Machine Learning Final Report
Facial Recognition with CNN and CRC

Gregory Williams
gwilli64@uncc.edu

Sam Luu
sluu3@uncc.edu

Nathan Langley
nlangle6@uncc.edu

***Abstract*—With the various techniques, algorithms, and solutions covered in ECGR 4090 - Introduction to Machine Learning, one of the most known applications of machine learning is facial recognition. The final project takes two techniques shown in the class to compare the accuracy and results of a simple facial recognition application. A known method in the deep learning community are convolutional neural networks (CNN). The other method that was introduced was the collaborative representation classifier. The sparse representation method was shown to work well with facial recognition or image classification.**

The project takes five different celebrities with approximately 700 images for training and 100 pictures for testing. The data set was created using OpenCV and utilizing the Pickle Python library to format the images and labels. Each method has its own program that separately utilizes the testing and training data. Before saving the data and labels to their own files, they were shuffled to enhance the training for both methods. The CNN and the CRC programs will import the shuffled data and labels for training and testing.

The implementations of the both programs were not as successful as was intended, because the accuracy for both were lower than expected. When comparing the results, the CNN method had a higher validation accuracy than the CRC method.

I. Introduction

A. Convolution Neural Network

Convolutional neural networks are extremely common in image recognition and classifications. There are three layers to a convolutional neural network: convolutional layer, pooling layer, and the fully-connected layer. The convolution and pooling layers can be repeated as many times as necessary before moving onto the fully connected layer. All the layers are shown in Fig 1 [1].

The convolution layer is the main part of the deep learning technique. It requires an input data, a filter, which results in a feature map. Depending on the size and the stride of the filter, the size of the feature map varies. The stride is the number of pixels that the filter will move around in the input matrix. Filters could include padding. The padding on the filter affects the feature map just as much as the size and stride of the filter. If the filter has zero-padding, then it will set all elements that fall outside the input matrix to zero, shown in Fig 2.

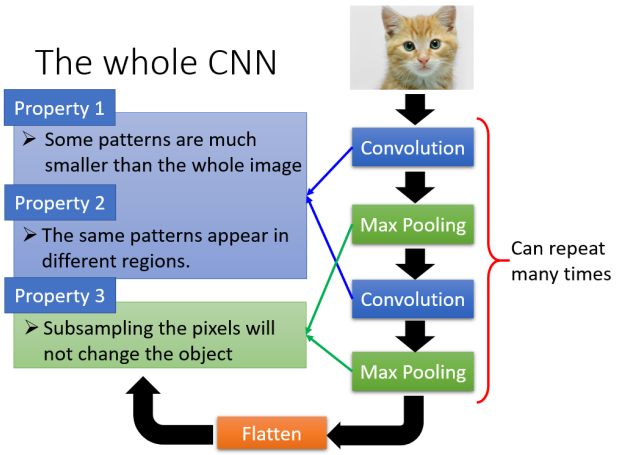
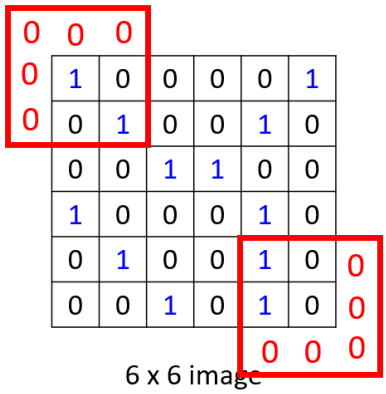
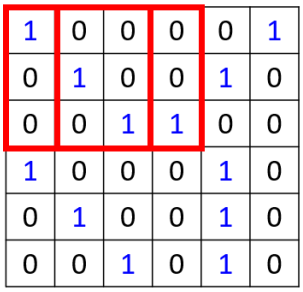


Fig. 1. Overview of CNN



(a)



(b)

Fig. 2. Zero-padding filter (a) vs No-padding filter (b)

The next layer after the convolution layer with filtering is the pooling layer. The pooling layer is another way to reduce the size of the feature map other than changing dimensions, stride, or padding on the filter. A feature map could be at 4x4 but reduced to 2x2 with pooling. The most common type is max pooling. A filter similar to that of the filter in the convolution layer will sweep across the feature map with a specific size and stride. Those filters determine the sub sections that will then be dwindled down to one unit. Each sub-field will have the largest value inputted into the smaller feature map, shown in Fig 3. Another type of pooling is average pooling. The format will the same as max pooling, but as the filter sweeps the map, it will input the average value of each sub-field into the final feature map [1].

