Capstone 3
Carsten Bruckner
March 2022

Detection of female and male eyes using a convolutional neural network

Problem statement
Classify cropped images of single eyes as either female or male with high accuracy.

Criteria for success
Be able to reproduce the results from another user's Jupyter Notebook of a Kaggle.com dataset.

Background
Image classification tasks are frequently handled by neural network models. The models are capable of segmentation of the image and labeling multiple image features. Model architecture, input data requirements, parameter tuning can be simplified if the classification requirements are simple. This project reduces the model requirements to assign one of two possible labels to the entire image: "femaleeyes" or "maleeyes".

Data
Kaggle.com submitter PavelBiz submitted a dataset (https://www.kaggle.com/pavelbiz/eyes-rtte) composed of faces cropped to a single eye.

- Folder femaleeyes contains 5202 .jpg images
- Folder maleeyes contains 6323 .jpg images

Most of the image files have less than 75x75 pixels, and some are grayscale.

Existing Jupyter Notebook template for this project created by LUCAS ARAÚJO:
https://www.kaggle.com/lucasar/detection-of-male-female-eyes-convnet-92-acc

Method
I originally repeated the execution of Lucas' notebook on my Google Colaboratory instance, using 12K input files I unzipped and then uploaded to my Google Drive (unzipping using online tools timed out). However CoLab would suspend execution after long periods of inactivity and I'd have to reload variables. In all, it took about 2 days to run through the whole notebook. Also, excecution times were a lot longer than I expected based on timings of original Notebook on Kaggle.

So I instead processed a modified version of the Notebook from my desktop, with all files local. Notes:

- set up a new tensorflow environment with older versions of Python and h5py, and other packages used by the notebook
- install pydot and graphviz via pip, as Anaconda installation method was not successful

I put a couple hours trying to read the source images directly from .zip, but ran out of time to handle this way. Instead I unzipped the files and accessed using template Notebook's existing code. It may also be that processing time is slowed down if input files are zipped with compression

Key software versions:

MODULE	VERSION
JUPYTERLAB	3.2.9
PYTHON	3.7.7
TENSORFLOW	2.0.0
SCIKIT-LEARN	1.0.2
H5PY	2.10.0
PANDAS	1.3.4

Data Cleaning and Feature Engineering

Fortunately the input images were already prepared and grouped into folders by expected label.

No input images were excluded.

The two labels "femaleeyes" and "maleeyes" were converted to numeric labels "0" and "1" using

```
LE = sklearn.preprocessing.LabelEncoder()
y_test = LE.fit_transform(testset_df["Label"])
```

The following function includes a step where input images were standarized to same size:

```
tensorflow.keras.preprocessing.image.ImageDataGenerator.flow_from_dataframe(d
ataframe = trainset_df,target_size = (75, 75),color_mode = "rgb")
```

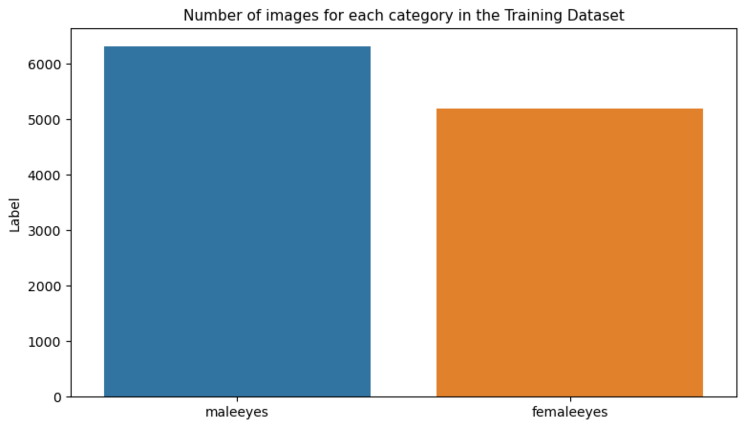
To increase the input data for model training purposes, the `ImageDataGenerator` also randomly modifies the images differently in each training epoch, with the following parameters:

```
train_datagen = ImageDataGenerator(rescale = 1./255,
                                    shear_range = 0.2,
                                    zoom_range = 0.1,
                                    rotation_range = 20,
                                    width_shift_range = 0.1,
                                    height_shift_range = 0.1,
                                    horizontal_flip = True,
                                    vertical_flip = True,
                                    validation_split = 0.1)
```

