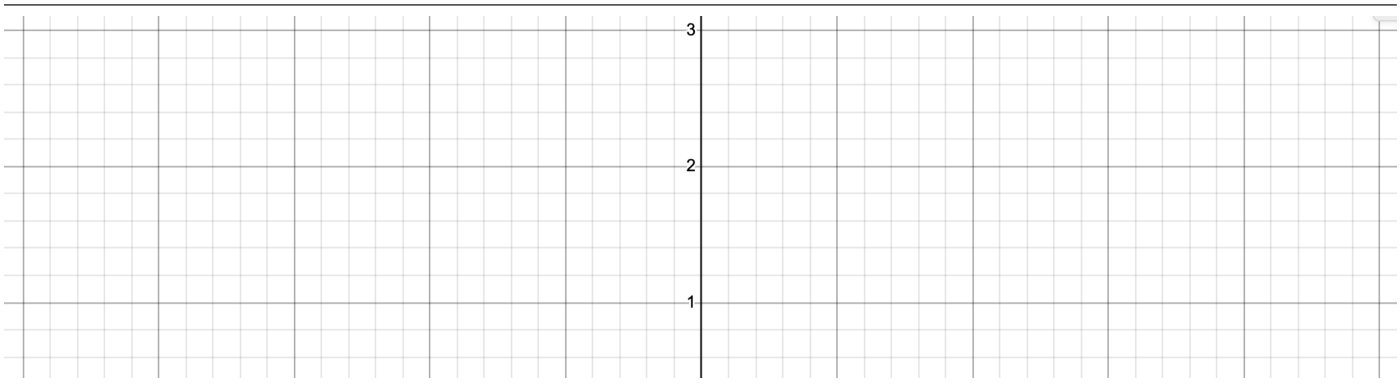
$$\frac{\partial \boldsymbol{J}}{\partial b} = \delta \in \mathbb{R}^{k \times 1}$$

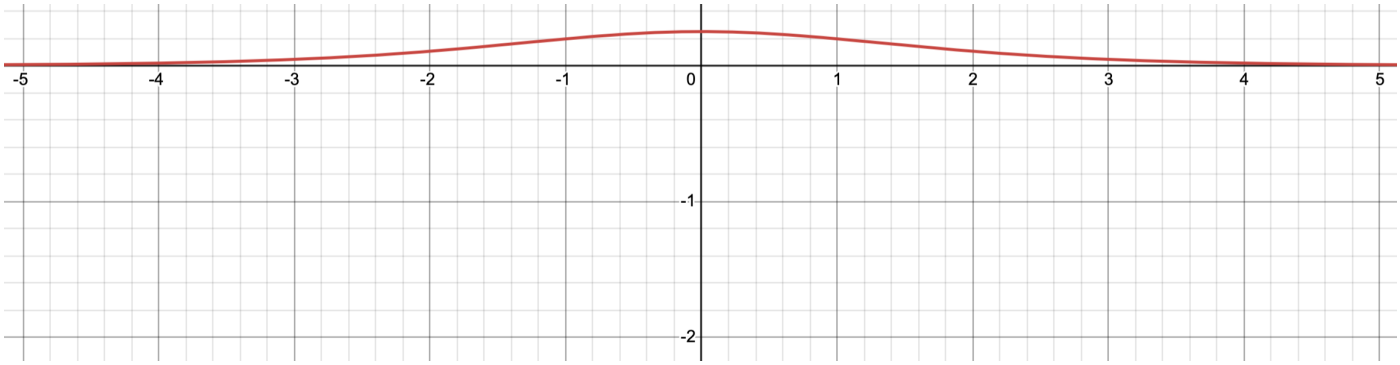
Q1.6

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)?





The derivative of the sigmoid function has a range of (0, 0.25]. During backpropogation, whenever we encounter a sigmoid function, the upstream gradient is multiplied by a value in (0, 0.25] to produce the downstream gradient. When the sigmoid function is used for many layers, the downstream gradient becomes smaller and smaller as the multiplication is compounded, eventually leading to a "vanishing gradient" as the downstream gradient approaches zero (and thus stalling the gradient updates for gradient descent).

Q1.6.2 (1 point)

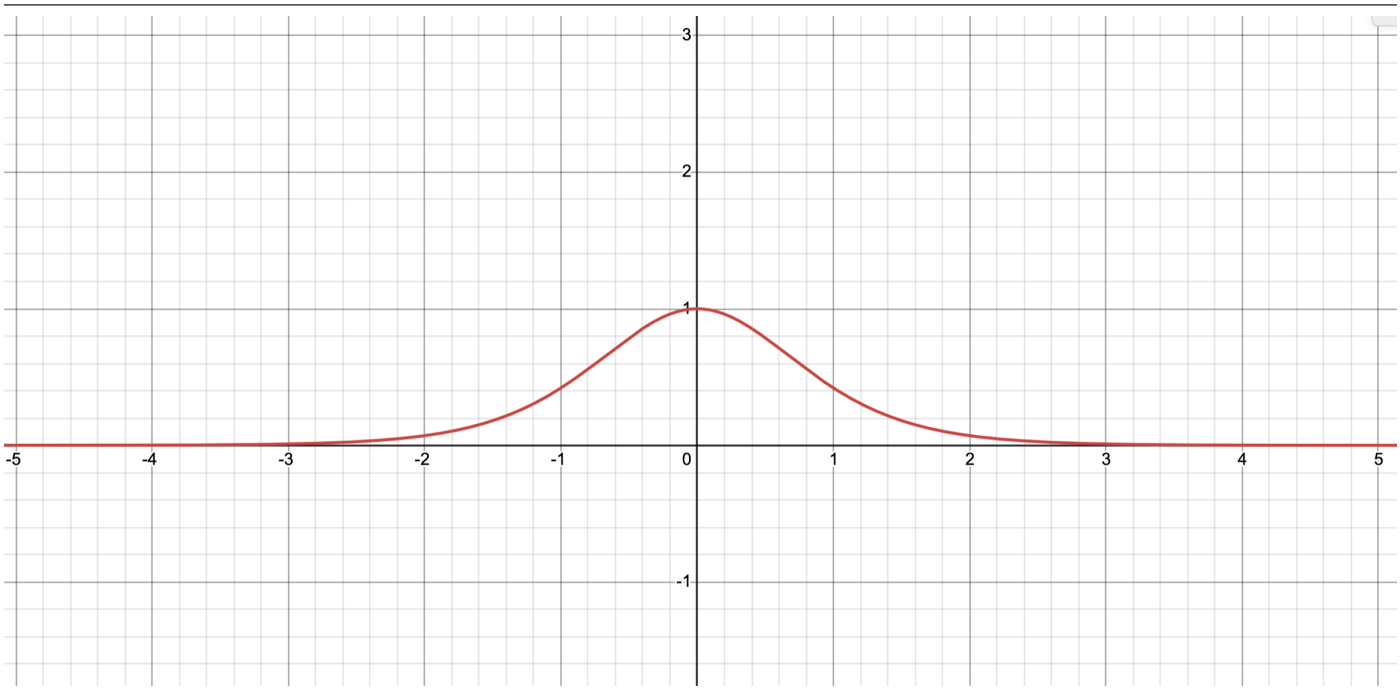
Often it is replaced with $\tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$. What are the output ranges of both $\tanh$ and sigmoid? Why might we prefer $\tanh$?

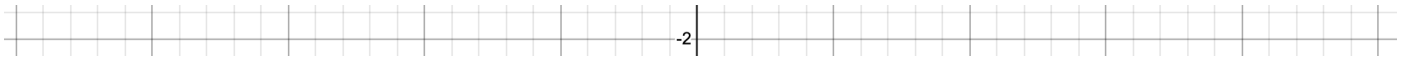
The output range of tanh is (-1, 1) while the output range of the sigmoid function is (0, 1). A few reasons as to why we might prefer tanh:

1. The sigmoid is centered around 0.5. This may induce a positive bias in the activation values. The tanh function is centered around 0 so there is no bias.
2. The gradient of the tanh function's range is 4 times of the gradient of the sigmoid. Thus the vanishing gradient problem is less.
3. Gradient descent using the sigmoid function will be in a zigzagging pattern due to the input vector all having the same sign (for multi-layered neural networks, the output of the sigmoid function eventually feeds into the input vector). This will cause convergence to be slow. The tanh function mitigates this problem since the input vector can have both positive and negative values.

Q1.6.3 (1 point)

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





The tanh(x) have less of a vanishing gradient problem since the gradient's range is (0,1], which is 4 times the gradient of the sigmoid function. Whenever we encounter a tanh(x) function during backprop, the upstream gradient would in general be multiplied by a larger value (albeit still less than 1) compared to the sigmoid, and thus retains a larger downstream gradient. This will cause the gradient to vanish slower, so there is less of a vanishing gradient problem.

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 $\sigma(x)$.

We have:

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

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

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

$$\therefore tanh(x) = 2\sigma(2x) - 1$$

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?

When we initialize a network with all zeros, each neuron in a layer produces the same output (and thus also have the same gradient during backprop). This causes the weights to update symmetrically for all neurons (or even not at all if the activation function is centered at 0), preventing the network from learning different features and converging to a suitable loss minima.

If we are using an activation function centered at 0 (e.g. ReLu or tanh), then the output after training would be all zeros. This is because there will be a zero gradient during backprop and so that network doesn't learn anything.

