

Case Study in Recognition of Emotion from Images

Carolyn Mason

4/18/2019

CSCI E-89 Deep Learning

Problem:

It can take a lot of time to sort through and choose images to share with friends and family. It can also be hard to come back to an album years later and find a specific sets of pictures. Deep learning tools can help speed up this process and pinpoint images of a specific style that you are searching for. This tool is an emotion detector, allowing the user to pick the top 'X' images from their photo album. The user can choose from angry, happy, sad, and neutral.

Software:

This project demonstrates python, Keras, and CNN (Convelutional Neural Networks).

Description of Data:

Data for this project was sourced from Kaggle.com. Kaggle is an online community, owned by Google, that offers machine learning competitions and is a platform of public data sets. This data used for this project was prepared by Pierre-Luc Carrier and Aaron Courville as part of a research project. The data can be downloaded here:

<https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge/data>

The FER+ data set consists of 28,709 pictures of framed faces. The images are grayscale and all cropped to be 48x48 pixels. There are seven different categories for the images. These include:

Labels: 0=Angry, 1=Disgust, 2=Fear, 3=Happy, 4=Sad, 5=Surprise, 6=Neutral

I downloaded 'fer2013.csv', which contains all data needed for this project. The csv file has two columns. The first is 'emotion', which will have a string label 0-6. The second is column is 'pixels' which contains a string of pixels for each image. Reading and processing the data will be shown in the following section. Here is a sample screen shot of the data:

	A	
1	emotion	pixels
2		0 70 80 82 72 58 58 60 63 54 58 60 48 89 115 121 119 115 110 98 91 84 84 90 99 11
3		0 151 150 147 155 148 133 111 140 170 174 182 154 153 164 173 178 185 185 189
4		2 231 212 156 164 174 138 161 173 182 200 106 38 39 74 138 161 164 179 190 201
5		4 24 32 36 30 32 23 19 20 30 41 21 22 32 34 21 19 43 52 13 26 40 59 65 12 20 63 9
6		6 4 0 0 0 0 0 0 0 0 0 0 3 15 23 28 48 50 58 84 115 127 137 142 151 156 155 149 1
7		2 55 55 55 55 55 54 60 68 54 85 151 163 170 179 181 185 188 188 191 196 189 194
8		4 20 17 19 21 25 38 42 42 46 54 56 62 63 66 82 108 118 130 139 134 132 126 113 9
9		3 77 78 79 79 78 75 60 55 47 48 58 73 77 79 57 50 37 44 56 70 80 82 87 91 86 80 7
10		3 85 84 90 121 101 102 133 153 153 169 177 189 195 199 205 207 209 216 221 225
11		2 255 254 255 254 254 179 122 107 95 124 149 150 169 178 179 179 181 181 184 1
12		0 30 24 21 23 25 25 49 67 84 103 120 125 130 139 140 139 148 171 178 175 176 17
13		6 39 75 78 58 58 45 49 48 103 156 81 45 41 38 49 56 60 49 32 31 28 52 83 81 78 7
14		6 219 213 206 202 209 217 216 215 219 218 223 230 227 227 233 235 234 236 237
15		6 148 144 130 129 119 122 129 131 139 153 140 128 139 144 146 143 132 133 134

Fig 1: Screenshot of FER++ data set

Code:

Here is a high level overview of the steps for this code. Each of these will be reviewed in greater detail:

- Download the data from Kaggle
- Setup test and train data sets
- Build the CNN
- Iterate until happy with the model results
- Run on your own images!

All code was written in python and is attached with this document. This section will review pieces of the code for better user understanding.

A) Download the data from Kaggle

The first step is to download and prepare the data. I downloaded 'fer2013.csv' from kaggle and saved it to my working directory.