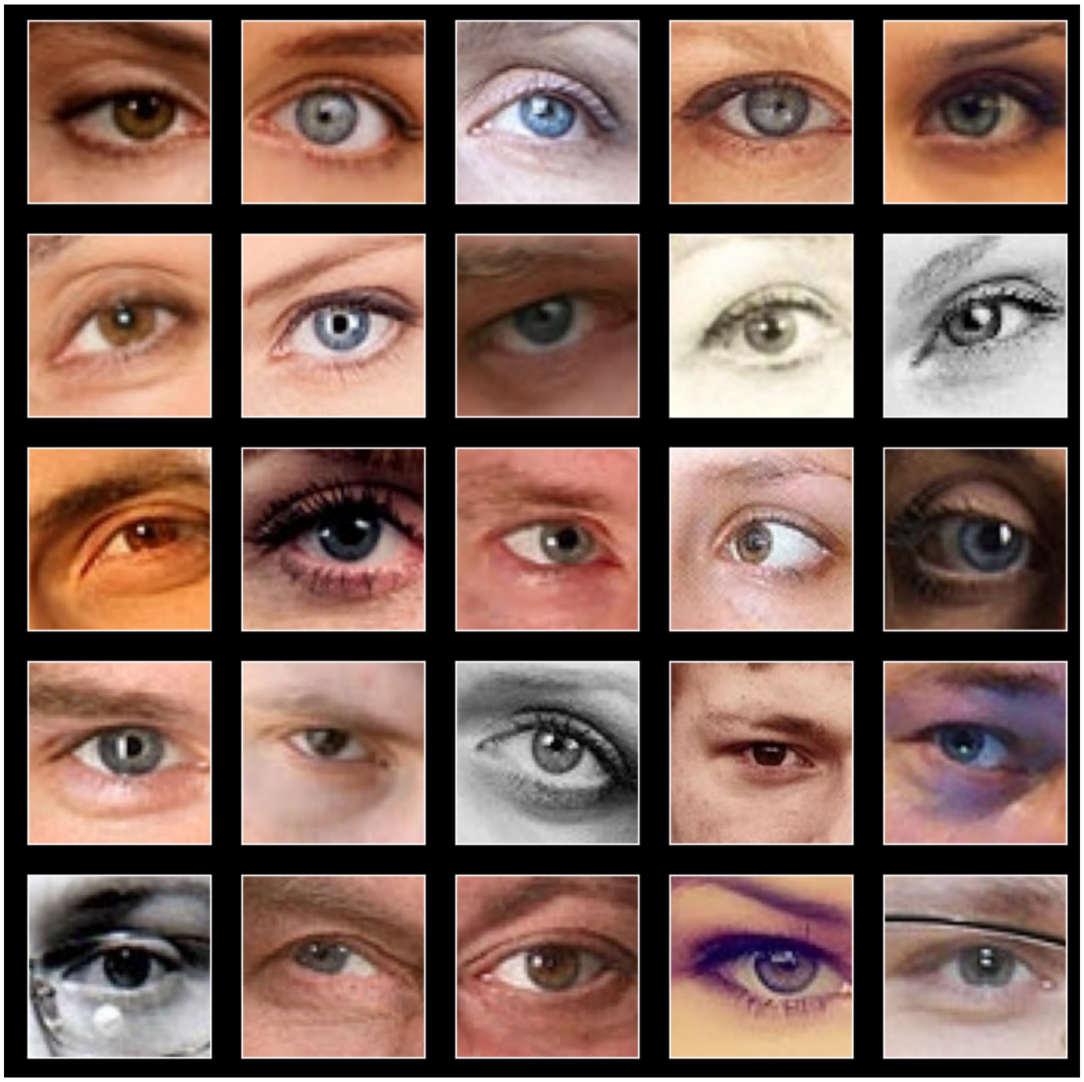
The second layer of the neural network is "BatchNormalization", applied immediately after the first convolutional layer is applied to the images modified by `ImageDataGenerator`. This applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1 (see https://www.tensorflow.org/api_docs/python/tf/keras/layers/BatchNormalization)

