EIASR 21Z - Face Mask Detection
Final report

Michał Smutkiewicz, 283538

Tomasz Miazga, 293071

Contents

Link to GitHub repository (to download models):
https://github.com/smutkiewicz/face-mask-detection

Task description

The objective of this project is to, in general, **recognize if people, identified on given pictures can be recognized as people with face masks**, a feature that can be useful in fighting a pandemic. Our main idea is to use convolutional neural networks to deal with the recognition. To get a network working like this, we will try to teach it to classify people with and without masks. As a result, based on given input images, system output should provide a classification of each input image.

PREPARATION PART

Choice of initial model

We will be taking advantage of *Transfer Learning*. As an initial model, we have chosen two options to experiment: pretrained ResNet50 model (which is quite big compared to our problem) and our own model (which might be better suited to our problem and easier to handle if properly constructed).

Changes comparing to prototype stage:

- *We've decided to experiment with simplified custom model,*
- *We've decided to change the input size of the network to a smaller one to reduce complexity of the solution (250 to 32).*

First model - ResNet50

As a first option, we chose **ResNet50**. It is a very popular model when it comes to image processing mainly due to its great accuracy. ResNet50 was appreciated and confirmed at the ILSVRC 2015 classification competition, achieving an error of only 3.57%. The name corresponds to the term *Residual Networks* and number 50 references the number of layers the model consists of.

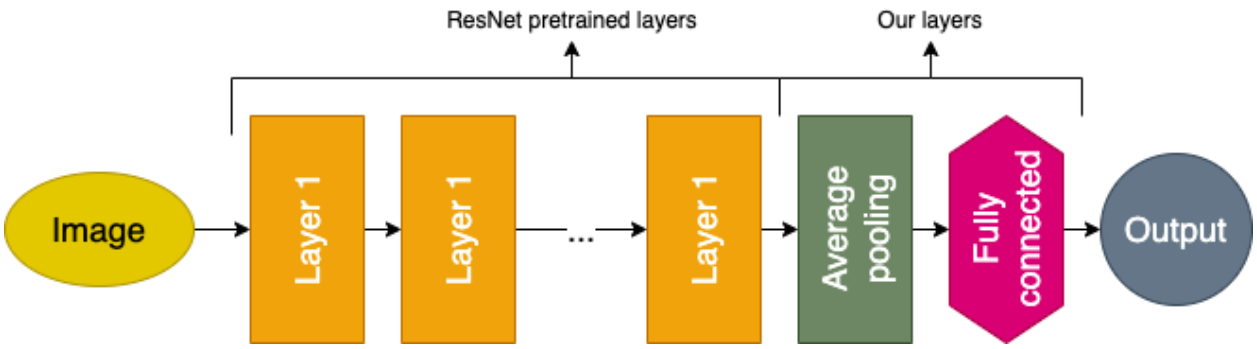

Fig. 1: ResNet solution overview

Base model is pretrained, so we froze those layers before training. Instead, we've added two more layers on top to train:

- **Average pooling**: required to downsample the detection of features in a feature map. It maps square patches (usually 2x2) into the average of values inside, dividing the size of the feature map by 2 in every dimension. This layer answers the problem of feature location in the input image sensitivity (local transition invariance).
- **Fully connected**: so called *dense layer*, is a part classification layer of a network. It is used for supporting the final classification, returning the probability of belonging to a specified class (in our case - whether a person has a mask on, or not).

Here's how added layers present on summary of the network (we've skipped pretrained part of the network):

```
Model: "ResNet50"
__________________________________________________________________________________________
___
 Layer (type)                 Output Shape              Param #   Connected to
==========================================================================================
===

input_1 (InputLayer)         [(None, 24, 22, 3  0                                          []
                             )]
… base model (skipped) …

global_average_pooling2d (Glob (None, 2048)          0           ['conv5_block3_out[0][0]']
alAveragePooling2D)

dense (Dense)                (None, 2)             4098      ['global_average_pooling2d[0][0]']

==========================================================================================
===
Total params: 23,591,810
Trainable params: 4,098
```

```
Non-trainable params: 23,587,712
_________________________________________________________________
```

Second model - custom architecture

As a second option, we decided to try some smaller network. The best way to do this was to create a custom model, which would better suit our problem that has rather low complexity. Better because of its smaller size and less complication than ResNet50.

