Northeastern University
Khoury College of
Computer Sciences

Facial Emotion Recognition

DS 4440 Final Project

Justin Littman, Matthew Martin

Khoury College of Computer Sciences, Boston, MA, U.S.A

Submitted 15 April 2019

Abstract

In this paper, we explored various convolutional neural network (CNN) architectures , such as VGG[1] and ResNet[2], and developed our own CNN architecture in hopes to classify human emotion based upon facial features. The dataset used was the Fer2013 dataset, which is comprised of over 30,000 grayscale images and contains seven emotions anger, disgust, fear, happy, sad, surprise, and neutral. We tested various modifications on the VGG and ResNet models such as model depth, optimizer used, and regularization techniques such as dropout. This dataset was used in a Kaggle competition in which the top competitor achieved an accuracy of 71%. Our models had varying performance. Through experimental results it was found that certain models such as ResNet were too complex for our data and that deep models lost a lot of information. The most optimal model performed at an accuracy of about 60% on the test set.

I. Introduction

Emotion is one of the key tools humans use to convey actions and assess the climate of situations. Due to an increase in human computer interaction, harnessing this assessment of emotion can improve many technologies such as virtual reality, driving assistants, and robotic systems. Due to the rising popularity of these HCI applications, a field of interest is Facial Emotion Recognition. In 2013, Kaggle hosted a competition called *Challenges in Representation Learning: Facial Expression*

Recognition Challenge, which aimed to classify images of facial expressions into one of seven emotions (Anger, Disgust, Fear, Happy, Sad, Surprise, and Neutral). The dataset contains 35,887 examples of 48x48 grayscale images, and the top performer was able to classify facial expressions with an accuracy of 71%.

The dataset, named fer2013, was originally introduced on International Conference on Machine Learning (ICML), and was made publicly available for the competition. This dataset was suitable for this project's focus, since the data was previously labeled by humans and allowed us to focus on constructing models.

Our objective was to explore different Convolutional Neural Network (CNN) architectures and develop our own to classify facial emotions. In this paper, we will explore different versions of architectures, such as VGG and ResNet, as well as explore a new architecture, which we will refer to as our model. In addition, we tested our model's performance in real time using OpenCV.

II. Experimental Setup

A. Data Preprocessing and Exploration

The dataset consists of seven emotions. The data is distributed as follows: 25% Happy, 17.3% Neutral, 16.9% Sad, 14.3% Fear, 13.8% Anger, 11.2% Surprise, and 1.5% Disgust. The training set consisted of 25,838 images, the validation set had 2,871, and the test set had 7,178. The images came in the form of an array of pixels, which needed to be reshaped to a 48x48 matrix. Figure 1 shows an example image from the dataset.

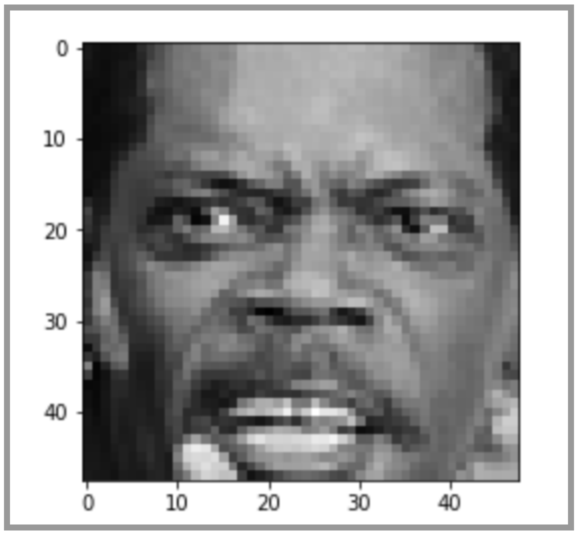
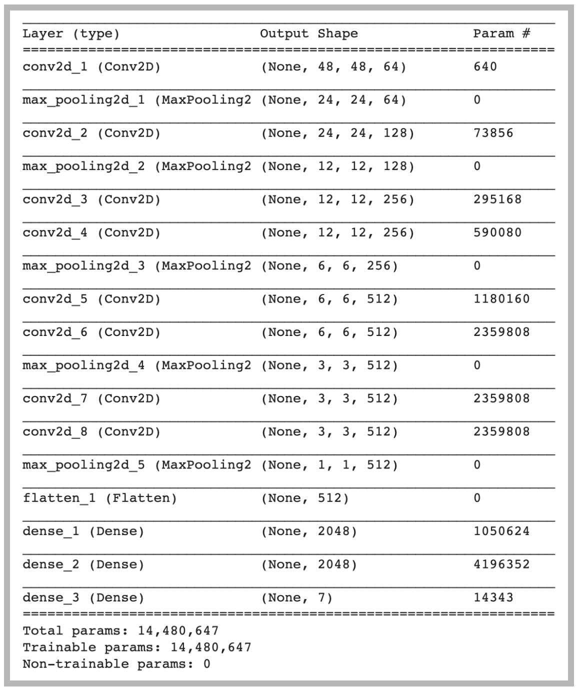