Exploratory Data Analysis

The number of maleeyes and femaleeyes input images are mostly balanced.



Picking 25 shuffled images, labeled by the source folder:



Variations across images besides sex:

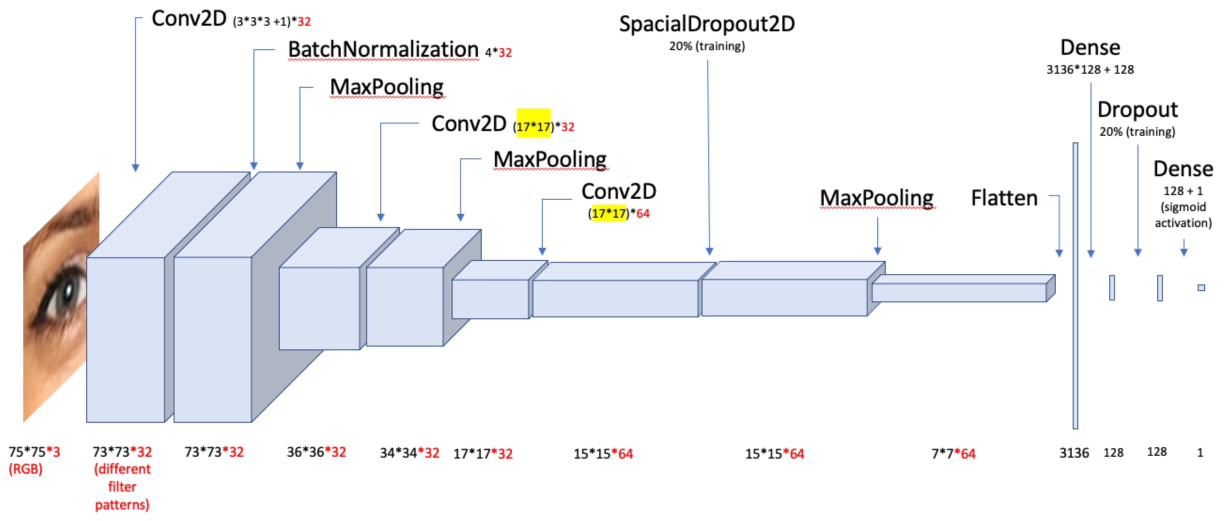
- image quality (number of input pixels, blurriness, noise, contrast)
- color vs grayscale
- magnification of eye
- partial obstructions (glasses, hair)
- lighting differences, impact on shadows
- orientation differences:
    - pupil location in eye,
    - facial orientation vs camera
    - location of eye vs center of picture

A random 25% of the images was held out for final testing of the model.

```
Training Dataset:
Number of images: 8643
Number of images with male eyes: 4729
Number of images with female eyes: 3914

Test Dataset:
Number of images: 2882
Number of images with male eyes: 1594
Number of images with female eyes: 1288
```

Algorithms and Machine Learning

A sequential convolutional neural network architecture was chosen, because of the 4 models tested in the template Notebook, this one had the best test accuracy. The specific architecture is described in the following graphic and table.



