Facial Recognition of Identical Twins using Convolutional Neural Networks

Capstone #2 Final Report

Anthony Johnson

0) Abstract

Beginning with the VGG-Face pretrained convolutional neural network (CNN) in keras/TensorFlow [ref. 1], transfer learning was used to train a model to distinguish between images of me and my identical twin brother. The final model had two extra Dense layers, one with 100 nodes and one with 2 nodes (the 'softmax' activation layer). The last 5 layers were set as trainable, with a learning rate of 0.001. The final testing accuracy was 88%, with an area under the Receiver Operating Characteristic (ROC) curve of 0.94. Overall, the model failed in cases with bad or inconsistent lighting or sometimes when we were wearing sunglasses.

1) Introduction

Facial recognition software has come a long way in the last few years, thanks to convolutional neural networks (CNNs). Such software now appears in every-day consumer gadgets, such as in the iPhone X as a way to unlock your phone. Or as an auto-tagging feature in Facebook or iPhoto. And we are just scratching the surface of its potential.

One important question is, how good exactly are CNNs? When it comes to security, such as with the iPhone X, anything less than 100% accuracy is a major concern. Since I am an identical twin, I want to know if CNNs tell identical twins apart. In my experience, the answer is "sometimes." On a number of occasions I've been able to unlock *my brother's* iPhone with *my* face. Thus, facial recognition, while excellent, is still not perfect. That is, it's still not as good as a human (my family, for instance, would never mistake me for my brother). A computer's inability to reliably tell identical twins apart is an important security issue that needs to be addressed.

For this capstone project, I used a convolutional neural network, along with images of me and my brother, to dig deeper into facial recognition of identical twins.

2) Model

I first created an *instantiate_model*() function that creates a model with the same architecture as VGG-Face, an open source convolutional neural network that has already been trained with hundreds of thousands of faces [1]. After setting up the layers, it loads the weights from the "*vgg_face_weights.h5*" file.

Next, the function needs to set up the model for transfer learning, to adapt the model to distinguish between me and brother. It first removes the last layer of the network, the Activation('softmax') layer. It then adds a new Dense layer with *num_dense* nodes, where *num_dense* is an input into the function. This value was tuned later, but for reference, the final value of *num_dense* was 100.

That layer is followed by a final Dense Activation('softmax') layer with 2 nodes. These final nodes will tell us which twin it thinks is associated with a given input image.

In order to preserve the weights in the early model layers, the function then sets all layers as untrainable, except for the last *num_trainable* layers. That means that when we train the model with new images, the optimization can only change the weights in the final few layers. In this way, we can piggy-back off of the previous model training, which had significant computational cost. This is transfer learning. Again, *num_trainable* is an input to the *instantiate_model*() function, and was tuned later to a value of 5.

Below is the final model architecture summary. Note that there are 145,265,380 total parameters, but only 11,004,836 of them are trainable. This is still an astronomical number of trainable variables. Neural networks explore an extremely high-dimensional space, and ultimately that is why they work.