If we are using a non-centered activation function (e.g. sigmoid), then the output after training would all have the same values. This is because the network weight updates are all in unison. This would cause the network to have very poor predictive properties.

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

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

    bound = np.sqrt(6 / (in_size + out_size))
    W = np.random.uniform(-bound, bound, (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 scale the initialization variance depending on layer size to maintain a consistent variance of activations and gradients across layers. This helps maintain a smooth gradient flow, and minimizes the chances of the activation/gradient exploding (which could destabilize the network) or vanishing (which could stall learning).

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

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

    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]

    ##########################
    ##### 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.

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

    x_stable = x - np.max(x, axis=1, keepdims=True)
    exp_x = np.exp(x_stable)
    res = exp_x / np.sum(exp_x, 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 $f : \mathbb{R}^D \rightarrow [0, 1]^C$ 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
    """
    loss, acc = None, None

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

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

    correct_preds = (np.argmax(probs, axis=1) == np.argmax(y, axis=1))
    acc = np.mean(correct_preds)

    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, num
    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 #####
    ##########################

    grad = activation_deriv(post_act)
    dLdy = grad * delta

    grad_W = np.dot(X.T, dLdy)
    grad_X = np.dot(dLdy, W.T)
    grad_b = np.sum(dLdy, 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: 0.0, 0.08
Q 2.1.2: 0.0, 0.07
Q 2.2.1: sigmoid outputs should be zero and one  0.0 1.0
Q 2.2.2: 0.08218644718718 0.9999999999999998 1.0000000000000002
Q 2.2.3 loss: 58.28663932156648, 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-3-15b4c7124231>:13: 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 = []

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

    num_examples = x.shape[0]
    indicies = np.random.choice(num_examples, size = (int(num_examples/batch_size), batch_size))

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

```
    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:
        #########################
        ##### your code here #####
        #########################
        # forward
        h1 = forward(xb, params, 'layer1', sigmoid)
        probs = forward(h1, params, 'output', softmax)

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

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

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

    avg_acc /= batch_num

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

```
itr: 00          loss: 61.15     acc : 0.22
itr: 100         loss: 35.56     acc : 0.70
itr: 200         loss: 29.08     acc : 0.80
itr: 300         loss: 24.34     acc : 0.82
itr: 400         loss: 21.27     acc : 0.82
```

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")
```

```
--2024-11-11 23:48:47--  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   354KB/s    in 11m 16s

2024-11-12 00:00:03 (313 KB/s) - '/content/data/data.zip' saved [216305627/216305627]

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

```
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)

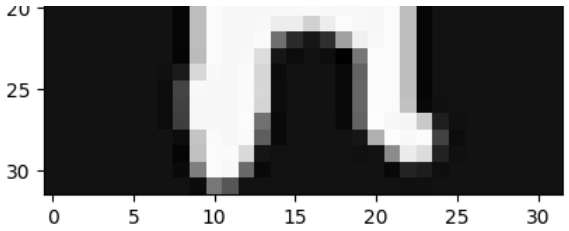
Train a network from scratch. Use a single hidden layer with 64 hidden units, and train for at least 50 epochs. The script will generate two plots:

(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
```



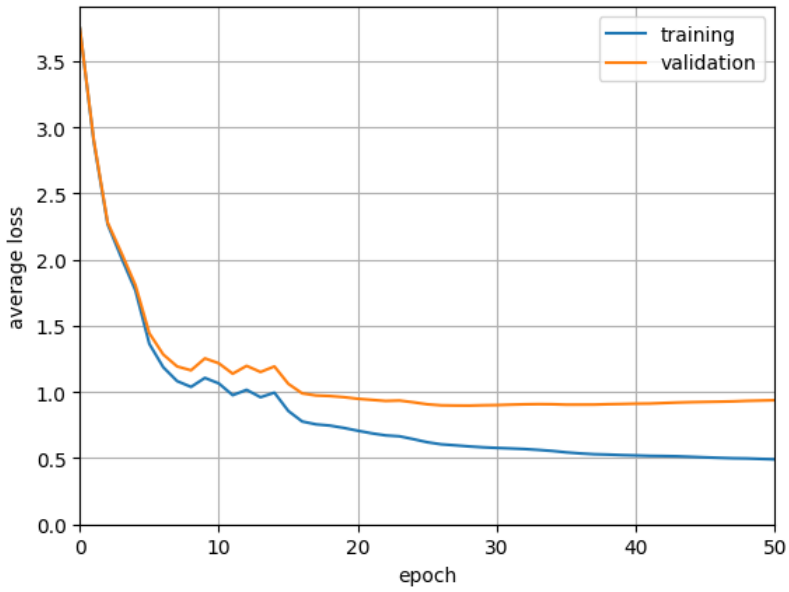

```
############################## Q 3.1 ##############################
max_iters = 50
# pick a batch size, learning rate
batch_size = 64
learning_rate = 10e-3
hidden_size = 64
##########################
##### your code here #####
##########################
np.random.seed(16720)

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

params = {}

# initialize layers
initialize_weights(train_x.shape[1], hidden_size, params, "layer1")
initialize_weights(hidden_size, train_y.shape[1], params, "output")
layer1_W_initial = np.copy(params["Wlayer1"]) # copy for Q3.3

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

    h1 = forward(valid_x,params,'layer1',sigmoid)
    probs = forward(h1,params,'output',softmax)
    loss, acc = compute_loss_and_acc(valid_y, probs)
    valid_loss.append(loss/valid_x.shape[0])
    valid_acc.append(acc)

    total_loss = 0
    avg_acc = 0
    for xb,yb in batches:
        # training loop can be exactly the same as q2!
        ##########################
        ##### your code here #####
        ##########################

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

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

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

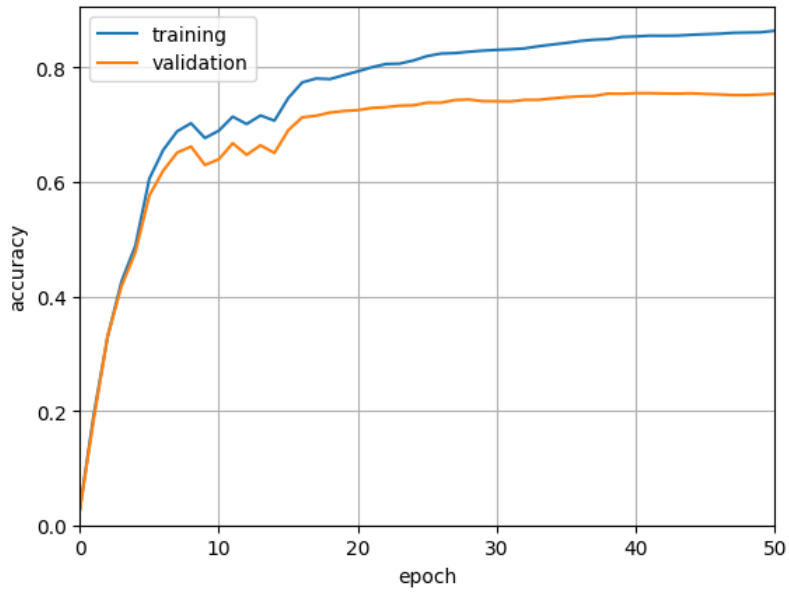
```
        # apply gradient to update the parameters
        params['Wlayer1'] -= learning_rate * params['grad_Wlayer1']
        params['blayer1'] -= learning_rate * params['grad_blayer1']
        params['Woutput'] -= learning_rate * params['grad_Woutput']
        params['boutput'] -= learning_rate * params['grad_boutput']

    avg_acc /= batch_num

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

# record final training and validation accuracy and loss
h1 = forward(train_x,params,'layer1',sigmoid)
probs = forward(h1,params,'output',softmax)
loss, acc = compute_loss_and_acc(train_y, probs)
train_loss.append(loss/train_x.shape[0])
train_acc.append(acc)

h1 = forward(valid_x,params,'layer1',sigmoid)
probs = forward(h1,params,'output',softmax)
loss, acc = compute_loss_and_acc(valid_y, probs)
valid_loss.append(loss/valid_x.shape[0])
valid_acc.append(acc)

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

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

```
itr: 00   loss: 35185.05   acc: 0.11
itr: 02   loss: 20512.77   acc: 0.46
itr: 04   loss: 14830.83   acc: 0.60
itr: 06   loss: 11698.65   acc: 0.69
itr: 08   loss: 10027.48   acc: 0.73
itr: 10   loss: 8791.77   acc: 0.76
itr: 12   loss: 7656.84   acc: 0.80
itr: 14   loss: 6788.12   acc: 0.82
itr: 16   loss: 6094.34   acc: 0.84
itr: 18   loss: 5545.46   acc: 0.86
itr: 20   loss: 5019.29   acc: 0.87
itr: 22   loss: 4666.72   acc: 0.88
itr: 24   loss: 4173.90   acc: 0.89
itr: 26   loss: 3874.07   acc: 0.90
itr: 28   loss: 3528.08   acc: 0.91
itr: 30   loss: 3315.56   acc: 0.92
itr: 32   loss: 3018.78   acc: 0.93
itr: 34   loss: 2888.20   acc: 0.93
itr: 36   loss: 2594.56   acc: 0.94
itr: 38   loss: 2374.85   acc: 0.95
itr: 40   loss: 2172.04   acc: 0.95
itr: 42   loss: 1991.89   acc: 0.96
itr: 44   loss: 1842.23   acc: 0.97
itr: 46   loss: 1718.83   acc: 0.97
itr: 48   loss: 1614.63   acc: 0.97
Validation accuracy:  0.7541666666666667
Test accuracy:  0.7561111111111111
```

