Ross Brancati
CS 589 – Homework Report 5
November 19, 2021

1 – Convolutional Neural Networks

1. Parameter Calculation
(a) To calculate the number of parameters in the fully connected layer, use the following equation:

$$\#params = (\#current\ layer\ neurons * \#previous\ layer\ neurons) + (1 * \#current\ layer\ neurons)$$

- *previous layer neurons = $N * N$*
- *current layer neurons = $N * N * C$*
- 1 represents the bias term

$$\#params = \big((N * N * C) * (N * N)\big) + \big(1 * (N * N * C)\big) = \boldsymbol{N^4C + N^2C}$$

I was a little bit confused on the wording of this question. I wasn't sure if the first layer was NxN and the second layer was NxNxC, or if you were just looking for the number of parameters in the NxNxC layer. If the question was only looking for the number of parameters in the NxNxC layer, it should be:

$$\big((\mathrm{N} * \mathrm{N} * 1) + 1\big) * \mathrm{C} = \mathbf{CN^2 + C}$$

(b) In a convolutional layer, the number of parameters is calculated as follows:

$$\#params = \big((filter\ height * filter\ width * \#filters\ in\ previous\ layer) + 1\big) * \#filters\ in\ current\ layer$$

- *filter height = k*
- *filter width = k*
- *# filters in previous layer = 1*
- *# filters in current layer = C*

$$\#params = \big((k * k * 1) + 1\big) * C = (k^2 + 1) * C = \boldsymbol{Ck^2 + C}$$

2. CNN Implementation
(a)

Table 1: Convolutional neural network layer details, output size, and number of parameters.

Layer	**Activation**	**# Filters**	**Filter Size**	**Stride**	**Padding**	**Output Size**	**# Params**
Input (28x28)							
Conv2d	ReLU	32	3x3	1	0	32x26x26	320
Conv2d	ReLU	64	3x3	1	0	64x24x24	18,496
MaxPool2d	-	-	2x2	2	0	64x12x12	0
Dropout2d (p=0.25)						64x12x12	0
Flatten()						9,216	0
Linear	ReLU	128	-	-	-	128	1,179,776
Dropout2d (p=0.5)						128	0
Linear	Softmax	10	-	-	-	10	1,290

(b)

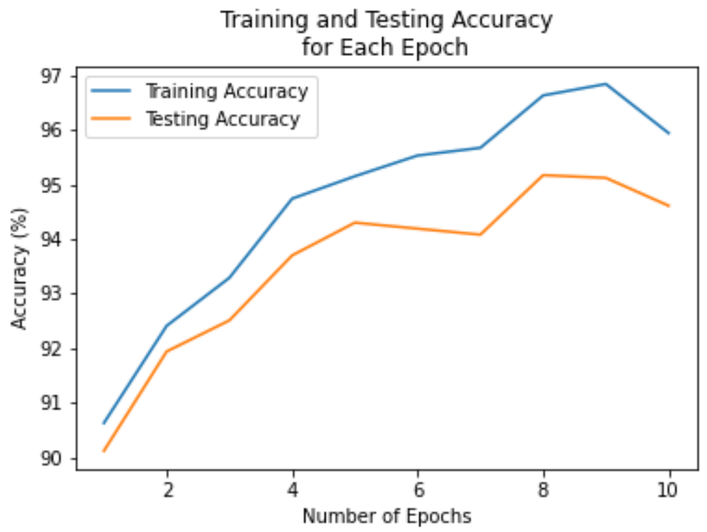

Figure 1: Training and testing accuracy for MNIST digits dataset. Training and testing data increase up to 8 and 9 epochs, respectively, then begins to slightly decline.

Table 2: Final training and testing accuracy after 10 epochs for the MNIST digits dataset.

Dataset	**Final Accuracy**
Training	96%
Testing	95%

(c)

Table 3: Percentage of parameters in each convolutional and fully connected layer.

Layer	**Number of Parameters**	**Percentage of Total Parameters**
Conv2d (1)	320	0.027%
Conv2d (2)	18,496	1.541%
Linear (1)	1,179,776	98.32%
Linear (2)	1,290	0.108%
Total Number of Parameters	1,199,882	100%

The results in *Table 3* are not surprising to me. I would expect that the convolutional layers have less parameters than the fully connected layers. After all, the point of having the convolutional layers is to improve efficiency of learning by the neural network. The convolutional layers are supposed to extract the best features in the data, whereas the fully connected layers extract all features. More features mean more training time and some of the features are not as important as others. To conclude, I am not surprised by these results because convolutional layers should have less parameters as they extract the important features from the data, whereas fully connected layers make no assumptions about the input and extract all features of the data.

