Face Mask Detector

Final Report

Sheldon Sebastian

Introduction

The goal of our project is to identify whether a person is **correctly** wearing a mask or not. A person correctly wears a mask when the mask completely covers his mouth and nose.

COVID-19, as we know, is a pandemic that has claimed millions of lives in the year 2020. Wearing a face mask has been identified as a successful method of preventing the spread of COVID amongst people. It is strongly recommended to wear a mask in public places. Most people follow the guidelines and wear masks. Some people do not wear it while others wear it incorrectly which doesn't cover their nose/mouth as it should.

Our project aims to train a model on images of people wearing masks and develop an interface to identify faces of people wearing the mask correctly, wearing it incorrectly or not wearing a mask.

This individual report contains the scripts to download data from One Drive folder for MaskedFace-Net, and to download data from Google Drive Folder for Flicker Face Dataset. Once the data is downloaded, the logic of splitting the data into train-test-validation and organizing the folder for ImageFolder API is described in the report. Finally, it describes how to perform occlusion experiment to interpret the CNN model.

Dataset

For our project we used the MaskedFace-Net dataset[1]. This dataset is a synthetic dataset created using Generative Adversarial Network (GAN) on the Flickr-Faces-HQ Dataset[2]. The MaskedFace-Net model created synthetic images of people wearing the mask correctly and incorrectly. For our project we also wanted to identify whether the person was wearing a mask or not. So, we added the original Flickr-Faces-HQ dataset images of people not wearing a mask to achieve this task.

The data was downloaded using CURL command and the python scripts are available in **the Data Download Scripts Folder of the Github repository**. The final combined dataset contains 60,000 images and is 15 GB in size.

Of the 60,000 images 20,000 images were of incorrect worn masks, 20,000 images were of correct worn masks and 20,000 images were of uncovered faces.

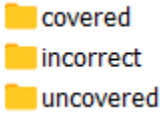
80% of the dataset was used for training and 20% was used as holdout or test set. The script to split into train-validation and holdout is found in **DataPreprocessing.py**.

```
# %% --------------------
# create 3 folders in specified path
# train
os.makedirs(BASE_DIR + "root_data/train/uncovered", exist_ok=True)
os.makedirs(BASE_DIR + "root_data/train/covered", exist_ok=True)
os.makedirs(BASE_DIR + "root_data/train/incorrect", exist_ok=True)

# val
os.makedirs(BASE_DIR + "root_data/validation/uncovered", exist_ok=True)
os.makedirs(BASE_DIR + "root_data/validation/covered", exist_ok=True)
os.makedirs(BASE_DIR + "root_data/validation/incorrect", exist_ok=True)

# holdout
os.makedirs(BASE_DIR + "root_data/holdout/uncovered", exist_ok=True)
os.makedirs(BASE_DIR + "root_data/holdout/covered", exist_ok=True)
os.makedirs(BASE_DIR + "root_data/holdout/incorrect", exist_ok=True)
```

```
# take random 80% indices from the 80% of original indices and cp to train
random.seed(seed)
# random.sample(..) gets elements without replacement
train_imgs = random.sample(arr, train_size)

# set difference to get 20 % of indices and cp those index files into holdout folder
holdout_imgs = set(arr) - set(train_imgs)
holdout_imgs = list(holdout_imgs)

# take 20 % of images from train indices
random.seed(seed)
# random.sample(..) gets elements without replacement
val_imgs = random.sample(train_imgs, val_size)

# subtract val_imgs from train_imgs
train_imgs = list(set(train_imgs) - set(val_imgs))

# copy all the files in the indexes to appropriate folder
```

The data was organized such that it was accessible using the ImageFolder API of Pytorch.

And inside each folder of holdout(test), train, and validation we have following folders:



The dataset contains the following 3 image labels:

covered

incorrect

uncovered

For the Data Download Scripts in CURL and Preprocessing all the code was written by me and no code was copied from the internet.