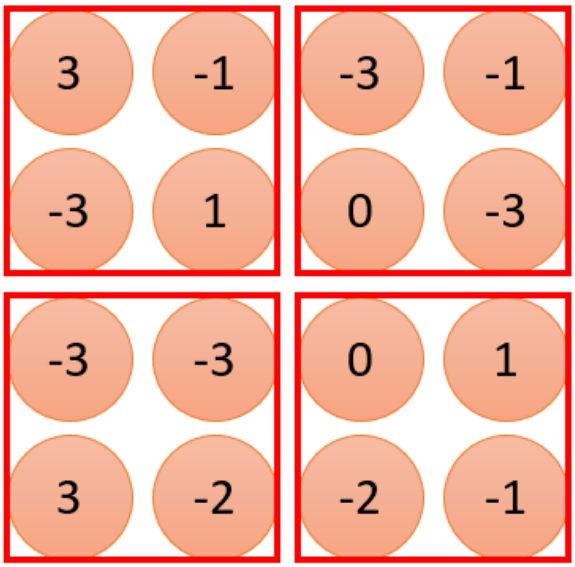


Fig. 3. Max pooling with a stride of 2

Depending on the application of the CNN, the convolution layer and the pooling layer may be repeated as many times a needed. The repetition of the layers could increase accuracy, but aren't necessary. The last layer is the fully-connected layer. The fully-connected layer will take each node in and connect it to a node in the previous layer. To perform classifications, the CNN typically needs an activation function. There are multiple such as the softmax or rectified linear unit (ReLU), shown in Fig 4. Those activation functions are implemented in the fully-connected layer [1] [2].

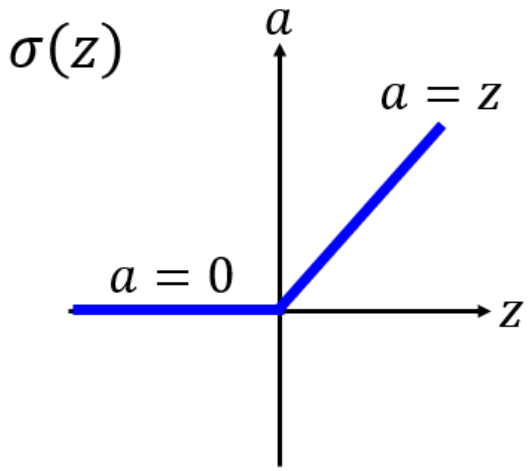


Fig. 4. ReLu activation function

B. *Collaborative Representation Classification*

Sparse coding is another widely used method in facial recognition and is considered an unsupervised learning approach. Two classification techniques that implement sparse coding are spare representation classification (SRC) and the collaborative representation classification (CRC) both using similar methods. In SRC, there are K classes, each class being a different face or image. The training images of k classes will be used as a linear combination to equate a test image. Table 1 shows the algorithm for the SRC and Table 2 provides the steps for the CRC method [3]. They are both similar, but the CRC is advantageous in that finding the projection matrix is faster with a more director solution [4]. The results between the SRC and CRC method will give similar or very close accuracy results, but CRC will be faster to compute and classify the images, making it a better choice between the two [4].

Table 1: The standard SRC Algorithm

1. Normalize the columns of $\boldsymbol{X}$ to have unit l_2-norm.
2. Code $\boldsymbol{y}$ over $\boldsymbol{X}$ via l_1-norm minimization

$$\hat{\boldsymbol{\alpha}} = \arg\min_{\boldsymbol{\alpha}} \left\{ \left\| \boldsymbol{y} - \boldsymbol{X}\boldsymbol{\alpha} \right\|_2^2 + \lambda \left\| \boldsymbol{\alpha} \right\|_1 \right\} \quad (1)$$

where λ is a positive scalar.

3. Compute the residuals

$$r_i = \left\| \boldsymbol{y} - \boldsymbol{X}_i \hat{\boldsymbol{\alpha}}_i \right\|_2 \quad (2)$$

where $\hat{\boldsymbol{\alpha}}_i$ is the coefficient vector associated with class i.