2 – Singular Value Decomposition

1. SVD Function Algorithm

```
def SVD(A, s, k):
    # TODO: Calculate probabilities p_i
    n,m = A.shape
    #create a matrix to store probabilities in
    p_i_all = np.zeros(n)
    for i in range(n):
        #each pi should be a scalar, so p_i_all should be vector of length n (number of rows)
        #in my python script, the next two lines are actually one, but it wouldn't fit on this page
        p_i = (np.linalg.norm(A[i,:])*np.linalg.norm(A[i,:]))/
        (np.linalg.norm(A, ord='fro')*np.linalg.norm(A, ord='fro'))
        p_i_all[i] = p_i

    # TODO: Construct S matrix of size s by m
    #create a matrix of zeros to store values
    S = np.zeros((s,m))
    #pick a random integer j with probability Pr(pick j)=p_j
    for i in range(s):
        #use random choice function to select integer and store in S
        j = np.random.choice(n, replace=False, p=p_i_all)
        S[i] = A[j,:]

    # TODO: Calculate SS^T
    sst = np.dot(S,np.matrix.transpose(S))

    # TODO: Compute SVD for SS^T
    w, s, vh = np.linalg.svd(sst)

    # TODO: Construct H matrix of size m by k
    # create matrix H to store data
    H = np.zeros((m,k))
    #loop over k rows and calculate h_t for each row
    for i in range(k):
        #in my python script, the next two lines are actually one, but it wouldn't fit on this page
        h_t = (np.dot(np.matrix.transpose(S),vh[i,:]))/
        (np.linalg.norm((np.dot(np.matrix.transpose(S),vh[i,:]))))
        #store h_t in matrix H
        H[:,i] = h_t

    # Return matrix H and top-k singular values sigma
    return H, s[0:k]
```

2. Sub-Optimal Calculation Explanation

To calculate the optimal approximation of the data matrix, X, we can used singular value decomposition which yields the right singular values. Interestingly, these are also the eigenvectors which can be shown by the relationship between principal component analysis and SVD. Take data matrix, *X*, where X_k is the k-rank approximation of X:

$$X_k^T X_k = (USV^T)^T(USV^T) = VS^TU^TUSV^T$$

$$U^TU = 1, and\ so\ VS^TU^TUSV^T = VDV^T$$

Where D is a matrix where diagonal entries are equal to the squares of diagonals of S because S itself is a diagonal matrix. The rows of V^T are the eigenvectors of X^TX by the definition of the eigenvector. So, the right singular values are the eigenvectors of X. In the algorithm in part 1, we calculate an approximation of the right singular values, H, which means that H is an approximation of the eigenvectors of X. Multiplying HH^T results in a diagonal matrix that is very close to the identity matrix. Since H is just a sub-optimal approximation of the eigenvectors, multiplying the original matrix *X* by HH^T yields a sub-optimal approximation of the original data matrix *X*.

3. Baboon Images

(a)

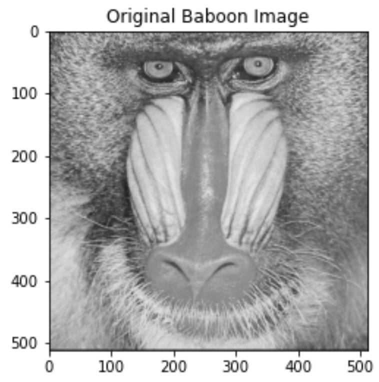

Figure 2: Plot of original image of baboon.

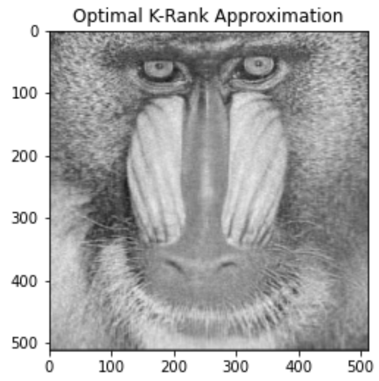

Figure 3: Plot of optimal k-rank approximation (X_k) of original baboon image.

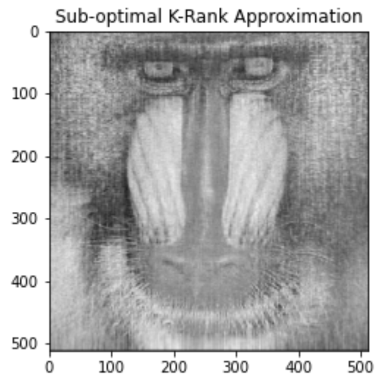

Figure 4: Plot of sub-optimal k-rank approximation ($\hat{X}_k$) of original baboon image.

(b)

Table 4: Error of optimal and sub-optimal 60 rank approximations.

Approximation	**Equation**	**Error**
Optimal	$\|A - A_k\|_F$	8827.53
Sub-Optimal	$\|A - \hat{A}_k\|_F$	11499.95

3 – Principal Component Analysis

1.

