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

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

Instructions

1. **Integrity and collaboration:** Students are encouraged to work in groups but each student must submit their own work. Include the names of your collaborators in your write-up. Code should **NOT** be shared or copied. Please **DO NOT** use external code unless permitted. Plagiarism is prohibited and may lead to failure in this course.
2. Answer each question (for points) marked with a **Q** in the corresponding titled boxes.
3. **Start early!** This homework may take a long time to complete.
4. **Attempt to verify your implementation as you proceed.** If you don't verify that your implementation is correct on toy examples, you will risk having a huge mess when you put everything together.
5. **In your PDF, start a new page for each question and indicate the answer/page(s) correspondence carefully when submitting on Gradescope.** For some questions, this may leave a lot of blank space. If you skip a written question, just submit a blank page for it. This makes your work much easier to grade.
6. **Some questions will ask you to "Include your code in the writeup".** For those questions, you can either copy/paste the code into a `verbatim` environment or include screenshots of your code.
7. If you have any questions or need clarifications, please post in Slack or visit the TAs during office hours.
8. **Submission:** The submission is on Gradescope, **you will be submitting both your writeup and code zip file**. The zip file, `<andrew-id.zip>` contains your code and any results files we ask you to save. **Note: You have to submit your write-up separately to Gradescope, and include results (and code snippets, when requested) in the write-up**.
9. **Do not** submit anything from the `data/` folder in your submission.
10. For your code submission, **do not** use any libraries other than *numpy, scipy, scikit-image, matplotlib* and (in the appropriate section) *pytorch*. Including other libraries (for example, cv2, etc.) **may lead to loss of credit** on the assignment.

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

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

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

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

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

13. Feel free to use

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

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

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

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

Contents

1 Theory

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

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

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

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

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

Q1.1

1.Prove: As $softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$, then $softmax(x_i + c) = \frac{e^{x_i+c}}{\sum_j e^{x_j+c}} = \frac{e^{x_i} * e^c}{\sum_j e^{x_j} * e^c}$
For $e^c! = 0$, then divide both numerators and denominators by e^c, $softmax(x_i + c) = \frac{e^{x_i}}{\sum_j e^{x_j}}$ $= softmax(x_i)$, so $softmax(x) = softmax(x + c) \quad \forall c \in \mathbb{R}$.
2. When c=0, e^{x_i} is from 0 to infinity When $c = -\max x_i$, $e^{x_i - \max x_i}$ is from 0 to $e^0 = 1$. Then range is much smaller, which can help normalization and prevent the number from becoming too large.

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

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

Q1.2.1

1.For $softmax(x_i) = \frac{1}{S} s_i = \frac{e^{x_i}}{\sum_i e^{x_i}}$, then range of each is from 0 to 1.
2. sum of all elements: $\sum_i \frac{e^{x_i}}{\sum_i e^{x_i}} = \frac{\sum_i e^{x_i}}{\sum_i e^{x_i}} = 1$

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

Q1.2.2

From Softmax function in Wikipedia
softmax takes an arbitrary real valued vector x and turns it into a probability distribution of it.

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

Q1.2.3

1. get $s = e^{x_i} - c$: make the value in a fixed range. 2. get $S = \sum s_i$: get sum of values, so that it can be in probability distribution. 3. get each as $\frac{s}{S}$, make it in range from 0 to 1, as probability distribution.

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

Q1.3

multi-layer neural networks: $y = f_n(w_n * x_n + b_n)$,$x_n = f_{n-1}(w_{n-1} * x_{n-1} + b_{n-1})$...
Without without a non-linear activation function: $y = k_n * (w_n * x_n + b_n) + g_n$(k,g are constant)
$y = k_n * w_n * x_n + k_n * b_n + g_n$
this can be written as:$y = w'_n * x_n + b'_n$
Then for, $x_n = w'_{n-1} * x_{n-1} + b'_{n-1}$, so $y = w'_n * (w'_{n-1} * x_{n-1} + b'_{n-1}) + b'_n = w'_n * w'_{n-1} * x_{n-1} + w'_n * b'_{n-1} + b'_n$
This can be written as: $y = w' * x_{n-1} + b'$ Same for $n - 2, ...2, 1$.
Finally $y = w * x + b$, which is the same as linear regression

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

Q1.4

gradient: $\frac{d\sigma(x)}{dx} = \frac{1}{1+e^{-x}} = \frac{-(-e^{-x})}{(1+e^{-x})^2} = \frac{e^{-x}}{(1+e^{-x})^2}$
In function of $\sigma(x)$:
$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{1+e^{-x}} * \frac{1}{1+e^{-x}}$
Since $\frac{1}{1+e^{-x}} = \sigma(x)$
$\frac{e^{-x}}{1+e^{-x}} = \frac{e^{-x}+1-1}{1+e^{-x}}$ $=1 - \frac{1}{1+e^{-x}} = 1 - \sigma(x)$
So $\frac{d\sigma(x)}{dx} = \sigma(x) * (1 - \sigma(x))$

+

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

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

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

Q1.5

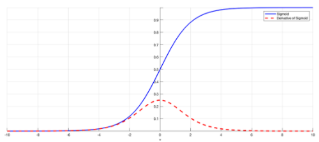
The shape of x,w,b's gradient should be the same as shape of x,w,b
Also, apply chain rule here:
1. $\frac{\partial J}{\partial W} = \frac{\partial J}{\partial y} * \frac{\partial y}{\partial W} = \delta * \frac{\partial y}{\partial W}$
Since $y = Wx + b$, then $\frac{\partial(Wx+b)}{\partial W} = x$
Because shape of w's gradient should be the same as shape of w
Then $\frac{\partial J}{\partial W} = \delta * x^T$
2. $\frac{\partial J}{\partial x} = \frac{\partial J}{\partial y} * \frac{\partial y}{\partial x} = \delta * \frac{\partial y}{\partial x}$
Since $y = Wx + b$, then $\frac{\partial(Wx+b)}{\partial x} = W$
Because shape of x's gradient should be the same as shape of x,
Then $\frac{\partial J}{\partial x} = W^T * \delta$
3. $\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y} * \frac{\partial y}{\partial b} = \delta * \frac{\partial y}{\partial b}$
Since $y = Wx + b$, then $\frac{\partial(Wx+b)}{\partial b} = 1$
Because shape of x's gradient should be the same as shape of x,
Then $\frac{\partial J}{\partial b} = \delta$

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

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

Q1.6.1

Plot of sigmid and derivative of the Sigmoid from Derivative of the Sigmoid function



From the plot, we can see that the gradient of Sigmoid function is between 0 and 0.25, also with x increasing, the gradient will decrease and become near to 0. So with many layers, the value will become smaller and smaller until 0, which is also the vanishing gradient.

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

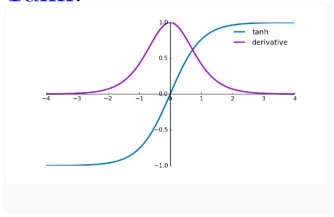
Q1.6.2

1. For sigmoid, the for e^{-x} is from 0 to +infinity, so $\frac{1}{1+e^{-x}}$ is from 0 to 1.
For tanh, the range is from -1 to 1.
2. We prefer tanh for it has higher gradient value and wilder range.

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

Q1.6.3

Plot of $\tanh(x)$ and derivative of the $\tanh(x)$ from Calculating the derivative of Tanh:



We can see that the range is from 0 to 1, which is much larger than 0.25 with same x value.So it has less of a vanishing gradient problem

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

Q1.6.4

$\tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}} = -\frac{e^{-2x}-1}{1+e^{-2x}} = -\frac{e^{-2x}+1-2}{1+e^{-2x}} = -\frac{e^{-2x}+1}{1+e^{-2x}} + 2 * \frac{1}{1+e^{-2x}} = -1 + 2 * \frac{1}{1+e^{-2x}}$
Since $\sigma(x) = \frac{1}{1+e^{-x}}$
So, $\tanh(x) = -1 + 2 * \sigma(2x)$

2 Implement a Fully Connected Network

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

2.1 Network Initialization

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

Q2.1.1

Reason: learned from slide and How to Initialize Weights in Neural Networks?:
If every layer has same weights and biases. Every neuron will do the same work and learn the same features of their inputs.
Then there will be no difference for each one. Then the result can be very bad.
Besides, giving different initial value can prevent situation that all neurons are in in local minima.

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

Q2.1.2

```
def initialize_weights(in_size,out_size,params,name=''):
        W, b = None, None
        ##########################
        ##### your code here #####
        ##########################
        # W = np.zeros((in_size,out_size))
        b = np.zeros((out_size))
        W_size = (in_size,out_size)
        # according to formula in paper:
        high = np.sqrt(6)/np.sqrt(in_size+out_size)
        low = - high
        W = np.random.uniform(low,high,(W_size))

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

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

Q2.1.3

1. For it can prevent that every neuron from doing the same work and learning the same features of their inputs.
2. From Fig 6 we can see that when initialization depending on layer size, the activation value in different will be similar. This can help delete the difference of different layers

2.2 Forward Propagation

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

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

Q2.2.1

```python
def sigmoid(x):
        res = None
        ##########################
        ##### your code here #####
        ##########################
        res = 1/(1+np.exp(-x))
        return res

def forward(X,params,name='',activation=sigmoid):
        pre_act, post_act = None, None
        # get the layer parameters
        W = params['W' + name]
        b = params['b' + name]
        ##########################
        ##### your code here #####
        ##########################
        # print(X.shape,W.shape,b.shape)
        num = X.shape[0]
        out_size = b.shape[0]
        pre_act = np.zeros((num, out_size))
        for i in range(num):
                value = X[i] @ W + b
                pre_act[i, :] = value

        post_act = activation(pre_act)

        # store the pre-activation and post-activation values
        # these will be important in backprop
        params['cache_' + name] = (X, pre_act, post_act)

        return post_act
```

Q2.2.2 Code [3 points] In `python/nn.py`, implement the softmax function. Be sure to use the numerical stability trick you derived in Q1.1 softmax. **Include your code in the write-up.**

Q2.2.2

```python
def softmax(x):
        res = None

        ##########################
        ##### your code here #####
        ##########################
        # print(x.shape)
        N, n = x.shape
        res = np.zeros((N, n))
        for i in range(N):
                # for each row
                # use the numerical stability trick in Q1.1 softmax
                c = - max(x[i, :])
                sum = 0
                for j in range(n):
                        res[i, j] = np.exp(x[i, j] + c)
                        sum += res[i, j]
                res[i, :] = res[i, :] / sum
        # print(res)

        return res
```

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

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

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

Q2.2.3

```python
def compute_loss_and_acc(y, probs):
        loss, acc = None, None

        ##########################
        ##### your code here #####
        ##########################
        # print(y)
        loss_matrix = y * np.log(probs)
        loss = -np.sum(loss_matrix)
        N = y.shape[0]
        correct_num = 0
        for i in range(N):
                prob_index = np.argmax(probs[i, :])
                y_index = np.argmax(y[i, :])
                if y_index == prob_index:
                        correct_num += 1
        acc = correct_num / N

        return loss, acc
```

2.3 Backwards Propagation

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

Q2.3