4. Output the identity of $\boldsymbol{y}$ as

$$\text{identity}(\boldsymbol{y}) = \arg\min_i \{r_i\} \quad (3)$$

Table 2: The CRC-RLS Algorithm

1. Normalize the columns of $\boldsymbol{X}$ to have unit l_2-norm.
2. Code $\boldsymbol{y}$ over $\boldsymbol{X}$ by

$$\hat{\boldsymbol{\alpha}} = \boldsymbol{P}\boldsymbol{y} \quad (11)$$

where $\boldsymbol{P} = \left(\boldsymbol{X}^T \boldsymbol{X} + \lambda \cdot \boldsymbol{I} \right)^{-1} \boldsymbol{X}^T$.

3. Compute the regularized residuals

$$r_i = \left\| \boldsymbol{y} - \boldsymbol{X}_i \hat{\boldsymbol{\alpha}}_i \right\|_2 / \left\| \hat{\boldsymbol{\alpha}}_i \right\|_2 \quad (12)$$

4. Output the identity of $\boldsymbol{y}$ as

$$\text{identity}(\boldsymbol{y}) = \arg\min_i \{r_i\} \quad (13)$$

II. Related Work

When researching about the CNN and CRC methods and how they are implemented with facial recognition, the research paper of "Facial Expression Recognition with CNN Ensemble" [5] and the paper of "Collaborative Representation based Classification for Face Recognition" [3] were explored.

A. Facial Expression Recognition with CNN Ensemble

The paper regarding CNN examined the architecture of CNNs and attempted to create one for the task of facial expression recognition. The power of a CNN relies on the architecture of the design as well as how many neurons are present. Large and complex CNNs can require a lot of data to train and they are susceptible to over-fitting the data set, making them more difficult to train. The paper proposes breaking up this CNN into three smaller CNNs and combining each of their outputs and averaging them for better results. It was found that this method was able to produce up to 81.20% accuracy with some facial expressions with only three CNN subnets each consisting of 8 to 10 compact layers that were easier for training. This paper also discussed each individual layer of a CNN and how they're used, which was very helpful when implementing the CNN proposed in this paper.

B. Collaborative Representation based Classification for Facial Recognition

In the paper pertaining to CRC, the main goal was to look at the application of CRC with facial recognition. It was clear that sparse representation was successful in using it for image processing or classifications. They not only looked at CRC, but also SRC as a starting point since the two are extremely similar. It was found that the normalization typically used in sparse based classifications, $l1$ is very time consuming. There were experimentation of using the $l1$-normalization or the $l2$-normalization. Both normalization methods had advantages and disadvantages to each. The $l1$-normalization was more robust while the $l2$-normalization was slightly faster but was not as accurate. Another main issue to tackle during the experimentation was the use between $l1$ and $l2$ normalization.

III. Proposed Method

Two methods that were mentioned to have worked well with facial recognition were the CRC and the CNN methods. For training and testing, five different celebrities pictures were used. There were approximately 700 pictures for each celebrity for training and approximately 100 pictures for testing.

A. Creating Data Set

First, training and testing images were separated into different directories and then were imported into the file as gray-scale images using OpenCV, shown in Fig 6. The images were resized to a size of 100x100 pixels and appended to a list along with their labels named test_data or train_data respectively as shown in Figure 5.

```
test_data = []
def create_test_data():
    for cel in CELEBS:
        path = os.path.join(TESTDATADIR, cel)
        class_num = CELEBS.index(cel)
        for img in os.listdir(path):
            #some images will fail so ignore them
            try:
                #Create array from image data then resize and append to list
                img_array = cv2.imread(os.path.join(path,img), cv2.IMREAD_GRAYSCALE)
                sized_img = cv2.resize(img_array, (IMG_SIZE, IMG_SIZE))
                test_data.append([sized_img, class_num])
            except Exception as e:
                pass

create_test_data()
```