Figure 1: Image from FER2013 Dataset

B. VGG Implementation

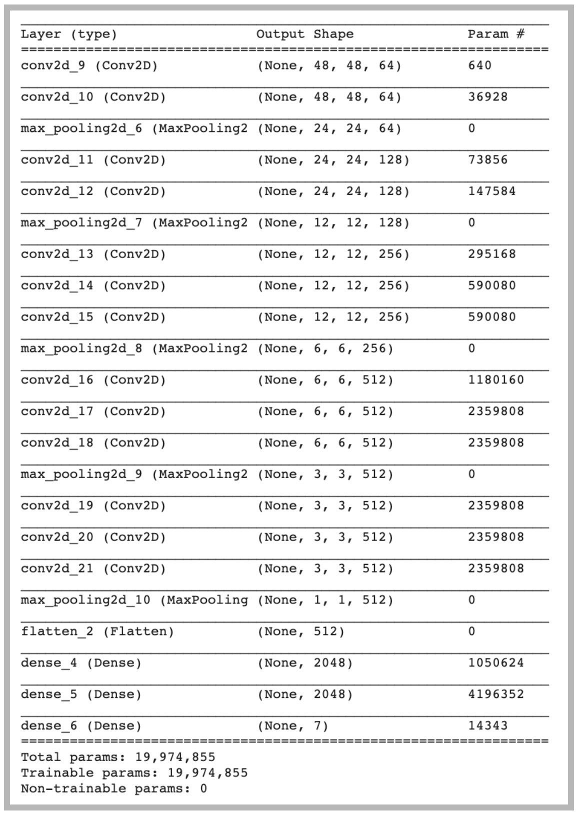
First, we implemented version of Oxford University's Visual Geometry Group (VGG) model that was created for the ImageNet competition. The versions we implemented were VGG11 and VGG16, the number represents the number of layers in the model. The VGG architecture explores how deeper models can retain information and perform better than shallower models. The architecture consists on multiple blocks. Each block consists of convolutional layers, a nonlinearity layer, and a max pooling layer. There are five blocks in every VGG architecture. These blocks are then followed by two fully connected layers and then finally run through a softmax. The model summary for VGG11 can be seen in figure 2.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 48, 48, 64)        640
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 24, 24, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 24, 24, 128)       73856
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 12, 12, 128)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 12, 12, 256)       295168
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 12, 12, 256)       590080
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 6, 6, 256)         0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 6, 6, 512)         1180160
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 6, 6, 512)         2359808
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 3, 3, 512)         0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 3, 3, 512)         2359808
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 3, 3, 512)         2359808
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 1, 1, 512)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 2048)              1050624
_________________________________________________________________
dense_2 (Dense)              (None, 2048)              4196352
_________________________________________________________________
dense_3 (Dense)              (None, 7)                 14343
=================================================================
Total params: 14,480,647
Trainable params: 14,480,647
Non-trainable params: 0
```

Figure 2: VGG11 model summary

VGG16 follows the same architecture as VGG11 except there is one more convolutional layer per block. Below the model summary for VGG16 can be seen in figure 3.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_9 (Conv2D)            (None, 48, 48, 64)        640
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 48, 48, 64)        36928
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 24, 24, 64)        0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 24, 24, 128)       73856
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 24, 24, 128)       147584
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 12, 12, 128)       0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 12, 12, 256)       295168
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 12, 12, 256)       590080
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 12, 12, 256)       590080
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 6, 6, 256)         0
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 6, 6, 512)         1180160
_________________________________________________________________
conv2d_17 (Conv2D)           (None, 6, 6, 512)         2359808
_________________________________________________________________
conv2d_18 (Conv2D)           (None, 6, 6, 512)         2359808
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 3, 3, 512)         0
_________________________________________________________________
conv2d_19 (Conv2D)           (None, 3, 3, 512)         2359808
_________________________________________________________________
conv2d_20 (Conv2D)           (None, 3, 3, 512)         2359808
_________________________________________________________________
conv2d_21 (Conv2D)           (None, 3, 3, 512)         2359808
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 1, 1, 512)         0
_________________________________________________________________
flatten_2 (Flatten)          (None, 512)               0
_________________________________________________________________
dense_4 (Dense)              (None, 2048)              1050624
_________________________________________________________________
dense_5 (Dense)              (None, 2048)              4196352
_________________________________________________________________
dense_6 (Dense)              (None, 7)                 14343
=================================================================
Total params: 19,974,855
Trainable params: 19,974,855
Non-trainable params: 0
```