Model Architecture

	Layer (type)	Output Shape	Param #
1)	zero_padding2d_131_input (In	(None, 224, 224, 3)	0
2)	zero_padding2d_131 (ZeroPadd	(None, 226, 226, 3)	0
3)	conv2d_161 (Conv2D)	(None, 224, 224, 64)	1792
4)	zero_padding2d_132 (ZeroPadd	(None, 226, 226, 64)	0
5)	conv2d_162 (Conv2D)	(None, 224, 224, 64)	36928
6)	max_pooling2d_51 (MaxPooling	(None, 112, 112, 64)	0
7)	zero_padding2d_133 (ZeroPadd	(None, 114, 114, 64)	0
8)	conv2d_163 (Conv2D)	(None, 112, 112, 128)	73856
9)	zero_padding2d_134 (ZeroPadd	(None, 114, 114, 128)	0
10)	conv2d_164 (Conv2D)	(None, 112, 112, 128)	147584
11)	max_pooling2d_52 (MaxPooling	(None, 56, 56, 128)	0
12)	zero_padding2d_135 (ZeroPadd	(None, 58, 58, 128)	0
13)	conv2d_165 (Conv2D)	(None, 56, 56, 256)	295168

```
14)   zero_padding2d_136 (ZeroPadd (None, 58, 58, 256)    0
15)   conv2d_166 (Conv2D)         (None, 56, 56, 256)    590080
16)   zero_padding2d_137 (ZeroPadd (None, 58, 58, 256)    0
17)   conv2d_167 (Conv2D)         (None, 56, 56, 256)    590080
18)   max_pooling2d_53 (MaxPooling (None, 28, 28, 256)    0
19)   zero_padding2d_138 (ZeroPadd (None, 30, 30, 256)    0
20)   conv2d_168 (Conv2D)         (None, 28, 28, 512)    1180160
21)   zero_padding2d_139 (ZeroPadd (None, 30, 30, 512)    0
22)   conv2d_169 (Conv2D)         (None, 28, 28, 512)    2359808
23)   zero_padding2d_140 (ZeroPadd (None, 30, 30, 512)    0
24)   conv2d_170 (Conv2D)         (None, 28, 28, 512)    2359808
25)   max_pooling2d_54 (MaxPooling (None, 14, 14, 512)    0
26)   zero_padding2d_141 (ZeroPadd (None, 16, 16, 512)    0
27)   conv2d_171 (Conv2D)         (None, 14, 14, 512)    2359808
28)   zero_padding2d_142 (ZeroPadd (None, 16, 16, 512)    0
29)   conv2d_172 (Conv2D)         (None, 14, 14, 512)    2359808
30)   zero_padding2d_143 (ZeroPadd (None, 16, 16, 512)    0
31)   conv2d_173 (Conv2D)         (None, 14, 14, 512)    2359808
32)   max_pooling2d_55 (MaxPooling (None, 7, 7, 512)      0
33)   conv2d_174 (Conv2D)         (None, 1, 1, 4096)     102764544
34)   dropout_21 (Dropout)        (None, 1, 1, 4096)     0
35)   conv2d_175 (Conv2D)         (None, 1, 1, 4096)     16781312
36)   dropout_22 (Dropout)        (None, 1, 1, 4096)     0
37)   conv2d_176 (Conv2D)         (None, 1, 1, 2622)     10742334
38)   flatten_11 (Flatten)        (None, 2622)           0
39)   dense_21 (Dense)           (None, 100)            262300
40)   dense_22 (Dense)           (None, 2)              202
=================================================================
Total params: 145,265,380
Trainable params: 11,004,836
Non-trainable params: 134,260,544
```

3) Data

For the data, I gathered digital photographs of me and my brother over the last 10 years. I square cropped each image to just our faces, and split the images into 3 datasets.

1) Training = 118 images per twin
2) Validation = 27 images each per twin
3) Testing = 33 images each per twin

The exact number in each dataset was arbitrary, though I chose the training set to be the largest by a factor of roughly 4-to-1.

To prepare the images for the model, I created a *getXY*() function that loads each image, resizes it to 224x224 pixels, and then stacks them together into a *trainX* numpy array. And into *valX* and *testX*, for the validation and testing datasets, respectively. The VGG-Face

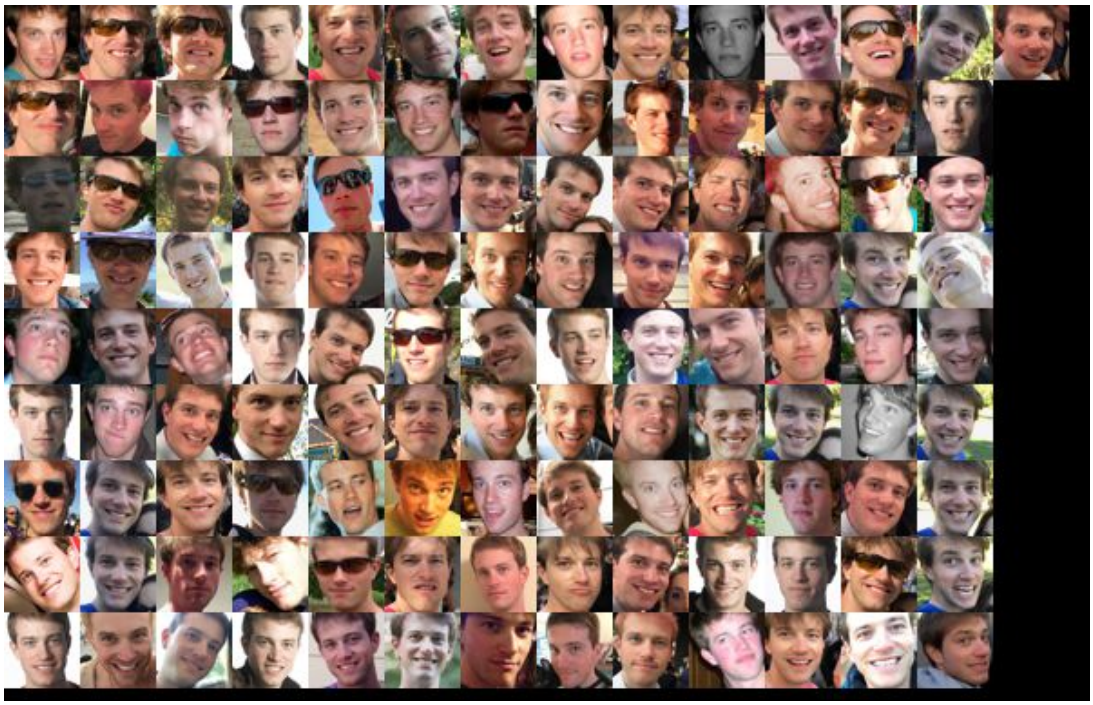
model requires images of 224 squared. The function also outputs *y*, the categorical dependent variable which simply has a 0 for me or 1 for my brother.