Fig. 5. Python code to create the data set

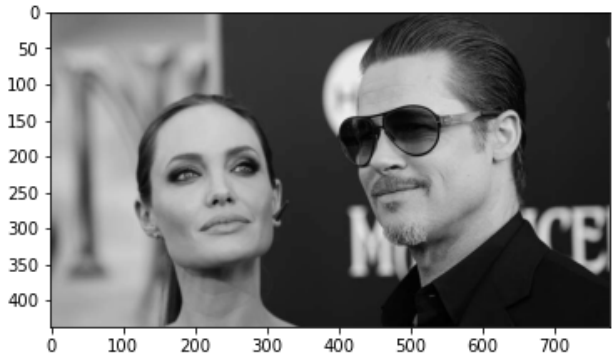


Fig. 6. Example image from running the OpemCV code to create the data set

The labels were then extracted into their own lists and the image data was converted into a NumPy array and then randomly shuffled for a better training set. The pickle Python library was then used to format the image data and labels into a files for use in both the CNN and CRC implementations. The code to write the data into the pickle file format is shown in Figure 7.

```
#writes data sets out to seperate files
data_out = open("XTrain.pickle", "wb")
pickle.dump(XTrain, data_out)
data_out.close()

data_out = open("YTrain.pickle", "wb")
pickle.dump(YTrain, data_out)
data_out.close()

data_out = open("XTest.pickle", "wb")
pickle.dump(XTest, data_out)
data_out.close()

data_out = open("YTest.pickle", "wb")
pickle.dump(YTest, data_out)
data_out.close()
```

Fig. 7. Python code to create the data files

B. CNN Approach

Before input to the CNN, the image data was imported into the file and normalized by dividing by the max value of a pixel, 255. The CNN was constructed using the Tensorflow and Keras Python libraries using the sequential model for

the implementation. Keras offers many different layers to add along with customization options for each layer. The different types of layers chosen are described below.

1) Convolution Layer: The kernel size chosen for each convolutional layer is 3x3, which is the smallest kernel size to capture surrounding information. The stride of these layers was set to 1 and no padding is used.

2) Activation Layer (Non-Linearity layer): The activation function used in this model is the "ReLU" activation function which can be described as $f(x) = max(0, x)$.

3) Pooling Layer: The network uses a 2x2 max pooling function to extract the most important data from the images. The pooling stride is set to two, allowing for no overlapping during the operation.

4) Flatten Layer: This layer is responsible for flattening the data into one dimension for the output layer.

5) Dense Layer: The dense layer is the regularly deeply connected neural network layer that performs the operation of $output = activation(dot(input, kernel) + bias)$. This is a fully connected layer that receives input from all the neurons in the previous layers.

The code in Figure 8 shows the construction of the network using the Keras toolkit. The model was then compiled with the options of the "adam" optimizer and Sparse Categorical Crossentropy for the loss parameter. The model was then trained and validated with the train and test data with a batch size of 64 over 20 epochs.

```
#Create CNN
model = Sequential()

model.add(Conv2D(32, (3,3), activation='relu', input_shape=XTrain.shape[1:]))
model.add(MaxPooling2D(2,2))

model.add(Conv2D(64, (3,3), activation='relu'))
model.add(Conv2D(64, (3,3), activation='relu'))
model.add(MaxPooling2D(2,2))
model.add(Conv2D(128, (3,3), activation='relu'))
model.add(Conv2D(128, (3,3), activation='relu'))
model.add(MaxPooling2D(2,2))
model.add(Conv2D(256, (3,3), activation='relu'))

model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dense(10))
```

Fig. 8. Python code creating the CNN architecture

C. CRC Approach

The implementation for the CRC approach began with implementing the algorithm shown in Table 2. The testing and training data was imported to the separate CRC Python file. The pictures themselves were normalized to a greyscale approach by dividing the color values by 255. The values were between 0-1 to make classification easier. The data was separated between labels and pictures with four variables: train-data, train-label, test-data, and test-label. Because of the large data set, the CRC was unable to run due to low memory.