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d (Conv2D)              (None, 73, 73, 32)        896       
_________________________________________________________________
batch_normalization (BatchNo (None, 73, 73, 32)        128       
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 36, 36, 32)        0         
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 34, 34, 32)        9248      
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 17, 17, 32)        0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 15, 15, 64)        18496     
_________________________________________________________________
spatial_dropout2d (SpatialDr (None, 15, 15, 64)        0         
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 7, 7, 64)          0         
_________________________________________________________________
flatten (Flatten)            (None, 3136)              0         
_________________________________________________________________
dense (Dense)                (None, 128)               401536    
_________________________________________________________________
dropout (Dropout)            (None, 128)               0         
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 129       
=================================================================
Total params: 430,433
Trainable params: 430,369
Non-trainable params: 64
_________________________________________________________________
None
```

Each of the three 2d convolutional layers is a 3x3 spacial convolution of the layer's input. For each of the 32 pattern filters in the first convolutional layer, there's a 3x3 convolution for each of 3 colors, which are linearly combined with a bias parameter, before going to a ReLu activation to get a single output. This means there are 32 * (3x3 spacial * 3 channels + 1 bias) = 896 parameters to be trained on just the input layer.

The number of trainable parameters are listed for each node. The weights for the first dense layer have the most parameters, because there is a full connection between each of the 3136 input values to each of the 128 nodes in the dense layer (3136*128=401536 weights).

The max_pooling layers each pass on only the maximum value of the adjacent 2x2 "pixel" inputs.