```
def backwards(delta,params,name='',activation_deriv=sigmoid_deriv):
        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]

        # do the derivative through activation first
        # (don't forget activation_deriv is a function of post_act)
        # then compute the derivative W, b, and X
        ##########################
        ##### your code here #####
        ##########################
        # do the derivative through activation
        action_deriv = activation_deriv(post_act)
        # derivative of cross-entropy:delta
        # print(action_deriv.shape)
        # print(delta.shape)
        # print(W.shape)
        # print(b.shape)
        # print(X.shape)
        deriv_f = action_deriv * delta
        # d(f(wx+b))/dw = x*f'
        grad_W = X.T @ deriv_f
        # d(f(wx+b))/db = I*f'
        N = deriv_f.shape[0]
        I = np.ones((1, N))
        grad_b = (I @ deriv_f).reshape(b.shape)
        # d(f(wx+b))/dx = w*f'
        grad_X = (W @ deriv_f.T).T

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

2.4 Training Loop: Stochastic Gradient Descent

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

Q2.4(part1)

```python
def get_random_batches(x,y,batch_size):
        batches = []
        ##########################
        ##### your code here #####
        ##########################
        # print(x.shape,y.shape)
        N = x.shape[0]
        num_per_batch = N // batch_size
        rand_index = np.random.permutation(N)
        start_index = 0
        # print(x)
        # print(rand_index)
        # the first n-1 batches
        for i in range(batch_size - 1):
                batch_index = rand_index[start_index:start_index + num_per_batch]
                start_index += num_per_batch
                batch_x = x[batch_index]
                batch_y = y[batch_index]
                batches.append((batch_x, batch_y))
        # the last batch(the rest)
        batch_index = rand_index[start_index:]
        batch_x = x[batch_index]
        batch_y = y[batch_index]
        # print(batch_x)
        batches.append((batch_x, batch_y))
        return batches
```

code for training loop:

```python
for itr in range(max_iters):
        total_loss = 0
        avg_acc = 0
        for xb,yb in batches:
                ##########################
                ##### your code here #####
                ##########################
                #pass
                # forward
                h1 = forward(xb, params, 'layer1')
                probs = forward(h1, params, 'output', softmax)
                # loss
                # be sure to add loss and accuracy to epoch totals
                loss, acc = compute_loss_and_acc(yb, probs)
                total_loss += loss
```

Please see the rest code for training loop in next page

Q2.4(part2)

```
                avg_acc += acc / batch_num
                # backward
                delta1 = probs - yb
                # apply gradient
                # gradients should be summed over batch samples
                # Implement backwards!
                delta2 = backwards(delta1, params, 'output', linear_deriv)
                backwards(delta2, params, 'layer1', sigmoid_deriv)
                # apply gradient
                # for layers
                for name in ('output', 'layer1'):
                    grad_W = params['grad_W' + name]
                    grad_b = params['grad_b' + name]
                    params['W' + name] -= learning_rate * grad_W
                    params['b' + name] -= learning_rate * grad_b

                # gradients should be summed over batch samples

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

2.5 Numerical Gradient Checker

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

Q2.5(part1)

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

        #########################
        ##### your code here #####
        #########################
        if 'W' in k:
                # for each value inside the parameter
                n, m = v.shape
                for i in range(n):
                        for j in range(m):
                                # add epsilon
                                value = params_orig[k][i, j]
                                value += eps
                                #   run the network
                                # copy and replace
                                new_params = copy.deepcopy(params_orig)
                                new_params[k][i, j] = value
                                h1 = forward(x, new_params, 'layer1')
                                probs = forward(h1, new_params, 'output',
                                ↪  softmax)
                                #   get the loss
                                loss1, acc1 = compute_loss_and_acc(y, probs)
                                #   subtract 2*epsilon
                                value -= eps * 2
                                new_params = copy.deepcopy(params_orig)
                                new_params[k][i, j] = value
                                h1 = forward(x, new_params, 'layer1')
                                probs = forward(h1, new_params, 'output',
                                ↪  softmax)
                                loss2, acc2 = compute_loss_and_acc(y, probs)
                                #   restore the original parameter value
                                #   compute derivative with central diffs
                                cen_diff = (loss1 - loss2) / (2 * eps)
                                params['grad_' + k][i, j] = cen_diff
        else:
                n = v.shape[0]
                for i in range(n):
                        # add epsilon
                        value = params_orig[k][i]
                        value += eps
                        #   run the network
see the rest code in next page
```

Q2.5(part2)

```
                    # copy and replace
                    new_params = copy.deepcopy(params_orig)
                    new_params[k][i] = value
                    h1 = forward(x, new_params, 'layer1')
                    probs = forward(h1, new_params, 'output', softmax)
                    #   get the loss
                    loss1, acc1 = compute_loss_and_acc(y, probs)
                    #   subtract 2*epsilon
                    value -= eps * 2
                    new_params = copy.deepcopy(params_orig)
                    new_params[k][i] = value
                    h1 = forward(x, new_params, 'layer1')
                    probs = forward(h1, new_params, 'output', softmax)
                    loss2, acc2 = compute_loss_and_acc(y, probs)
                    #   restore the original parameter value
                    #   compute derivative with central diffs
                    cen_diff = (loss1 - loss2) / (2 * eps)
                    params['grad_' + k][i] = cen_diff
```

3 Training Models

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

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

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

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

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

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

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

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

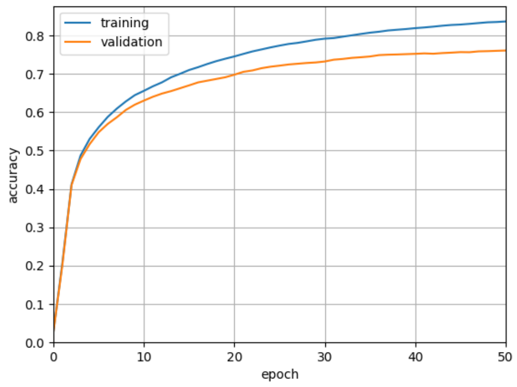
(1) the accuracy on both the training and validation set over the epochs, and

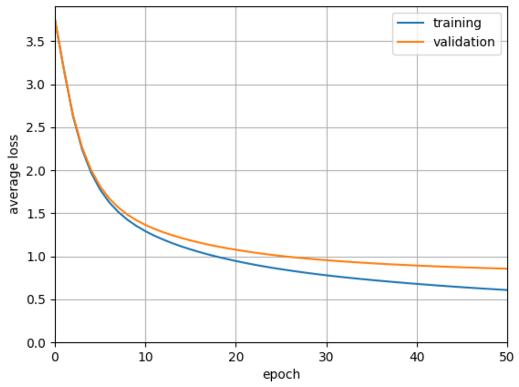
(2) the cross-entropy loss averaged over the data.

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

Q3.1

result:(fixed random seed:np.random.seed(1))(batch_size = 100 learning_rate = 2e-3)
Validation accuracy: 0.7608333333333334
Test accuracy: 0.7605555555555555
plot:
1.acc in left and 2.loss in the right:





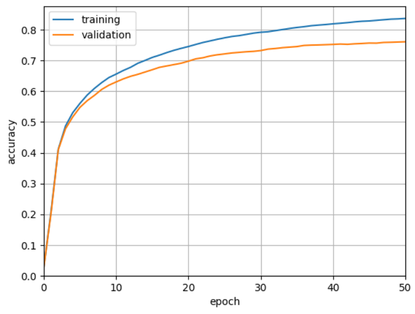
Code:

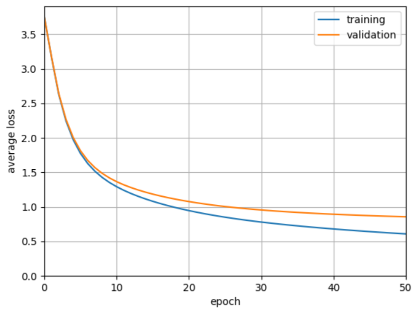
```
for xb,yb in batches:
        # training loop can be exactly the same as q2!
        ##########################
        ##### your code here #####
        ##########################
        # img = np.transpose(xb[0].reshape((32,32)))
        # plt.imshow(img)
        # plt.show()
        h1 = forward(xb, params, 'layer1')
        probs = forward(h1, params, 'output', softmax)
        # loss
        # be sure to add loss and accuracy to epoch totals
        loss, acc = compute_loss_and_acc(yb, probs)
        total_loss += loss
        avg_acc += acc / batch_num
        # backward
        # the derivative of cross-entropy(softmax(x)) is probs - 1[correct
        ↪  actions]
        delta1 = probs - yb
        delta2 = backwards(delta1, params, 'output', linear_deriv)
        # Implement backwards!
        backwards(delta2, params, 'layer1', sigmoid_deriv)
        # apply gradient
        # for layers
        for name in ('output', 'layer1'):
                grad_W = params['grad_W' + name]
                grad_b = params['grad_b' + name]
                params['W' + name] -= learning_rate * grad_W
                params['b' + name] -= learning_rate * grad_b
```

Q3.2 Tune the Learning rate - Writeup [3 points] Use the script in Q3.1 to train and generate accuracy and loss plots for each of these three networks:
(1) one with your tuned learning rate,
(2) one with 10 times that learning rate, and
(3) one with one-tenth that learning rate.
Include total of six plots in your write-up. Comment on how the learning rates affect the training, and report the final accuracy of the best network on the test set. *Hint: Use fixed random seeds to improve reproducibility.*

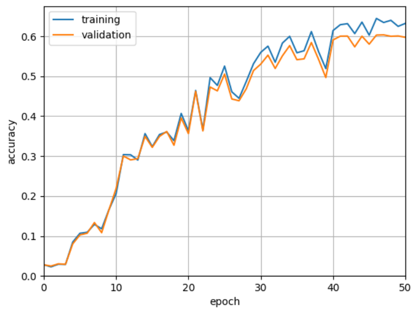
Q3.2

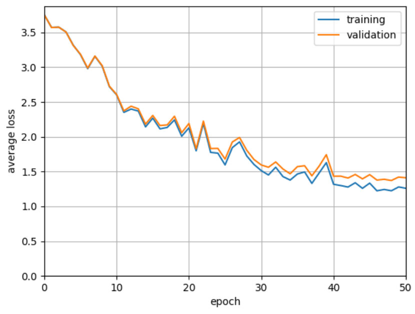
fixed random seed:np.random.seed(1)
(1)tuned learning rate:learning_rate = 2e-3:
Validation accuracy: 0.7608333333333334
Test accuracy: 0.7605555555555555
plot:
1.acc in left and 2.loss in the right:



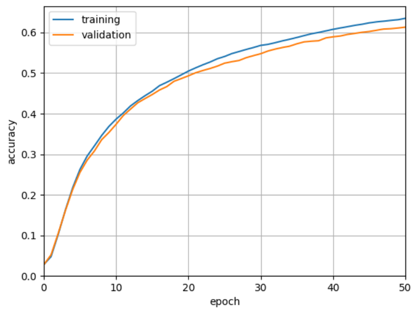


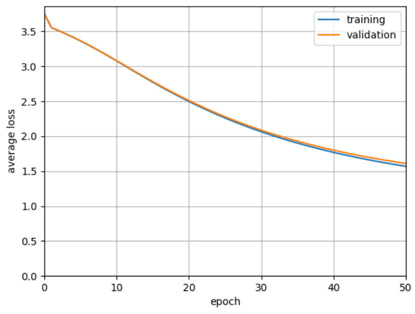
(2) with 10 times that learning rate: 2e-2:
1.acc in left and 2.loss in the right:





(3) one-tenth that learning rate: 2e-4:
1.acc in left and 2.loss in the right:



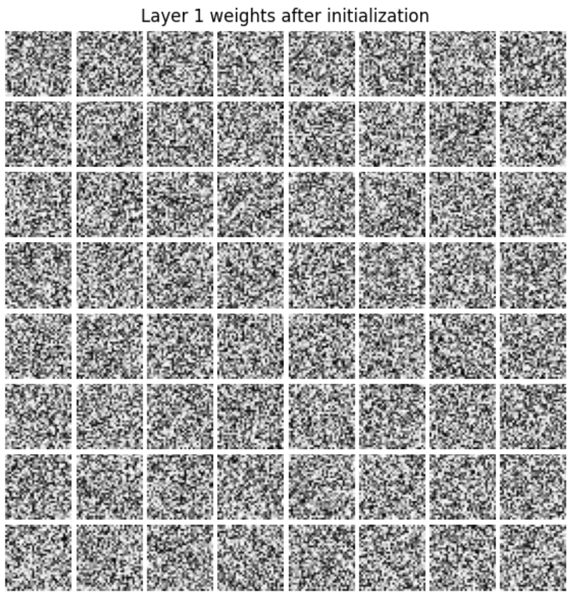


Comment: for w = w - lr*dw, if learning rate is too large, the parameters will change sharply in each epoch, the plot is also sharp, then it may jump around the best value, which makes training bad. If it's too small, the change will be little, the plot can be smooth, then it may take lots of more steps to reach the best values and the training is also not that good.
Report:
Validation accuracy: 0.7705555555555555
Test accuracy: 0.7677777777777778

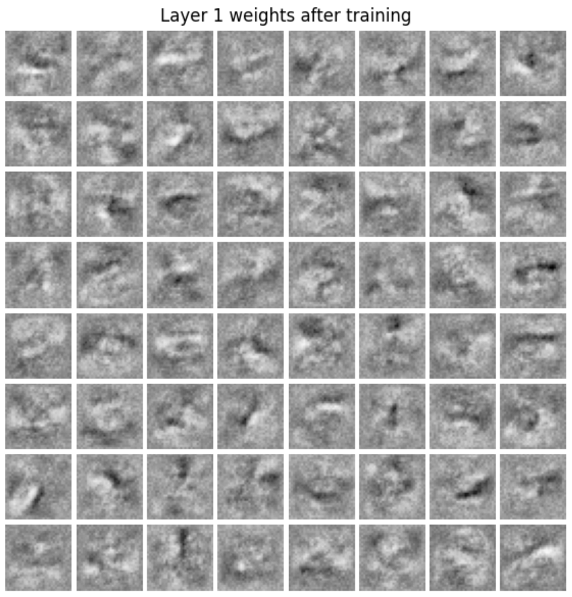
Q3.3 Learned Weights - Writeup [3 points] The script will visualize the first layer weights as 64 32x32 images, both immediately after initialization and after full training. Include both visualizations in your write-up. Comment on the learned weights and compare them to the initialized weights. Do you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

Q3.3

before training:



after training:

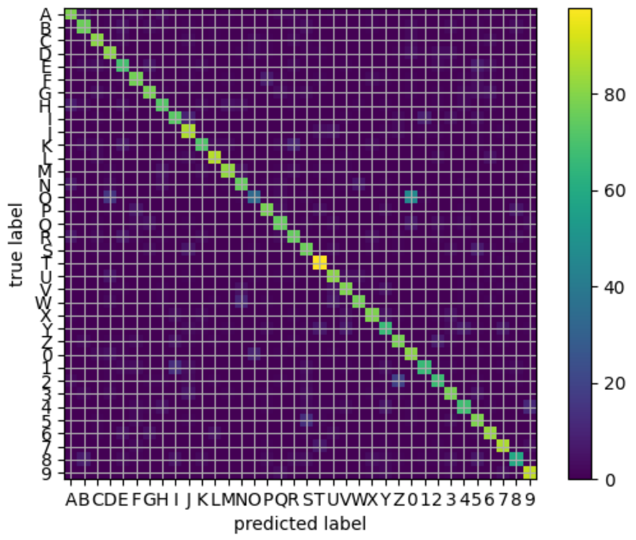


comment on learned weights and compare them to the initialized weights:
For the initialized weights are random, it looks like with lots of noise. After training the weights look like very smooth. The values nearby are very close.
patterns: values become similar in closed area and smooth.

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

Q3.4

confusion matrix:



The most commonly confused pairs: 'O' with '0', 'O' with 'D', 'Z' with '2'. Because they look very similar in shape. It's easy to be confused

4 Extract Text from Images

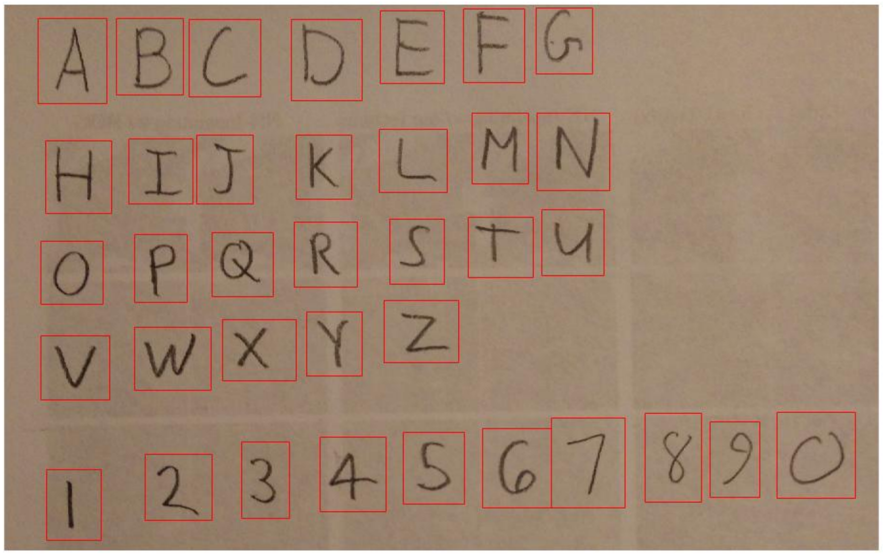


Figure 1: Sample image with handwritten characters annotated with boxes around each character.

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

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

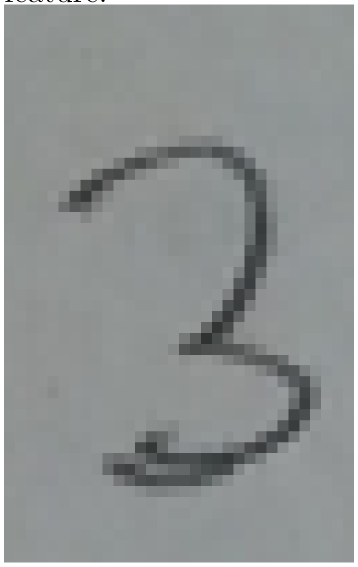
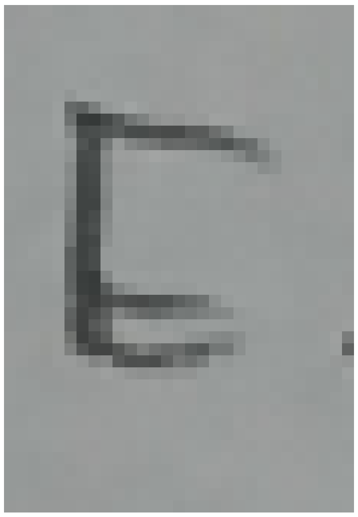
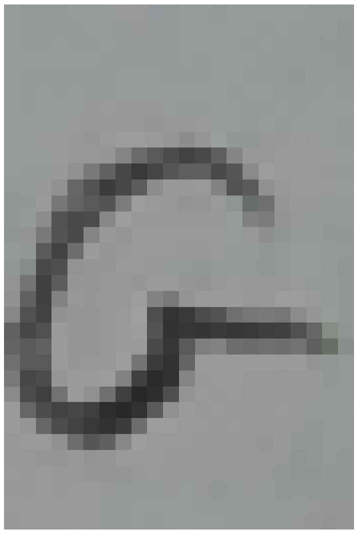
1. Process the image (blur, threshold, opening morphology, etc., perhaps in that order) to classify all pixels as being part of a character or background.
2. Find connected groups of character pixels (see skimage.measure.label). Place a bounding box around each connected component.
3. Group the letters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

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

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

Q4.1

Assumption1: The characters are in good shape and clear which means it's complete and every feature is keeped. If it's written fast, some features may be lost or extra features are added:
Example the characters not in good shape and lose features or have extra wrong feature:

Assumption2 and 3: All letters have font forms similar to the training set letter font types and are all uppercase letters:
Examples that may be wrong(lowercase letters or rare font):