Here's how model architecture present on summary of the network:

```
Model: "model_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_5 (InputLayer)        [(None, 32, 32, 3)]       0

 conv2d_16 (Conv2D)          (None, 30, 30, 10)        280

 conv2d_17 (Conv2D)          (None, 28, 28, 10)        910

 max_pooling2d_8 (MaxPooling  (None, 14, 14, 10)       0
 2D)

 conv2d_18 (Conv2D)          (None, 12, 12, 10)        910

 conv2d_19 (Conv2D)          (None, 10, 10, 10)        910

 max_pooling2d_9 (MaxPooling  (None, 5, 5, 10)         0
 2D)

 flatten_8 (Flatten)         (None, 250)               0

 dense_8 (Dense)             (None, 8)                 2008

 dropout_8 (Dropout)         (None, 8)                 0

 flatten_9 (Flatten)         (None, 8)                 0

 dense_9 (Dense)             (None, 2)                 18

 dropout_9 (Dropout)         (None, 2)                 0

=================================================================
Total params: 5,036
Trainable params: 5,036
Non-trainable params: 0
_________________________________________________________________
```

We decided to go with:

- Input layer of size 32x32x3 to keep problem input size and thus network complexity as simple as possible,
- Two sets of: 2x convolutional layer + max pooling layer,
- Two sets of: flatten layer (to “resize” inputs for next dense layer) + dense layer (of size 8 and then 2 for two-class output) + dropout layers (to prevent overlearning, we experimented with rate from 0.1 to 0.2),
- All layers were trainable this time.

Main algorithm description

We divided the algorithm into two main phases: detection and classification. For detection, we will use the OpenCV library and for face classification, we will use pre pre-trained convolutional neural network - ResNet50.

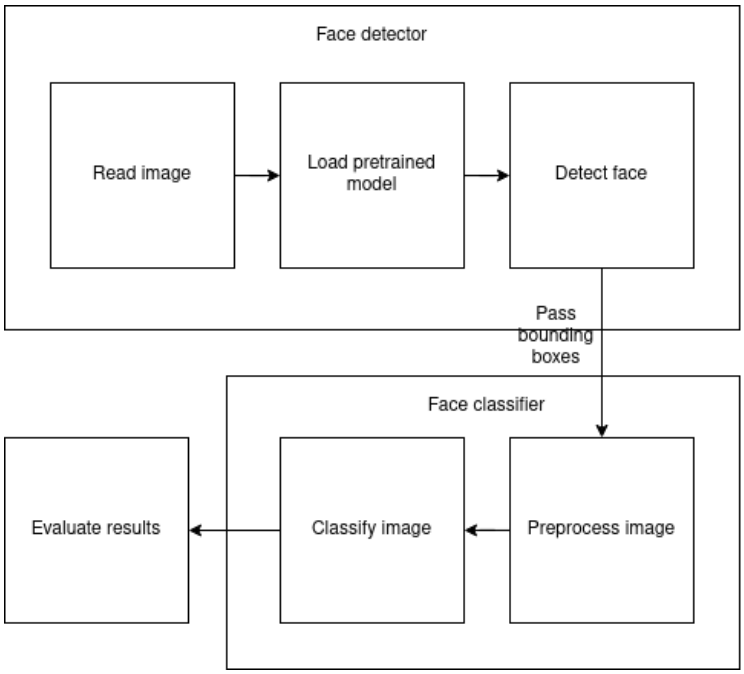

Fig. 2: Main algorithm diagram

1. Face detection

   Our first general step is to detect all possible faces on a given image. We will be using a pre-trained cascade classifier in OpenCV.