Below are collages for the training images of both me (AJ) and my brother (PJ), made with a custom *create_collage*() function. Here are some of the sources of variability in the images:

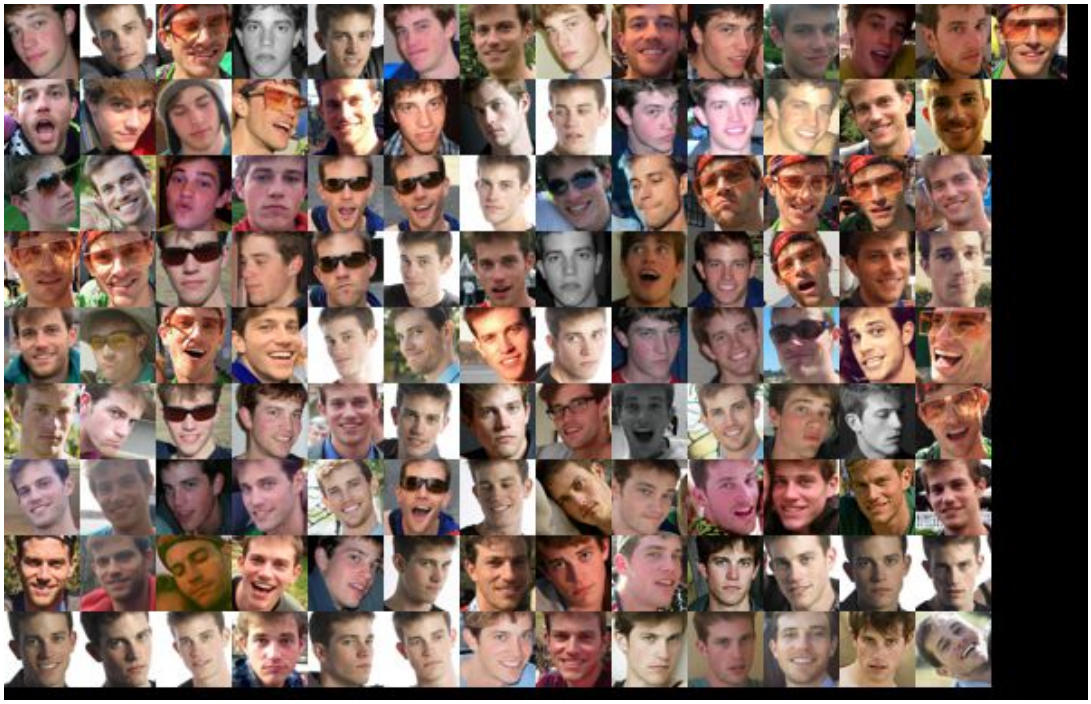
1) Orientation of the face
2) Lighting
3) Some have sunglasses, some not
4) Some are smiling, some not
5) Some are from 2011, some are from 2019

With these stacks of images, we are now ready to train and optimize the model.

