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

Q1.1)

$s(x_i) = \text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$

i) Softmax is invariant to translation

$$s(x_i + c) = \frac{e^{x_i+c}}{\sum_j e^{x_j+c}} = \frac{e^{x_i} \cdot e^{c}}{e^{c} \cdot \sum_j e^{j}} = \frac{e^{x_i}}{\sum_j e^{j}} = s(x_i)$$

Hence, it's invariant.

ii) The exp() of even a moderately large positive value may cause overflow issue. On top of that dividing a huge number may also cause arithmetic errors. But since, we just proved that softmax is translation invariant, we can subtract with max x_i to prevent these issues. Hence, generally its a good idea to use $c = -\max x_i$ while calculating softmax.

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.) The range of each element of the softmax(x) would be [0, 1]. And, the sum of all elements in the softmax(x) would be 1.

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

Q1.2.2

2.) Probability Distribution

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

Q1.2.3

3.) Step 1: $s_i = e^{x_i}$

Calculates the exp() of i^{th} element of the d dimensional vector x.

Step 2: $\sum_j s_j = S$.

Calculates the sum of all exp() values.

Step 3: $\frac{s_i}{S}$

Intuitively the most important which makes use of s_i and S so that each real valued element of vector x can be interpreted as a probability.

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

Q1.3

Let x_i be the input to the i^{th} layer then assuming w_i and b_i to be weights and biases of the i^{th} layer the output y_i can be written as follows

$$x_{i+1} = w_i^T x_i + b_i \quad , \quad x_i \in \mathbb{R}^d \quad b_i \in \mathbb{R}^k \quad w_i \in \mathbb{R}^{d \times k}$$

Similarly, for $i+2^{th}$ layer

$$x_{i+2} = w_{i+1}^T x_{i+1} + b_{i+1} \quad , \quad x_{i+1} \in \mathbb{R}^k \quad b_{i+1} \in \mathbb{R}^{k'} \quad w_{i+1} \in \mathbb{R}^{k' \times k}$$

$$= w_{i+1}^T (w_i^T x_i + b_i) + b_{i+1}$$

$$= w_{i+1}^T w_i^T x_i + w_{i+1}^T b_i + b_{i+1}$$

$$= w' x_i + b' \quad , \text{ where } w' \in \mathbb{R}^{k' \times d} \quad , \quad b' \in \mathbb{R}^{k'}$$

Hence from above we can see that the final output would be a linear combination of a learned weight(w) and bias (b) which is equivalent to 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

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

$$\frac{d\sigma(x)}{dx} = \frac{(1+e^{-x})\cdot(0) - (-e^{-x})(1)}{(1+e^{-x})^2}$$

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

$$= \frac{1}{(1+e^{-x})} - \left[\frac{1}{(1+e^{-x})}\right]^2$$

$$= \sigma(x) - \sigma^2(x)$$

$$= \sigma(x)\left[1 - \sigma(x)\right] \text{ //.}$$

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

Q1.5)

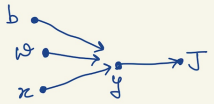
$$y = Wx + b$$

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

$W \in \mathbb{R}^{k \times d}$, $x \in \mathbb{R}^d$, $b \in \mathbb{R}^k$, $y \in \mathbb{R}^k$

$\frac{dJ}{dy} = \delta \in \mathbb{R}^k$

- Flow Graph



$\frac{dJ}{dW} = \frac{dJ}{dy} \cdot \frac{dy}{dW}$

$\frac{dJ}{dx} = \frac{dJ}{dy} \cdot \frac{dy}{dx}$, $\frac{dJ}{db} = \frac{dJ}{dy} \cdot \frac{dy}{db}$. ['s' based on flow graph.]

Now, $\frac{dy_i}{dx_j} = W_{ij}$, $\frac{dy_i}{dW_{ij}} = x_j$, $\frac{dy_i}{db_i} = 1$

Hence for all y_i's.

$$\frac{dy}{db} = \begin{bmatrix} \frac{dy_1}{db_1} & \cdots & \frac{dy_1}{db_k} \\ \vdots & & \\ \frac{dy_k}{db_1} & \cdots & \frac{dy_k}{db_k} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & & \vdots \\ \vdots & & \ddots & \\ 0 & \cdots & & 1 \end{bmatrix}_{k \times k}$$

$$\frac{dy}{dW} = \begin{bmatrix} \frac{dy_1}{dW_{11}} & \cdots & \frac{dy_1}{dW_{1d}} \\ \vdots & & \\ \frac{dy_k}{dW_{k1}} & \cdots & \frac{dy_k}{dW_{kd}} \end{bmatrix} = \begin{bmatrix} x_1 & \cdots & x_d \\ & \vdots & \\ x_1 & \cdots & x_d \end{bmatrix}_{k \times d}$$