Figure 3: VGG16 model summary

VGG16 has over 5 million more parameters to learn than VGG11. Both models needed the input layer to be reshaped from the original model architecture to suit our data. The new input shape for both models was 48x48x1 which is the image dimensions by the the number of channels. For these two implementation we tested the performance of each with batch sizes of 32, 64, and 128. In addition to these model we wanted to see how dropout affected model performance. We implemented dropout models using the VGG11 architecture. First, we added a 50% dropout layer after every block. Then, we implemented an architecture with dropout at every other block.

C. ResNet Implementation

Next, we implemented versions of the ResNet architecture, which placed first in the ImageNet 2015 competition for classification. ResNet introduced the concept of residual learning, which allows information from earlier blocks to be passed forward to a further layer in the network. This helped combat the vanishing gradient problem that caused some deeper networks to actually perform worse on training data than shallower networks. The ResNet architecture is popular for modern classification tasks, because it is capable of handling deeper networks.

All ResNet architectures consist of four modules, with small filter sizes of either 3x3 or 1x1. The default architecture uses 224x224 images, with three color channels (RGB). The fer2013 dataset has 48x48 images, with a single color channel, so we had to change the input shape in the network to account for this. The versions we implemented were ResNet18 and ResNet50, where the number refers to the number of layers in the network.

Our implementation of the ResNet18 architecture contains a convolutional block, which is called at the start of each module to resize the input as the dimensions increase, followed by an identity block, which acts the same as the convolutional block aside from passing the input through a convolution. Each block contains two layers. ResNet50 follows a similar pattern, except each module has a varying number identity blocks, as specified by the architecture. Each block in ResNet50 has three layers, rather than two.

Traditional ResNet architectures still have a high level of complexity, so we also explored a modified architecture which we refer to as ResNet10. This architecture has two modules, rather than the traditional four modules in other ResNet architectures, and the blocks in ResNet10 follow the same pattern as the blocks in ResNet18. These architectures were trained on Adam and RMSProp optimizers, using varying batch sizes of 32, 64, and 128.