```
# 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()
```



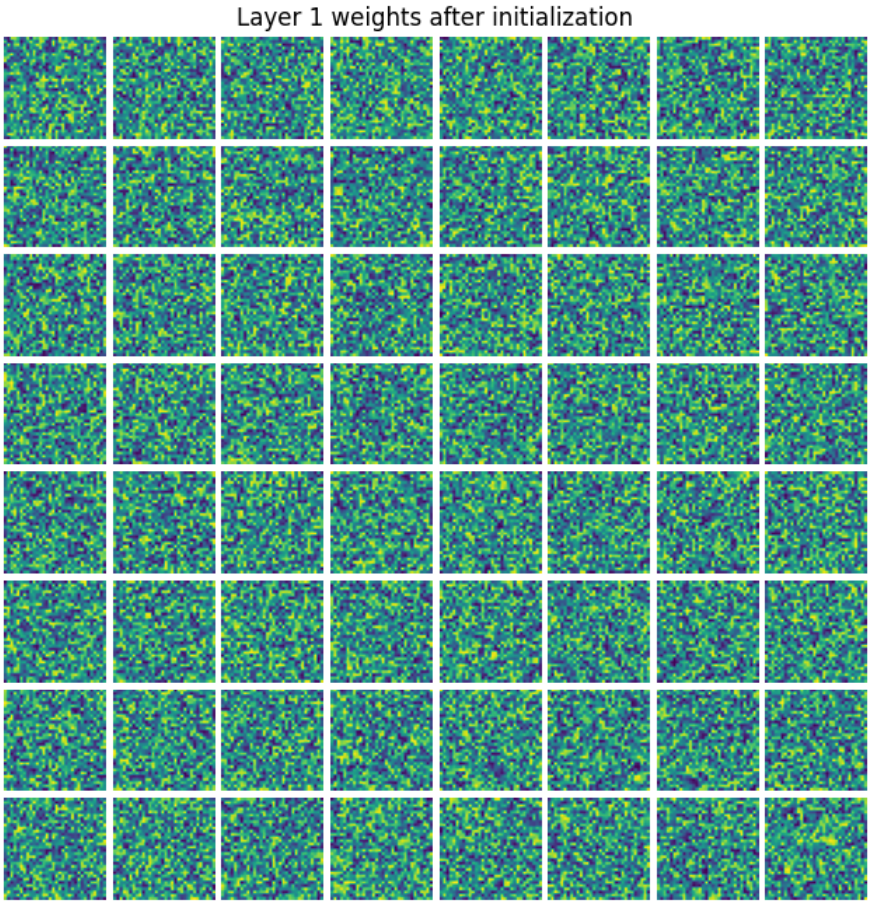


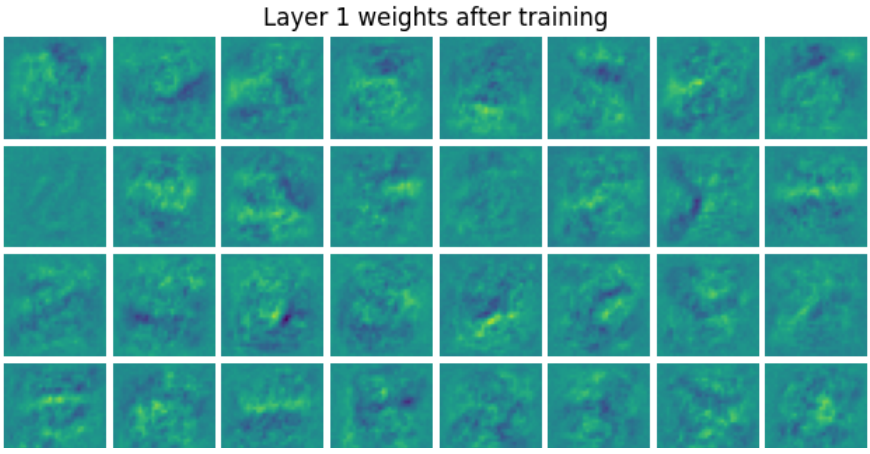
Q3.2 (3 points)

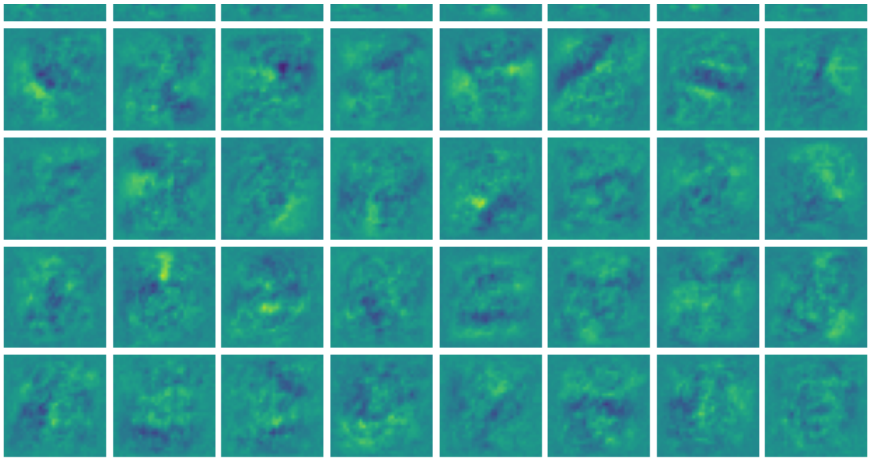
The provided code will visualize the first layer weights as 64 32x32 images, both immediately after initialization and after full training. Generate both visualizations. Comment on the learned weights and compare them to the initialized weights. Do you notice any patterns?

```
############################### 0.3.2 ###############################
```

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







From the figures above, we can see that the weights immediately after initialization has no discernable pattern and is just random noise. This is expected since we initialized it according to a uniform distribution using Xavier initialization.

The general pattern after learning is that each weight image appears to have learned an unique "template image" that is mostly different than the other weight images. There are also a few weight images that failed to learn (e.g. the image in the 2nd row 1st column), but most of the weight images have learned some distinct feature (e.g. horizontal/vertical/diagonal edges, abstract edges, textures, localized blobs, etc) that help in recognizing parts of the images in the dataset.

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.

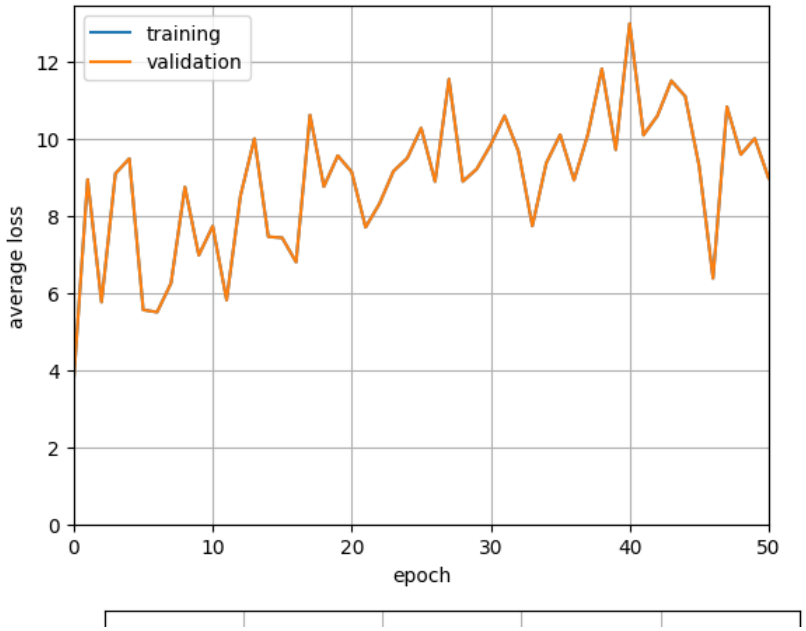
Include total of six plots (two will be the same from Q3.1). Comment on how the learning rates affect the training, and report the final accuracy of the best network on the test set. Hint: Use fixed random seeds to improve reproducibility.

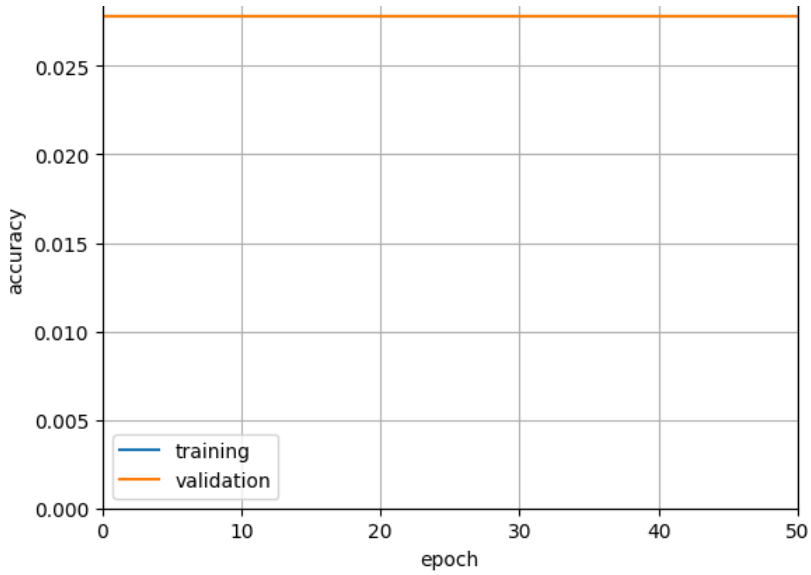
```
############################## Q 3.3 ##############################
#########################
##### your code here #####
#########################

max_iters = 50
# pick a batch size, learning rate
batch_size = 64
learning_rates = [100e-3, 1e-3, 10e-3]
hidden_size = 64

for learning_rate in learning_rates:
    print("learning rate: ", learning_rate)
    np.random.seed(16720)