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

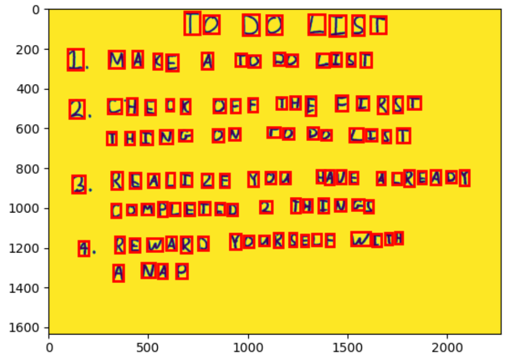
Q4.2

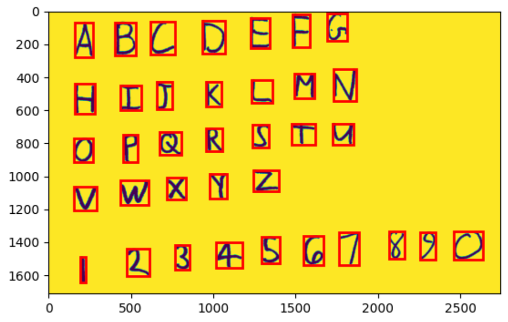
```python
def findLetters(image):
        bboxes = []
        bw = None
        ##########################
        ##### your code here #####
        ##########################
        # estimate noise
        # denoise
        denoised = skimage.restoration.denoise_bilateral(image,channel_axis=-1)#, sigma_color=0.05, sigma_spatial=15)
        # -> greyscale
        grayscale = skimage.color.rgb2gray(denoised)
        # -> threshold
        thresh = skimage.filters.threshold_otsu(grayscale)
        binary = grayscale < thresh
        # -> morphology
        #tophat = skimage.morphology.black_tophat(binary)
        dilation =skimage.morphology.binary_dilation(binary)
        dilation = skimage.morphology.binary_dilation(dilation)
        dilation = skimage.morphology.binary_dilation(dilation)
        dilation = skimage.morphology.binary_dilation(dilation)
        # -> label
        label_image = skimage.measure.label(dilation)
        regions = skimage.measure.regionprops(label_image)
        # -> skip small boxes
        # check
        #skimage.io.imshow(denoised)
        #skimage.io.imshow(binary)
        # skimage.io.imshow(label_image)
        # skimage.io.imshow(dilation)
        areas=[]
        for region in regions:
        areas.append(region.area)
        max_area = np.max(areas)
        for region in regions:
        #if region.area >= 500:
        if region.area >= max_area*0.3:
        bboxes.append(region.bbox)
        # with the characters in black and the background in white.
        # should be between 0.0 to 1.0
        dilation = ~dilation
        dilation=dilation.astype(float)
        bw = dilation
        return bboxes, bw
```

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

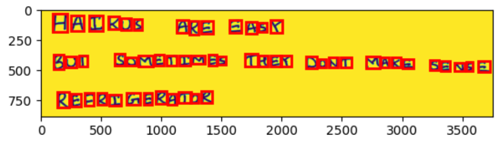
Q4.3

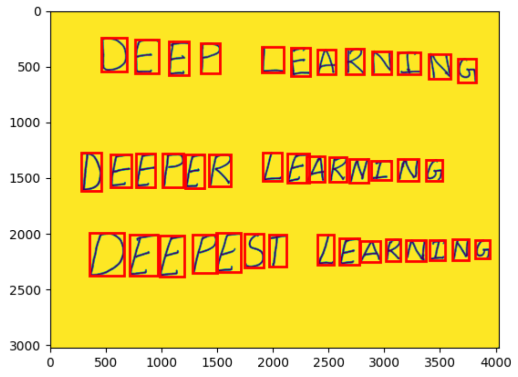
Result1 and 2:





Result3 and 4:





We can see that all letters are recognized correctly.

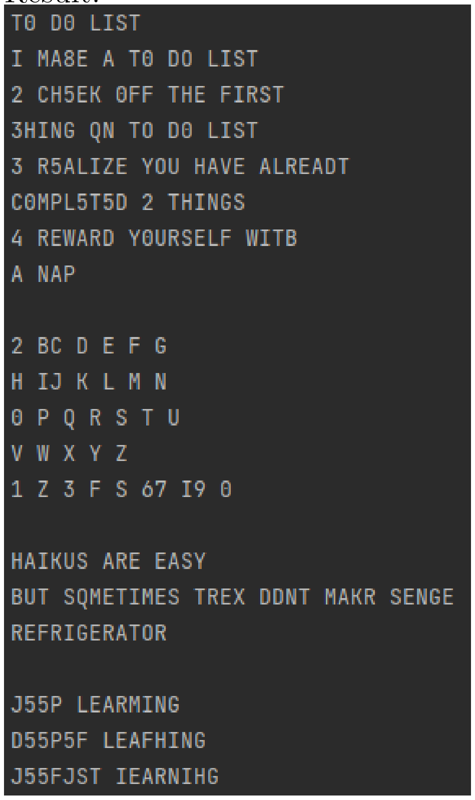
Q4.4 Eval Letter Detection - Code/Writeup [**8 points**] In `python/run_q4.py`, you will now load the image, find the character locations, classify each one with the network you trained in **Q3.1**, and return the text contained in the image. Be sure you try to make your detected images look like the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

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

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

Q4.4(part1)

Result:



```
T0 D0 LIST
I MA8E A T0 DO LIST
2 CH5EK 0FF THE FIRST
3HING QN TO D0 LIST
3 R5ALIZE YOU HAVE ALREADT
C0MPL5T5D 2 THINGS
4 REWARD Y0URSELF WITB
A NAP

2 BC D E F G
H IJ K L M N
0 P Q R S T U
V W X Y Z
1 Z 3 F S 67 I9 0

HAIKUS ARE EASY
BUT SQMETIMES TREX DDNT MAKR SENGE
REFRIGERATOR

J55P LEARMING
D55P5F LEAFHING
J55FJST IEARNIHG
```

Image1:
T0 D0 LIST
I MA8E A T0 DO LIST
2 CH5EK 0FF THE FIRST
3HING QN TO D0 LIST
3 R5ALIZE YOU HAVE ALREADT
C0MPL5T5D 2 THINGS
4 REWARD Y0URSELF WITB
A NAP
Accuracy = 98/115 = 85.2%
Image2:
2 B C D E F G
H I J K L M N
0 P Q R S T U
V W X Y Z
1 Z 3 F S 6 7 I 9 0
Accuracy = 30/36 = 83.3%
Image3:
HAIKUS ARE EASY
BUT SQMETIMES TREX DDNT MAKR SENGE
REFRIGERATOR
Accuracy = 49/54 = 90.7%
(see the rest in next page)

Q4.4(part2)

Image4:
J55P LEARMING
D55P5F LEAFHING
J55FJST IEARNIHG
Accuracy = 27/41 = 65.9%
Overall accuracy = 204/246= 82.9% It's larger than 80%
code:

```
# find the rows using..RANSAC, counting, clustering, etc.
##########################
##### your code here #####
##########################
# Group the letters based on which line of the text they are a part of, and sort
↪   each
# group so that the letters are in the order they appear on the page.
row_lists = []
# box: minr, minc, maxr, maxc
# sort with row
bboxes.sort(key=lambda x: x[0])
#h = bboxes[0][2] - bboxes[0][2]
threshold_row = 100
row_judge = bboxes[0][0]
#print(row_num)
#print(len(bboxes))
row_list = []
for bbox in bboxes:
        row = bbox[0]
        if abs(row - row_judge) < threshold_row:
                row_list.append(bbox)
        else:
                # sort each group with col and append
                row_list.sort(key=lambda x: x[1])
                row_lists.append(row_list)
                # start new row
                row_list = []
                row_list.append(bbox)
        #update new
        row_judge = row
# last row
row_list.sort(key=lambda x: x[1])
row_lists.append(row_list)
#print(row_lists)
(see the rest code in next page)
```

Q4.4(part2)

```
# consider doing a square crop, and even using np.pad() to get your images
↪  looking more like the dataset
##########################
##### your code here #####
##########################
chars=[]
counts = []
for row_list in row_lists:
        count = 0
        for bbox in row_list:
                count+=1
                minr, minc, maxr, maxc = bbox
                crop_img = bw[minr:maxr,minc:maxc]
                # plt.imshow(crop_img)
                # plt.show()
                # square crop, and even using np.pad()
                h = maxr-minr
                w = maxc-minc
                diff = int(abs(h-w)//2)
                pad_h = int(h/7)
                pad_w = int(h / 7)
                if h > w:
                        pad_img = np.pad(crop_img, ((pad_h, pad_h), (diff+pad_w,
                        ↪  diff+pad_w)), 'maximum')
                else:
                        pad_img = np.pad(crop_img, ((diff+pad_h,
                        ↪  diff+pad_h),(pad_w, pad_w)), 'maximum')
                # plt.show()')
                # resize: input images are 32 × 32 images
                resized_img = skimage.transform.resize(pad_img, (32, 32))
                # before you flatten, transpose the image
                trans_img = np.transpose(resized_img)
                #plt.imshow(resized_img)
                #plt.show()
                flatten_img = trans_img.flatten()
                chars.append(flatten_img)
        counts.append(count)
#print(counts)
see the rest in next part
```

Q4.4(part3)

```
##########################
##### your code here #####
##########################
char_with = bboxes[0][3] - bboxes[0][1]
#print(char_with)
line_num = len(counts)
chars = np.array(chars)
h1 = forward(chars, params, 'layer1')
probs = forward(h1, params, 'output', softmax)
res = []
k=0
count = counts[k]
for prob in probs:
        if counts[k]-count>0 and \
        (row_lists[k][counts[k]-count][1] -
        row_lists[k][counts[k]-count-1][3])>char_with*0.8:
                res.append(' ')
        index = np.argmax(prob)
        letter = letters[index]
        res.append(letter)
        count -= 1
        if count == 0:
                res.append('\n')
                k += 1
                if k < line_num:
                        count = counts[k]
print(''.join(res))
```

5 (Extra Credit) Image Compression with Autoencoders

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

5.1 Building the Autoencoder

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

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

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

Q5.1.1

```
# initialize layers here
##########################
##### your code here #####
##########################

initialize_weights(1024, 32, params, "layer1")
# zero-initialized momentum accumulators
params['Mw' + 'layer1'] = np.zeros((1024,32))
params['Mb' + 'layer1'] = np.zeros((32))

initialize_weights(32, 32, params, "layer2")
params['Mw' + 'layer2'] = np.zeros((32,32))
params['Mb' + 'layer2'] = np.zeros((32))

initialize_weights(32, 32, params, "layer3")
params['Mw' + 'layer3'] = np.zeros((32,32))
params['Mb' + 'layer3'] = np.zeros((32))

initialize_weights(32, 1024, params, "output")
params['Mw' + 'output'] = np.zeros((32,1024))
params['Mb' + 'output'] = np.zeros((1024))
# should look like your previous training loops

##########################
##### your code here #####
##########################
# forward
h = forward(xb, params, 'layer1',relu)
h = forward(h, params, 'layer2',relu)
h = forward(h, params, 'layer3',relu)
probs = forward(h, params, 'output', sigmoid)
#  squared error for the output image compared to the input image
loss = np.sum((xb-probs)**2)
total_loss+=loss
# backward
delta1 = -2*(xb-probs)
delta2 = backwards(delta1, params, 'output', sigmoid_deriv)
delta3 = backwards(delta2, params, 'layer3', relu_deriv)
delta4 = backwards(delta3, params, 'layer2', relu_deriv)
delta5 = backwards(delta4, params, 'layer1', relu_deriv)
# apply gradient
# for layers
for name in ('output','layer3','layer2', 'layer1'):
        grad_W = params['grad_W' + name]
        grad_b = params['grad_b' + name]
        params['W' + name] -= learning_rate * grad_W
        params['b' + name] -= learning_rate * grad_b
```