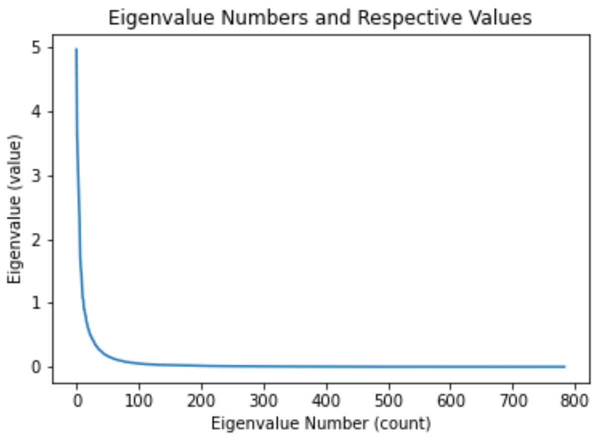

Figure 5: Plot of eigenvalues in descending order. The x-axis is the eigenvalues number for the respective eigenvector, and the y-axis represents the value of the respective eigenvalue. Note: the np.linalg.eigh calculates the eigenvectors and eigenvalues in ascending order, so I reversed the order to create this plot. So the first eigenvalue is actually representative of the last eigenvalue in the matrix generated by the python function.

Based on *Figure 4*, I would choose somewhere between 50-100 eigenvectors in my analysis. the eigenvalues represent the amount of variance explained by the eigenvector, and I would only be interested in the eigenvalues that explain a portion of the variance in the dataset. From *Figure 4,* you can see that the eigenvalues sharply decline at the beginning and plateau to around 0 at about 100 eigenvalues. So, eigenvalues further than somewhere around 100 wouldn't really help to explain the variance in the data, and thus would be wasteful to include, especially if you are using PCA as a dimensionality technique. It should also be noted that the eigenvectors that only explain a small portion of the variance can be very important in some scenarios, such as in my field of study (biomechanics), where we use PCA to analyze waveform data. Although we typically choose all of the eigenvectors that collectively explain 90-95% of the variance in the data, the eigenvectors that explain just 1-2% of the data may be important for determining differences in movement patterns between a healthy individual and one with a pathology such as knee osteoarthritis.

2.

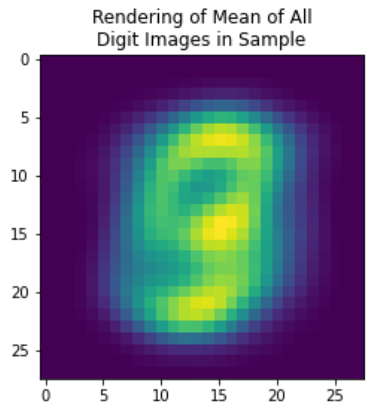

Figure 6: Rendering of the sample mean of the dataset. In this image, you can see portions of numbers, such as a 3, the upper loop of a 9, and possibly even the upper portion of an 8.

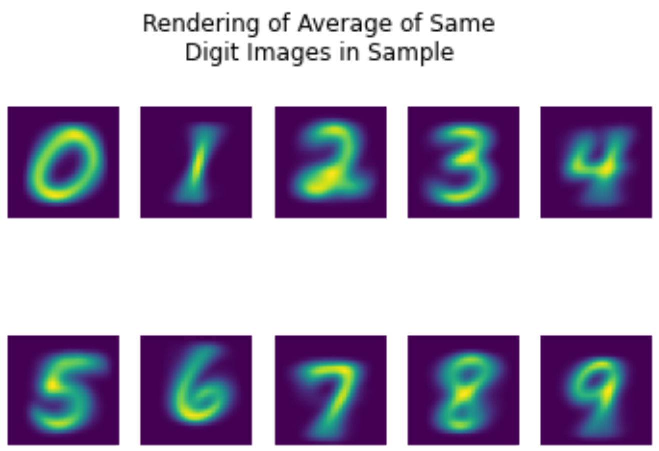

Figure 7: Rendering of the mean of each digit in the dataset. It is obvious that even if we take the mean value of the numbers, the plot of the mean will look like the number. Essentially, this tells us the handwritten digits in the dataset were written in very similar fashions.

3.

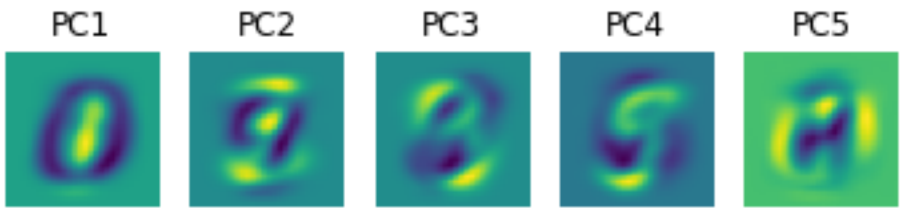

Figure 8: Top five eigenvectors (also known as principal components) plotted as images. Eigenvector 1 (principal component 1) is underneath PC1, eigenvector 2 is under PC2, etc. PC stands for principal component.