<https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge/data>

Please note- that the original data set has seven different emotions. I created a 'light' version in order to focus on model accuracy for four emotions. I found 'seven' to take a long time to run and had issues with smaller sized sets for some of the emotions. Here is code that I used to created the 'light' csv.

```
import pandas as pd
from sklearn.utils import shuffle
```

```

# Grab csv
base_dir = '/home/carolyn/Documents/Classes/DeepLearning/week14_finalProjects/
FERPlus_data/data'
data = '/fer2013/fer2013.csv'
fer_path = base_dir+data

df = pd.read_csv(fer_path,
                 header=0,
                 names=['Emotion', 'data', 'etc'])

# Simplify the emotions
l1 = df.loc[df['Emotion'] == 0]
l2 = df.loc[df['Emotion'] == 3]
l3 = df.loc[df['Emotion'] == 4]
l4 = df.loc[df['Emotion'] == 6]

light = pd.concat([l1,l2,l3,l4])

# Fix values
light.Emotion.loc[(light['Emotion'] == 3)] = 1
light.Emotion.loc[(light['Emotion'] == 4)] = 2
light.Emotion.loc[(light['Emotion'] == 6)] = 3

# re-shuffle
light = shuffle(light)

# Save csv
light.to_csv(base_dir+'light.csv')

```

I then checked the csv and renamed it 'datalight.csv'. Once the data can be accessed by your computer, copy the path and set it up in your code.

```

base_dir = '/home/carolyn/Documents/Classes/DeepLearning/week14_finalProjects/
FERPlus_data/data'
data = '/fer2013/datalight.csv'
fer_path = base_dir+data

```

For later use, it will be important to denote the labels associated with each numerical code. Create a dictionary of values.

```

# List of labels
emotion_table = {'0' : 'anger',      # 4953
                 '1' : 'happy',      # 8989
                 '2' : 'sad',        # 6077
                 '3' : 'neutral'}    # 6198

```

B) Setup test and train data sets

In this section, we will setup all information that is needed for the model. The main items are the test and train data sets. The train set is set to be 80% of the data and the test set to 20%. These values can be altered as needed.

The first items to set are the total number of classes (what output is possible from the model?), the batch size, and the number of epochs.

```
# Setup important information for the model
num_classes = len(emotion_table)
batch_size = 128
epochs = 15
```

Next we set up the arrays for the test and train sets for their data and labels. The data comes from the 'pixel' csv column and the labels come from the 'emotion' csv column. Each line from the csv is read and added to one of the four arrays. As noted above, I used 60% of the data for the train data set and 40% for test. I also set up the data to be normalized floats. Normalization helps the model be less sensitive to large values.

```
# Read csv
count = 0
train_data = []
train_labels = []
test_data = []
test_labels = []

with open(fer_path, 'r') as csvfile:
    dataNum = len(csvfile.readlines())-1
    trainNum = ceil(dataNum*0.8)

with open(fer_path, 'r') as csvfile:
    fer_rows = csv.reader(csvfile, delimiter=',')
    for row in islice(fer_rows, 1, None):
        if(count < trainNum):
            train_data.append([float(s)/255. for s in row[1].split(' ')])
            train_labels.append(row[0])
        else:
            test_data.append([float(s)/255. for s in row[1].split(' ')])
            test_labels.append(row[0])
        count+=1
```

The data needs to be a numpy array for the following calculations.

```
# Settle data
train_data = np.array(train_data)
train_labels = np.array(train_labels)
test_data = np.array(test_data)
test_labels = np.array(test_labels)
```

For this sort of model, the labels need to use one-hot encoding. That means that instead of using the numerical value of '2', we will create a matrix of nXnum_classes. For the value '2', a one will appear in the 2nd column. This is done for all labels and uses the to_categorical function.

```
# Labels
train_labels = to_categorical(train_labels, num_classes)
test_labels = to_categorical(test_labels, num_classes)
```

Finally, the data needs to be re-shaped for the model.

```
# Reshaping
img_rows, img_cols = 48, 48
train_data = train_data.reshape(train_data.shape[0], img_rows, img_cols, 1)
test_data = test_data.reshape(test_data.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1)
```

C) Build the CNN

The next step is to build the model. The pieces used in this framework can/may include:

Layer options:

- **Sequential:** Common architecture to keep in order of input to output
- **Conv2D:** First argument is the number of output channels. Next is the kernel size which is an NxN moving window.
- **Drop Out:** A regularization technique used to reduce overfitting. This may occur if there isn't enough data, the network is too big, or the model is trained for too long.
- **Max Pooling:** A NxN search action around the picture. The key role is to reduce the size of the image while highlighting key shapes. The flashlight search action also helps to reduce overfitting by looking at a small piece of information at a time.
- **Flatten:** Used after max pooling to make a single column for processing
- **Dense:** Finishes with two dense layers. First is the number of nodes. Second is the number of possible label output answers.