↑ [This would be a 3D tensor but we are only considering a plane in that 3D tensor all values in that plane will be $[x_1, \ldots x_d] \to x^T$]

$$\frac{dy}{dx} = \begin{bmatrix} \frac{dy_1}{dx_1} & \frac{dy_1}{dx_2} & \cdots & \frac{dy_1}{dx_d} \\ \vdots & \vdots & & \\ \frac{dy_k}{dx_1} & \cdots & & \frac{dy_k}{dx_d} \end{bmatrix} = \begin{bmatrix} W_{11} & W_{12} & \cdots & W_{1d} \\ & \vdots & & \\ W_{k1} & \cdots & & W_{kd} \end{bmatrix} = W.$$

So,

$$\frac{dJ}{dW} = \delta x^T, \quad \frac{dJ}{db} = \delta, \quad \frac{dJ}{dx} = W^T \delta$$

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

1.) Derivative of sigmoid

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

and since $\sigma(x) \in (0,1)$ then $\sigma'(x)$ would range in $(0, 0.25]$. We can see from this range as when we take derivative across multiple layers, the overall value would drastically decrease hence leading to vanishing gradient problem.

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

2.)

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

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

$$= \frac{e^{x}+e^{-x}-2e^{-x}}{e^{x}+e^{-x}}$$

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

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

$$= 1 - 2[1-\sigma(2x)]$$

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

Output ranges of $\sigma(x)$ is $(0, 1)$ and since $\tanh(x) = 2\sigma(2x) - 1$ (a rescaled version of sigmoid)

$$0 < \sigma(2x) < 1$$

$$0 < 2\sigma(2x) < 2 \text{ , Multiplying by 2 both sides}$$

$$-1 < 2\sigma(2x) < 1 \text{ , Subtracting 1 both sides}$$

So, $\tanh(x)$ has range $(-1, 1)$

We prefer $\tanh(x)$ over $\sigma(x)$ because sigmoid has a max derivative of 0.25 whereas tanh has a max derivative of 1. Hence, if we used sigmoid activations then derivate across multiple layers would be small but since tanh derivatives are larger compared to $\sigma(x)$ it'll have less of a vanishing gradient problem. The other reason might be that $\tanh(x)$ has a wider range where +ve numbers are mapped to +ve and -ve numbers are mapped to -ve value.

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

3.) As mentioned in earlier question, the tanh derivatives range [0, 1] . and hence unlike sigmoid whose derivatives range (0, 0.25] tanh (x) would have 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

4.) This was shown in the earlier question when I calculated tanh (x) in terms of sigmoid to find the range of values.

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

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

Let's assume we initialized all the weights and biases of the network with zeros. Then for the first iteration, the output will be all zeros. Although all outputs will be zeros, the more important issue is that all outputs are the same, hence the gradient updates during backpropagation will remain the same for all the Ws and bs. Hence, in each iteration, all features learned would be symmetric which is not desired.

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
    b = np.zeros((out_size,))
    t = np.sqrt(6) / np.sqrt(in_size + out_size)
    W = np.random.uniform(low=-t, high=t, size=(in_size, out_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

Initialization with random numbers prevents the network from learning symmetric features so as to not hamper the learning capability of the network. To keep the layer-wise variance during the forward and backward pass the same we scale the initialization depending on layer size.

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

```
def sigmoid(x):
    res = None
    with np.errstate(invalid="ignore", over="ignore"):
        res = np.where(x < 0, np.exp(x) / (1 + np.exp(x)), 1 / (1 + np.exp(-x)))
    return res
```

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

```
def softmax(x):
    e_x = np.exp(x)
    res = e_x / (np.sum(e_x, axis=1))[:, np.newaxis]
    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):
    true_labels = np.argmax(y, axis=1)
    pred_labels = np.argmax(probs, axis=1)
    num_labels = list(range(y.shape[0]))
    loss, acc = (
        np.sum(-np.log(probs)[num_labels, true_labels]),
        np.sum(true_labels == pred_labels) / y.shape[0],
    )
    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

```python
def backwards(delta, params, name="", activation_deriv=sigmoid_deriv):
    grad_X, grad_W, grad_b = None, None, None
    W = params["W" + name]
    b = params["b" + name]
    X, pre_act, post_act = params["cache_" + name]
    grad_b = np.zeros_like(b)
    grad_W = np.zeros_like(W)
    grad_X = np.zeros_like(X)
    grad_W = X.T @ (delta * activation_deriv(post_act))
    grad_b = (np.ones((1, X.shape[0])) @ (delta *
    ↪ activation_deriv(post_act))).squeeze(
        axis=0
    )
    grad_X = delta * activation_deriv(post_act) @ W.T
    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

```python
def get_random_batches(x, y, batch_size):
    batches = []
    n = x.shape[0]
    indices = np.random.permutation(list(range(n)))
    num_batches = (n // batch_size) + (1 if n % batch_size != 0 else 0)
    for i in range(num_batches):
        start = batch_size * i
        end = batch_size * (i + 1)
        if end < len(indices):
            ind = indices[start:end]
        else:
            ind = indices[start:]
        x_batch = x[ind, :]
        y_batch = y[ind]
        batches.append((x_batch, y_batch))
    return batches
# Train Loop
for itr in range(max_iters):
    total_loss = 0
    avg_acc = 0
    for xb, yb in batches:
        h1 = forward(xb, params, "layer1")
        probs = forward(h1, params, "output", softmax)
        loss, acc = compute_loss_and_acc(yb, probs)
        total_loss += loss
        avg_acc += acc
        delta1 = probs - yb
        delta2 = backwards(delta1, params, "output", linear_deriv)
        backwards(delta2, params, "layer1", sigmoid_deriv)
        for k, v in sorted(list(params.items())):
            if "grad" in k:
                name = k.split("_")[1]
                params[name] = params[name] - learning_rate * v
    avg_acc = avg_acc / batch_num
    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