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

Q5.1.2

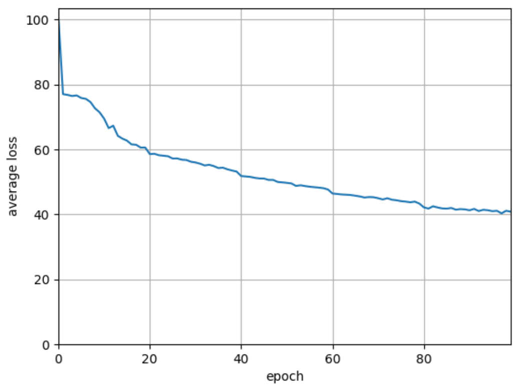
```python
# initialize layers here
##########################
##### your code here #####
##########################

initialize_weights(1024, 32, params, "layer1")
# zero-initialized momentum accumulators
params['Mw' + 'layer1'] = np.zeros((1024,32))
params['Mb' + 'layer1'] = np.zeros((32))

initialize_weights(32, 32, params, "layer2")
params['Mw' + 'layer2'] = np.zeros((32,32))
params['Mb' + 'layer2'] = np.zeros((32))

initialize_weights(32, 32, params, "layer3")
params['Mw' + 'layer3'] = np.zeros((32,32))
params['Mb' + 'layer3'] = np.zeros((32))

initialize_weights(32, 1024, params, "output")
params['Mw' + 'output'] = np.zeros((32,1024))
params['Mb' + 'output'] = np.zeros((1024))
# should look like your previous training loops

##########################
##### your code here #####
##########################
...
# apply gradient
# for layers
for name in ('output','layer3','layer2', 'layer1'):
        grad_W = params['grad_W' + name]
        grad_b = params['grad_b' + name]
        params['Mw' + name] = 0.9 * params['Mw' + name] - learning_rate * grad_W
        params['W' + name] = params['W' + name] + params['Mw' + name]
        params['Mb' + name] = 0.9 * params['Mb' + name] - learning_rate * grad_b
        params['b' + name] = params['b' + name] + params['Mb' + name]
```

5.2 Training the Autoencoder

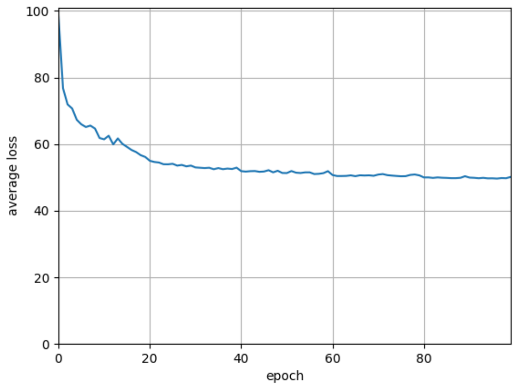
Q5.2 (Extra Credit) Writeup/Code [5 points] Using the provided default settings, train the network for 100 epochs. Plot the training loss curve and include it in the writeup. What do you observe?

Q5.2

Not using momentum:



Use momentum:



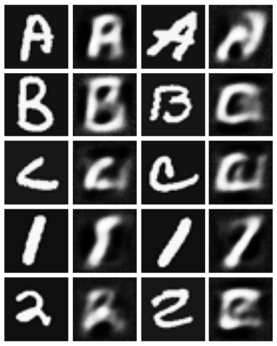
The loss get decreased and become stable when meet one epoch value

5.3 Evaluating the Autoencoder

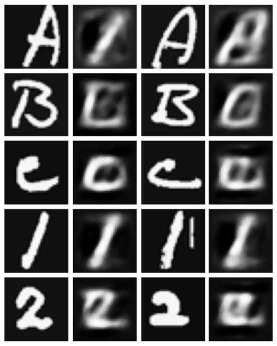
Q5.3.1 (Extra Credit) Writeup/Code [5 points] Now let's evaluate how well the autoencoder has been trained. Select 5 classes from the total 36 classes in the validation set and for each selected class include in your report 2 validation images and their reconstruction. What differences do you observe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

Q5.3.1

Result: Not using momentum:



Use momentum:



Difference:
The reconstructed ones Look more blurry and the boundaries are not as clear as the original image
Code:

```
h = forward(visualize_x, params, 'layer1',relu)
h = forward(h, params, 'layer2',relu)
h = forward(h, params, 'layer3',relu)
reconstructed_x = forward(h, params, 'output', sigmoid)
```

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

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

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

Q5.3.2

Result:
Not using momentum:
Average PSNR : 13.379990823521801
using momentum:
Average PSNR: 11.322928512093558

```
h = forward(valid_x, params, 'layer1',relu)
h = forward(h, params, 'layer2',relu)
h = forward(h, params, 'layer3',relu)
result = forward(h, params, 'output', sigmoid)
rsnr_total=0
for i in range(len(valid_x)):
        rsnr_total+=peak_signal_noise_ratio(result[i],valid_x[i])
print("Average PSNR: "+str(rsnr_total/len(valid_x)))
```

6 PyTorch

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

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

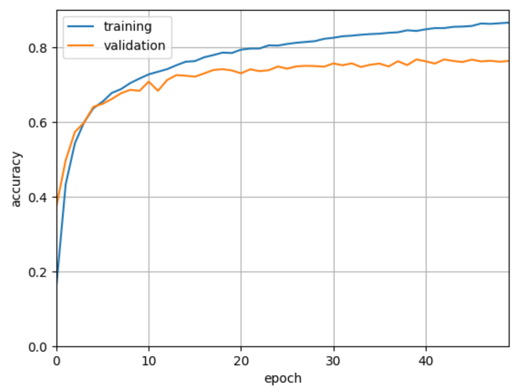
For this section, you're free to implement these however you like. All of the tasks required here are fairly small and don't require a GPU if you use small networks. Include the neural network code for each part in the writeup.

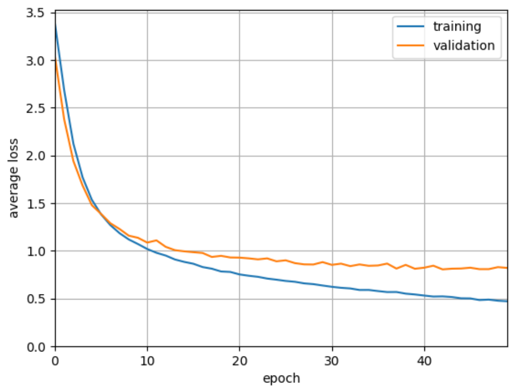
6.1 Train a neural network in PyTorch

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

Q6.1.1

plot:acc(left) loss(right)





(please see code in next page)

runq6.1.1.py

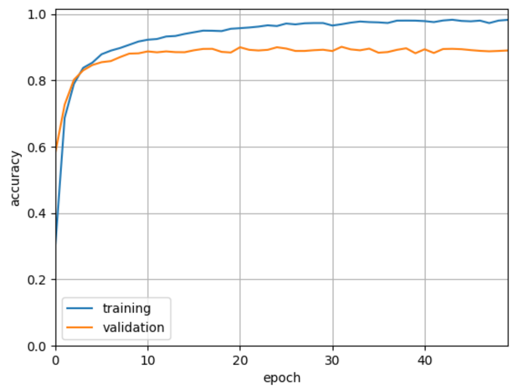
```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import scipy.io
from torch.utils.data import Dataset,DataLoader
import matplotlib.pyplot as plt


def plot(train_loss, valid_loss, train_acc, valid_acc):
    # 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()

class Dataset(Dataset):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __getitem__(self, index):
        x = torch.Tensor(self.x[index])
        y = torch.Tensor(self.y[index])
        return (x, y)

    def __len__(self):
        count = self.x.shape[0]
        return count
```

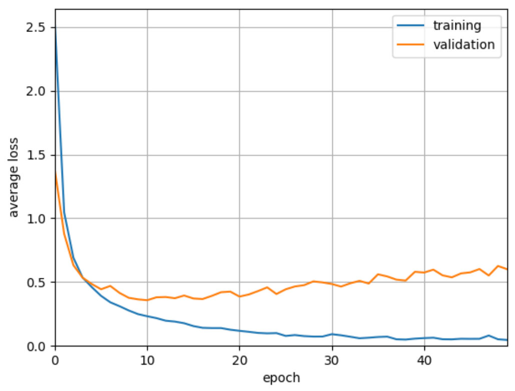
```python


class FC_Net(nn.Module):
    def __init__(self, size_in, size_out):
        super().__init__()
        self.fc1 = nn.Linear(size_in,64)
        self.fc2 = nn.Linear(64, size_out)

    def forward(self,x):
        x = self.fc1(x)
        x = F.sigmoid(x)
        out = self.fc2(x)
        return out


train_data = scipy.io.loadmat('../data/nist36_train.mat')
valid_data = scipy.io.loadmat('../data/nist36_valid.mat')
test_data = scipy.io.loadmat('../data/nist36_test.mat')
train_x, train_y = train_data['train_data'], train_data['train_labels']
valid_x, valid_y = valid_data['valid_data'], valid_data['valid_labels']
test_x, test_y = test_data['test_data'], test_data['test_labels']

# model and loss, optimizer
model = FC_Net(train_x.shape[1], train_y.shape[1])
print(model)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.002)

# load data, shuffle is necessary!!!
train_data = Dataset(train_x,train_y)
train_loader = DataLoader(train_data, batch_size=100, shuffle=True)

test_data = Dataset(test_x,test_y)
test_loader = DataLoader(test_data, batch_size=100)

valid_data = Dataset(valid_x,valid_y)
valid_loader = DataLoader(valid_data, batch_size=100)

epoch = 50
train_loss = []
valid_loss = []
train_acc = []
valid_acc = []
for i in range(epoch):
    # training part
    print("Training iteration: " + str(i+1))
    size = len(train_loader.dataset)
    num_batches = len(train_loader)
    model.train()
    loss_val, correct = 0, 0
    for data in train_loader:
        x, y = data
        pred = model(x)
        loss = loss_fn(pred, y)
        #pred = nn.Softmax(dim=1)(pred)
        correct += (pred.argmax(1) == y.argmax(1)).type(torch.float).sum().item()
        loss_val+=loss.item()

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    loss_val /= num_batches
    correct /= size
    train_loss.append(loss_val)
    train_acc.append(correct)
    print("Train loss: " + str(loss_val) + ", Accuracy: " + str(correct * 100) + "%")


    # validation part
    size = len(valid_loader.dataset)
    num_batches = len(valid_loader)
    model.eval()
    loss, correct = 0, 0
    with torch.no_grad():
        for X, y in valid_loader:
            pred = model(X)
            loss += loss_fn(pred, y).item()
            #pred = nn.Softmax(dim=1)(pred)
            correct += (pred.argmax(1) == y.argmax(1)).type(torch.float).sum().item()
    loss /= num_batches
    correct /= size
    valid_loss.append(loss)
    valid_acc.append(correct)
    print("Validation loss: "+str(loss)+", Accuracy: " + str(correct*100)+"%")

plot(train_loss, valid_loss, train_acc, valid_acc)
```