    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 = []
```

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

    h1 = forward(valid_x,params,'layer1',sigmoid)
    probs = forward(h1,params,'output',softmax)
    loss, acc = compute_loss_and_acc(valid_y, probs)
    valid_loss.append(loss/valid_x.shape[0])
    valid_acc.append(acc)

    total_loss = 0
    avg_acc = 0
    for xb,yb in batches:
        # training loop can be exactly the same as q2!
        ##########################
        ##### your code here #####
        ##########################

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

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

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

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

    avg_acc /= batch_num

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

# record final training and validation accuracy and loss
h1 = forward(train_x,params,'layer1',sigmoid)
probs = forward(h1,params,'output',softmax)
loss, acc = compute_loss_and_acc(train_y, probs)
train_loss.append(loss/train_x.shape[0])
train_acc.append(acc)

h1 = forward(valid_x,params,'layer1',sigmoid)
probs = forward(h1,params,'output',softmax)
loss, acc = compute_loss_and_acc(valid_y, probs)
valid_loss.append(loss/valid_x.shape[0])
valid_acc.append(acc)

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

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

# plot loss curves
```

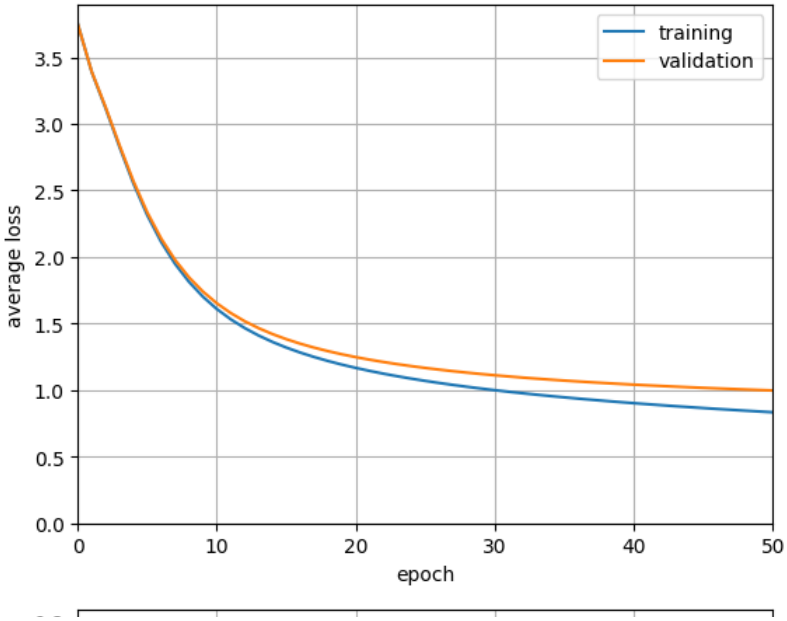
```
# plot loss curves
plt.plot(range(len(train_loss)), train_loss, label="training")
plt.plot(range(len(valid_loss)), valid_loss, label="validation")
plt.xlabel("epoch")
plt.ylabel("average loss")
plt.xlim(0, len(train_loss)-1)
plt.ylim(0, None)
plt.legend()
plt.grid()
plt.show()

# 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()
```

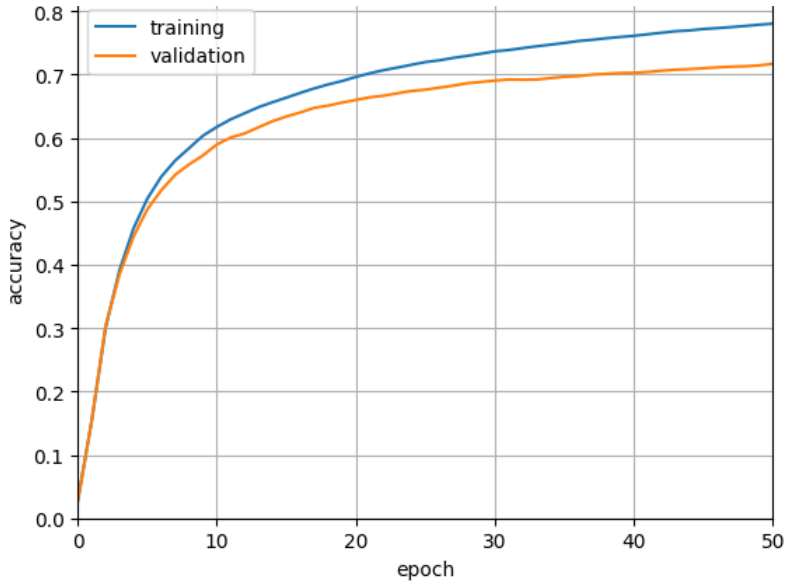
```
learning rate:  0.1
itr: 00   loss: 136967.93   acc: 0.03
itr: 02   loss: 103533.28   acc: 0.03
itr: 04   loss: 81415.23   acc: 0.03
itr: 06   loss: 80571.14   acc: 0.03
itr: 08   loss: 100006.85   acc: 0.03
itr: 10   loss: 102626.40   acc: 0.03
itr: 12   loss: 98516.54   acc: 0.03
itr: 14   loss: 100354.28   acc: 0.03
itr: 16   loss: 122693.74   acc: 0.03
itr: 18   loss: 122583.14   acc: 0.03
itr: 20   loss: 118588.38   acc: 0.03
itr: 22   loss: 121299.07   acc: 0.03
itr: 24   loss: 127692.45   acc: 0.03
itr: 26   loss: 121088.95   acc: 0.03
itr: 28   loss: 132873.89   acc: 0.03
itr: 30   loss: 123298.90   acc: 0.03
itr: 32   loss: 124769.24   acc: 0.03
itr: 34   loss: 117915.80   acc: 0.03
itr: 36   loss: 121652.18   acc: 0.03
itr: 38   loss: 132524.38   acc: 0.03
itr: 40   loss: 128179.83   acc: 0.03
itr: 42   loss: 121911.68   acc: 0.03
itr: 44   loss: 123999.99   acc: 0.03
itr: 46   loss: 116272.91   acc: 0.03
itr: 48   loss: 127831.32   acc: 0.03
Validation accuracy:  0.027777777777777776
Test accuracy:  0.027777777777777776
```