```
eps = 1e-6
for k, v in params.items():
    if "_" in k:
        continue
    # we have a real parameter!
    # for each value inside the parameter
    #   add epsilon
    if k[0] == "W":
        r, c = params[k].shape
        for i in range(r):
            for j in range(c):
                v[i][j] += eps
                # run the network
                h1 = forward(x, params, "layer1")
                probs = forward(h1, params, "output", softmax)
                # get the loss
                loss1, acc1 = compute_loss_and_acc(y, probs)
                # subtract 2*epsilon
                v[i][j] -= 2 * eps
                # run the network
                h2 = forward(x, params, "layer1")
                probs2 = forward(h2, params, "output", softmax)
                # get the loss
                loss2, acc2 = compute_loss_and_acc(y, probs2)
                # compute derivative with central diffs
                grad = (loss1 - loss2) / (2 * eps)
                params["grad_" + k][i][j] = grad
                # restore
                v[i][j] += eps
```

Q2.5

```
    else:
        r = len(params[k])
        for i in range(r):
            v[i] += eps
            # run the network
            h1 = forward(x, params, "layer1")
            probs = forward(h1, params, "output", softmax)
            # get the loss
            loss1, acc1 = compute_loss_and_acc(y, probs)
            # subtract 2*epsilon
            v[i] -= 2 * eps
            # run the network
            h2 = forward(x, params, "layer1")
            probs2 = forward(h2, params, "output", softmax)
            # get the loss
            loss2, acc2 = compute_loss_and_acc(y, probs2)
            # compute derivative with central diffs
            grad = (loss1 - loss2) / (2 * eps)
            params["grad_" + k][i] = grad
            # restore
            v[i] += eps
```

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:

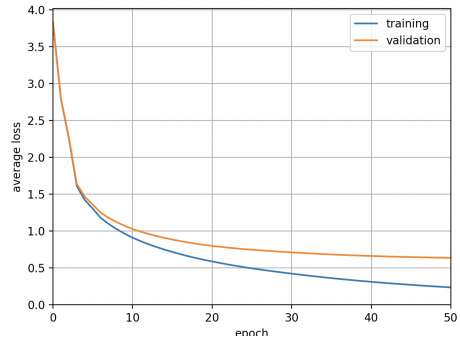
(1) the accuracy on both the training and validation set over the epochs, and

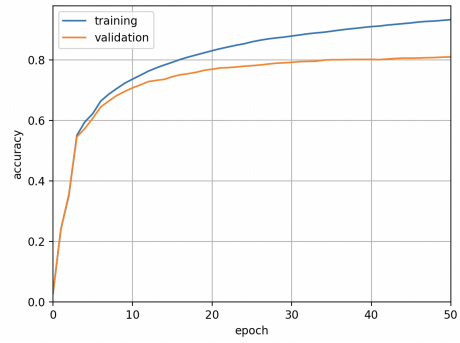
(2) the cross-entropy loss averaged over the data.

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