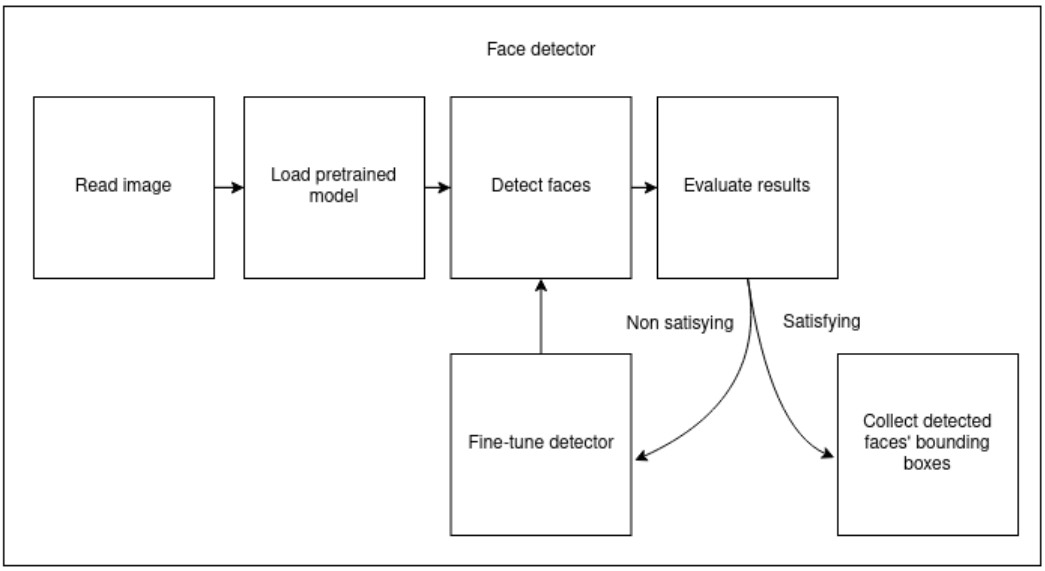

Fig. 3: Face detection algorithm diagram

- **Reading image**: randomly pick and load an image from our validation image set.
- **Loading pretrained model**: we will use the *Open Frontal Face Detection* Model from OpenCV Github project resources.
- **Detecting faces**: we will use the Haar Cascade Classifier, which is already trained with several hundred positive sample views of a particular object and arbitrary *negative* images of the same size. After the classifier is trained it can be applied to a region of an image and detect the object by moving a search window across the image and checking every location for the classifier.
- **Evaluating results**: depending on how accurate the detection will be on images from our validation images set, we can fine-tune the cascade classifier by modifying *scale factor* or *minimum neighbors* parameters.
- **Collection of detected faces' bounding boxes**: cascade classifier should provide us boundaries as x and y coordinates of corners of rectangle over each detected face.

2. Face classification

Fig. 4: Face classification diagram

We will leverage our trained model which has an easy API to run the prediction process. We only have to resize the input image, which is a cropped part of the original image (part on which the detector previously found face).

Mask detection model training

1. Testing images selection

Classifier will be taught in two classes: “has mask” and “no mask”, so we need to gather a dataset containing information about two classes of images. We also need a set for verification of our classifier accuracy. We have chosen *kaggle* platform as our source of example dataset:

- **Face masks**: https://www.kaggle.com/andrewmvd/face-mask-detection

Changes comparing to prototype stage:

- *We’ve included only two classes (previously we also included masks weared incorrectly), making the problem binary,*
- *To increase dataset size, we decided to do data augmentation on original dataset,*
- *We’ve decided to use only one dataset which we will divide for several separated parts: for training, validation (separated with some validation split value on training phase) and test (might be original or augmented).*

2. Training algorithm

   Algorithm is as following:

   - **Data augmentation**: for a more diverse dataset and for better results, we tried *data augmentation*. Augmentation can include image preprocessing operations like rotation or scaling.
   - **Images resizing**: for compatibility with our initial model (min 32x32), we have to resize images to size of 32x32.
   - **[only for ResNet50 model] Freeze weights**: freeze weights of pretrained bottom layers.
   - **[only for ResNet50 model] Add top layers**: use additional global average pooling and dense layers for training of the network.
   - **Configure training**: choose optimal number of epochs, choose learning rate for optimizer.
   - **Start training**: compile and train the classifier.
   - **Enable model checkpoint/early stopping**: after each epoch without validation loss value improvement, bring back weights from the last better epoch. Stop training after some epochs in a row without improvement.
   - **Save classifier.**

Changes comparing to prototype stage:

- *We've set early stopping epochs to 4 instead of 3.*
- *We were not freezing layers for our simplified network.*

Face detection

The face detector is a module used for real-life testing of an already trained classifier. Prepared Python script enables easy-to-use usage while keeping its algorithm very simple and robust. What must be indicated, it is prepared for the usage of solely JPG images. The flowchart of the program can be represented in such a way:

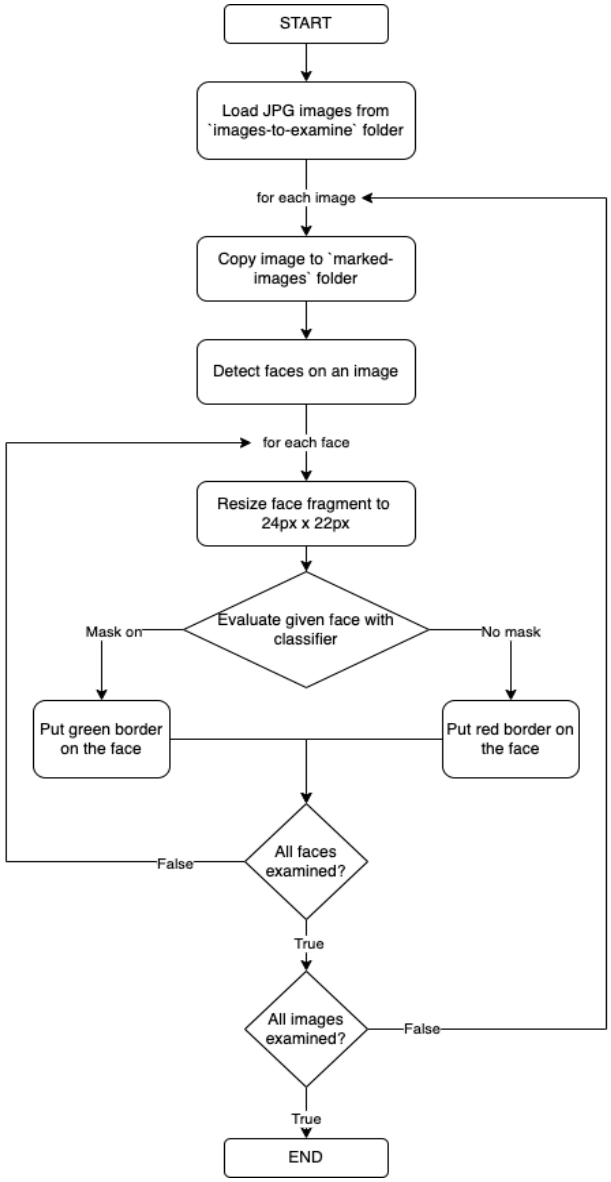

Fig. 5: Face detector flowchart

OpenCV Python library proved to be a significantly useful tool for testing out already trained classifiers with a set of real-life images. However, one problem is present and persists unsolvable. There is one assumption that we took: the detector shall not detect anything other than a face. Despite numerous tries in calibrating the detector input arguments (*scale factor* and *minimum neighbors*), we could not ensure that every masked and unmasked face is consistently detected with regard to the assumption. The detector finds the vast majority of them, especially for unmasked faces, yet for masked ones we can ensure about 80% efficiency.

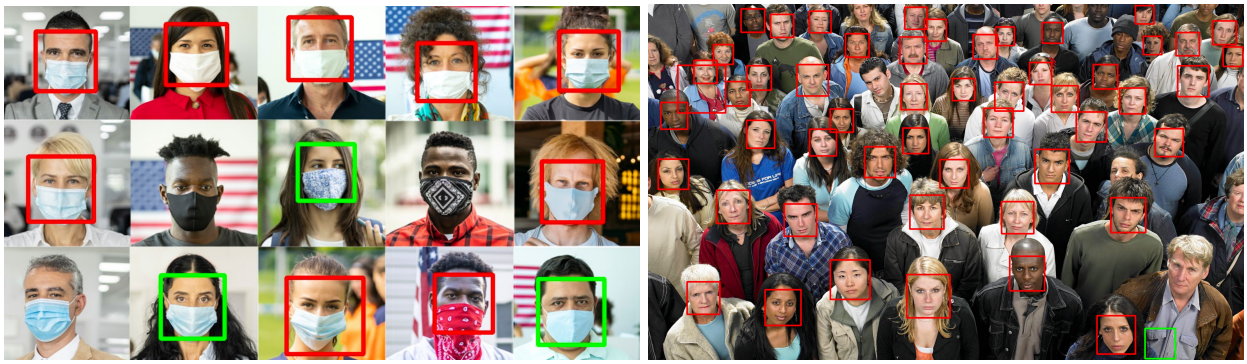

Fig. 6: Comparison of masked and not masked faces

The possible explanation for such a behavior is that we want to find faces belonging to two groups: masked and unmasked. If we focused on finding just one group of faces, we could achieve different, much more reliable results. That could be tested with preparing two separate face detectors, each calibrated for masked or unmasked faces, however that would require a too complex implementation, checking whether the `face-boxes` overlay each other etc. The current state of the detector is satisfactory and allows us to test the trained classifier.

Data augmentation

Data augmentation module is responsible for enlarging the datasets we have used for training the model. The enlargement process is simple - take an image, and modify it in three ways:

- Brightening
- Flipping horizontally
- Combination of both

During the preparations for the model training we saw a drawback. Our initial dataset had a much larger number of images belonging to a class `with_mask` compared to `withou_mask`. In order to balance the quantities of graphics belonging to different classes we had to add functionality of modifying only a chosen group of files without manual searching through thousands of images to our program. Hereby, our Python script can be used in two ways:

- Test mode - to start testing, put some images into the `test-images` folder. After that, run the script. Modified versions of the graphics can be now found in the `modified-images` folder.

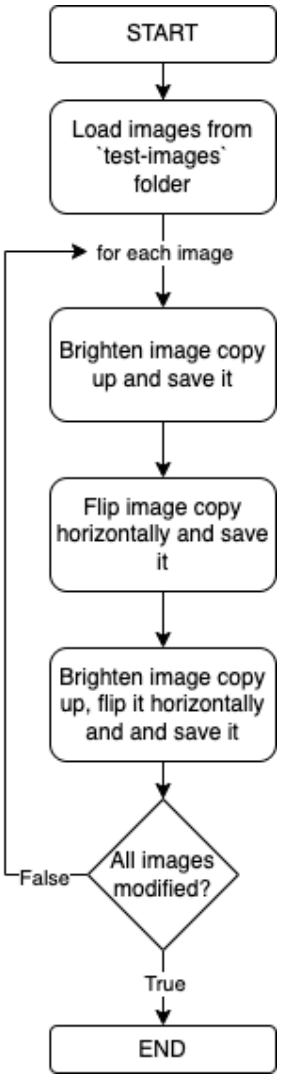

Fig. 7: Test mode flowchart

- Real application mode - this mode accepts the whole dataset of images and an CSV file describing the images. It filters the required class images, creates their modified copies and generates a new CSV file containing the data of the new set.