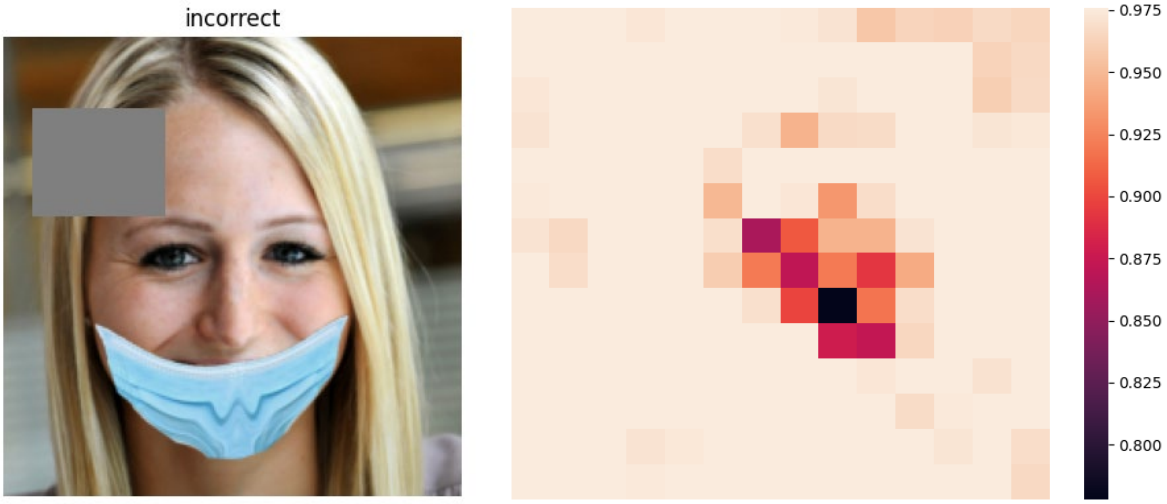
Model Interpretation

Once the model was trained with strong validation accuracy and low validation loss, we interpreted the CNN model on the training dataset using Occlusion experiment.

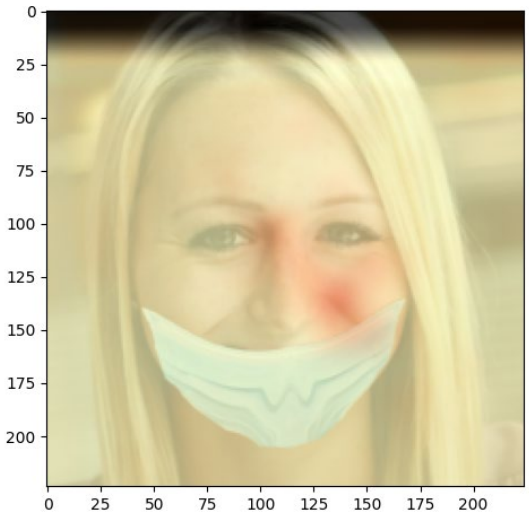
Occlusion experiments are performed to determine which patches of the image contribute maximally to the output of the neural network.

In the occlusion experiment, we iterate over all the regions of the image systematically by occluding(blocking) a part of the image with a grey patch and monitor the probability of the classifier using a heatmap.

```
# custom function to conduct occlusion experiments
def occlusion(model, image, label, occ_size=50, occ_stride=50, occ_pixel=0.5):
    # get the width and height of the image
    width, height = image.shape[-2], image.shape[-1]

    # setting the output image width and height
    output_height = int(np.ceil((height - occ_size) / occ_stride))
    output_width = int(np.ceil((width - occ_size) / occ_stride))

    # create a white image of sizes we defined
    heatmap = torch.zeros((output_height, output_width))

    # iterate all the pixels in each column
    for h in range(0, height):
        for w in range(0, width):

            h_start = h * occ_stride
            w_start = w * occ_stride
            h_end = min(height, h_start + occ_size)
            w_end = min(width, w_start + occ_size)

            if (w_end) >= width or (h_end) >= height:
                continue

            input_image = image.clone().detach()

            # replacing all the pixel information in the image with occ_pixel(grey) in the specified location
            input_image[:, :, w_start:w_end, h_start:h_end] = occ_pixel

            # run inference on modified image
            output = model(input_image)
            output = nn.functional.softmax(output, dim=1)
            prob = output.tolist()[0][label]

            # setting the heatmap location to probability value
            heatmap[h, w] = prob

    return heatmap
```