```
max_iters = 50
batch_size = 128
learning_rate = 5e-3
hidden_size = 256
# Validation Accuracy: ~ 82%
```





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,

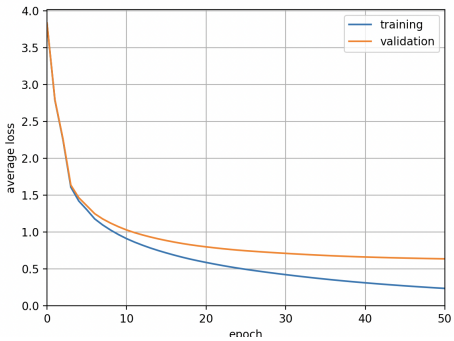
(2) one with 10 times that learning rate, and

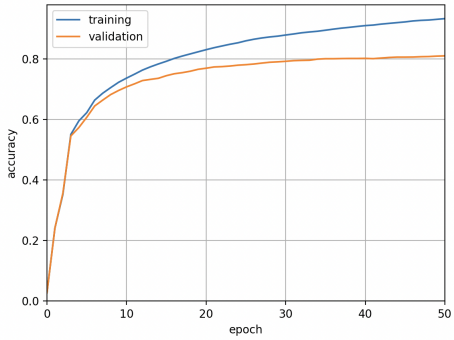
(3) one with one-tenth that learning rate.

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.*

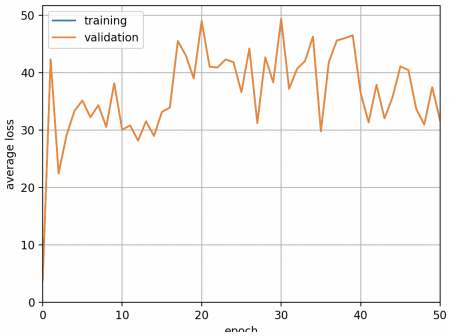
Q3.2

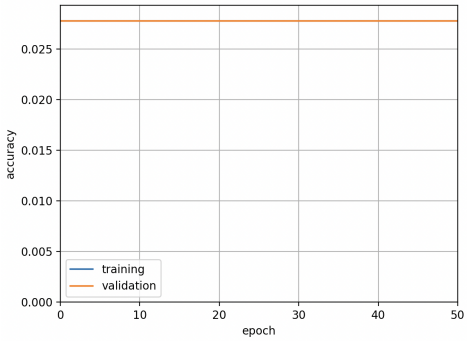
Tuned Learning Rate ($5e^{-}3$):



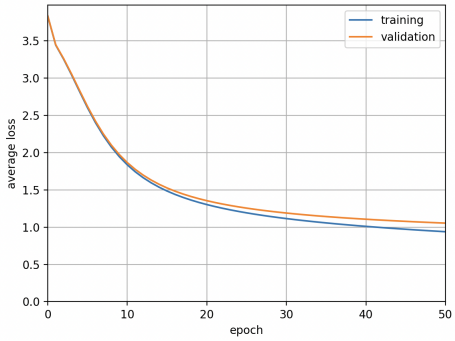


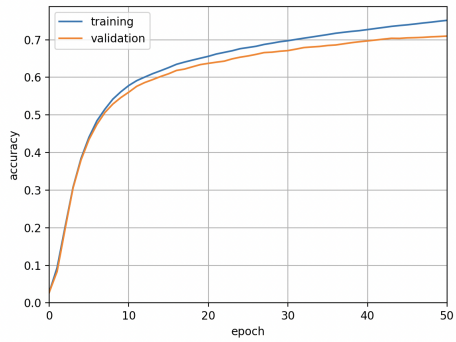
10 times Tuned Learning Rate ($5e^{-}2$):





Tuned Learning Rate by 10 ($5e^{-}4$):

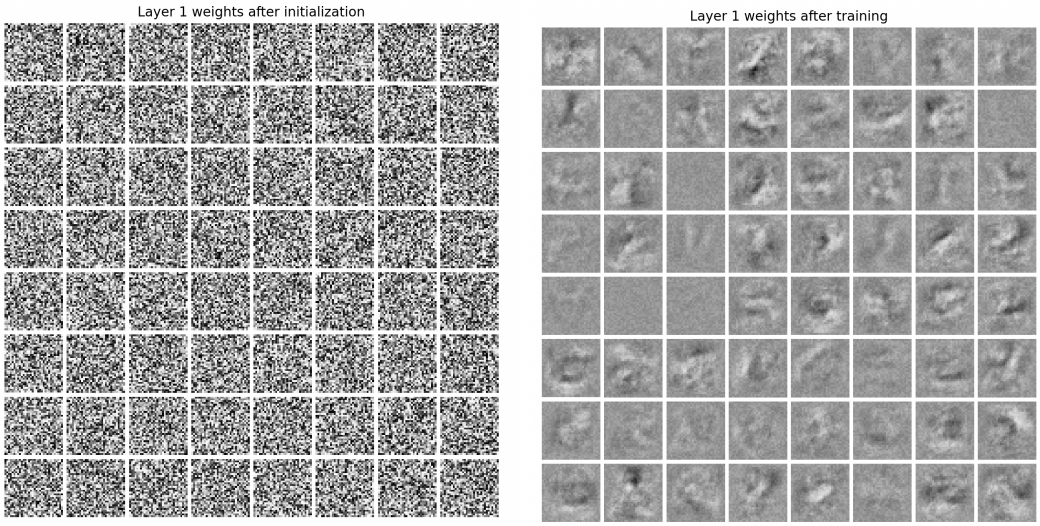




As can be seen from the graphs above, the learning rate is an essentially factor in solving the learning problem. When we use 10 times the tuned learning rate, the loss fluctuates but the accuracy doesn't improving showing signs we are overshooting in the loss curvature. When the we use tuned learning rate by 10 then we can see that both loss and acc hasnt converged yet.

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?

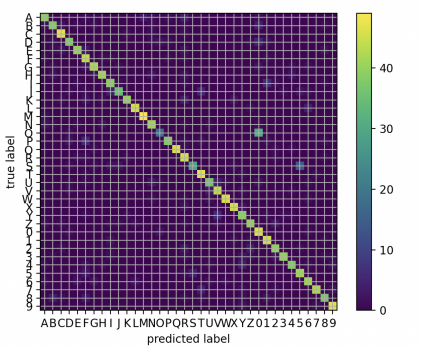
Q3.3



The initialized random weights are noisy and we dont see any upfront visible pattern. In the learned weights (on the right), we see some clear patterns.

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



To inspect the top few pairs of classes that are mostly confused, I qualitatively inspected the plot and noticed letter 'O' and number '0', letter 'S' and number '5', and letter 'I' and '1' were confused a lot.

4 Extract Text from Images

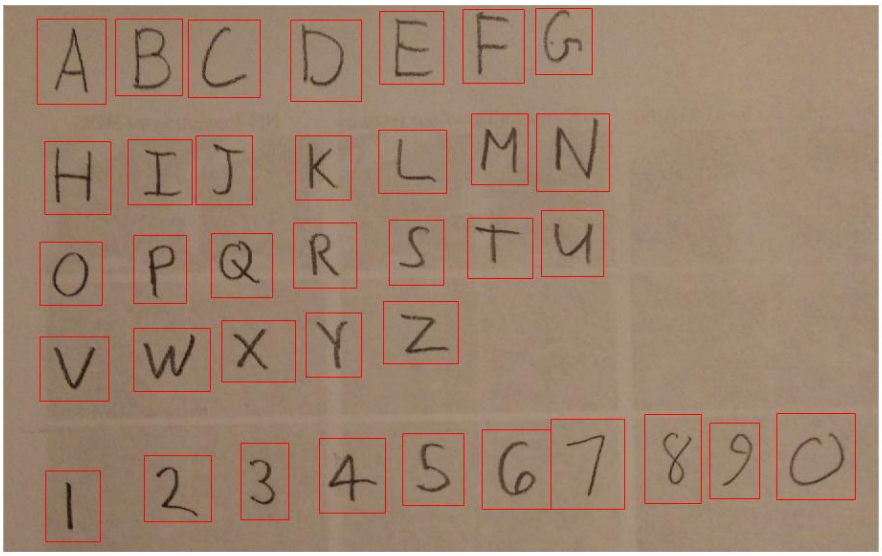


Figure 1: Sample image with handwritten characters annotated with boxes around each character.

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.

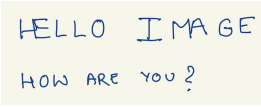
4. Take each bounding box one at a time and resize it to 32×32, classify it with your network, and report the characters in order (inserting spaces when it makes sense).

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

The two big assumptions I made in my implementation is that, no two characters are connected. So, there should be some space between two letters. The second assumption was that the rows have a distance of atleast 55 pixels or else the number of rows calculation logic would fail.
Two example images where I suspect the current implementation to miserably fail is as follows:



As can be seen in the first image two letters M, A and H, E are connected which might result in the bounding boxes to be not correct and hence prediction will most probably be wrong. Second image theres a non letter '?' which might be misclassified as 2.