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

Q6.1.2

plot:acc(left) loss(right)





From plot we can see:The result(accuracy around 90%) is much more better than previous fully-connected network(accuracy around 78%)

runq6.1.2.py

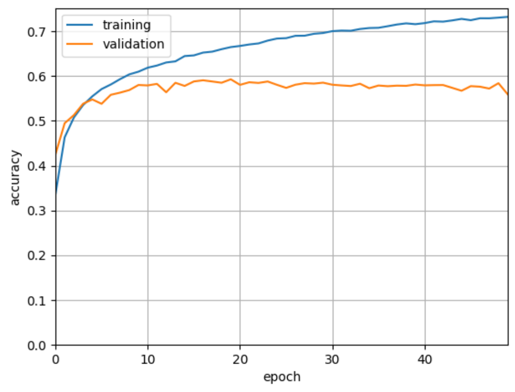
```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import scipy.io
from torch.utils.data import Dataset,DataLoader
import matplotlib.pyplot as plt


def plot(train_loss, valid_loss, train_acc, valid_acc):
    # 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()


class Dataset(Dataset):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __getitem__(self, index):
        x = torch.Tensor(self.x[index])
        y = torch.Tensor(self.y[index])
        return (x, y)

    def __len__(self):
        count = self.x.shape[0]
        return count


class CNN_Net(nn.Module):
    def __init__(self, size_in, size_out):
        super().__init__()
        self.cnn_model = nn.Sequential(
            nn.Conv2d(1, 6, 5),
            nn.ReLU(),
            nn.MaxPool2d(2,2),
            nn.Conv2d(6, 16, 5),
            nn.ReLU(),
            nn.MaxPool2d(2,2),
            nn.Flatten(),
            nn.Linear(16 * 5 * 5, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, size_out)
        )

    def forward(self,x):
        out = self.cnn_model(x)
        return out


train_data = scipy.io.loadmat('../data/nist36_train.mat')
valid_data = scipy.io.loadmat('../data/nist36_valid.mat')
test_data = scipy.io.loadmat('../data/nist36_test.mat')
train_x, train_y = train_data['train_data'], train_data['train_labels']
valid_x, valid_y = valid_data['valid_data'], valid_data['valid_labels']
test_x, test_y = test_data['test_data'], test_data['test_labels']

# model and loss, optimizer
model = CNN_Net(train_x.shape[1], train_y.shape[1])
print(model)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.002)

# load data, shuffle is necessary!!!
train_data = Dataset(train_x,train_y)
print(train_x.shape,train_y.shape)
train_loader = DataLoader(train_data, batch_size=100, shuffle=True)

test_data = Dataset(test_x,test_y)
test_loader = DataLoader(test_data, batch_size=100)

valid_data = Dataset(valid_x,valid_y)
valid_loader = DataLoader(valid_data, batch_size=100)

epoch = 50
train_loss = []
valid_loss = []
train_acc = []
valid_acc = []
for i in range(epoch):
    # training part
    print("Training iteration: " + str(i+1))
    size = len(train_loader.dataset)
    num_batches = len(train_loader)
    model.train()
    loss_val, correct = 0, 0
    for data in train_loader:
        x, y = data
        N = x.shape[0]
        # N,channel,h,w
        x = x.reshape(N,1,32,32)
```

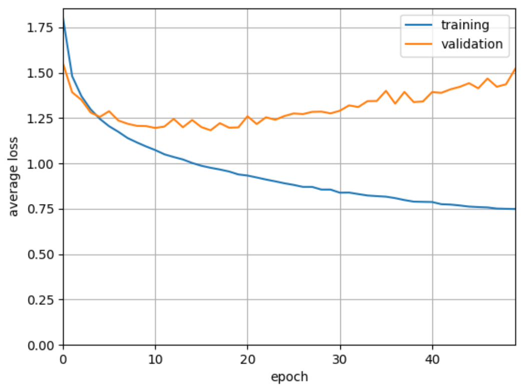
```
            pred = model(x)
            loss = loss_fn(pred, y)
            #pred = nn.Softmax(dim=1)(pred)
            correct += (pred.argmax(1) ==
             ↪  y.argmax(1)).type(torch.float).sum().item()
            loss_val+=loss.item()

            # Backpropagation
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        loss_val /= num_batches
        correct /= size
        train_loss.append(loss_val)
        train_acc.append(correct)
        print("Train loss: " + str(loss_val) + ", Accuracy: " + str(correct *
         ↪  100) + "%")


        # validation part
        size = len(valid_loader.dataset)
        num_batches = len(valid_loader)
        model.eval()
        loss, correct = 0, 0
        with torch.no_grad():
            for X, y in valid_loader:
                N = X.shape[0]
                # N,channel,h,w
                X = X.reshape(N, 1, 32, 32)
                pred = model(X)
                loss += loss_fn(pred, y).item()
                #pred = nn.Softmax(dim=1)(pred)
                correct += (pred.argmax(1) ==
                 ↪  y.argmax(1)).type(torch.float).sum().item()
        loss /= num_batches
        correct /= size
        valid_loss.append(loss)
        valid_acc.append(correct)
        print("Validation loss: "+str(loss)+", Accuracy: " +
         ↪  str(correct*100)+"%")

plot(train_loss, valid_loss, train_acc, valid_acc)
```

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

Q6.1.3

plot:acc(left) loss(right)





runq6.1.3.py

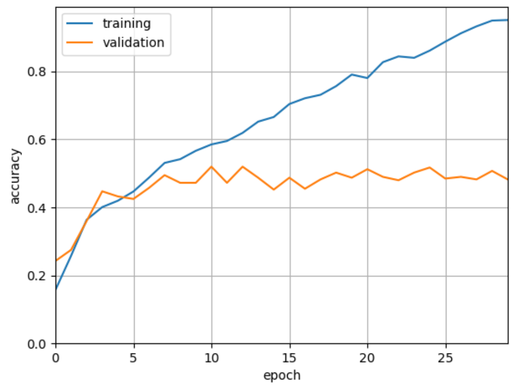
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import scipy.io
from torch.utils.data import Dataset,DataLoader
import matplotlib.pyplot as plt
import torchvision


def plot(train_loss, valid_loss, train_acc, valid_acc):
    # 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()

```

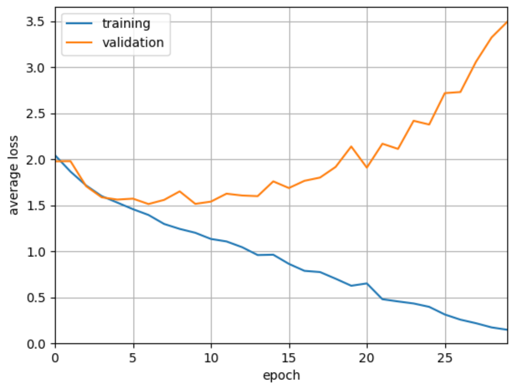
```python
    # plot accuracy curves
    plt.plot(range(len(train_acc)), train_acc, label="training")
    plt.plot(range(len(valid_acc)), valid_acc, label="validation")
    plt.xlabel("epoch")
    plt.ylabel("accuracy")
    plt.xlim(0, len(train_acc) - 1)
    plt.ylim(0, None)
    plt.legend()
    plt.grid()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


class Dataset(Dataset):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __getitem__(self, index):
        x = torch.Tensor(self.x[index])
        y = torch.Tensor(self.y[index])
        return (x, y)

    def __len__(self):
        count = self.x.shape[0]
        return count


class CNN_Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.cnn_model = nn.Sequential(
            nn.Conv2d(3, 6, 5),
            nn.ReLU(),
            nn.MaxPool2d(2,2),
            nn.Conv2d(6, 16, 5),
            nn.ReLU(),
            nn.MaxPool2d(2,2),
            nn.Flatten(),
            nn.Linear(16 * 5 * 5, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 10)
        )

    def forward(self,x):
        out = self.cnn_model(x)
        return out


train_data = torchvision.datasets.CIFAR10(root='../data', train=True,
↪  download=True, transform=torchvision.transforms.ToTensor())
test_data = torchvision.datasets.CIFAR10(root='../data', train=False,
↪  download=True, transform=torchvision.transforms.ToTensor())

# model and loss, optimizer
model = CNN_Net()
print(model)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.002)

# load data, shuffle is necessary!!!
train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
valid_loader = DataLoader(test_data, batch_size=64)


epoch = 50
train_loss = []
valid_loss = []
train_acc = []
valid_acc = []
for i in range(epoch):
    # training part
    print("Training iteration: " + str(i+1))
    size = len(train_loader.dataset)
    num_batches = len(train_loader)
    model.train()
    loss_val, correct = 0, 0
    for data in train_loader:
        x, y = data
        pred = model(x)
        loss = loss_fn(pred, y)
        #pred = nn.Softmax(dim=1)(pred)
        correct += (pred.argmax(1) == y).type(torch.float).sum().item()
        loss_val+=loss.item()

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    loss_val /= num_batches
    correct /= size
    train_loss.append(loss_val)
    train_acc.append(correct)
    print("Train loss: " + str(loss_val) + ", Accuracy: " + str(correct *
     ↪  100) + "%")


    # validation part
    size = len(valid_loader.dataset)
    num_batches = len(valid_loader)
    model.eval()
    loss, correct = 0, 0
    with torch.no_grad():
        for X, y in valid_loader:
            pred = model(X)
            loss += loss_fn(pred, y).item()
            #pred = nn.Softmax(dim=1)(pred)
            correct += (pred.argmax(1) ==
             ↪  y).type(torch.float).sum().item()
    loss /= num_batches
    correct /= size
    valid_loss.append(loss)
    valid_acc.append(correct)
    print("Validation loss: "+str(loss)+", Accuracy: " +
     ↪  str(correct*100)+"%")

plot(train_loss, valid_loss, train_acc, valid_acc)
```

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

Q6.1.4

plot:acc(left) loss(right)





We can see that the accuracy is around 50%, it's not as good as HW1 method. So for scene classification, only use basic CNN is not enough. Some other methods like data transforming are still needed. OR better CNN can be built

runq6.1.4.py