1) CRC Function: The function that controls the CRC implementation takes in five inputs: training images, training labels, testing images, testing labels. Fig 9 shows the main set up for the function where n is the amount of training labels, testNum is the amount of testing/validation images. The array ptrain[] is the projection matrix that is holding the the alpha values from step 2 of Table 2.

```
def CRC(imTrain, imVal, labTrain, labVal, lam):
    n = len(labTrain)
    testNum = len(imVal)
    ptrain = []
```

Fig. 9. Parameters and declarations for the CRC function

The projection matrix is computed with the transpose of the training image is divided by the multiplication of transposition and the original training images. The learning rate, lambda, is multiplied by an identity matrix that is the same size as the amount of training labels, shown in Fig 10.

```
numerator = inv(((X) + (lam * np.identity(n))))
ptrain = numerator @ np.transpose(imTrain)
```

Fig. 10. Parameters and declarations for the CRC function

Since there are five classes in the project, with there being five different celebrities. Each class will go through training and validation. Each class has approximately 130-140 images for each celebrity. When running through the CRC, where the training images match the labels need to be identified. Finding which image match the label and the alpha values subtracted will result in finding the residuals, like in step 3 of Table 2. A nested for loop was necessary to dissect the classes from the training data and locate the matches, shown in Fig 6. The first for loop being to traverse between the different testing images, shown in Fig 11, and the second to look at each class one at a time. To find the matching images and labels, the alpha array needs to be checked each time. The alpha array, or in the code notated as rho-hat, being the result of step 2 from Table 2: the multiplication of the projection matrix and the validation/testing images.

```
for i in range(testNum):
    y = imTrain[:,i]
    rho_hat = np.dot(ptrain, y)
    r = np.zeros(classNum)
```

Fig. 11. First for loop to look at each class individually

In Fig 11, the residuals from the step 3 of Table 2 will be held in array r initialized temporarily with zeros. The residuals will hold the minimum of the computation. The second for loop takes care of finding where the training label is the same as the class number, being 1 through 5, with the variable being rho-hat-k, shown in Fig 12.

```
for j in range(classNum):
    rho_hat_k = rho_hat[labTrain == j]
    xk = imTrain[:, labTrain == j]
    r[j] = min(y - (xk @ rho_hat_k ))
    #print(r)
val, min_idx = min((val, idx) for (idx, val) in enumerate(r))
ivect[i] = min_idx
```

Fig. 12. Second for loop that checks which training labels match the training images

To find the identity vector, in step 4 of Table 2, the minimum of the residuals is found and then added to the identity vector. The vector will be used to check the accuracy of the training. The accuracy is found by finding the sum of when the elements of the identity matrix match the validation/testing labels and then divided by the number of testing images, shown in Fig 13. The accuracy can be varied with changing the learning rate, lambda.

```
acc = sum(ivect == labVal)
acc = acc / testNum
```

Fig. 13. Finding the accuracy of the training using the identity matrix and the validation/testing labels

2) Fixing Memory Issues: Initially our CRC did not run because of memory issues. The problem stemmed from combining the matrix into a 4-D matrix for multiplication. After separating and flattening the matrix we were left with a 10,000 x 3,481 matrix containing our train data and a 3,481 x 1 matrix containing our labels. The program was based off a previous homework that was for implementing CRC. Since the homework utilized the same algorithm, the team was able to convert homework 3 into a python CRC program. Unexpectedly, after conversion the accuracy dropped, most likely due to differences between functions in MatLab and python. Another error was that our data was stored in a matrix transposed to what was expected of X.

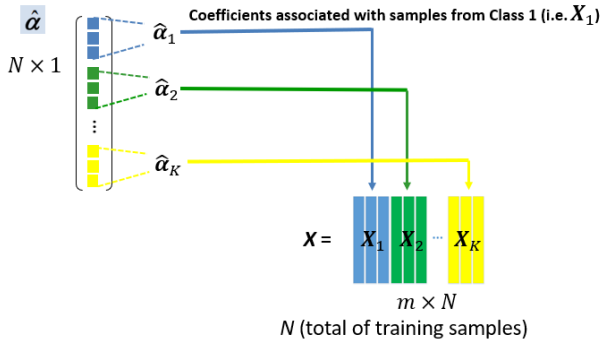


Fig. 14. Flattening of data for use in CRC

As mentioned before the data is flattened and then added to the matrix as seen in Fig 14 [4].