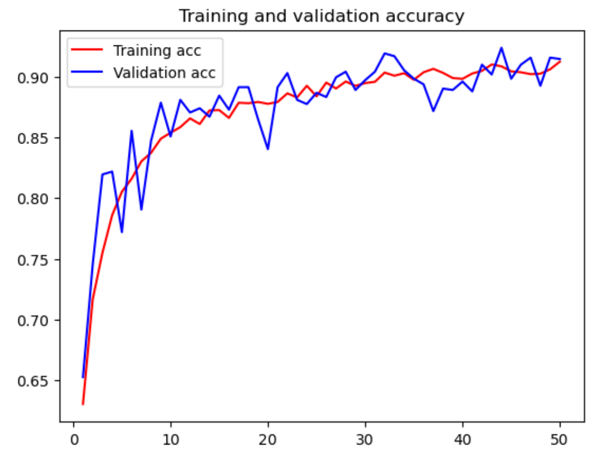
The dropout layers mask out a random 20% of the input values from passing through to the next layer. This is done to reduce the dependence of the model on specific nodes in the next layer, to reduce overfitting when working with small datasets like this one. I _*believe*_ without evidence that the masked values are randomized once for each batch of 32 images within each epoch, during the training phase. Note that the dropout behavior only happens during the training phase, NOT the prediction phase.

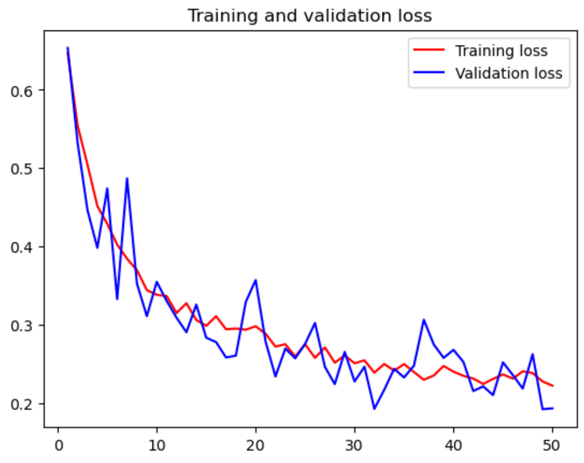
The final dense layer connects the previous layer's 128 outputs to a single input node, and uses 128 weights + 1 normalization parameter to generate a single value. This value goes to a sigmoid activation function to generate a continuous output ranging from 0 to 1. When this output is rounded to the nearest integer, the 0 prediction predicts the "femaleeyes" class, and the 1 prediction predicts the "maleeyes" class.

Parameter Tuning

I did not do specific hyperparameter tuning, like adjusting the %dropout in the dropout layers, or the max number of epochs, or how the images were randomly modified to increase diversity.

The 50 epochs of training took around an hour of real time. 10% of the training samples were used to measure validation accuracy and loss after each epoch. According to the training history plots below, going beyond 50 epochs is not likely to improve model performance significantly.





Predictions

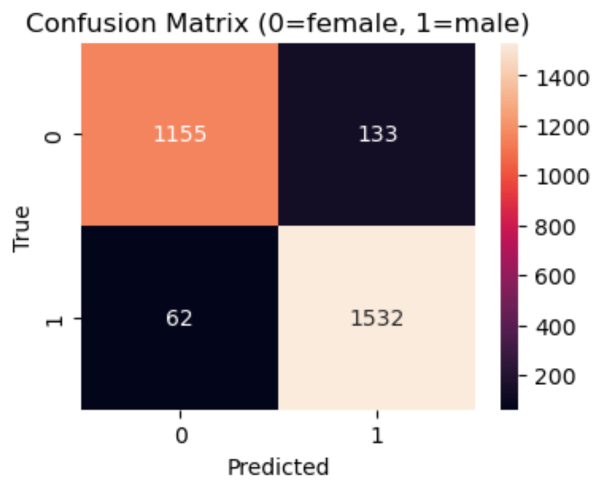
The trained model was used to predict the sex of the held-out test set of 2882 images.
label 0 = femaleyes
label 1 = maleeyes

The overall test set prediction accuracy is 93.2%. Male eyes were correctly identified as male 96% of the time (recall), and female eyes were correctly identified as female 90% of the time. However, a higher percentage of images labeled as female actually are female (95% precision) than for labeled male images actually being male (92% precision). So the model errored on the side of overcalling males.

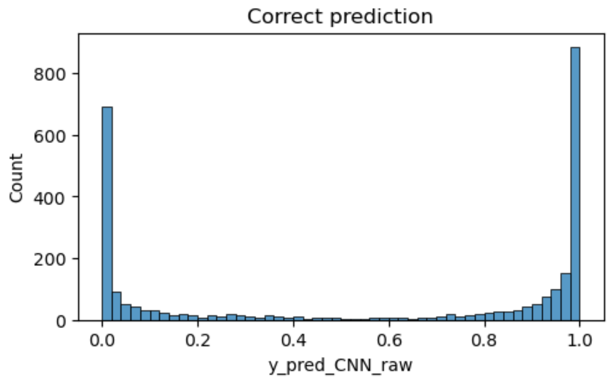
The full classification report:

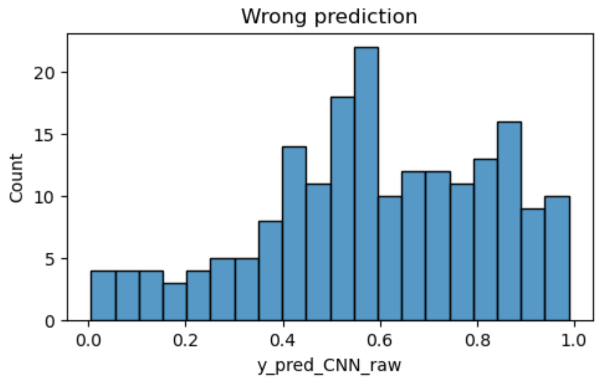
```
              precision    recall  f1-score   support