```
import os.path

import skimage
import torch
import torch.nn as nn
from PIL import Image
from torch.utils.data import Dataset,DataLoader
import matplotlib.pyplot as plt
import numpy as np



def plot(train_loss, valid_loss, train_acc, valid_acc):
    # 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()

img_size = 64

class Dataset(Dataset):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __getitem__(self, index):
        x = torch.Tensor(self.x[index])
        y = torch.Tensor(self.y[index])
        return (x, y)

    def __len__(self):
        count = self.x.shape[0]
        return count


class CNN_Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.cnn_model = nn.Sequential(
            nn.Conv2d(1, 16, 5),
            nn.ReLU(),
            nn.MaxPool2d(2,2),

            nn.Conv2d(16, 32, 5),
            nn.ReLU(),
            nn.MaxPool2d(2,2),

            nn.Conv2d(32, 64, 5),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),

            nn.Flatten(),
            nn.Linear(1024, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 8)
        )

    def forward(self,x):
        out = self.cnn_model(x)
        return out


def get_data(path,x_txt,y_txt):
    # for img
    f = open(x_txt,'r')
    lines = f.readlines()
    #print(lines)
    f.close()
    x = []
    for line in lines:
        line = line.replace("\n", "")
        img_path = path+line
        img = Image.open(img_path)
        img = img.resize((img_size, img_size))
        img = np.array(img.convert('L'))
        img=img/255.0
        x.append(img)

    f = open(y_txt, 'r')
    lines = f.readlines()
    f.close()
    y = []
    for line in lines:
        line = line.replace("\n", "")
        y.append(int(line))

    y = np.array(y)
    # print(y)
    x = np.array(x)
    # N = x.shape[0]
    # # N,channel,h,w
    # x = x.reshape(N, 1, 64, 64)
    y=y.reshape(-1,1)
    # print(y)
    print(x.shape)
    print(y.shape)
    return x,y


# read
train_x_path = '../data/SUN/train_files.txt'
train_y_path = '../data/SUN/train_labels.txt'
test_x_path = '../data/SUN/test_files.txt'
test_y_path = '../data/SUN/test_labels.txt'
path = '../data/SUN/'
# save
x_train_save = '../data/sun_train_x.npy'
y_train_save = '../data/sun_train_y.npy'
x_test_save = '../data/sun_test_x.npy'
y_test_save = '../data/sun_test_y.npy'

if os.path.exists(x_train_save):
    train_x = np.load(x_train_save)
    train_y = np.load(y_train_save)
    valid_x = np.load(x_test_save)
    valid_y = np.load(y_test_save)
else:
    train_x, train_y = get_data(path, train_x_path, train_y_path)
    valid_x, valid_y = get_data(path, test_x_path, test_y_path)
    np.save(x_train_save,train_x)
    np.save(y_train_save, train_y)
    np.save(x_test_save, valid_x)
    np.save(y_test_save, valid_y)

# model and loss, optimizer
model = CNN_Net()
print(model)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.002)

# load data, shuffle is necessary!!!
train_data = Dataset(train_x,train_y)
train_loader = DataLoader(train_data, batch_size=32, shuffle=True)

valid_data = Dataset(valid_x,valid_y)
valid_loader = DataLoader(valid_data, batch_size=32)

epoch = 30
train_loss = []
valid_loss = []
train_acc = []
```

```
valid_acc = []

for i in range(epoch):
    # training part
    print("Training iteration: " + str(i+1))
    size = len(train_loader.dataset)
    num_batches = len(train_loader)
    model.train()
    loss_val, correct = 0, 0
    for data in train_loader:
        x, y = data
        N = x.shape[0]
        # N,channel,h,w
        x = x.reshape(N,1,img_size,img_size)
        y=y.reshape(-1)
        #print(y.dtype)
        pred = model(x)
        y = y.long()

        loss = loss_fn(pred, y)
        #pred = nn.Softmax(dim=1)(pred)
        correct += (pred.argmax(1) == y).type(torch.float).sum().item()
        loss_val+=loss.item()

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    loss_val /= num_batches
    correct /= size
    train_loss.append(loss_val)
    train_acc.append(correct)
    print("Train loss: " + str(loss_val) + ", Accuracy: " + str(correct *
    ↪  100) + "%")


    # validation part
    size = len(valid_loader.dataset)
    num_batches = len(valid_loader)
    model.eval()
    loss, correct = 0, 0
    with torch.no_grad():
        for X, y in valid_loader:
            N = X.shape[0]
            # N,channel,h,w
            X = X.reshape(N, 1, img_size, img_size)
            pred = model(X)
            y = y.reshape(-1)
            y = y.long()
            loss += loss_fn(pred, y).item()
            #pred = nn.Softmax(dim=1)(pred)
            correct += (pred.argmax(1) ==
             ↪  y).type(torch.float).sum().item()
    loss /= num_batches
    correct /= size
    valid_loss.append(loss)
    valid_acc.append(correct)
    print("Validation loss: "+str(loss)+", Accuracy: " +
     ↪  str(correct*100)+"%")

plot(train_loss, valid_loss, train_acc, valid_acc)
```

6.2 Fine Tuning

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

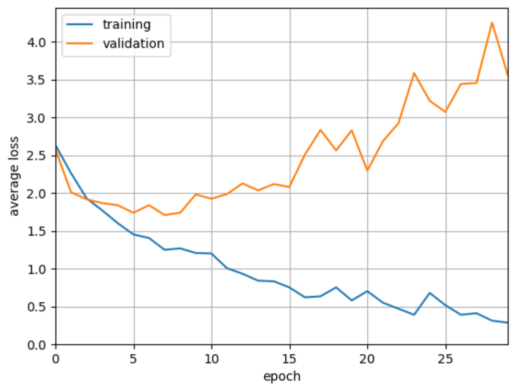
These days, it is most common to initialize a network with weights from another deep network that was trained for a different purpose. This is because, whether we are doing image classification, segmentation, recognition etc..., most real images share common properties. Simply copying the weights from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

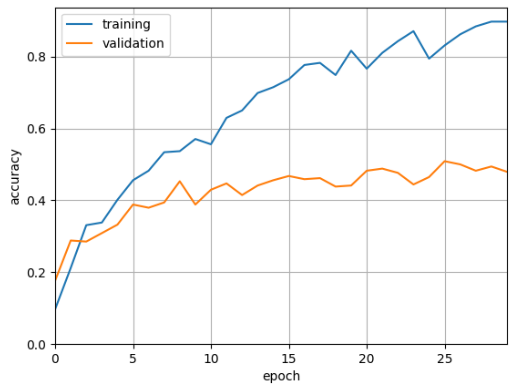
Q6.2 Code/Writeup [**5 points**] Fine-tune a single layer classifier using pytorch on the flowers 17 (or flowers 102!) dataset using squeezenet1_1, as well as an architecture you've designed yourself (for example 3 convolutional layers followed 2 fully connected layers, it's standard slide 6) and trained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

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

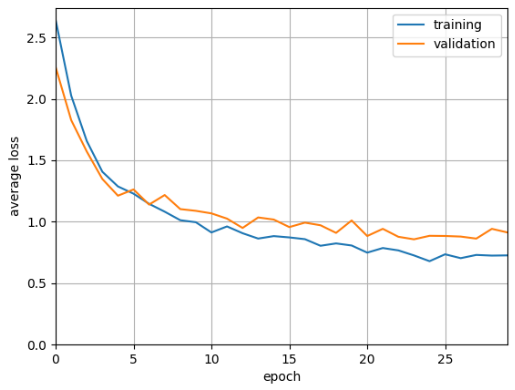
Q6.2

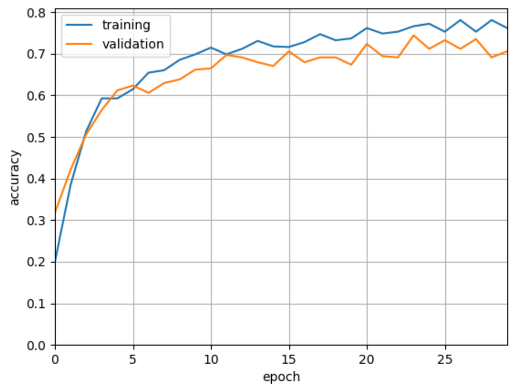
plot: My network





plot: Use fine tuning of queezenet1_1 can be much better.





We can see that using my network is not as good as fine tuning and even cause overfitting when epoch is large.
So using fine tuning

6.2.2.py

```
import torch
import torchvision
from torchvision import transforms, datasets
import os.path

import skimage
import torch
import torch.nn as nn
from PIL import Image
from torch.utils.data import Dataset,DataLoader
```

```python
import matplotlib.pyplot as plt
import numpy as np
img_size = 32
input_size = 224
def plot(train_loss, valid_loss, train_acc, valid_acc):
    # 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()

data_transforms = transforms.Compose([
    transforms.RandomResizedCrop(input_size),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])
trans_dataset =
↪   datasets.ImageFolder(root='../data/oxford-flowers17/train',
                                             transform=data_transforms)
train_loader = torch.utils.data.DataLoader(trans_dataset,
                                               batch_size=4, shuffle=True)
trans_dataset = datasets.ImageFolder(root='../data/oxford-flowers17/val',
                                             transform=data_transforms)
valid_loader = torch.utils.data.DataLoader(trans_dataset,
                                               batch_size=4, shuffle=True)

# model and loss, optimizer
model = torchvision.models.squeezenet1_1(pretrained=True)
feature_extract = True
```

```python
# set false
for param in model.parameters():
    param.requires_grad = False
# change classifier layer
model.classifier[1] = nn.Conv2d(512, 17, kernel_size=(1,1), stride=(1,1))
model.num_classes = 17
input_size = 224
print(model)
loss_fn = nn.CrossEntropyLoss()
# select parameters
params_to_update = model.parameters()
print("Params to learn:")
# only change
if feature_extract:
    params_to_update = []
    for name,param in model.named_parameters():
        if param.requires_grad == True:
            params_to_update.append(param)
            print("\t",name)
else:
    for name,param in model.named_parameters():
        if param.requires_grad == True:
            print("\t",name)

# Observe that all parameters are being optimized
optimizer = torch.optim.SGD(params_to_update, lr=0.001)

epoch = 30
train_loss = []
valid_loss = []
train_acc = []
valid_acc = []

for i in range(epoch):
    # training part
    print("Training iteration: " + str(i+1))
    size = len(train_loader.dataset)
    num_batches = len(train_loader)
    model.train()
    loss_val, correct = 0, 0
    for data in train_loader:
        x, y = data
        pred = model(x)
        loss = loss_fn(pred, y)
        #pred = nn.Softmax(dim=1)(pred)
        correct += (pred.argmax(1) == y).type(torch.float).sum().item()
        loss_val+=loss.item()

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    loss_val /= num_batches
    correct /= size
    train_loss.append(loss_val)
    train_acc.append(correct)
    print("Train loss: " + str(loss_val) + ", Accuracy: " + str(correct * 100) + "%")