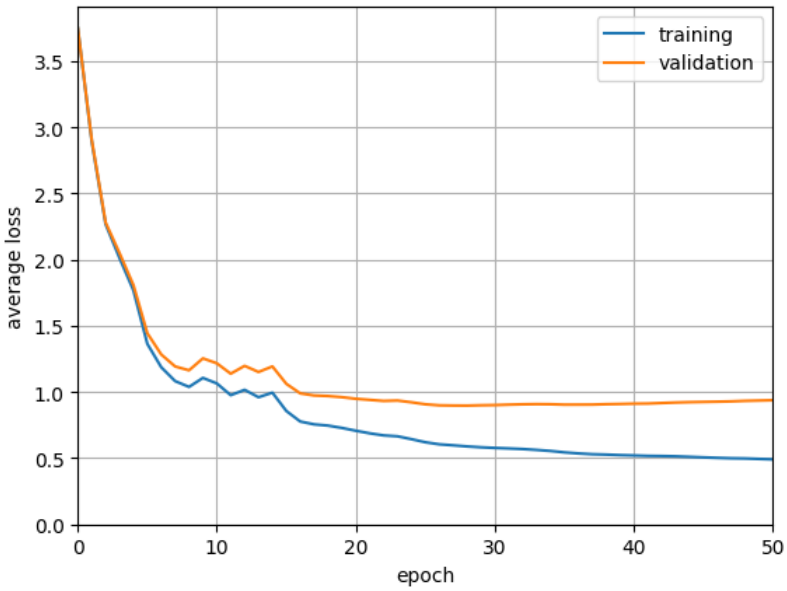


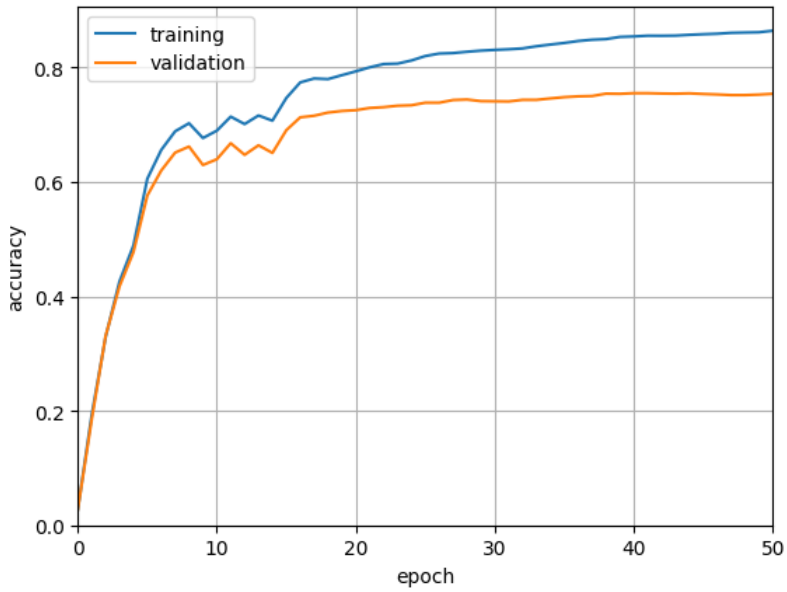

```
learning rate:  0.001
itr: 00   loss: 37646.12   acc: 0.09
itr: 02   loss: 31839.33   acc: 0.36
itr: 04   loss: 25917.78   acc: 0.50
itr: 06   loss: 21518.83   acc: 0.57
itr: 08   loss: 18532.18   acc: 0.61
itr: 10   loss: 16472.70   acc: 0.64
itr: 12   loss: 15000.49   acc: 0.67
itr: 14   loss: 13898.69   acc: 0.68
itr: 16   loss: 13037.81   acc: 0.70
itr: 18   loss: 12339.87   acc: 0.71
itr: 20   loss: 11756.55   acc: 0.72
itr: 22   loss: 11256.95   acc: 0.73
itr: 24   loss: 10820.56   acc: 0.74
itr: 26   loss: 10433.31   acc: 0.75
itr: 28   loss: 10085.17   acc: 0.76
itr: 30   loss: 9768.83   acc: 0.77
itr: 32   loss: 9478.79   acc: 0.77
itr: 34   loss: 9210.81   acc: 0.78
itr: 36   loss: 8961.60   acc: 0.78
itr: 38   loss: 8728.51   acc: 0.79
itr: 40   loss: 8509.41   acc: 0.80
itr: 42   loss: 8302.57   acc: 0.80
itr: 44   loss: 8106.54   acc: 0.81
itr: 46   loss: 7920.12   acc: 0.81
itr: 48   loss: 7742.29   acc: 0.81
Validation accuracy:  0.7166666666666667
Test accuracy:  0.7327777777777778
```






```
learning rate:  0.01
itr: 00   loss: 35185.05   acc: 0.11
itr: 02   loss: 20512.77   acc: 0.46
itr: 04   loss: 14830.83   acc: 0.60
itr: 06   loss: 11698.65   acc: 0.69
itr: 08   loss: 10027.48   acc: 0.73
itr: 10   loss: 8791.77   acc: 0.76
itr: 12   loss: 7656.84   acc: 0.80
itr: 14   loss: 6788.12   acc: 0.82
itr: 16   loss: 6094.34   acc: 0.84
itr: 18   loss: 5545.46   acc: 0.86
itr: 20   loss: 5019.29   acc: 0.87
itr: 22   loss: 4666.72   acc: 0.88
itr: 24   loss: 4173.90   acc: 0.89
itr: 26   loss: 3874.07   acc: 0.90
itr: 28   loss: 3528.08   acc: 0.91
itr: 30   loss: 3315.56   acc: 0.92
itr: 32   loss: 3018.78   acc: 0.93
itr: 34   loss: 2888.20   acc: 0.93
itr: 36   loss: 2594.56   acc: 0.94
itr: 38   loss: 2374.85   acc: 0.95
itr: 40   loss: 2172.04   acc: 0.95
itr: 42   loss: 1991.89   acc: 0.96
itr: 44   loss: 1842.23   acc: 0.97
itr: 46   loss: 1718.83   acc: 0.97
itr: 48   loss: 1614.63   acc: 0.97
Validation accuracy:  0.7541666666666667
Test accuracy:  0.7561111111111111
```





When the learning rate is 10 times my tuned learning rate, the training becomes very poor and the accuracy stays low. This is because the learning rate is too high and consistently oversteps the local minimum (as seen from the zigzagging average loss plot).

When the learning rate is one-tenth of my tuned learning rate, training does occur and the loss goes down over epochs. However, the learning rate is very slow as we are only making baby steps in the direction of the local minimum. This leads to a worse off performance over 50 epochs as we have not sufficiently converged to the local minimum.

My tuned learning rate of 0.01 seems to work the best out of the three. It is a tradeoff between fast convergence (which requires a large learning rate) against accurate convergence (which requires a small learning rate).

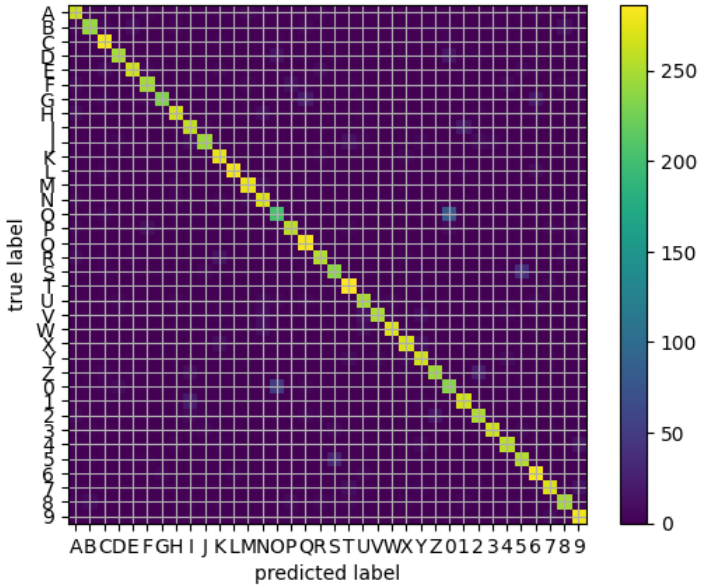
The final accuracy obtained is the following: **Validation accuracy: 75.42%**, **Test accuracy: 75.61%**. It is achieved with 50 iterations, a batch size of 64, and a learning rate of 0.01.

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.

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

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

h1 = forward(train_x, params, 'layer1', sigmoid)
probs = forward(h1, params, 'output', softmax)
pred_labels = np.argmax(probs, axis=1)
true_labels = np.argmax(train_y, axis=1)

for true, pred in zip(true_labels, pred_labels):
    confusion_matrix[true, pred] += 1

# visualize confusion matrix
import string
plt.imshow(confusion_matrix,interpolation='nearest')
plt.grid()
plt.xticks(np.arange(36),string.ascii_uppercase[:26] + ''.join([str(_) for _ in range(10)]))
plt.yticks(np.arange(36),string.ascii_uppercase[:26] + ''.join([str(_) for _ in range(10)]))
plt.xlabel("predicted label")
plt.ylabel("true label")
plt.colorbar()
```

```
plt.show()
```



The top few pairs of classes that are most commonly confused are the number '0' with the letter 'O', and the number '5' with the letter 'S'. Additionally, the number '1' with the letter 'I', and the number '2' with the letter 'Z' are also confused, albeit to a lesser extent. Heuristically this makes sense since the strokes of the confused characters are very similar in appearance, thereby causing the confusion as they are harder to tell apart.

Q4 Object Detection and Tracking

Initialization

Run the following code, which imports the modules you'll need and defines helper functions you may need to use later in your implementations.

```
import cv2
import numpy as np
import torchvision
from torchvision.io.image import read_image
from torchvision.models.detection import fasterrcnn_resnet50_fpn_v2, FasterRCNN_ResNet50_FPN_V2_Weights
import torch
import argparse
from PIL import Image
from matplotlib import pyplot as plt
import os
import glob

# Utility functions

coco_names = [
    '__background__', 'person', 'bicycle', 'car', 'motorcycle', 'airplane', 'bus',
    'train', 'truck', 'boat', 'traffic light', 'fire hydrant', 'N/A', 'stop sign',
    'parking meter', 'bench', 'bird', 'cat', 'dog', 'horse', 'sheep', 'cow',
    'elephant', 'bear', 'zebra', 'giraffe', 'N/A', 'backpack', 'umbrella', 'N/A', 'N/A',
    'handbag', 'tie', 'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball',
    'kite', 'baseball bat', 'baseball glove', 'skateboard', 'surfboard', 'tennis racket',
    'bottle', 'N/A', 'wine glass', 'cup', 'fork', 'knife', 'spoon', 'bowl',
    'banana', 'apple', 'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog', 'pizza',
    'donut', 'cake', 'chair', 'couch', 'potted plant', 'bed', 'N/A', 'dining table',
    'N/A', 'N/A', 'toilet', 'N/A', 'tv', 'laptop', 'mouse', 'remote', 'keyboard', 'cell phone',
    'microwave', 'oven', 'toaster', 'sink', 'refrigerator', 'N/A', 'book',
```

```
    'clock', 'vase', 'scissors', 'teddy bear', 'hair drier', 'toothbrush'
]