This model was built like:

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=input_shape))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Dropout(0.25))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dropout(0.25))
model.add(layers.Dense(512, kernel_regularizer=regularizers.l2(1e-4),
activation='relu'))
model.add(layers.Dropout(0.4))
model.add(layers.Dense(num_classes, activation='sigmoid'))
```

Here is the model summary:

```
# Print summary
model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 46, 46, 32)	320
max_pooling2d_1 (MaxPooling2D)	(None, 23, 23, 32)	0
conv2d_2 (Conv2D)	(None, 21, 21, 64)	18496
max_pooling2d_2 (MaxPooling2D)	(None, 10, 10, 64)	0
conv2d_3 (Conv2D)	(None, 8, 8, 128)	73856
max_pooling2d_3 (MaxPooling2D)	(None, 4, 4, 128)	0
dropout_1 (Dropout)	(None, 4, 4, 128)	0
conv2d_4 (Conv2D)	(None, 2, 2, 128)	147584
max_pooling2d_4 (MaxPooling2D)	(None, 1, 1, 128)	0
flatten_1 (Flatten)	(None, 128)	0
dropout_2 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 512)	66048
dropout_3 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 4)	2052
Total params: 308,356		
Trainable params: 308,356		
Non-trainable params: 0		

Lastly, the model is compiled. I used ‘categorical_crossentropy’ for the loss type. This allows for the model to pick the emotion best related to each image. I also used an adam optimizer. There are many different types of optimizers to use. This is common, so I stuck with it.

```
# Compilation
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['acc'])
```

In order to create the model, we use a ‘fit’ function. From variables set at the top of the script, batch_size = 128 and epochs=15. I ran one model out to 25 epochs, but found that the model was nearly converge by 15 epochs and there was no reason to run the model further. After the model is created it will be saved. Here is the code.

```
# Data pre-processing
history = model.fit(train_data, train_labels,
                    batch_size=batch_size,
                    epochs=epochs,
                    verbose=1,
                    validation_data=(test_data, test_labels))
```

```
model.save('facial_expressions_l2_'+str(lmbda)+'h5')
```

D) Iterate until happy with the model results

Below are a set of items to consider when building CNN's. These can affect the model's performance and behavior. It is good to test a few items out to make sure the model is not over fitting or needlessly run for too many epochs.

Here are some potential knobs one can use to edit the model. Every item will tie into 'accuracy', but here are a few loose categories that can be used when iterating the model.

Avoid overfitting data:

- 1) L2 Regularization: ie. lambda: 0.001, 0.005 and 0.0005
- 2) Dropout

Used for small data sets:

- 1) Data augmentation
- 2) Label weights (even distribution of label sets)
- 3) K-fold (used for testing model skill)

Time to run model:

- 1) Number of epochs
- 2) Image size/ shape
- 3) Learning rates
- 4) Type of optimizer when compiling: (ie. adam)

Accuracy:

- 1) Using a pre-trained model
- 2) Dense layer, activation function type: (none, sigmoid, relu, tanh)
- 3) Loss function: (mean squared error (L2 loss), categorical_crossentropy, binary_crossentropy)