Collage for AJ



Collage for PJ



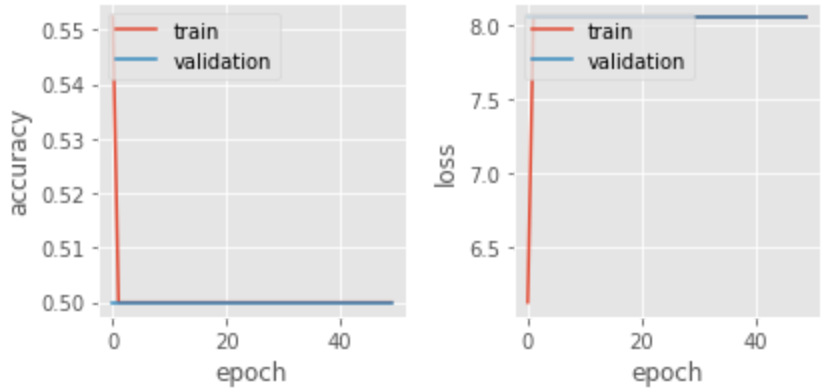
4) Model Training & Optimization

I experimented with a few hyperparameters and optimizers, in order to find a model that makes the best predictions. Below is a list of the things I varied:

1) The number of layers that are trainable, *num_trainable*
2) The learning rate, *lr*
3) The number of nodes, *num_dense*, in the penultimate layer
4) The decay rate, *decay*, of the learning rate
5) The optimizer: SGD vs. Adam

Iteration #1

In the first iteration, I only varied the number of trainable layers *num_trainable*, using values of [3, 4, 5, 6, 7, 8, 9]. The models all had 120 nodes in the penultimate layer. I used the SGD optimizer with learning rate = 0.1, decay = 0, loss = 'categorical_crossentropy', and metrics = 'accuracy.' As seen below, these runs suffered from overfitting and thus did not converge. The models predicted every image to be the same twin. I suspected the learning rate was too high, and that I could play with a few other parameters to get the results to converge. I tried using the Adam optimizer, with similar results.



Iteration #2

In addition to varying the number of trainable layers, I also changed:

1) The learning rate (*lr*) = [0.001, .005, .01, .05, .1, 1]
2) The number of nodes in the penultimate layer (*num_dense*) = [10, 50, 100, 200]
3) I also include an EarlyStopping condition, with a patience of 4, monitoring the validation loss metric (*val_loss*).

These runs did converge, having final accuracies between 90%-100% and validation accuracies between 85%-95%. The results were quite consistent across the board, so it was difficult to choose which model was best.

Iteration #3

Finally, I varied the decay of the learning rate, with values [.000005,.00001,.00002,.00004]. And I increased the patience to 6. These runs also converged, though I didn't see any improvement in the accuracy compared to Iteration #2.

Best Model

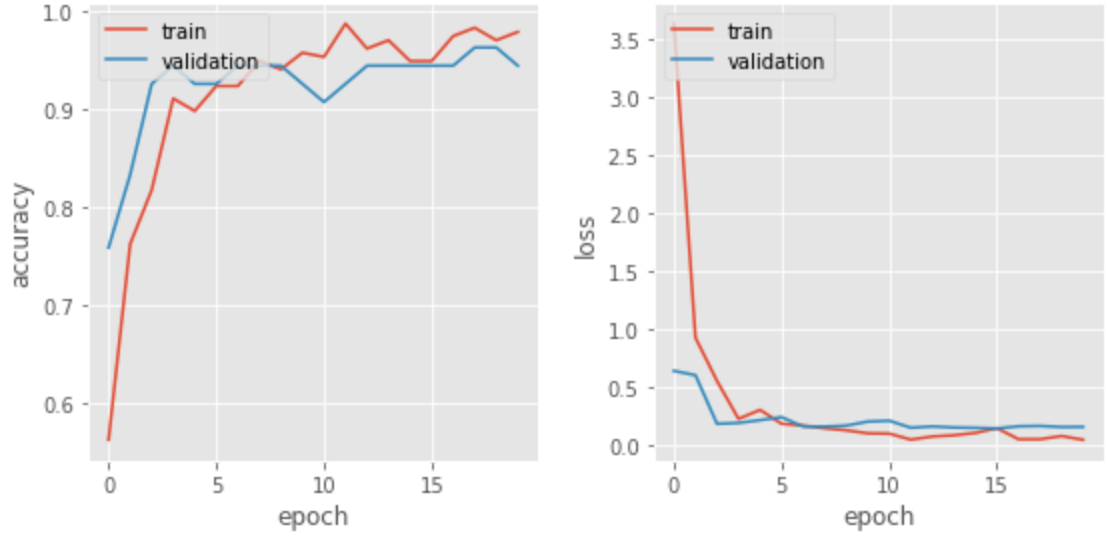
The best model had the parameters below:

1) Num_dense = 100
2) Num_trainable = 5
3) Learning rate = 0.001
4) Decay = 0
5) Optimizer = SGD
6) EarlyStopping patience of 6 epochs

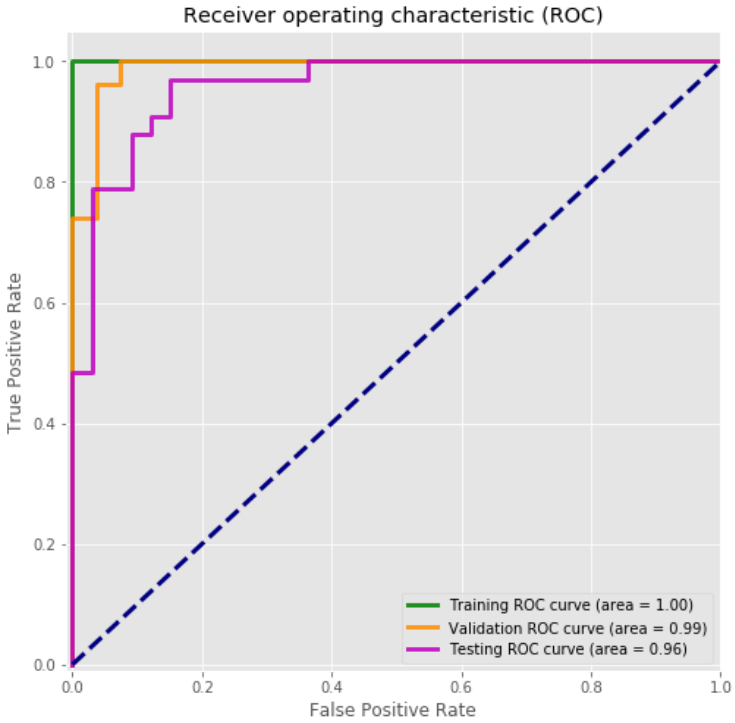
Model accuracy:

1) Training accuracy: 1.0
2) Validation accuracy: 0.94
3) Testing accuracy: 0.88

Below we see the training history for this model. It took 19 epochs to converge, and the validation accuracy reached 0.94 within 3 epochs.



Below is an ROC plot for the final model. The final testing area-under-the-curve was 0.96, not bad for identical twins.



Below are the confusion matrices for training, validation, and testing. In the validation set, there was one false PJ (predicted AJ, but actually PJ), and two false AJs (predicted PJ, but actually AJ). In 2 out of 3 misclassifications, there were sunglasses that partially obstructed the face.

TRAINING SET	PREDICTED - AJ	PREDICTED - PJ
ACTUAL - AJ	118	0
ACTUAL - PJ	0	118

VALIDATION SET	PREDICTED - AJ	PREDICTED - PJ
ACTUAL - AJ	25	2
ACTUAL - PJ	1	26

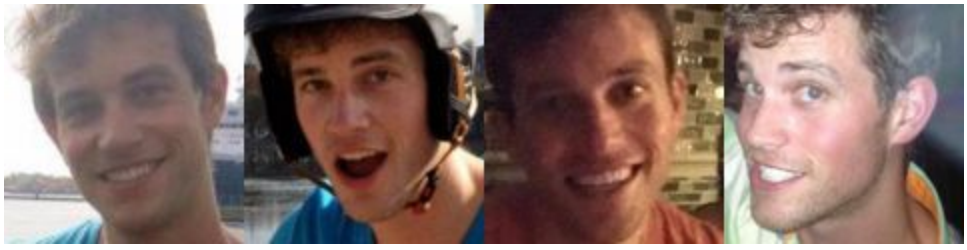
TESTING SET	PREDICTED - AJ	PREDICTED - PJ
ACTUAL - AJ	29	4
ACTUAL - PJ	4	29

In the final testing set, there are 4 false PJs and 4 false AJs (images shown below). Potential causes of these misclassifications:

1) Lighting is inconsistent
2) Sunglasses
3) Slightly unusual facial expressions

Actual = AJ, Predicted = PJ

Actual = PJ, Predicted = AJ



5) Model Interpretation

One question is, *how* is the model telling us apart? In other words, what specific features does it look for that differentiate us?