# Create different colors for each class.
COLORS = np.random.uniform(0, 255, size=(len(coco_names), 3))

def draw_boxes(boxes, labels, image):
    """
    Draws the bounding box around a detected object, also with labels
    """
    image = image.copy()
    lw = max(round(sum(image.shape) / 2 * 0.003), 2)  # Line width
    tf = max(lw - 1, 1) # Font thickness.
    for i, box in enumerate(boxes):
        color = COLORS[labels[i]]
        cv2.rectangle(
            img=image,
            pt1=(int(box[0]), int(box[1])),
            pt2=(int(box[2]), int(box[3])),
            color=color[::-1],
            thickness=lw
        )
        cv2.putText(
            img=image,
            text=coco_names[labels[i]],
            org=(int(box[0]), int(box[1]-5)),
            fontFace=cv2.FONT_HERSHEY_SIMPLEX,
            fontScale=lw / 3,
            color=color[::-1],
            thickness=tf,
            lineType=cv2.LINE_AA
        )
    return image


def draw_single_track(all_frames, track, track_idx):
    """
    Visualize a track
    """
    image_vis_list = []
    start_frame = track['start_frame']
    num_frames_in_track = len(track['bboxes'])
    print('Visualizing track {} with {} frames, starting from frame {}'.format(track_idx, num_frames_in_track, start_f

    for track_frame_num in range(num_frames_in_track):
        frame_num = start_frame + track_frame_num
        image, _, _, _ = all_frames[frame_num]
        bbox = track['bboxes'][track_frame_num]
        image_viz = image.copy()

        # print('Frame: {}, Bbox: {}'.format(frame_num, bbox))
        cv2.rectangle(image_viz, (bbox[0], bbox[1]), (bbox[2], bbox[3]), (0, 255, 0), 4)

        xcentroid, ycentroid = (bbox[0] + bbox[2]) // 2, (bbox[1] + bbox[3]) // 2
        text = "ID {}".format(track_idx)

        cv2.putText(image_viz, text, (xcentroid - 10, ycentroid - 10), cv2.FONT_HERSHEY_SIMPLEX, 1.0, (0, 255, 0), 3)
        cv2.circle(image_viz, (xcentroid, ycentroid), 6, (0, 255, 0), -1)

        image_vis_list.append(image_viz)

    return image_vis_list


def draw_multi_tracks(all_frames, tracks):
    """
    Visualize multiple tracks
    """
    # Mapping from frame number to a list of (bbox, track_idx) tuples
    viz_per_frame = {}

    # Image visualization list
```

```
    image_vis_list = []

    # Track idx to color (each track idx has a color)
    track_to_color = {}

    # Loop through the tracks and got the proper info
    for track_idx, track in enumerate(tracks):
        start_frame = track['start_frame']
        num_frames_in_track = len(track['bboxes'])
        print('Visualizing track {} with {} frames, starting from frame {}'.format(track_idx, num_frames_in_track, sta

        for track_frame_num in range(num_frames_in_track):
            frame_num = start_frame + track_frame_num
            bbox = track['bboxes'][track_frame_num]

            if frame_num not in viz_per_frame:
                viz_per_frame[frame_num] = []
            viz_per_frame[frame_num].append((bbox, track_idx))

    # Loop through the frames and draw the boxes
    for frame_num, (image, bboxes, confidences, class_ids) in enumerate(all_frames):
        image_viz = image.copy()

        if frame_num not in viz_per_frame:
            continue

        for bbox, track_idx in viz_per_frame[frame_num]:
            if track_idx not in track_to_color:
                track_to_color[track_idx] = np.random.randint(0, 255, size=3)

            color = track_to_color[track_idx]
            color = (int(color[0]), int(color[1]), int(color[2]))

            cv2.rectangle(image_viz, (bbox[0], bbox[1]), (bbox[2], bbox[3]), color, 4)

            xcentroid, ycentroid = (bbox[0] + bbox[2]) // 2, (bbox[1] + bbox[3]) // 2
            text = "ID {}".format(track_idx)

            cv2.putText(image_viz, text, (xcentroid - 15, ycentroid - 15), cv2.FONT_HERSHEY_SIMPLEX, 1.5, color, 4)
            cv2.circle(image_viz, (xcentroid, ycentroid), 10, color, -1)

        image_vis_list.append(image_viz)

    return image_vis_list
```

Set up data

```
if not os.path.exists('car_frames_simple.zip'):
  !wget https://www.andrew.cmu.edu/user/kvuong/car_frames_simple.zip -O car_frames_simple.zip
  !unzip -qq "car_frames_simple.zip"
  print("downloaded and unzipped data")
```

```
--2024-11-12 00:03:08--  https://www.andrew.cmu.edu/user/kvuong/car_frames_simple.zip
Resolving www.andrew.cmu.edu (www.andrew.cmu.edu)... 128.2.42.53
Connecting to www.andrew.cmu.edu (www.andrew.cmu.edu)|128.2.42.53|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 10778523 (10M) [application/zip]
Saving to: ‘car_frames_simple.zip’

car_frames_simple.z 100%[===================>]  10.28M  1.98MB/s    in 5.4s

2024-11-12 00:03:14 (1.91 MB/s) - ‘car_frames_simple.zip’ saved [10778523/10778523]

downloaded and unzipped data
```

Problem 4.1: Object Detection with Faster-RCNN

```
def get_model(device):
```

```
    """
    Load the pretrained model + inference transform
    """
    # Load the model
    weights = FasterRCNN_ResNet50_FPN_V2_Weights.DEFAULT
    model = fasterrcnn_resnet50_fpn_v2(weights=weights)
    # Load the model onto the computation device
    model = model.eval().to(device)
    # inference transform
    preprocess = weights.transforms()

    return model, preprocess


def predict(image, model, device, detection_threshold):
    """
    Predicts bounding boxes, scores, and class labels for objects detected in an image.
    Only returns detections with confidence above the specified threshold.

    Args:
        image (torch.Tensor): The input image tensor.
        model (torchvision.models.detection.FasterRCNN): The object detection model.
        device (torch.device): The device to perform computations on.
        detection_threshold (float): Confidence threshold for filtering detections.

    Returns:
        boxes (numpy.ndarray): Bounding boxes of detected objects above the confidence threshold. Shape (N, 4),
            where N is the number of detections. Bbox format: (x1, y1, x2, y2)
        scores (numpy.ndarray): Confidence scores for the detected objects. Shape (N,)
        labels (numpy.ndarray): Class labels for the detected objects. Shape (N,)
    """

    # TODO: Move the input image to the specified device (GPU)
    image = image.to(device)

    # TODO: Add a batch dimension to the image tensor
    batch = image.unsqueeze(0)

    # TODO: Run the forward pass (with torch.no_grad()) to get model outputs
    with torch.no_grad():
        outputs = model(batch)

    # TODO: Extract the scores, bounding boxes, and labels from the model outputs
    boxes = outputs[0]['boxes'].cpu().numpy()
    scores = outputs[0]['scores'].cpu().numpy()
    labels = outputs[0]['labels'].cpu().numpy()

    # TODO: Apply the detection threshold to filter out low-confidence predictions
    indices = np.where(scores >= detection_threshold)[0]
    boxes = boxes[indices]
    scores = scores[indices]
    labels = labels[indices]

    return boxes, scores, labels


def run_detector(image_path, model, preprocess, det_threshold=0.9):
    """
    Runs the object detector on a given image and retrieves bounding boxes, confidence scores,
    and class labels for detected objects.

    Args:
        image_path (str): Path to the image file to detect objects in.
        model (torchvision.models.detection.FasterRCNN): The object detection model.
        preprocess (callable): Preprocessing function for the image.
        det_threshold (float): Confidence threshold for detections.

    Returns:
        image_np (numpy.ndarray): Original image in numpy array format (for visualization later)
        bboxes (numpy.ndarray): Bounding boxes of detected objects.
        confidences (numpy.ndarray): Confidence scores for the detected objects.
        class_ids (numpy.ndarray): Class labels for the detected objects.
```

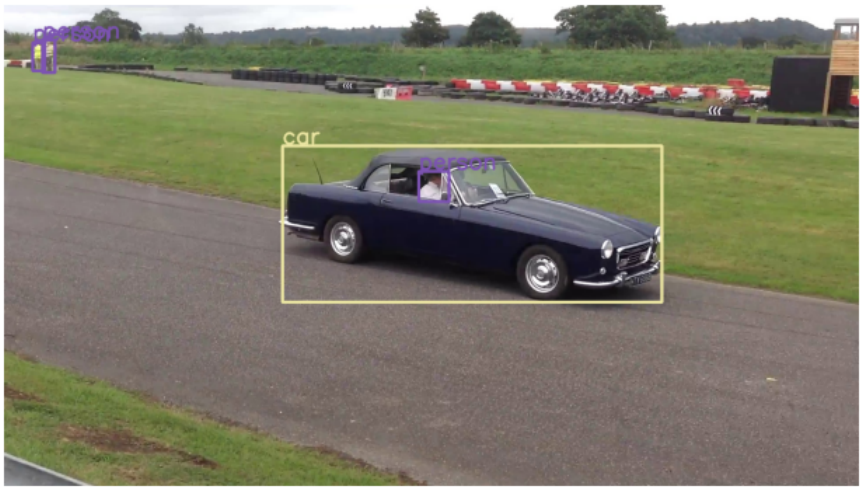
```
    """
    # Read image to tensor (0-255 uint8)
    image_torch = read_image(image_path)
    image_np = image_torch.permute(1, 2, 0).numpy()

    # TODO: Apply the preprocess to preprocess the image (normalization, etc.) (see more at https://pytorch.org/vision
    image_processed = preprocess(image_torch)

    # TODO: Run the predict function on image_processed to obtain bounding boxes, scores, and class IDs
    bboxes, confidences, class_ids = predict(image_processed, model, device, det_threshold)

    return (image_np, bboxes, confidences, class_ids)


# Define the computation device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
if device == 'cpu':
  print('!!! WARNING: USING CPU ONLY, THIS WILL BE VERY SLOW !!!')