D. *Our Model*

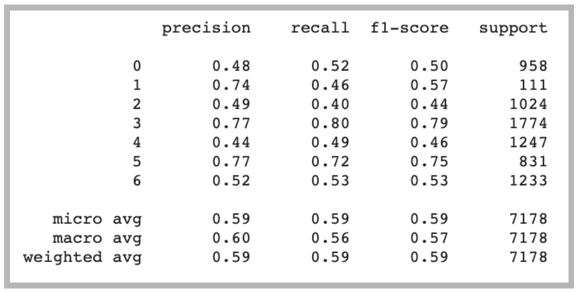
For our model we used VGG11 as a base since it had the best performance with our data. From there was wanted to explore how changing the filter sizes affected the model since VGG uses a strict 3x3 filter size. The model increases the filter size after each block. The filter sizes are 3x3, 5x5, and 7x7. This method was chosen based upon an article referring to Neural Network construction practices. In addition, we implemented Batch Normalization after each layer. As an addition, we added the option of an attention layer. This layer emphasizes the important areas of the image during classification, using a weighting system to focus on specific regions. This is the followed by two fully connected layers and a softmax, just like in the VGG architectures. This model was trained using an Adam optimizer and a batch size of 64. These were the ideal parameters for the VGG11 implementation that this model was based on.

III. Results and Discussion

In this section we will discuss the results for the best models of each architecture.

A. *VGG Results*

Based on the testing data VGG11 with batch size of 64 yielded the best results with an accuracy of 58.9%. Below the precision, recall, and F1-score can be seen for each class in Figure 4. This model performed strongest on instances of happy and surprised faces. These emotions are very distinct, and it is reasonable to see that they were the easiest for the model to classify. The model was also able to identify disgust with precision of 74%, but it was reluctant to classify images as disgust with a recall of 46%. It is reasonable to attribute this to the frequency of disgust throughout the dataset. It was the rarest of the seven emotions, and only accounted for 1.5% of the entire dataset.



```
              precision    recall  f1-score   support

           0       0.48      0.52      0.50       958
           1       0.74      0.46      0.57       111
           2       0.49      0.40      0.44      1024
           3       0.77      0.80      0.79      1774
           4       0.44      0.49      0.46      1247
           5       0.77      0.72      0.75       831
           6       0.52      0.53      0.53      1233

   micro avg       0.59      0.59      0.59      7178
   macro avg       0.60      0.56      0.57      7178
weighted avg       0.59      0.59      0.59      7178
```

Figure 4: Result statistics for optimal VGG11

For all VGG implementations, we tested the model with different optimizers. Adam was the only optimizer that resulted in significant improvement to the loss and accuracy. The results for all VGG implementations can be found in Figure 5.

Model	Batch Size	Accuracy
VGG11	32	58.4%
VGG11	**64**	**58.9%**
VGG11	128	56.8%
VGG16	32	57.8%
VGG16	64	55.6%
VGG16	128	54.7%

Figure 5: VGG model results

We also explored adding dropout, a popular regularization technique, to see if it could improve the performance of our optimal VGG11 model. We tried two implementations of dropout, the first adding a dropout layer after every block, and the second adding a dropout layer after every other block. Each dropout layer had a dropout probability of 50%. When adding dropout after every block, the network lost some valuable information, and the test accuracy fell to 38%. Simplifying the dropout to add a dropout layer only after every other block saw the accuracy return to 58.7%. This was similar to the performance of VGG11 with no dropout, and we concluded that dropout did not have a significant effect for this model on this dataset.

B. *ResNet Results*

Out of the three architectures (ResNet50, ResNet18, ResNet10), the shallowest architecture yielded the best results. ResNet10 with an Adam optimizer and a batch size of 32 had a test accuracy of 43.2%. The precision, recall, and F1-score from this model can be seen in Figure 6. Despite being the best performing ResNet model, it performed worse than the best VGG model in classifying each class. It did still manage to perform best on the happy and surprised classes, The precision for the other five classes was similar, but poor across all classes.