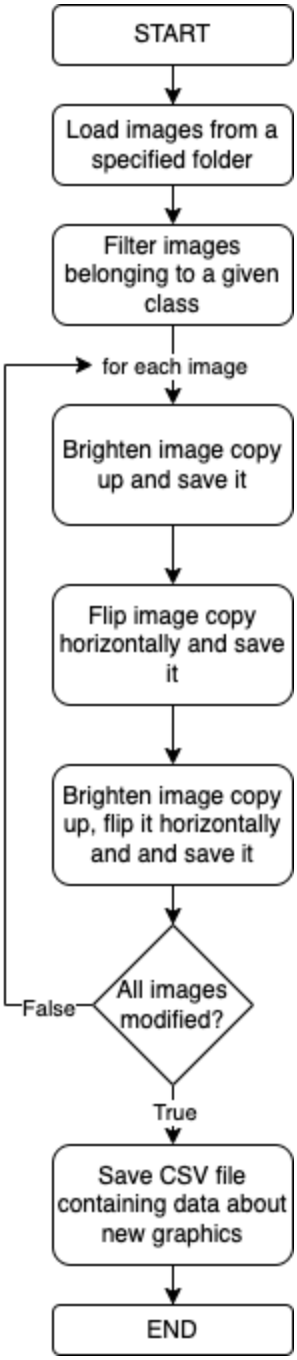

Fig. 8: Real application mode flowchart

The data augmentation module turned out to be very useful and very intuitive to prepare. That is how the resulting graphics look like:

Fig. 9: Modified images (original on the left)

RESULTS EVALUATION PART

After we initially didn't succeed with training ResNet50 to better results, we decided to go with our own model. After a lot of experiments involving: changing learning rate, changing optimizer, number of epochs, batch size, dropout layer rate we went with not the best results we could have, but we couldn't find a better one despite lots of experiments. In this chapter, we present the results of our experiments.

1. ResNet50 model

Parameters: pretrained ResNet50 model, learning rate: 0.01, optimizer: Adam, epochs: 24, batch size: 32, validation split: 0.25.

Part of training output (last three epochs):

```
…
Epoch 16/18
12/12 [==============================] - ETA: 0s - loss: 0.5302 - accuracy: 0.7322
Epoch 00016: val_loss did not improve from 0.48077
12/12 [==============================] - 5s 426ms/step - loss: 0.5302 - accuracy: 0.7322 -
val_loss: 0.5060 - val_accuracy: 0.7657
Epoch 17/18
12/12 [==============================] - ETA: 0s - loss: 0.4986 - accuracy: 0.7545
Epoch 00017: val_loss improved from 0.48077 to 0.46462, saving model to
drive/MyDrive/models/classifier-2.h5
12/12 [==============================] - 6s 544ms/step - loss: 0.4986 - accuracy: 0.7545 -
val_loss: 0.4646 - val_accuracy: 0.7950
Epoch 18/18
12/12 [==============================] - ETA: 0s - loss: 0.5336 - accuracy: 0.7545
Epoch 00018: val_loss improved from 0.46462 to 0.46301, saving model to
drive/MyDrive/models/classifier-2.h5
12/12 [==============================] - 6s 482ms/step - loss: 0.5336 - accuracy: 0.7545 -
val_loss: 0.4630 - val_accuracy: 0.7866
```

Accuracy achieved on the validation set was about 79%.