In order to plot results and see the effects of each of these changes, here is the code that I used. As you can see, I set the model up to test different L2 lambda values. With this setup, each iteration is added to the plot for comparison.

```
# Create and run model -----

history1 = createAndRun(0.001)
history2 = createAndRun(0.0005)
history3 = createAndRun(0.0001)

all_history = {'0.001':history1,'0.005':history2,'0.0001':history3}

# Create Plots -----
```

```

color = {'0.001':'r','0.005':'b','0.0001':'g'}

for val in color.keys():
    acc = all_history[val].history['acc']
    val_acc = all_history[val].history['val_acc']
    plt.plot(range(1,epochs+1), acc, 'o'+color[val], label='Training acc: '+val)
    plt.plot(range(1,epochs+1), val_acc, color[val], label='Validation acc: '+val)

plt.title('Training and validation accuracy')
plt.legend()

plt.savefig('Training_and_validation_accuracy.png')
plt.figure()

for val in color.keys():
    loss = all_history[val].history['loss']
    val_loss = all_history[val].history['val_loss']
    plt.plot(range(1,epochs+1), loss, 'o'+color[val], label='Training loss: '+val)
    plt.plot(range(1,epochs+1), val_loss, color[val], label='Validation loss: '+val)

plt.title('Training and validation loss')
plt.legend()

plt.savefig('Training_and_validation_loss.png')
plt.show()

# Results
for val in color.keys():
    val_acc = all_history[val].history['val_acc'][-1]
    print('L2 value: ' +str(val) + ', Accuracy: ' + str(val_acc))

```

E) Run on your own images!

In order to run the model on images from the test or on your own images, you must load in the model we just setup. It is also important to note the labels, so we can decode the prediction.

```

model =
load_model('/home/carolyn/Documents/Classes/DeepLearning/week14_finalProjects/
FERPlus_data/code/facial_expressions.h5')

# List of labels
emotion_table = {'0' : 'anger',      # 4953
                 '1' : 'happy',      # 8989
                 '2' : 'sad',        # 6077
                 '3' : 'neutral'}    # 6198

```

I set up the code to load all images from a given directory and predict the result.

```

basePath = "/home/carolyn/Documents/Classes/DeepLearning/week14_finalProjects/
FERPlus_data/code/myPics"

```



```
imageList = [f for f in os.listdir(basePath) if os.path.isfile(basePath+f)]
```

Here is the code that loads, resizes, and predicts the image outcome. I set up the prediction to only report back an answer if the total sum of the responses is great than 20%. If there is a 'match', the results will be returned! If the model does not get a good fit, it will return 'No face'.

```
for image in imageList:
    # New image
    print('-----')
    print('Name: ' + image)

    # Load
    pic = cv2.imread(basePath+image,cv2.IMREAD_GRAYSCALE)

    # Resize and Trim = 32x32 pixels
    sz = 48
    r = sz / pic.shape[0]
    dim = (int(pic.shape[1] * r),sz)
    resized = cv2.resize(pic, dim, interpolation = cv2.INTER_AREA)
    img = resized[0:sz,0:sz]

    # Option too save for reference
    #cv2.imwrite(basePath+"/cropped.jpg", img)

    # Preprocess the image into a 4D tensor
    img2 = np.expand_dims(img, axis=0)
    img_tensor = np.expand_dims(img2, axis=3)






    # Predict
    ans = model.predict(img_tensor)[0]
    if (sum(ans) > 0.02 ):
        #norm = [float(i)/sum(ans)*100 for i in ans]
        for i in range(0,len(ans)):
            print(emotion_table[str(i)] + ': ' + str(round(ans[i]*100)) + '%')
    else:
        print(ans)
        print(sum(ans))
        print('No face')
```

Demonstration (Results and Visualization):






In this section I will show prediction results. I took images from my computer, google search results, and from the validation directory. The images start as all different shapes, sizes, and colors. Before the model is run, all images are set in grayscale and resized to 48x48. Each image and result are shown below. The code is setup to translate the one-hot encoded result to one of the original labels. It uses % values for the likelihood of each result.

For my sample data sets angry 4/5, happy 5/5, neutral 4/5, and sad 1/5. More result details will be discussed in the next section.