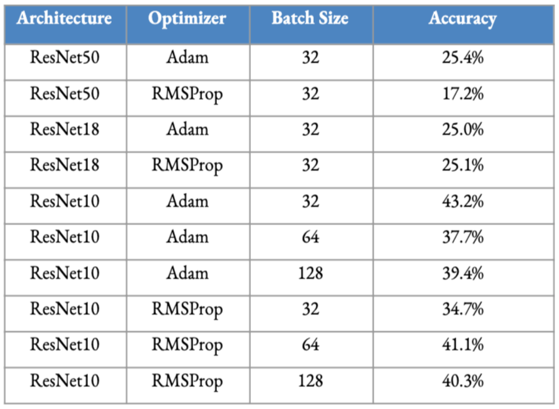
```
              precision    recall  f1-score   support

           0       0.30      0.30      0.30       958
           1       0.33      0.28      0.30       111
           2       0.33      0.25      0.29      1024
           3       0.57      0.65      0.60      1774
           4       0.32      0.28      0.30      1247
           5       0.64      0.58      0.61       831
           6       0.37      0.44      0.41      1233

   micro avg       0.43      0.43      0.43      7178
   macro avg       0.41      0.40      0.40      7178
weighted avg       0.43      0.43      0.43      7178
```

Figure 6: Results statistics for optimal ResNet10

For all ResNet10 implementations, we tried implementations across batch sizes of 32, 64, and 128. We also tried both Adam and RMSProp optimizers to see if one can yield better results. For ResNet50 and ResNet18 implementations, we used both optimizers, but only trained it using a batch size of 32. The full set of results can be seen in Figure 7.



Architecture	Optimizer	Batch Size	Accuracy
ResNet50	Adam	32	25.4%
ResNet50	RMSProp	32	17.2%
ResNet18	Adam	32	25.0%
ResNet18	RMSProp	32	25.1%
ResNet10	Adam	32	43.2%
ResNet10	Adam	64	37.7%
ResNet10	Adam	128	39.4%
ResNet10	RMSProp	32	34.7%
ResNet10	RMSProp	64	41.1%
ResNet10	RMSProp	128	40.3%

Figure 7: ResNet model results

ResNet50 and ResNet18 both struggled to perform well on this dataset. We suspect that the complexity of these architectures was too high for the dataset we were using. When using a smaller architecture with fewer parameters like ResNet10, we saw more comparable results to the earlier models.

C. *Our Model Results*

Out of all of the models that we trained, our model had the best performance on the test data. Running this model with an Adam optimizer and a batch size of 64 resulted in a test accuracy of 61.4%. The main difference between this model and the other models we trained was the attention component that we included. We were able to see where the model was emphasizing the image during classification by visualizing the image in its state after going through the attention layer. From this we were able to see that anger is emphasized in the nostril and mouth region of the face. This can be seen in Figure 8 below.

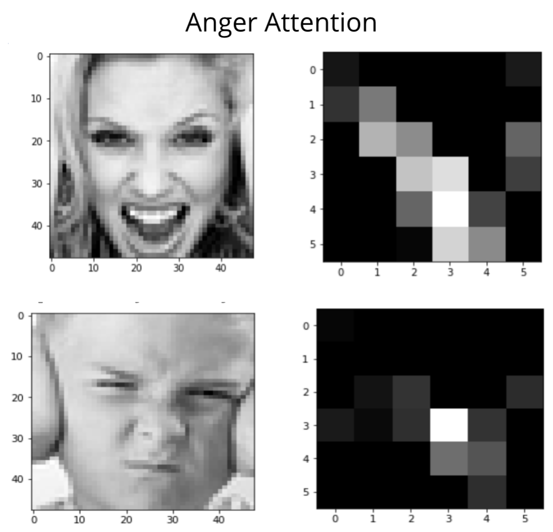
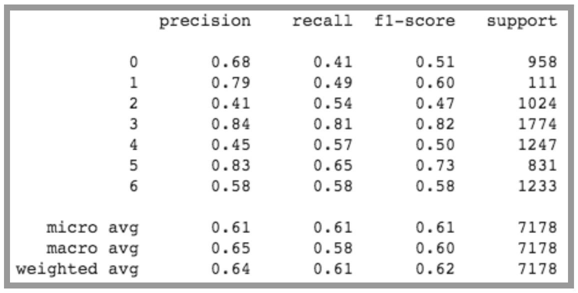