# First, load the model and preprocessor
model, preprocess = get_model(device)
```

```
Downloading: "https://download.pytorch.org/models/fasterrcnn_resnet50_fpn_v2_coco-dd69338a.pth" to /root/.cache/to
100%|██████████| 167M/167M [00:03<00:00, 54.8MB/s]
```

```
# TODO: either use wget or manually upload the image to temporary storage (please don't use the same image as the examp
# My hometown Zhengzhou :-)
!wget "https://upload.wikimedia.org/wikipedia/commons/e/eb/20220907_Huanghe_Road_in_Zhengzhou.jpg" -O example.png
image_path = "example.png"

# Run the detector on the image
output_det = run_detector(image_path, model, preprocess, det_threshold=0.9)
image, bboxes, confidences, class_ids = output_det
image_with_boxes = draw_boxes(bboxes, class_ids, image)
plt.imshow(image_with_boxes)
plt.axis('off')
plt.tight_layout()
plt.show()
```

```
--2024-11-12 00:03:26--  https://upload.wikimedia.org/wikipedia/commons/e/eb/20220907_Huanghe_Road_in_Zhengzhou.jp
Resolving upload.wikimedia.org (upload.wikimedia.org)... 208.80.153.240, 2620:0:860:ed1a::2:b
Connecting to upload.wikimedia.org (upload.wikimedia.org)|208.80.153.240|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1871417 (1.8M) [image/jpeg]
Saving to: ‘example.png’

example.png         100%[===================>]   1.78M  4.83MB/s    in 0.4s

2024-11-12 00:03:27 (4.83 MB/s) - ‘example.png’ saved [1871417/1871417]
```

```
# TODO: run object detector on every image inside the data folder
image_folder = "./car_frames_simple"
image_paths = sorted(glob.glob(os.path.join(image_folder, "*.jpg")))

output_detections = []
for image_path in image_paths:
    output_det = run_detector(image_path, model, preprocess, det_threshold=0.9)
    output_detections.append(output_det)

# Visualize a few images (first and last image for example)
indices = [0, len(output_detections) - 1]
for idx in indices:
    image, bboxes, confidences, class_ids = output_detections[idx]
    image_with_boxes = draw_boxes(bboxes, class_ids, image)
    plt.imshow(image_with_boxes)
    plt.axis('off')
    plt.tight_layout()
    plt.show()
```



We visualize above a few images in the **car_frames_simple** folder and Huanghe road in Zhengzhou, China (my hometown). We notice that in general, increasing the confidence threshold decreases the number of observations, and vice versa. Additionally, when the confidence threshold is decreased to approximately below 0.6, some of the entities are starting to get incorrect labels. Again, this is to be expected since a lower confidence threshold means that entities that have a higher label uncertainty are not being discarded. When the confidence threshold is increased to above 0.99, we get almost no detections since the confidence requirement is too strict.

Problem 4.2: Multi-object-tracking with IOU-based tracker

Compute IOU

```
def iou(bbox1, bbox2):
    """
    Calculates the intersection-over-union of two bounding boxes.

    Args:
        bbox1 (numpy.array, list of floats): bounding box in format x1,y1,x2,y2.
        bbox2 (numpy.array, list of floats): bounding box in format x1,y1,x2,y2.

    Returns:
        float: intersection-over-onion of bbox1, bbox2
    """

    # TODO: Calculate the coordinates for the intersection rectangle
    x_left = max(bbox1[0], bbox2[0])
    y_top = max(bbox1[1], bbox2[1])
    x_right = min(bbox1[2], bbox2[2])
    y_bottom = min(bbox1[3], bbox2[3])

    # TODO: Return 0 if there's no overlap. If yes, calculate the ratio of the overlap to each ROI size and the unifie
    if x_right < x_left or y_bottom < y_top:
        return 0

    bbox1_area = (bbox1[2] - bbox1[0]) * (bbox1[3] - bbox1[1])
    bbox2_area = (bbox2[2] - bbox2[0]) * (bbox2[3] - bbox2[1])

    size_intersection = (x_right - x_left) * (y_bottom - y_top)
    size_union = bbox1_area + bbox2_area - size_intersection

    return size_intersection / size_union
```

IOU-based Tracker

```
def track_iou(detections, sigma_l, sigma_h, sigma_iou, t_min):
    """
    Implements a simple IoU-based multi-object tracker. Matches detections to existing tracks based on IoU.
    Detections with IoU above a threshold are linked to existing tracks; otherwise, new tracks are created.

    See "High-Speed Tracking-by-Detection Without Using Image Information by Bochinski et al. for
    more information.

    Args:
         detections (list): list of detections per frame
         sigma_l (float): low detection threshold.
         sigma_h (float): high detection threshold.
         sigma_iou (float): IOU threshold.
         t_min (float): minimum track length in frames.

    Returns:
        list: list of tracks. Each track is a dict containing 'bboxes': a list of bounding boxes, 'max_score': the
        maximum detection score, and 'start_frame': the frame index of the first detection.
    """

    # Initialize an empty list to store active and completed tracks
    tracks_active = []
    tracks_finished = []

    # Loop over each frame's detections
    for frame_num, detections_frame in enumerate(detections):
        # TODO: Apply low threshold sigma_l to filter low-confidence detections
        dets = []
        for det in detections_frame:
            if det['score'] >= sigma_l:
                dets.append(det)

        updated_tracks = []
```

```
        for track in tracks_active:
            track_updated = False

            # If there are detections for this frame
            if len(dets) > 0:
                # TODO: get det with highest iou
                best_match = None
                max_iou = 0

                for det in dets:
                    current_iou = iou(track['bboxes'][-1], det['bbox'])
                    if current_iou >= max_iou:
                        max_iou = current_iou
                        best_match = det

                # If IoU of best_match, exceeds sigma_iou, then extend the track by adding the detection to the track,
                # update the max_score, then remove that detection from the dets. Remember to set track_updated to Tru
                if iou(track['bboxes'][-1], best_match['bbox']) >= sigma_iou:
                    # TODO: fill in the code here ...
                    track['bboxes'].append(best_match['bbox'])
                    track['max_score'] = max(track['max_score'], best_match['score'])
                    dets.remove(best_match)
                    track_updated = True
                    updated_tracks.append(track)

            # If track was not updated
            if not track_updated:
                # TODO: finish track when the conditions are met by appending the track to tracks_finished
                if track['max_score'] >= sigma_h and len(track['bboxes']) >= t_min:
                    tracks_finished.append(track)

        # create new tracks
        new_tracks = [{'bboxes': [det['bbox']], 'max_score': det['score'], 'start_frame': frame_num} for det in dets]
        tracks_active = updated_tracks + new_tracks

    # finish all remaining active tracks
    tracks_finished += [track for track in tracks_active
                        if track['max_score'] >= sigma_h and len(track['bboxes']) >= t_min]

    return tracks_finished
```

Run Tracker

```
def run_tracker(frames):
    # Track objects in the video
    detections = []
    for frame_num, (image, bboxes, confidences, class_ids) in enumerate(frames):
        dets = []
        for bbox, confidence, class_id in zip(bboxes, confidences, class_ids):
            dets.append({'bbox': (bbox[0], bbox[1], bbox[2], bbox[3]),
                         'score': confidence,
                         'class': class_id})
        detections.append(dets)

    print('Running tracker...')
    tracks = track_iou(detections, sigma_l=0.4, sigma_h=0.7, sigma_iou=0.3, t_min=2)
    print('Tracker finished!')
    return tracks


# TODO: From the detections, run the tracker to obtain a list of tracks
output_tracks = run_tracker(output_detections)

# Convert bbox values into integers as draw_multi_tracks uses cv2 functions which cannot parse floats
for track in output_tracks:
    for i in range(len(track['bboxes'])):
        track['bboxes'][i] = [int(coord) for coord in track['bboxes'][i]]

# Visualize the tracks
image_vis_list = draw_multi_tracks(output_detections, output_tracks)
```

```
# TODO: Visualize a few images (here we show first, middle, and last image for example)
indices = [0, len(output_detections) // 2, len(output_detections) - 1]
for idx in indices:
    plt.imshow(image_vis_list[idx])
    plt.axis('off')
    plt.tight_layout()
    plt.show()
```

```
Running tracker...
Tracker finished!
Visualizing track 0 with 20 frames, starting from frame 0
Visualizing track 1 with 20 frames, starting from frame 0
```

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")
```

```
--2024-11-12 00:04:11--  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   363KB/s    in 6.9s

2024-11-12 00:04:18 (459 KB/s) - '/content/images/images.zip' saved [3248168/3248168]

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

```
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?

Two major assumptions are:

1. Each letter is assumed to be fully connected (e.g. the letter i is not fully connected as it has a dot on top) and isolated (i.e. adjacent letters are not connected).
2. Each letter is assumed to have roughly the same pixel dimensions (or else blurring and thresholding could filter them out).