           0       0.95      0.90      0.92      1288
           1       0.92      0.96      0.94      1594

    accuracy                           0.93      2882
   macro avg       0.93      0.93      0.93      2882
weighted avg       0.93      0.93      0.93      2882
```

…and confusion matrix:

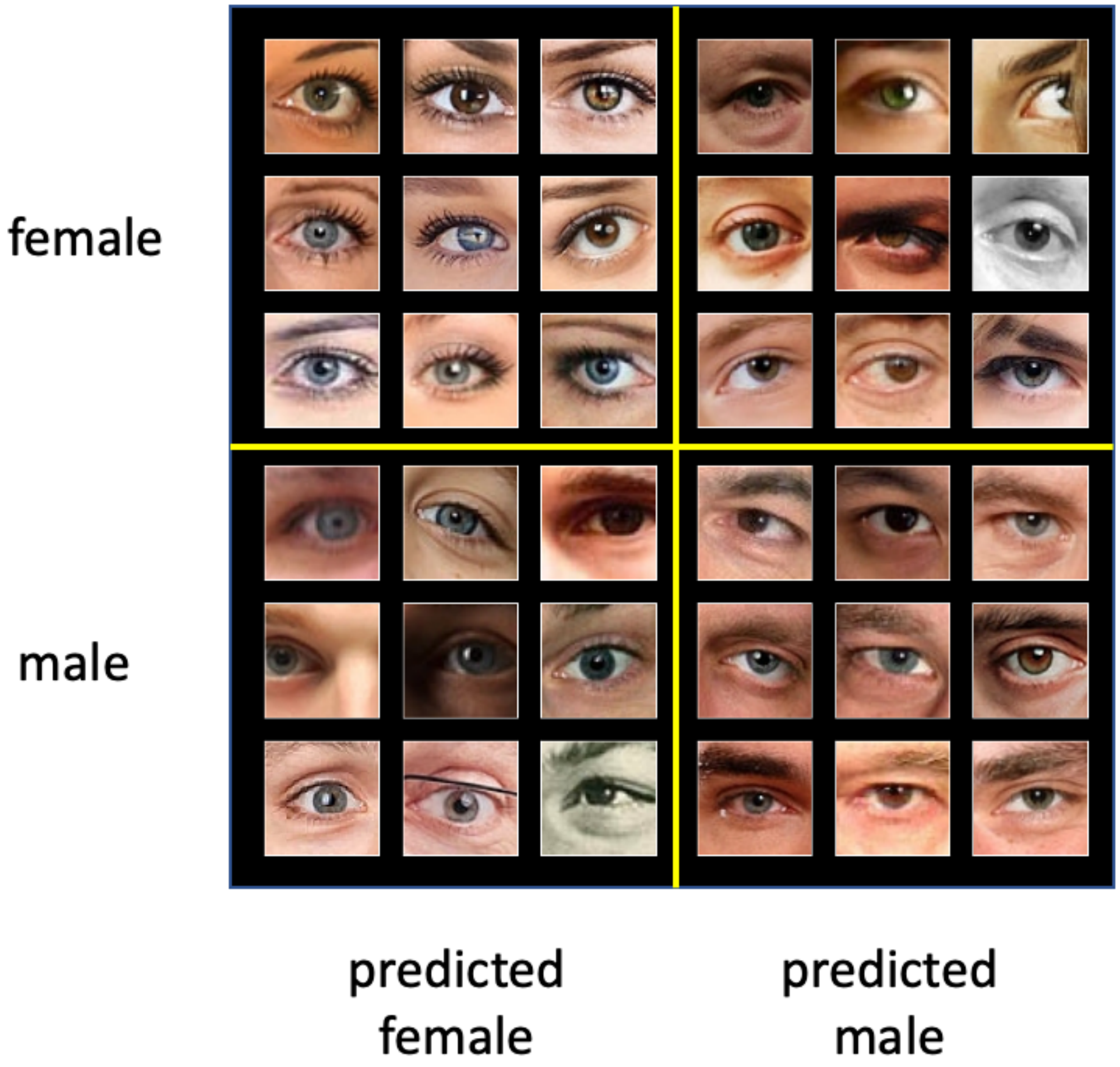


The raw prediction score ranges from 0 to 1. This score is rounded to the nearest integer to assign the binary label prediction. Correctly-classified images usually have a raw prediction score close to 0 or 1. I assumed that incorrectly classified images would have a raw prediction score closer to the middle of the range. Proportionally the misclassified images do have more middle range raw scores, but there are some misclassified images with a raw prediction score close to 0 and 1.



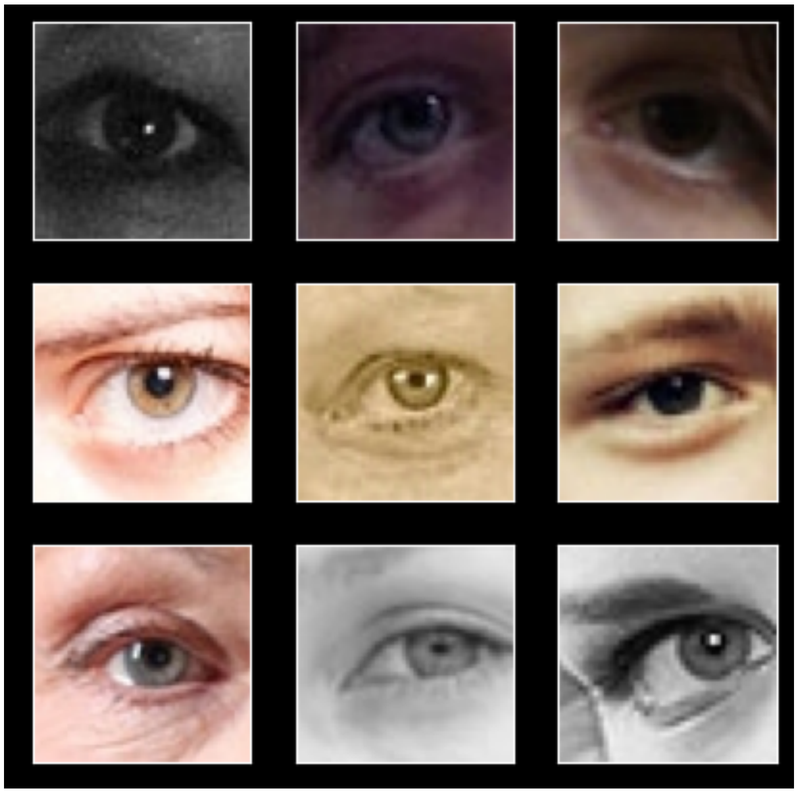


Let’s look at the most confident predictions from each of the 4 quadrants of the confusion matrix:



The correctly-predicted female eyes images all have strong eyelash features. The male images and the incorrectly-predicted female images do not have strong eyelash features. Maybe the stonger eyebrows on males are helping to predict the male label.

Which images did the model have the most difficulty predicting? It would be those with a raw prediction score between 0.49 to 0.51:



I think a human can correctly classify these images better than the model.

How this model can be used

I can't think of a practical use for this model.

Future improvements

I'm interested in getting the prediction accuracy as least as good as human accuracy. I could randomize about 500 pictures and manually label them, and from there get a baseline on human prediction accuracy.

I could try to better balance the prediction accuracy for both genders by retraining with a custom weighted metric that weights female image accuracy higher than male image accuracy.

https://stackoverflow.com/questions/60399983/how-to-create-and-use-weighted-metrics-in-keras

Even though the male and female eyes classes are not that imbalanced, I could try to fully balance by changing the class_weight parameter when calling model.fit()
https://www.tensorflow.org/tutorials/structured_data/imbalanced_data

I could try changing the architecture of the model by perhaps adding an additional Convolutional layer, or adjusting the number of filters.

I could adjust the model hyperparameters like dropout rate in the two dropout layers, and increase the epochs 4x to see if I can squeeze out another 1% of accuracy.

An expert Kaggle notebook contributor to this project was able to achieve 96% accuracy https://www.kaggle.com/code/gpiosenka/multi-function-callback-to-try-f1-score-96, so I could evaluate his model as well.