Angry

	<p>Name: angry0.JPG</p> <p>anger: 100.0%</p> <p>happy: 0.0%</p> <p>sad: 0.0%</p> <p>neutral: 0.0%</p>
	<p>Name: angry1.png</p> <p>anger: 100.0%</p> <p>happy: 0.0%</p> <p>sad: 0.0%</p> <p>neutral: 0.0%</p>
	<p>Name: angry2.png</p> <p>anger: 100.0%</p> <p>happy: 0.0%</p> <p>sad: 0.0%</p> <p>neutral: 0.0%</p>
	<p>Name: angry3.JPG</p> <p>anger: 0.0%</p> <p>happy: 100.0%</p> <p>sad: 0.0%</p> <p>neutral: 0.0%</p>
	<p>Name: angry4.JPG</p> <p>anger: 100.0%</p> <p>happy: 0.0%</p> <p>sad: 0.0%</p> <p>neutral: 0.0%</p>


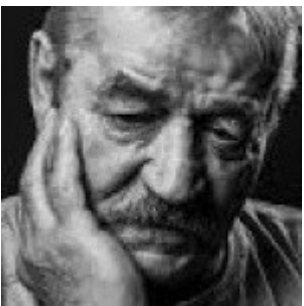

Happy

	Name: happy0.png anger: 0.0% happy: 100.0% sad: 0.0% neutral: 0.0%
	Name: happy1.JPG anger: 0.0% happy: 100.0% sad: 0.0% neutral: 0.0%
	Name: happy2.JPG anger: 0.0% happy: 100.0% sad: 0.0% neutral: 0.0%
	Name: happy3.JPG anger: 0.0% happy: 100.0% sad: 0.0% neutral: 0.0%
	Name: happy4.png anger: 0.0% happy: 100.0% sad: 0.0% neutral: 0.0%

Neutral

	<p>Name: neutral0.png</p> <p>anger: 0.0%</p> <p>happy: 0.0%</p> <p>sad: 0.0%</p> <p>neutral: 100.0%</p>
	<p>Name: neutral1.JPG</p> <p>[0. 0. 0. 0.]</p> <p>0.0</p> <p>No face</p>
	<p>Name: neutral2.jpeg</p> <p>anger: 0.0%</p> <p>happy: 0.0%</p> <p>sad: 0.0%</p> <p>neutral: 3.0%</p>
	<p>Name: neutral3.jpeg</p> <p>anger: 0.0%</p> <p>happy: 0.0%</p> <p>sad: 0.0%</p> <p>neutral: 100.0%</p>
	<p>Name: neutral4.jpeg</p> <p>anger: 0.0%</p> <p>happy: 0.0%</p> <p>sad: 0.0%</p> <p>neutral: 100.0%</p>