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.

```
############################## Q 5.2 ##############################
def findLetters(image):
    """
    takes a color image
    returns a list of bounding boxes and black_and_white image
    """
    bboxes = []
    bw = None
```

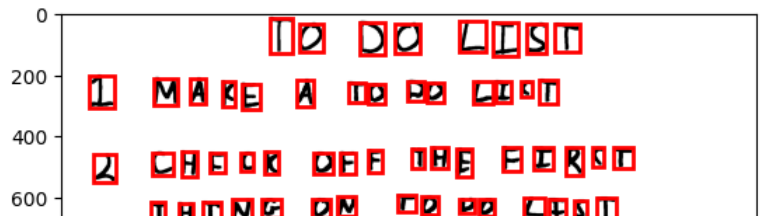
```
    # insert processing in here
    # one idea estimate noise -> denoise -> greyscale -> threshold -> morphology -> label -> skip small boxes
    # this can be 10 to 15 lines of code using skimage functions

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

    # Can optionally add noise, but not necessary for the given datasets and will worsen performance
    # image = skimage.filters.gaussian(image, sigma=1)
    image = skimage.restoration.denoise_bilateral(image, channel_axis=-1)

    gray_image = skimage.color.rgb2gray(image)
    threshold = skimage.filters.threshold_otsu(gray_image)
    bw = gray_image < threshold

    # Remove small objects and close small gaps in characters
    bw = skimage.morphology.remove_small_objects(bw, min_size=256)
    bw = skimage.morphology.closing(bw, footprint=skimage.morphology.square(5))

    # Making letters thicker
    bw = skimage.morphology.dilation(bw, footprint=skimage.morphology.square(8))

    # Find bboxes
    labeled_image = skimage.measure.label(bw, connectivity=2, background=0)
    for region in skimage.measure.regionprops(labeled_image):
        if region.area >= 256:
            bboxes.append(region.bbox)

    # Invert the colors as nist36 has 1 as black and 0 as white
    bw = 1.0 - bw

    return bboxes, bw
```

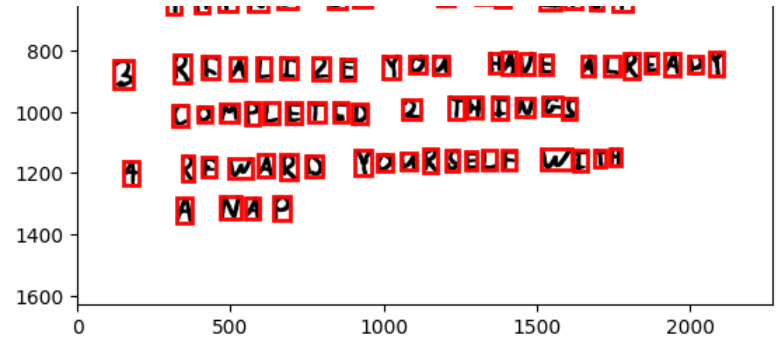
Q5.3 (Extra Credit) (3 points)

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.

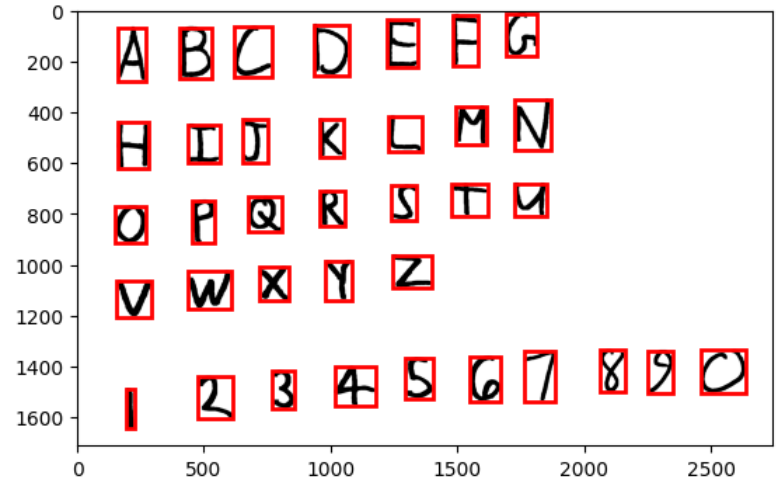
```
############################## Q 5.3 ##############################
# do not include any more libraries here!
# no opencv, no sklearn, etc!
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=UserWarning)

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

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

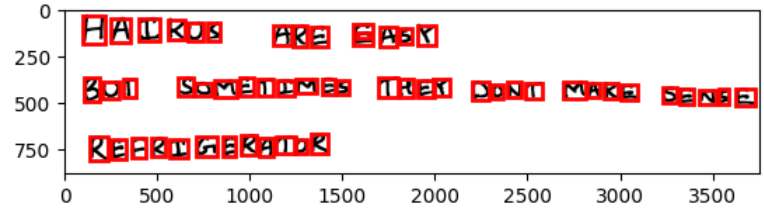




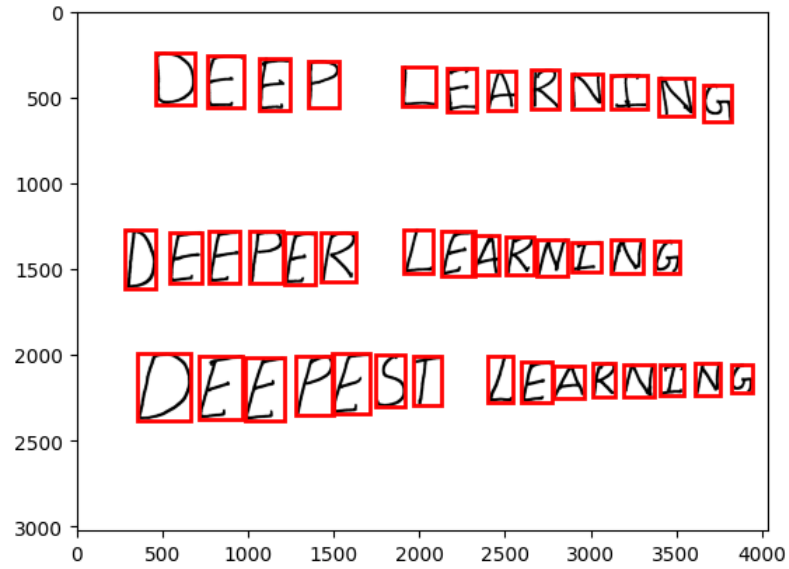
02_letters.jpg



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.

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

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

    # Determine average height of letters to approximate row height
    letter_heights = [bbox[2] - bbox[0] for bbox in bboxes]
    avg_letter_height = sum(letter_heights) / len(letter_heights)

    # Sort the boxes by rows
    bboxes.sort(key=lambda x: x[2])
    bboxes_by_row = [[]]
    curr_row = 0
    curr_base = bboxes[0][2]

    for bbox in bboxes:
        minr, minc, maxr, maxc = bbox

        if maxr > (curr_base + avg_letter_height):
            curr_row += 1
            curr_base = maxr
            bboxes_by_row.append([])
        bboxes_by_row[curr_row].append(bbox)

    # Sort the boxes by columns
    for row in bboxes_by_row:
        row.sort(key=lambda x: x[1])

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

    crops = []
    for row in bboxes_by_row:
        crops_in_row = []

        for bbox in row:
            minr, minc, maxr, maxc = bbox
            cropped_letter = bw[minr:maxr, minc:maxc]
            resized_letter = skimage.transform.resize(cropped_letter, (24, 24))
            padded_letter = np.pad(resized_letter, ((4, 4), (4, 4)), mode='constant', constant_values=1)

            transposed_letter = np.transpose(padded_letter)
            flattened_letter = transposed_letter.flatten()
            crops_in_row.append(flattened_letter.reshape(1,-1))

        crops.append(crops_in_row)

    # load the weights
    # run the crops through your neural network and print them out
    import pickle
```

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

for row in crops:

    row_output = ''

    for letter in row:

        h1 = forward(letter, params, 'layer1', sigmoid)
        probs = forward(h1, params, 'output', softmax)
        prediction = np.argmax(probs)
        row_output += letters[prediction]
    print(row_output)


01_list.jpg
TQDQLIST
INAKEATOZQLZET
2CHIRKDFF7HEFIRQT
THTNGQNT0UOLI5T
3RIALI2EYQUHAWEALRGAUY
CQMPLETED2YHING3
QR8WARDY0URXELFWITH
ANAP

02_letters.jpg
ABCDEFG
HIIKLMN
Q8QKSTM
VWXYZ
BX3FSQ78PQ

03_haiku.jpg
HAIKUSAREEMASK
EUTSQMETIMESTHEXDDWTMAKZSRNGR
REFRIGERATQR

04_deep.jpg
DEEYLEARMING
DEEPEKL5AKNING
DE55ESTLEARNING
```

01_list.jpg:

TQ DQ LIST

I NAKE A TO ZQ LZET

2 CHIRK DFF 7HE FIRQT

THTNG QN T0 UO LI5T

3 RIALI2E YQU HAWE ALRGAUY

CQMPLETED 2 YHING3

Q R8WARD Y0URXELF WITH

A NAP

Accuracy: 72.07%

02_letters.jpg

A B C D E F G

H I I K L M N

Q 8 Q K S T M

V W X Y Z

B X 3 F S Q 7 8 P Q

Accuracy: 66.67%

03_haiku.jpg

HAIKUS ARE EMASK

EUT SQMETIMES THEX DDWT MAKZ SRNGR

REFRIGERATQR

Accuracy: 78.18%

04_deep.jpg

DEEY LEARMING

DEEPEK L5AKNING

DE55EST LEARNING

Accuracy: 82.93%