To explore this, I used the open-source LIME package, which stands for Local Interpretable Model-Agnostic Explanations [ref. 2]. In particular, I used the *LimeImageExplainer*(), which essentially identifies features in an image that increase the probability (green) or decrease the probability (red) of a certain class. For instance, in the collages below, red patches indicate a higher probability of PJ and green patches indicate a higher probability of AJ. These are from the validation dataset.

Image Explainer (AJ)

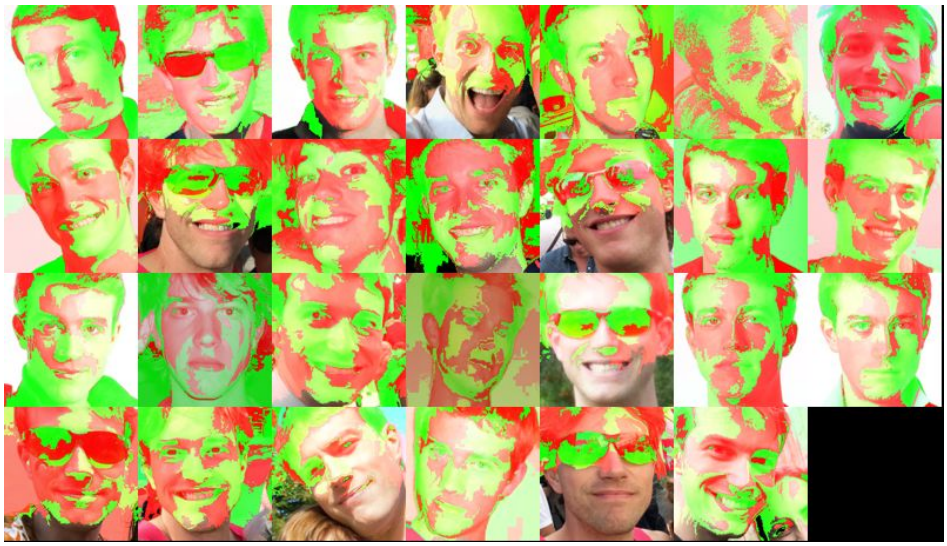
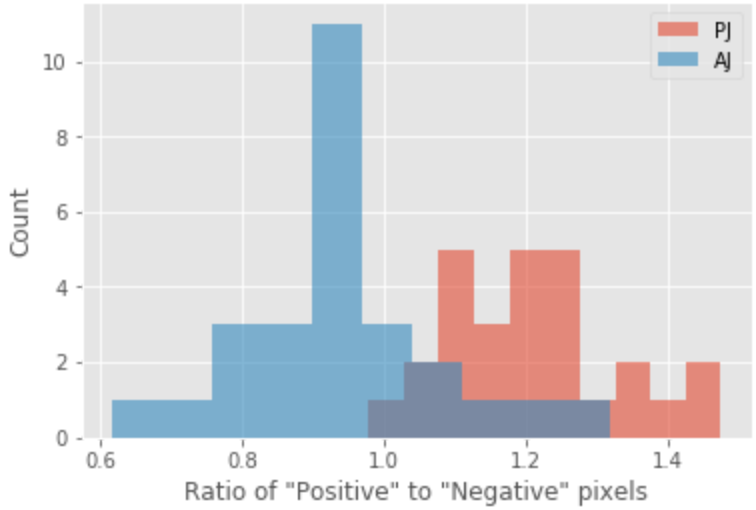


Image Explainer (PJ)

I summed the total number of green and red pixels for both AJ and PJ. I then, for each class, divided the “positive” pixels by the “negative” pixels. In the images with a PJ label, the ratio of red to green pixels was between 1.0 and 1.5, averaged to 1.21. But for AJ images, the equivalent ratio (green to red in this case) varied much more widely, between 0.6 and 1.3. Below is a histogram of these ratios.