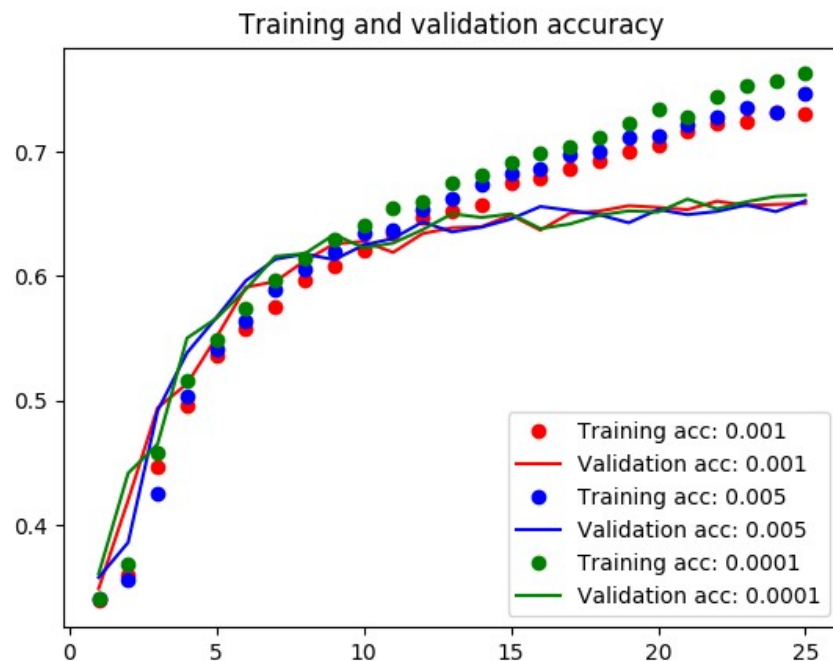
Sad

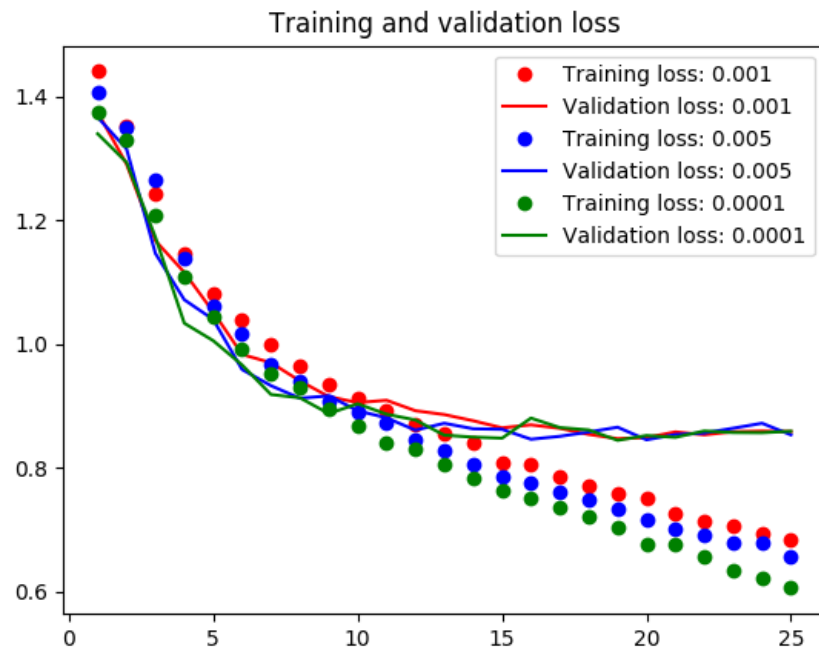
	<p>Name: sad0.jpeg</p> <p>[0. 0. 0. 0.00846662]</p> <p>0.008466624654829502</p> <p>No face</p>
	<p>Name: sad1.jpeg</p> <p>[0.0000000e+00 0.0000000e+00 1.4279881e-14 7.4760398e-23]</p> <p>1.427988119890005e-14</p> <p>No face</p>
	<p>Name: sad2.jpeg</p> <p>[4.9207607e-09 4.5433485e-30 0.0000000e+00 0.0000000e+00]</p> <p>4.920760687809889e-09</p> <p>No face</p>
	<p>Name: sad3.jpeg</p> <p>[0. 0. 0. 0.]</p> <p>0.0</p> <p>No face</p>
	<p>Name: sad5.png</p> <p>anger: 0.0%</p> <p>happy: 0.0%</p> <p>sad: 100.0%</p> <p>neutral: 0.0%</p>