IV. Experiment Results

A. CNN Experiment Results

Early implementations of the CNN had very few layers and achieved a high accuracy of 99% but could only reach a validation accuracy of 59%. To try and increase the accuracy of the CNN, more convolution layers were added with varying amounts of filters. Another 2x2 max pooling layer was also added to try and reduce the amount of data for the high filter count convolutional layers at the end of the network. The results of this new network are shown in Figure 15.

```
Epoch 15/20
55/55 [==============================] - 45s 813ms/step - loss: 0.5858 -
accuracy: 0.7948 - val_loss: 1.3198 - val_accuracy: 0.5718
Epoch 16/20
55/55 [==============================] - 37s 669ms/step - loss: 0.3470 -
accuracy: 0.8824 - val_loss: 1.3531 - val_accuracy: 0.5776
Epoch 17/20
55/55 [==============================] - 35s 637ms/step - loss: 0.2318 -
accuracy: 0.9310 - val_loss: 1.9477 - val_accuracy: 0.5747
Epoch 18/20
55/55 [==============================] - 45s 821ms/step - loss: 0.1909 -
accuracy: 0.9392 - val_loss: 1.8780 - val_accuracy: 0.6293
Epoch 19/20
55/55 [==============================] - 37s 677ms/step - loss: 0.1285 -
accuracy: 0.9573 - val_loss: 2.2873 - val_accuracy: 0.5517
Epoch 20/20
55/55 [==============================] - 37s 677ms/step - loss: 0.1195 -
accuracy: 0.9609 - val_loss: 2.2163 - val_accuracy: 0.5891
```

Fig. 15. Results of CNN Method

The highest validation accuracy the network was able to achieve was demonstrated epoch 18 which was 63%. This is only a performance increase of 4% compared to the first model created, but upon addition of more even more layers, the model would perform with a lower accuracy than the first model with a validation accuracy at around 57%. The model was also compiled with different optimizers other than Adam, although Adam consistently showed the best results.

B. CRC Experiment Results

The accuracy of our CRC program was drastically lower than expected. As stated earlier, this is most likely due to the conversion between MatLab and python. Many functions did not behave the same and needed to be modified for it to run.

The implementation of the CRC method was initially based off of the previous homework assignment of CRC in MatLab. There, the accuracy was around 85-90% but re-implemented, has an accuracy of approximately 22%, shown in Fig 16.

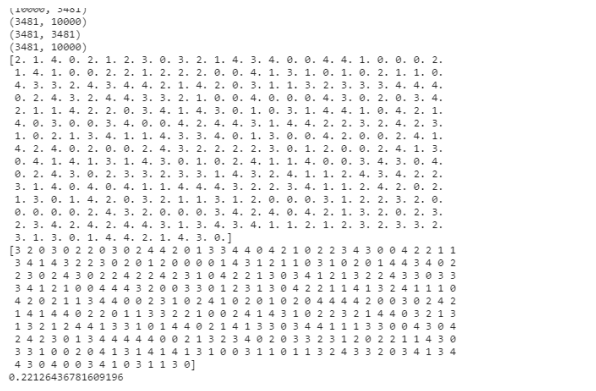


Fig. 16. Results of the CRC method

Figure 16 shows the label prediction in the first array vs true label data in the second array. The accuracy being shown at the bottom.

V. Conclusion

The goal of the project with facial recognition was to compare between two methods: convolutional neural networks and the collaborative representation classifier. From the results of both the CNN and CRC experiments, the CNN method worked better. It had a higher accuracy than the CRC method and a lot less errors.

For the CNN, the main issue previously was the lack of layers to increase training. With more added layers, the validation accuracy increased, but not dramatically. The CRC had a very low accuracy. When comparing how each method was programmed, the CNN was easier to program and expand upon because of imported Python libraries with built-in functions directly related to the CNN layers, like Keras. Since the CRC algorithm is mainly built on matrix mathematics, it is more likely that human errors can occur that could effect the accuracy.

References

[1] D. C. Chen, *ML-Lecture-16-DL-2*. UNC Charlotte, 2020.

[2] I. C. Education. Convolutional neural networks. [Online]. Available: https://www.ibm.com/cloud/learn/convolutional-neural-networks

[3] L. Zhang, M. Yang, X. Feng, Y. Ma, and D. Zhang, "Collaborative representation based classification for face recognition," vol. 1204, 04 2012.

[4] D. C. Chen, *ML-Lecture-11-Sparse Coding*. UNC Charlotte, 2020.

[5] K. Liu, M. Zhang, and Z. Pan, "Facial expression recognition with cnn ensemble," in *2016 International Conference on Cyberworlds (CW)*, 2016, pp. 163–166.