    # validation part
    size = len(valid_loader.dataset)
    num_batches = len(valid_loader)
    model.eval()
    loss, correct = 0, 0
    with torch.no_grad():
        for X, y in valid_loader:
            pred = model(X)
            loss += loss_fn(pred, y).item()
            #pred = nn.Softmax(dim=1)(pred)
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    loss /= num_batches
    correct /= size
    valid_loss.append(loss)
    valid_acc.append(correct)
    print("Validation loss: "+str(loss)+", Accuracy: " + str(correct*100)+"%")

plot(train_loss, valid_loss, train_acc, valid_acc)
```

6.2.1.py

```
import torch
from torchvision import transforms, datasets
import os.path

import skimage
import torch
import torch.nn as nn
from PIL import Image
```

```python
from torch.utils.data import Dataset,DataLoader
import matplotlib.pyplot as plt
import numpy as np
img_size = 64

def plot(train_loss, valid_loss, train_acc, valid_acc):
    # 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()

class CNN_Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.cnn_model = nn.Sequential(
            nn.Conv2d(3, 16, 5),
            nn.ReLU(),
            nn.MaxPool2d(2,2),

            nn.Conv2d(16, 32, 5),
            nn.ReLU(),
            nn.MaxPool2d(2,2),

            nn.Conv2d(32, 64, 5),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),

            nn.Flatten(),
            nn.Linear(1024, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 17)
        )

    def forward(self,x):
        out = self.cnn_model(x)
        return out

data_transforms = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.Resize((img_size, img_size)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                             std=[0.229, 0.224, 0.225])
    ])
trans_dataset =
↪   datasets.ImageFolder(root='../data/oxford-flowers17/train',
                                       transform=data_transforms)
train_loader = torch.utils.data.DataLoader(trans_dataset,
                                         batch_size=4, shuffle=True)
trans_dataset = datasets.ImageFolder(root='../data/oxford-flowers17/val',
                                       transform=data_transforms)
valid_loader = torch.utils.data.DataLoader(trans_dataset,
                                         batch_size=4, shuffle=True)

# model and loss, optimizer
model = CNN_Net()
print(model)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.002)

epoch = 30
train_loss = []
valid_loss = []
train_acc = []
valid_acc = []

for i in range(epoch):
    # training part
    print("Training iteration: " + str(i+1))
    size = len(train_loader.dataset)
    num_batches = len(train_loader)
    model.train()
    loss_val, correct = 0, 0
    for data in train_loader:
        x, y = data
        pred = model(x)
        loss = loss_fn(pred, y)
        #pred = nn.Softmax(dim=1)(pred)
        correct += (pred.argmax(1) == y).type(torch.float).sum().item()
        loss_val+=loss.item()

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    loss_val /= num_batches
    correct /= size
    train_loss.append(loss_val)
    train_acc.append(correct)
    print("Train loss: " + str(loss_val) + ", Accuracy: " + str(correct * 100) + "%")


    # validation part
    size = len(valid_loader.dataset)
    num_batches = len(valid_loader)
    model.eval()
    loss, correct = 0, 0
    with torch.no_grad():
        for X, y in valid_loader:
            pred = model(X)
            loss += loss_fn(pred, y).item()
            #pred = nn.Softmax(dim=1)(pred)
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    loss /= num_batches
    correct /= size
    valid_loss.append(loss)
    valid_acc.append(correct)
    print("Validation loss: "+str(loss)+", Accuracy: " + str(correct*100)+"%")

plot(train_loss, valid_loss, train_acc, valid_acc)
```

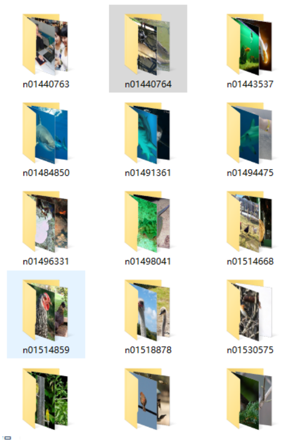
6.3 Neural Networks in the Real World

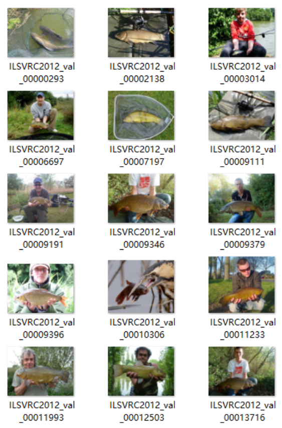
Often, we train neural networks on standard datasets and evaluate on held out validation data. How would a model trained on ImageNet perform in the wild?

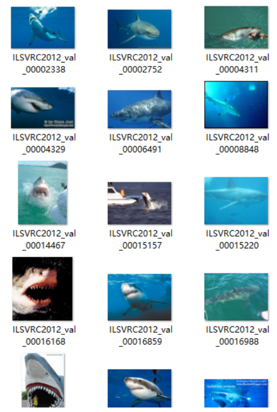
Q6.3 (Extra Credit) Neural Networks in the Real World [20 points] Download an ImageNet pretrained image classification model of your choice from torchvision. Using this model, pick a single category (out of the 1000 used to train the model) and evaluate the validation performance of this category. You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

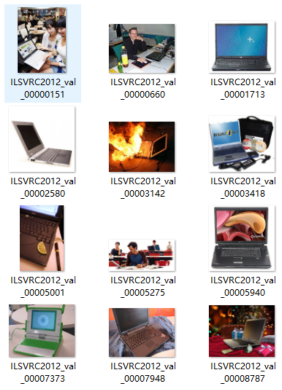
Q6.3

Create account and download validation data(test with 4 categories):

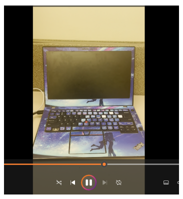








Test with categories:
Accurancy for category: 0 is 92.0
Accurancy for category: 1 is 92.0
Accurancy for category: 2 is 90.0
Also, use laptop(681) in dataset to test:
Accurancy for category: 681 is 54.0
Test with my video frames (also choose laptop 681):



Accurancy for category: 681 is 48.18181818181818% (106.0/220)
We can see that the result in validation dataset is much better than video I take. Because the image size, scale and processing may be different and the model is trained to fit better for validation data.
Ways: Train model with more datasets including my own data, do things like data transforming to make it better.
(code in next page:)

6.3.py

```python
import torch
import torchvision
from torchvision import transforms, datasets
import os.path

import skimage
import torch
import torch.nn as nn
from PIL import Image
from torch.utils.data import Dataset,DataLoader
import matplotlib.pyplot as plt
import numpy as np
from torchvision.models import resnet152,ResNet152_Weights
input_size = 224

data_transforms = transforms.Compose([
    transforms.RandomResizedCrop(input_size),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])

val_data=torchvision.datasets.ImageNet(root="D:\CV", split='val',
↪  transform=data_transforms)
val_data = DataLoader(val_data, batch_size=10)

# model and loss, optimizer
model = resnet152(weights=ResNet152_Weights.IMAGENET1K_V2)
model.eval()
k = 0
acc=0
sum=0
# tench and goldfish
with torch.no_grad():
    for X, y in val_data:
        if int(y[0])!=k:
            # accurancy
            res = (acc/sum)*100
            print("Accurancy for category: "+str(k) +" is "+str(res)+"%")
            # next category
            k+=1
            sum=0
            acc=0
        # only get accurancy for first 2
        if k==3:
            break
        pred = model(X)
        acc+=(pred.argmax(1) == y).type(torch.float).sum().item()
        sum+=10
```

6.3real.py

```
import torch
import torchvision
from torchvision import transforms, datasets
import os.path

import skimage
import torch
import torch.nn as nn
from PIL import Image
from torch.utils.data import Dataset,DataLoader
import matplotlib.pyplot as plt
import numpy as np
from torchvision.models import resnet152,ResNet152_Weights
img_size = 240
input_size = 224

data_transforms = transforms.Compose([
    transforms.Resize((img_size, img_size)),
    transforms.CenterCrop((input_size, input_size)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])

# car
real = 681
hymenoptera_dataset = datasets.ImageFolder(root='../data/real_data',
                                           transform=data_transforms)
val_loader = torch.utils.data.DataLoader(hymenoptera_dataset,
                                            batch_size=10, shuffle=True)

# model and loss, optimizer
model = resnet152(weights=ResNet152_Weights.IMAGENET1K_V2)
model.eval()
acc=0
sum=0
with torch.no_grad():
    for X, y in val_loader:
        pred = model(X)
        acc+=(pred.argmax(1) == real).type(torch.float).sum().item()
        sum+=10
    res = (acc / sum) * 100
    print("Accurancy for category: " + str(real) + " is " + str(res) +
     ↪  "%")
```

6.3save.py

```
import cv2

cap = cv2.VideoCapture('../data/real.mp4')

i = 0
while (cap.isOpened()):
    ret, frame = cap.read()
    if ret == True:

        filename = '../data/real_data/1/' + str(i) + '.jpg'
        cv2.imwrite(filename, frame)
        i+=1
    else:
        break

cap.release()
cv2.destroyAllWindows()
```

Deliverables

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

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

References

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

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

Appendix: Neural Network Overview

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

Mathematical overview

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

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

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

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

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

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

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

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

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

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

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

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

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

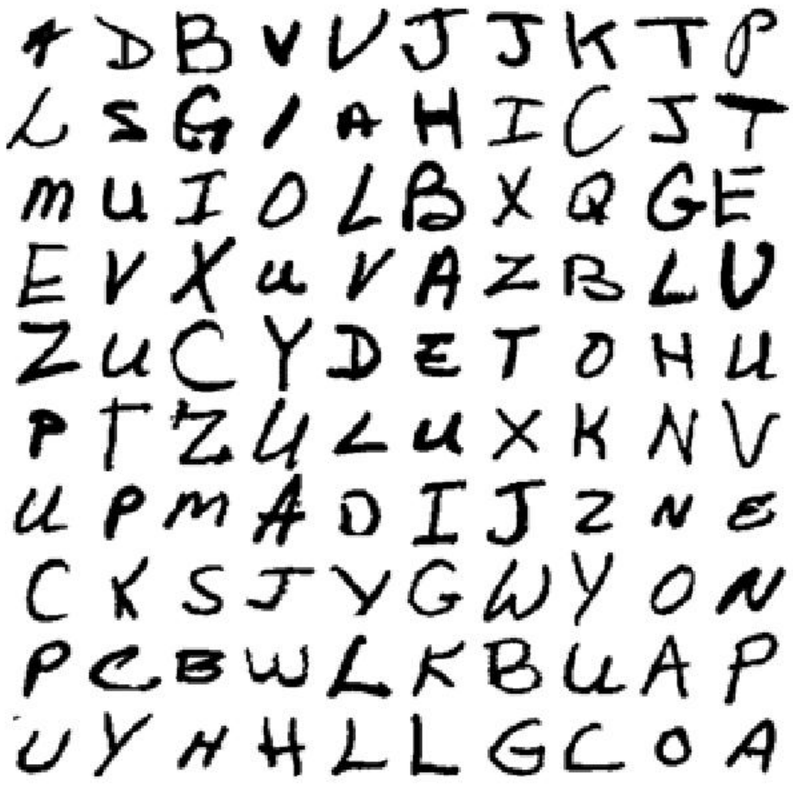


Figure 2: Samples from NIST Special 19 dataset [2]

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

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

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

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

Backprop

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

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

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

Debugging Checklist for Training Neural Networks

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