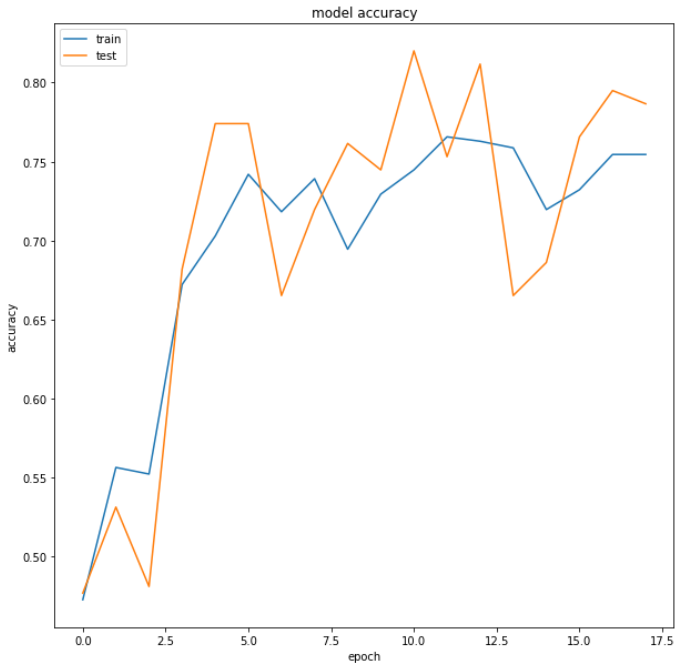

Fig. 10: Model accuracy - simplified model

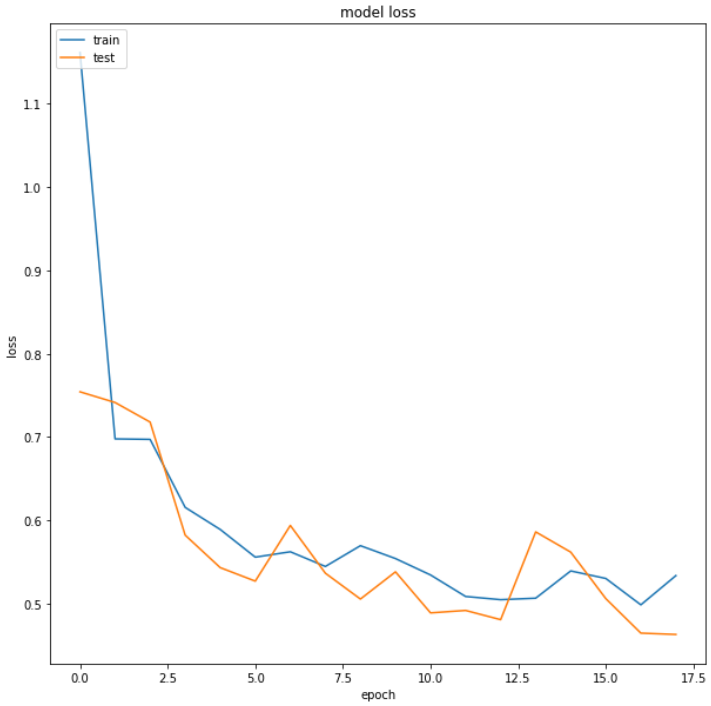

Fig. 11: Model loss - ResNet50 model

Loss is being quite unstable here, in this experiment we also tried to achieve the most stable state as we could, but it was not perfect.

When it came to prediction results, accuracy on the testing dataset was poor (about 51%), so we aimed to have at least dominant true positive values diagonal.

Fig. 12: ResNet50 model - confusion matrix

Chosen confusions matrices for some of our other experiments with simplified model are shown below. We were looking for luck in learning rate range of 0.1 - 0.0001.

Fig. 13: ResNet50, rate 0.0001

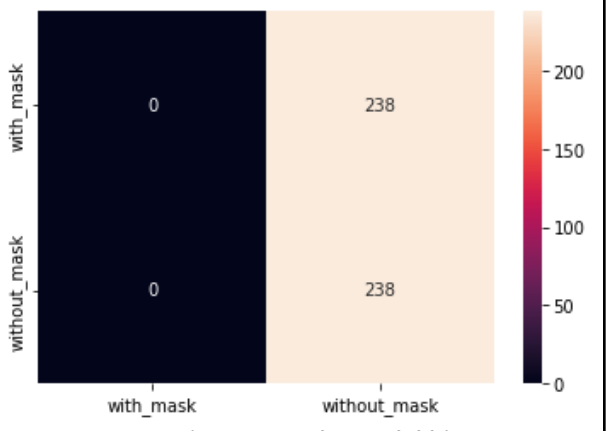

Fig. 14: ResNet50, rate 0.001

Fig. 15: ResNet50, rate 0.01

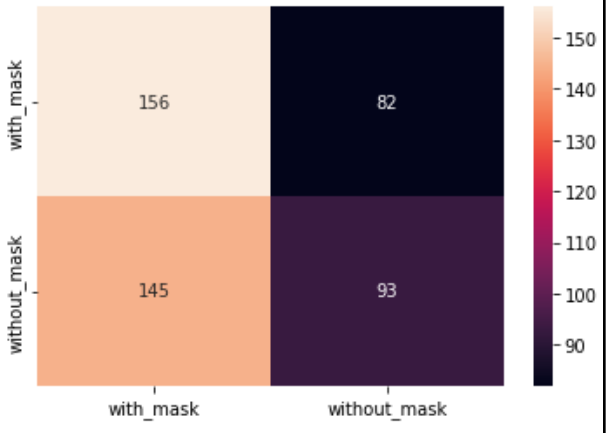

Fig. 16: ResNet50, rate 0.05

2. Simplified model

Parameters: *Custom model, learning rate: 0.001, optimizer: Adam, max epochs: 18, batch size: 64, validation set split: 0.25.*