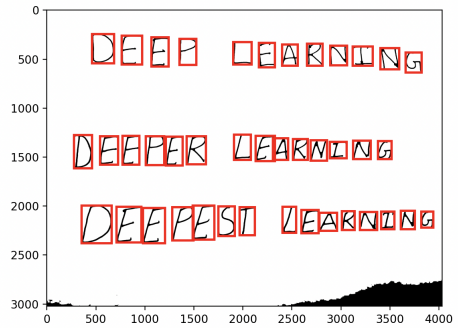
Q4.2 Find Letters - Code [10 points] In `python/q4.py`, implement the function to find letters in the image. Given an RGB image, this function should return bounding boxes for all of the located handwritten characters in the image, as well as a binary black-and-white version of the image `im`. Each row 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.**

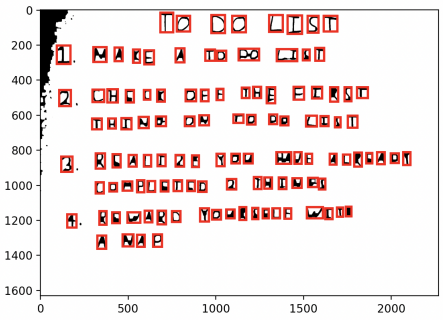
Q4.2

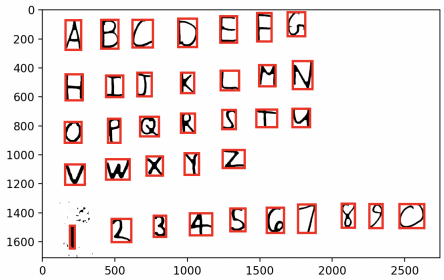
```python
def findLetters(image):
    bboxes = []
    bw = None
    gray = skimage.color.rgb2gray(image)
    gray = skimage.exposure.equalize_adapthist(gray)
    blur = skimage.filters.gaussian(gray, sigma=2.5)
    thresh = skimage.filters.threshold_otsu(blur)
    bw = blur < thresh
    bw = skimage.morphology.closing(bw, footprint=skimage.morphology.square(15))
    label_image = skimage.measure.label(
        skimage.segmentation.clear_border(bw), background=0
    )
    regions = skimage.measure.regionprops(label_image)
    bboxes = [region.bbox for region in regions if region.area > 400]
    remove_indices = []
    for i in range(len(bboxes)):
        q_minr, q_minc, q_maxr, q_maxc = bboxes[i]
        for j in range(len(bboxes)):
            if i != j:
                k_minr, k_minc, k_maxr, k_maxc = bboxes[j]
                if (q_minr >= k_minr and q_minc >= k_minc) and (
                    q_maxr <= k_maxr and q_maxc <= k_maxc
                ):
                    remove_indices.append(i)
                    break
    bboxes = [bboxes[i] for i in range(len(bboxes)) if i not in remove_indices]
    return bboxes, 1 - bw
```

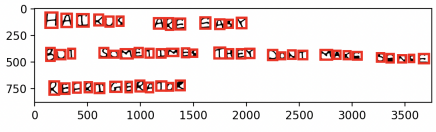
Q4.3 Eval Bounding Box Accuracy - Writeup [5 points] Using `python/run_q4.py`, visualize all of the located boxes on top of the binary image to show the accuracy of your `findLetters(..)` function. Include all the resulting images in your write-up.

Q4.3









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

True	Predicted
DEEP LEARNING DEEPER LEARNING DEEPEST LEARNING	0CEF CE2RNING DEEFEF LEARNING 05CFE5T LEARNING
TO DO LIST 1.MAKE A TO DO LIST 2.CHECK OFF THE FIRST THING ON TO DO LIST 3. REALIZE YOU HAVE ALREADY COMPLETED TWO THINGS 4. REWARD YOURSELF WITH A NAP	T0 D0 LIST I MAKE A T0 UUST 2 CHECK 0FF THE FIRST THIN0 0N T0 D0 LIST 3 REALIZE Y0U HUE ALREADT COMPLETED 2 THINFI 4 REWARD YOUREELF WITQ A NAP
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 1 2 3 4 5 6 7 8 9 0	A B C 0 E F G H I J K L M N 0 P Q R 5 T 4 V W X Y Z Q 2 3 4 5 G 7 8 7 0
HAIKUS ARE EASY BUT SOMETIMES THEY DONT MAKE SENSE REFRIGERATOR	HAIKUS ARE EASX BUT S0METIMES THEX 00NT MAK6 S6NGE REFRIGERAT0R

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