Starting with eigenvalue 1 and moving to eigenvector 5 in *Figure 7*, you can see 0, 9, 8, 9, and 8, respectively. However, this plot was generated with a color map that should also be mentioned. I believe that the dark values (purplish) represent greater values and lighter (yellowish) represent smaller values, which is important for interpretation. So, I think that the first PC captures a lot of the variance in the edges of the numbers, which is why it looks like a circle. All numbers have edges to them, but not all numbers have that middle space that is yellowish, such as 1. So, I think that the first PC is representative of outside edges of numbers. My interpretation of the second PC is that it captures variance around the top half of numbers where there is a loop. A lot of numbers (2, 3, 5, 8, and 9) have a portion that has this upper loop or has a portion of the top that is circular looking, so these dark areas in this image most likely capture that variance. PC3 is challenging because the darker areas look like a blur, but I think that this is capturing variance between the numbers in the middle section of them but is not so much capturing the loop mentioned for the second PC. The fourth PC is in a similar boat. I think it may be capturing variance in the bottom left, top left, and bottom right of the numbers. Lastly, PC5 appears to be capturing variance in the handwritten digits across the middle where you can see that darker line streaking across. This is likely coming from numbers such as 2, 3, 4, 5, 6, and 8, all of which have a darker region across the middle. Overall, interpreting the meaning of eigenvectors or principal components, especially ones that have larger eigenvalues, is challenging because you are reducing many data points into one vector. If you were to look at the PCs that explain only a very small portion of the variance, I would guess that only one of the pixels in the image would be dark.

4.

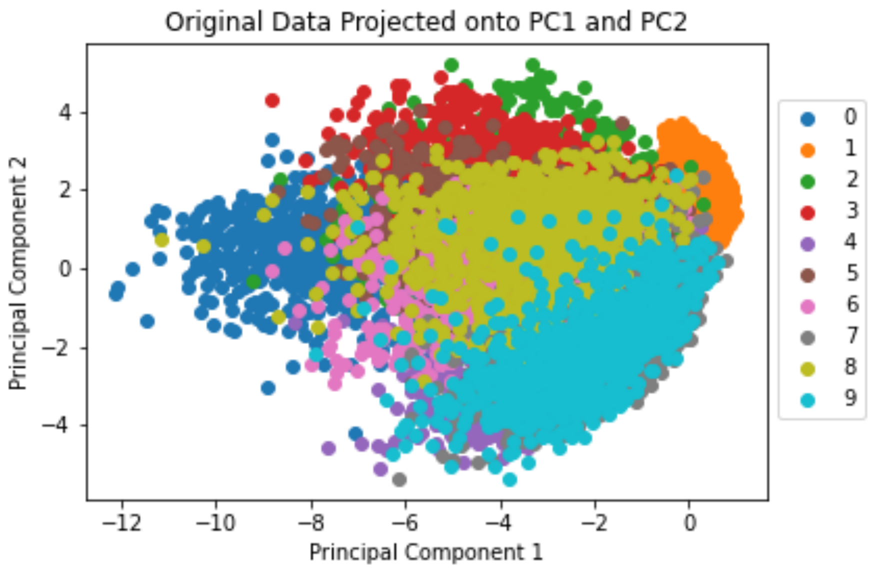

Figure 9: Plot of projections of original data onto the first and second principal components with each color representing a different digit in the dataset.

Figure 8 is a color-coded projection of each digit onto the first and second principal components. From the plot and the legend, we can see that there is some subtle separation between digits. On the left side of the plot is 0, which is not really similar to any other digits which is why it is somewhat on its own over there. Digits 2, 3, and 5 are all higher up on the plot and are towards the center of the first principal component, which makes sense because they look somewhat similar compared to other digits. In the same fashion, digits 4, 7, and 9 are mostly in the bottom right corner of the cloud. Again, this makes sense because they all look similar. The 1 digit is an interesting observation because its standard deviation is so low. It is literally just a single line down the page when you write it, so this isn't surprising because it's not possible to have high variability when writing a 1. Lastly, digits 6 and 8 are more toward the middle of the cloud, and this is because they also look similar, especially because one of the most prominent features of both of these numbers is the loop at the bottom. In fact, if you wanted to use a pen to turn a 6 into an 8, all you would really have to do is continue that top portion of the 6 and connect it back down to the loop. This example shows that principal component analysis is a valuable tool for reducing dimensions and separating data for classification purposes.