Part of training output (last two epochs):

```
...
Epoch 17/18
12/12 [==============================] - ETA: 0s - loss: 1.1216 - accuracy: 0.8759
Epoch 00017: val_loss improved from 0.34303 to 0.33116, saving model to
```

```
drive/MyDrive/models/classifier-2.h5
12/12 [==============================] - 2s 205ms/step - loss: 1.1216 - accuracy: 0.8759 -
val_loss: 0.3312 - val_accuracy: 0.9331
Epoch 18/18
12/12 [==============================] - ETA: 0s - loss: 0.8900 - accuracy: 0.8954
Epoch 00018: val_loss did not improve from 0.33116
12/12 [==============================] - 2s 192ms/step - loss: 0.8900 - accuracy: 0.8954 -
val_loss: 0.3339 - val_accuracy: 0.9331
```

Accuracy achieved on the validation set was better than what we got on ResNet50 and 93% is a good result.

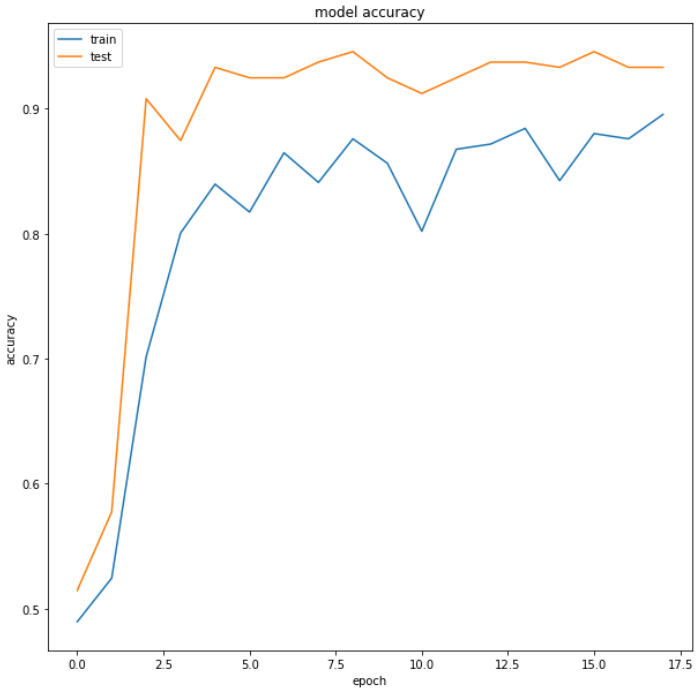

Fig. 17: Model accuracy - simplified model

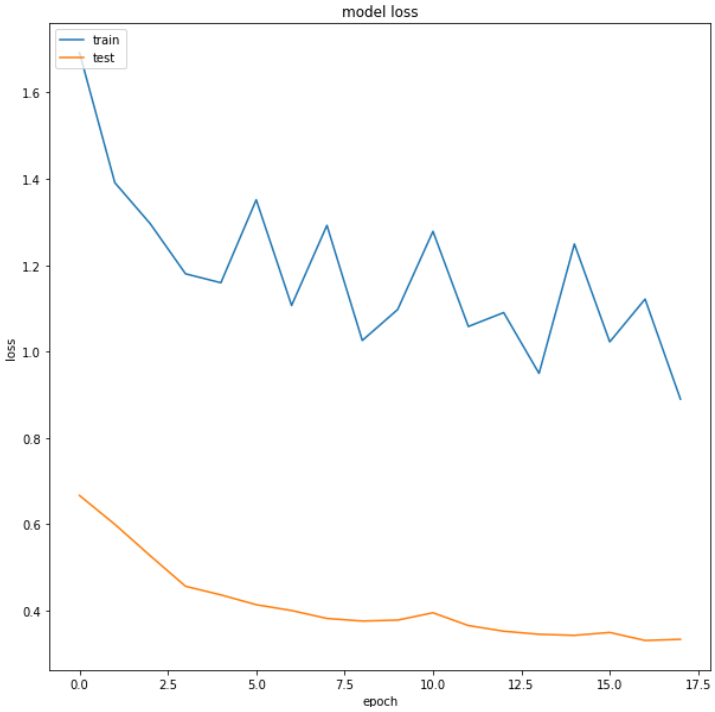

Fig. 18: Model loss - simplified model

Loss evolution during training process is being quite unstable here. In this experiment we tried to achieve the most stable state as we could, but it was not perfect.

Fig. 19: Simplified model - confusion matrix

When it came to prediction results, we were hoping for better accuracy, but instead we only achieved ~53%, which was disappointing (at least the diagonal of the matrix is somehow dominant). That's why we tried out some other parameter configurations of the same network to improve this, unfortunately with no better result.