```
hidden_size = 32

initialize_weights(1024, hidden_size, params, "layer1")
initialize_weights(hidden_size, hidden_size, params, "layer2")
initialize_weights(hidden_size, hidden_size, params, "layer3")
initialize_weights(hidden_size, 1024, params, "output")
```

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

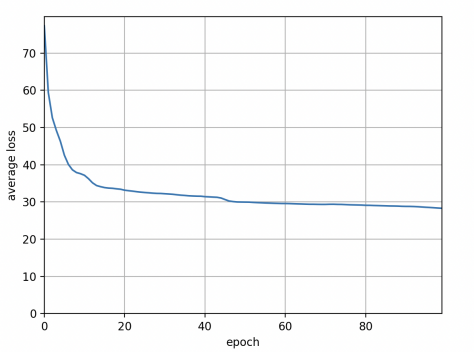
```
params["m_Wlayer1"] = np.zeros((1024, hidden_size))
params["m_blayer1"] = np.zeros(hidden_size)
params["m_Wlayer2"] = np.zeros((hidden_size, hidden_size))
params["m_blayer2"] = np.zeros(hidden_size)
params["m_Wlayer3"] = np.zeros((hidden_size, hidden_size))
params["m_blayer3"] = np.zeros(hidden_size)
params["m_Woutput"] = np.zeros((hidden_size, 1024))
params["m_boutput"] = np.zeros(1024)
losses = []
for itr in range(max_iters):
    total_loss = 0
    for xb, _ in batches:
        h1 = forward(xb, params, "layer1", relu)
        h2 = forward(h1, params, "layer2", relu)
        h3 = forward(h2, params, "layer3", relu)
        out = forward(h3, params, "output", sigmoid)
        loss = np.sum((xb - out) ** 2)
        total_loss += loss
        delta1 = 2 * (out - xb)
        delta2 = backwards(delta1, params, "output", sigmoid_deriv)
        delta3 = backwards(delta2, params, "layer3", relu_deriv)
        delta4 = backwards(delta3, params, "layer2", relu_deriv)
        backwards(delta4, params, "layer1", relu_deriv)
        for k, v in sorted(list(params.items())):
            if "grad" in k:
                name = k.split("_")[1]
                params[f"m_{name}"] = 0.9 * params[f"m_{name}"] - learning_rate *
                ↪  v
                params[name] += params[f"m_{name}"]