Results:

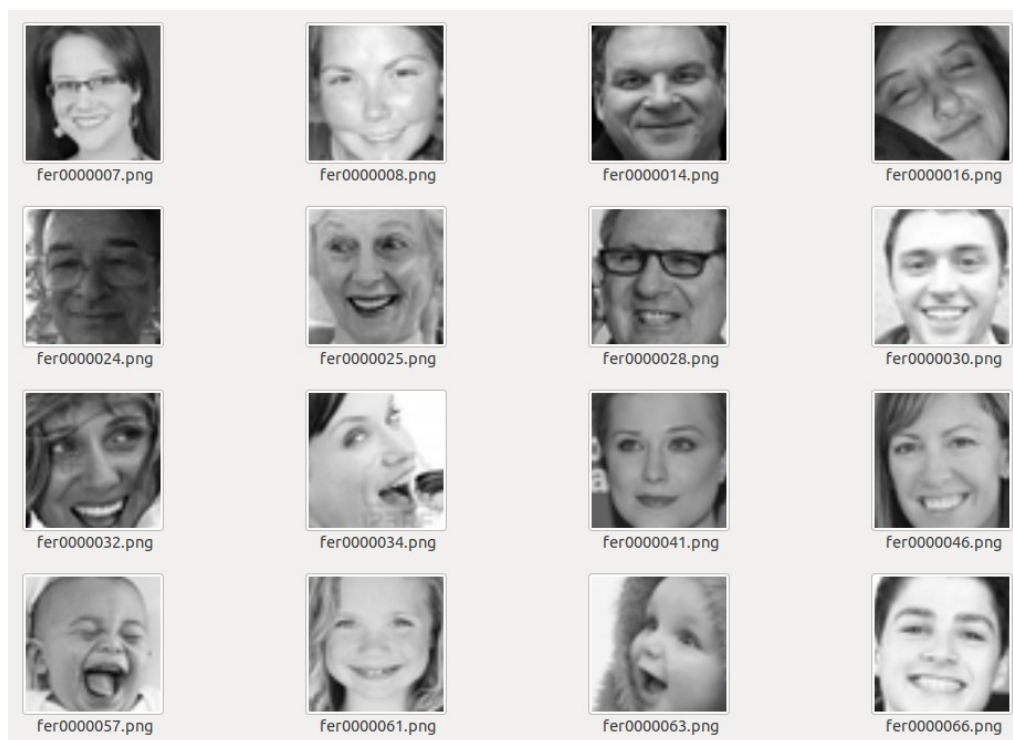
The final accuracy of the model is 66.5% for predicting four emotions of cropped pictures of the face. The intention was to take any picture and read the emotion, but this adds on a layer of complexity. The code would need an automatic face detector, so the model does not get confused on the rest of the image. This means that this is not a useful tool for sorting through old vacation photos. The tool is more hands on and requires manual cropping of individual faces. This is something that is addressed in the future work section.