Figure 8: Attention areas of anger images

The F1 scores for this model were slightly higher for each class in this model than the VGG models we explored. It also performed better on some of the less represented classes that were difficult to classify earlier. The full results for precision, recall, and F1-score from this model can be found in Figure 9.



```
              precision    recall  f1-score   support

           0       0.68      0.41      0.51       958
           1       0.79      0.49      0.60       111
           2       0.41      0.54      0.47      1024
           3       0.84      0.81      0.82      1774
           4       0.45      0.57      0.50      1247
           5       0.83      0.65      0.73       831
           6       0.58      0.58      0.58      1233

   micro avg       0.61      0.61      0.61      7178
   macro avg       0.65      0.58      0.60      7178
weighted avg       0.64      0.61      0.62      7178
```

Figure 9: Result statistics for Our Model

An interesting observation is that this model was more inclined to predict fear and less inclined to predict surprised than the VGG model. The recall for fear and surprised in our model were 54% and 65% respectively, while they were 40% and 72% respectively in the VGG model.

D. *Live Stream*

We were able to implement a live stream classification seen in Figure 10. We used the OpenCV library to run the stream in real-time. We used a pre-trained Haar Cascade Classifier[3] in the OpenCV library to extract all of the faces in each frame. Each face was converted to a 48x48 grayscale image before being forwarded to our model for prediction. The model was trained in a Colab notebook, and exported to a local drive where the OpenCV code would be able to connect to a camera.

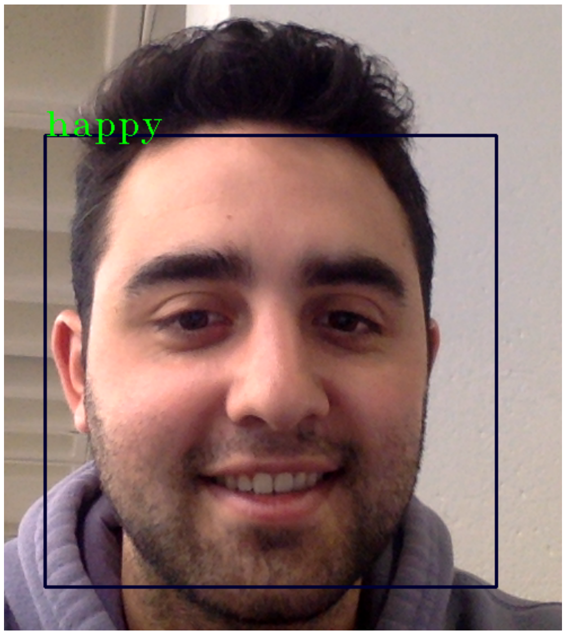


Figure 10: Live Stream Classification

IV. Conclusion/ Future Work

The fer2013 dataset remains a difficult dataset to use for classifying facial emotions. The varying facial orientations and limited resolution make it difficult to create a very accurate model. In the future, it would be interesting to explore these architectures on a dataset with more information to see if the results improve.

We were pleased with the performance of the VGG11 model, especially when we modified it to add attention. Attention was the only method we added that actually improved the overall performance of the model. It was fascinating to see which features were prioritized in the fer2013 images, and we would like to explore this further. If this model had been deployed in the Kaggle competition, it would have finished in the top 10.

Unfortunately, ResNet did not perform as well as we had hoped. We trained the weights from scratch, and in the future we would like to try and implement it with pretrained weights. It is very possible that using these weights would lead to a significant improvement in performance.

Another area of interest is to explore demographic biases in more depth. The dataset was not labeled by gender or sex, and labeling would have been a time consuming task for this exploration. It would be interesting to see if training a model on the fer2013 dataset has some implicit biases that we could not identify.

References

1. K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In ICLR, 2015. arXiv:1409.1556v6
2. K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," arXiv:1512.03385, 2015.
3. https://github.com/opencv/opencv/blob/master/data/haarcascades/haarcascade_frontalface_default.xml