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

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

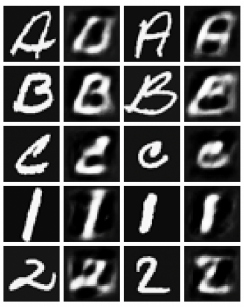


The loss decreases drastically for the first 15 epochs and then it kind of saturates but is still visibly decreasing. Overall it smoothly decreases with no visible oscillations across iterations. It seems that the training can be continued a bit more longer as the loss hasnt converged yet.

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



Based on the visualization above, we can that the reconstructed validation images resemble the original ones but are blurry. This means that the encoder has not yet learned to capture all the relevant features which makes sense since it wasn't trained long enough epochs.

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

Avg. PSNR: 15.317337913343552

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).

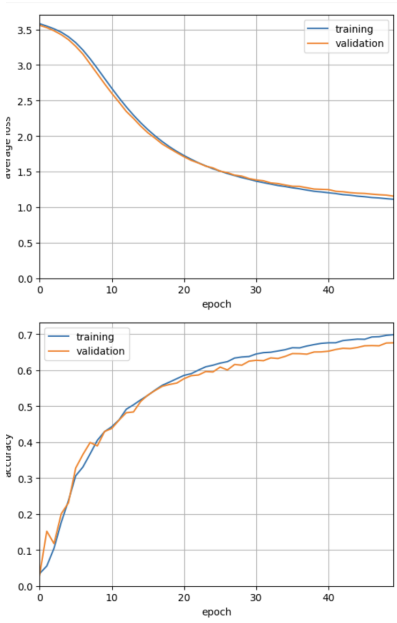
For this section, you're free to implement these however you like. All of the tasks required here are fairly small and don't require a GPU if you use small

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.

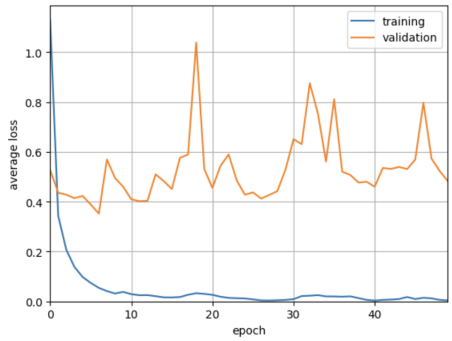
Q6.1.1

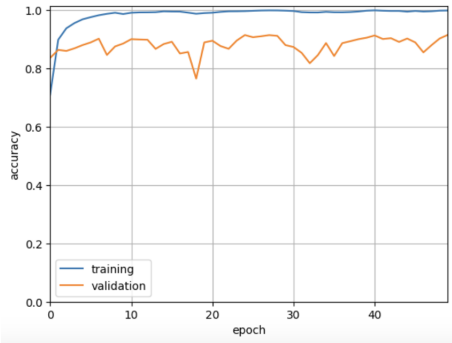


MLP Net NIST Test Accuracy: **69%**

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



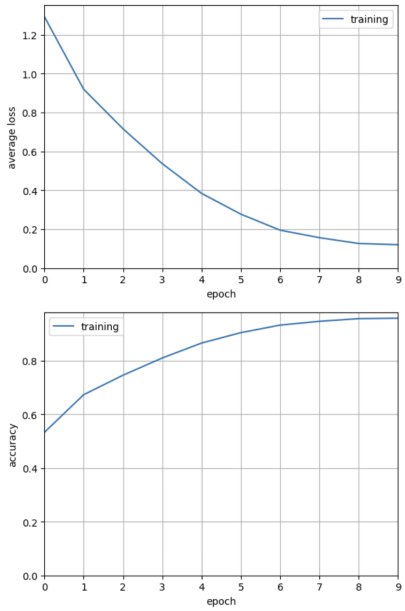


Layer	Kernel Shape	Output Shape	Params	Mult-Adds
0_backbone.Conv2d_0	[1, 64, 3, 3]	[128, 64, 18, 18]	640.0	186624.0
1_backbone.BatchNorm2d_1	[64]	[128, 64, 18, 18]	128.0	64.0
2_backbone.GELU_2	-	[128, 64, 18, 18]	NaN	NaN
3_backbone.Conv2d_3	[64, 128, 5, 5]	[128, 128, 10, 10]	204928.0	20480000.0
4_backbone.BatchNorm2d_4	[128]	[128, 128, 10, 10]	256.0	128.0
5_backbone.GELU_5	-	[128, 128, 10, 10]	NaN	NaN
6_head.Flatten_0	-	[128, 12800]	NaN	NaN
7_head.Linear_1	[12800, 128]	[128, 128]	1638528.0	1638400.0
8_head.BatchNorm1d_2	[128]	[128, 128]	256.0	128.0
9_head.GELU_3	-	[128, 128]	NaN	NaN
10_head.Dropout_4	-	[128, 128]	NaN	NaN
11_head.Linear_5	[128, 36]	[128, 36]	4644.0	4608.0