The biggest point of concern, for this model, is over fitting. I was able to improve this metric by adding dropout and an L2 llambda value. While the results can still improve, I will present the final plots for the model I created. As a final test, I ran a series of L2 llambda values of 0.001, 0.005, and 0.0001. I chose an L2 llambda value of 0.0001, since all validation accuracies were the same and it had the least overfitting. Here are the plots, showing my final results:

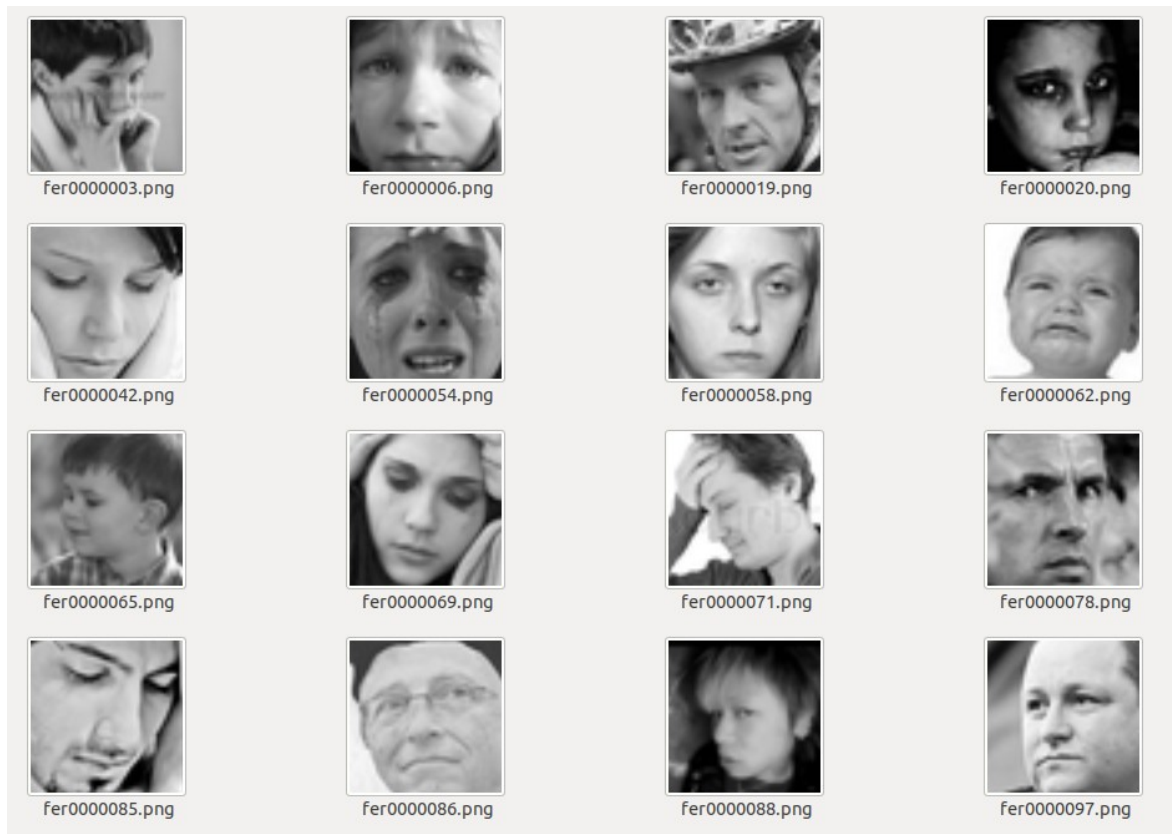




When I ran the demonstration pictures, I found that emotions such as ‘happy’ performed really well, where sad was hardly recognized. From the Demonstration section, I found that in my sample data sets angry 4/5, happy 5/5, neutral 4/5, and sad 1/5. Since the data is consistent and all of the same relative size, some emotions must be more complicated and harder to train than others. Everyone who is happy have similar characteristics– wide, upward turned mouth, and often teeth.



The sad group of photos, are more complex. Eyes can be open or closed, mouth is often flat or turned slightly down, and hands may obscure the image. Some of these traits may get confused with the 'neutral' category.



I re-ran the model with every 'anger', 'happy', 'sad', or 'neutral' image from the test and training sets to see how the model performed. I found, similar results to my mini test. The table compares correct/total = percent correct

Anger: 1329/ 4924 = 27%

Happy: 7356/ 8953 = 82%

Sad: 643/ 6037 = 10.6%

Neutral: 3026/ 6160 = 49%

This table shows a wide range of success. Sad images are only correct 10.6% of the time, while happy images are correct 82% of the time. While training set size may make a difference, neutral did much better than sad and had a similarly sized set. In order to make sure that there was no fluke in dividing the test/ train data sets, I ran a count on each of the labels. We know that '0' : 'anger', '1' : 'happy', '2' : 'sad', '3' : 'neutral'. The split does not indicate any red flags:

```
>>> unique, counts = np.unique(train_labels, return_counts=True)
>>> dict(zip(unique, counts))
{'0': 3038, '1': 5341, '2': 3672, '3': 3680}
```



```
>>> unique, counts = np.unique(test_labels, return_counts=True)
>>> dict(zip(unique, counts))
{'0': 1915, '1': 3648, '2': 2405, '3': 2518}
```

To me, the skewed results show that some results are more difficult to train than others. This will be a key item to address in future projects. If perceived individually, how can each category perform best? Are there any other strategies to help out these categories?

Lessons Learned & Pros/Cons:

The model has an accuracy of 66.5% for four emotions. As a reference, the Kaggle competition held for this data set had a top accuracy of 71% for seven emotions (<https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge/leaderboard>). There is still work that can be done to improve this model, but the initial results are not bad.

This project taught me how to train a model with categorical data and what knobs could help improve over fitting. Including dropout and L2 regularization were helpful, but not enough to align the training and validation accuracies. I think the key lesson learned it addressing the data and improving the results from categories that do not perform as well as the others.

Future Work:

The next step toward improving this model is determining how to better train the model on the sad dataset. It would be good to get each emotion's results above 50%. Once the 4-emotion model is finished, the next step will be to reach higher accuracies while using 7+ emotions.

In the future, it would also be beneficial to add an automatic face detection/ cropping option in order to find faces in normal, more complicated images. In order to automatically find every vacation photo with 'happy' people, it would be important.

YouTube URLs:

2 minute: <https://youtu.be/nrLpXucMcLM>

15 minute: <https://youtu.be/mJR3sblItPs>

References:

Zoran's class notes

Kaggle Data set: <https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge/data>

Deep Facial Expression Recognition: A Survey: <https://arxiv.org/pdf/1804.08348.pdf>

Variety of papers on facial recognition: <https://paperswithcode.com/task/facial-expression-recognition>
Deep Learning For Smile Recognition: <https://arxiv.org/pdf/1602.00172v2.pdf>