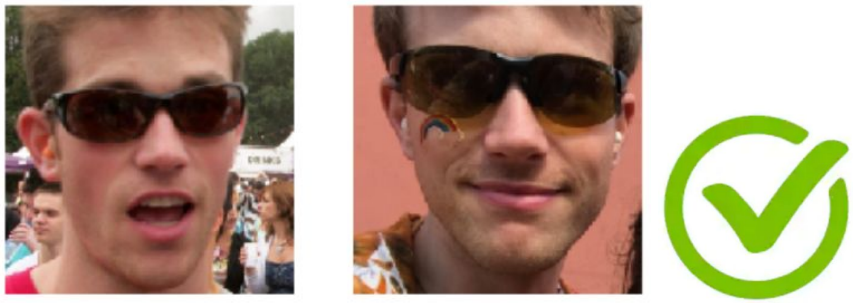


It's unclear why exactly the ratios are so different between AJ and PJ, but it ultimately means that the model has an easier time recognizing PJ than AJ. This pattern has been consistent; nearly every model trained in this project resulted in more false AJs than false PJs. One explanation could be that the PJ images are "better," and this difference is due to chance in picking the images.

I also used a *verifyFace*() function [ref. 3] to compare the final vectors for validation set images img14, img30, img31. These images correspond to PJ, AJ, AJ, respectively. The model predictions were PJ, PJ, and AJ, respectively, i.e. the second image was misclassified. These are the face comparison results:

```
Cosine similarity:  0.4106149673461914
Euclidean distance:  35.715675
unverified! they are not same person!
```

```
================================================
Cosine similarity:  0.13973468542099
Euclidean distance:  21.714138
verified... they are same person
```

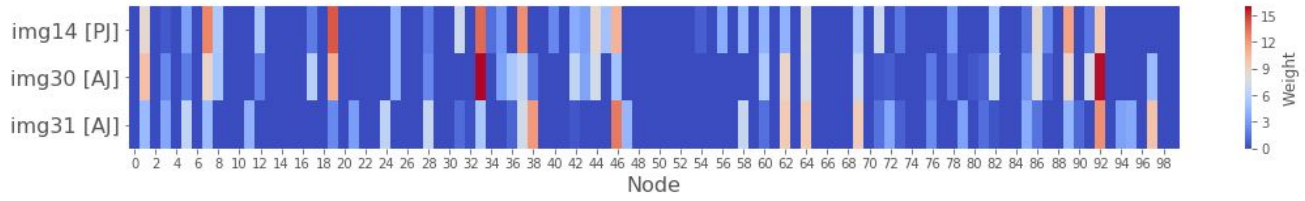


```
================================================
Cosine similarity:  0.36515021324157715
Euclidean distance:  32.855408
verified... they are same person
```

```
================================================
```

Curiously, the model can tell that img14 is different from img31, and that img30 is img31, but NOT that img14 is different from 30.

Below is a heatmap of the 3 final layer vectors associated with each of the 3 images from above. Qualitatively, there is significant similarity between the top two lines (img14, img30), which makes sense based on the misclassification. We would expect the bottom two rows to be more similar since they are both AJ.



6) Conclusions and Future Work

The final convolutional neural network had a Dense layer with 100 nodes followed by a final Dense layer with 2 nodes. By looping through multiple values of the learning rate, decay, number of trainable layers, and number of nodes in the final layers, I was able to get the weights optimization to converge. In some cases, I saw convergence in as few as 2 or 3 epochs. The final testing accuracy was 88%, with an ROC AUC of 0.96. The training accuracy was 100%. Overall, the model failed in cases with bad or inconsistent lighting or sometimes when we were wearing sunglasses.

Using LIME and a function to compare faces, I was able to better understand how the model views each of our faces. However, the results aren't exactly conclusive and are themselves hard to interpret. Interpreting neural networks could be a career in itself, and so these techniques truly just scratch the surface.

While decent for identical twins, this model would be unacceptable in a security application. To improve the model, the first and most obvious step would be to train more images, at a computational cost. In the realm of neural networks, a training set with 236 images is considered miniscule. In most real world applications, models can improve themselves over time, as increasingly more images are collected.

It is also possible that better accuracy could be achieved by further optimizing the hyperparameters in the model, and by optimizing the optimizer itself. Overall, given more images and greater computational resources, I am confident that I could achieve a testing accuracy close to 100%.

Perhaps in a future project, I would like to dig deeper into interpreting and visualizing convolutional neural networks. Simply due to the large number of parameters involved, CNNs are difficult to interpret. But this is still a relatively new field, and anything is possible.

Github Repository:

https://github.com/Aejohnso/Springboard/tree/master/Capstone_2_Project

Sources

[1] https://gist.github.com/EncodeTS/6bbe8cb8bebad7a672f0d872561782d9
[2] https://github.com/marcotcr/lime
[3] https://sefiks.com/2018/08/06/deep-face-recognition-with-keras/