Chosen confusions matrices for some of our other experiments with simplified model are shown below. We were looking in range 0.1 - 0.001. We came to the conclusion that rates below 0.001 were not giving good effects and above 0.1 were too high for this network.

Fig. 20: Simplified, rate 0.01, batch 64

Fig. 21: Simplified, rate 0.01, batch 128

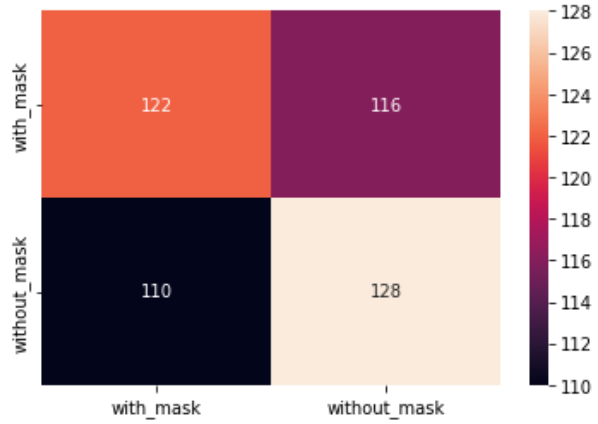

Fig. 22: Simplified, rate 0.005, batch 64

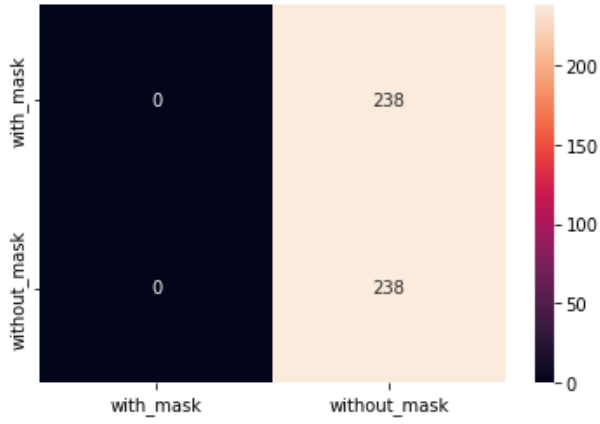

Fig. 23: Simplified, rate 0.1, batch 64

INSTRUCTION

Running face detector script:

1. Put the JPG files you want to examine into the *detector/images-to-examine* folder
2. Put the trained classifier file into the *detector* and make sure its name is appropriate for the *CLASSIFIER_PATH* constant in *face-detector.py* (line 10)
3. In the command line, navigate to the *detector* directory
4. Execute command *python3 face-detector.py*
5. Examined images will appear in *detector/marked-images*

Running data augmentation (test mode):

1. Put the JPG files you want to modify into the *data-augmentation/test-images* folder
2. In the command line, navigate to the *data-augmentation* directory
3. Execute command *python3 data-augmentation.py*
4. Modified images will appear in *data-augmentation/modified-images*

Running data augmentation (real application mode):

1. Put the files in a directory (name is to choose) and put this folder into *data-augmentation*
2. Provide the CSV file describing the data in *data-augmentation* folder
3. In *data-augmentation.py* change:
    a. *IMAGES_TO_MODIFY_FOLDER_PATH* (line 11) to the name of your folder
    b. *CSV_TO_READ* (line 12) to the name of your CSV file
    c. *CLASS_TO_AUGMENT* (line 13) to the value of class property you want to augment

   d. In line 87, set the proper index of the column where the class label is specified
   e. Uncomment line 116 and comment 115
4. In the command line, navigate to the *data-augmentation* directory
5. Execute command *python3 data-augmentation.py*
6. Modified images will appear in *data-augmentation/modified-images* along with generated CSV file in *data-augmentation*

CONCLUSIONS

- We identified that after some initial training on high learning rates (0.6 for example) gave good validation dataset accuracy on ResNet50, but resulted in poor one class prediction, which was an obvious error.
- We identified that training batch size can have an impact on prediction, the optimal one for us was 64 (32, 64 and 128 were tested).
- Although we got good validation dataset accuracy, we were surprised by poor prediction results.
- We spent a lot of time trying to improve prediction accuracy, but despite the efforts, the best test dataset accuracy we got was about 53%.
- We got poor training results, which may be caused by dataset selection (or the way we processed it). However, we made sure we correctly divided training, validation and testing datasets. Also, the training dataset might have been too small for our problem.
- Finally, we didn't use our augmented dataset as we were suspecting the result images of being a problem somehow.
- We used OpenCV to preprocess images, as we noticed after some errors, we had to make sure we used the correct color convention, as OpenCV uses BGR instead of RGB.