MLP Net NIST Test Accuracy: **91%**
As can be seen from the above avg. loss and acc. plots across iterations, the conv net architecute performs better than fully connected network. This network converges much faster than MLP net and the test accuracy is 91% compared to MLP net test acc of 69% (more than 20%).

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

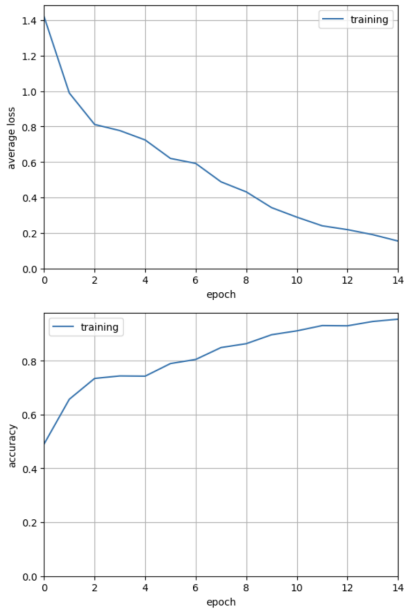


Test Accuracy: **70%**

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

Achieved a test Accuracy: **70%** with the same conv net architecture as in the previous question (Q6.1.3). In HW1 for scene classification I achieved best classification accuracy of 61% with 30 clusters for K-Means. Whereas with the current model achieved test accuracy of 70% which is more than 10% of the old result.



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.

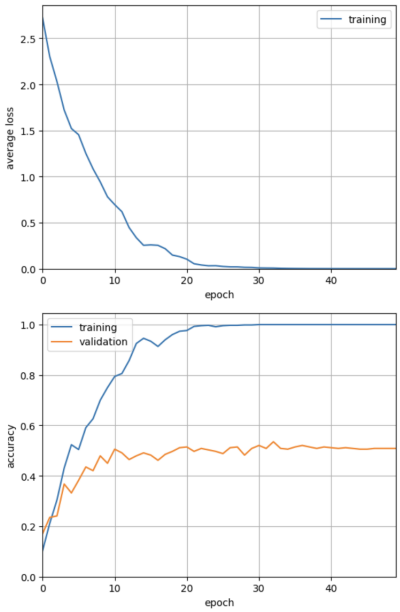
These days, it is most common to initialize a network with weights from another deep network that was trained for a different purpose. This is because, whether we are doing image classification, segmentation, recognition etc..., most real images share common properties. Simply copying the weights from 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

Conv Net (Similar to previous question q6.1.3) performance is as follows:



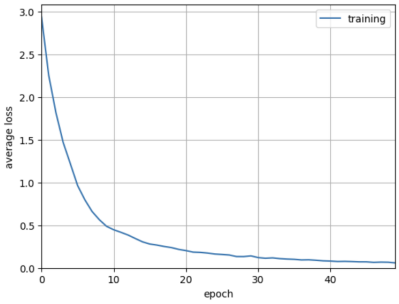
Achieved Test Accuracy: 0.4970588235294118.
Fine tuning SqueezeNet1_1:

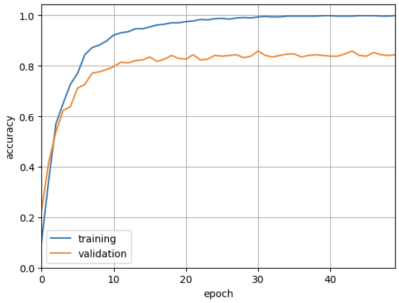
```
class SqueezeNet(torch.nn.Module):
  def __init__(self, in_channels=3, out_size=17):
    super(SqueezeNet, self).__init__()
    self.backbone =
    ↪  torchvision.models.squeezenet1_1(weights="SqueezeNet1_1_Weights.IMAGENET1K_V1").features
    self.classifier = torch.nn.Sequential(
        torch.nn.Dropout(0.1),
        torch.nn.Conv2d(512, out_size, kernel_size=1),
        torch.nn.ReLU(),
        torch.nn.AdaptiveAvgPool2d((1, 1))
    )
  def forward(self, x):
    embedding = self.backbone(x)
    out = self.classifier(embedding)
    out = torch.flatten(out, 1)
    return out

for param in snet.parameters():
  param.requires_grad = False

for param in snet.classifier.parameters():
  param.requires_grad = True
```

Q6.2





Achieved Test Accuracy: 0.8205882352941176

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

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:

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

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.