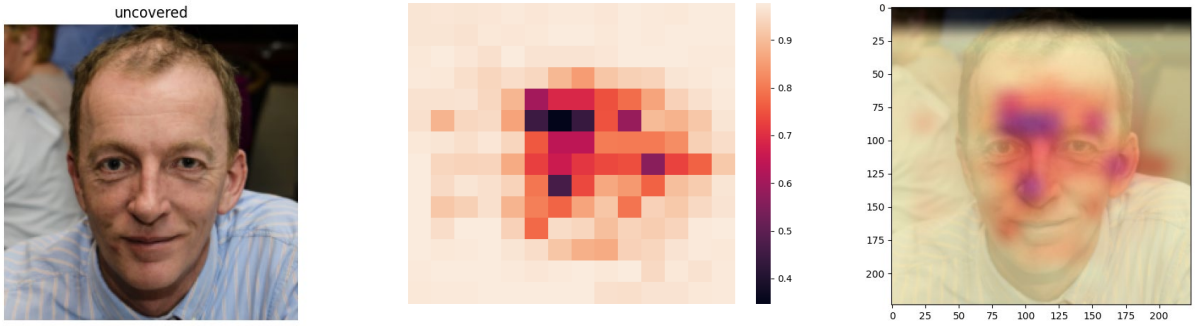


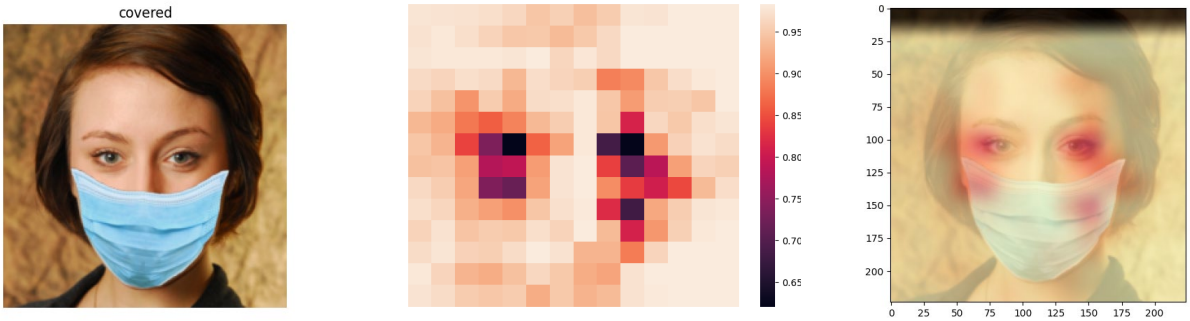
After blending the probability heatmap and the original image we get the following output:



We notice that we get low probability for the classifier when the patches of images containing nose or upper mouth are occluded. Thus, the occlusion experiment tells us that the CNN model is learning relevant patterns in the image.

For other image labels the occlusion experiment results are as follows:





We can summarize that the model is looking at relevant features in the image to make classification predictions. For the occlusion experiment and viewing the blended images which is present in **OcclusionExperiment.py**, about 50% of the code was copied from the internet[3].

Summary and Conclusion

Thus, the dataset was downloaded, split into train, test and validation split and was then used in the training of the model. After the model was trained to a satisfactory level of validation accuracy and loss, the model was interpreted using the occlusion experiment.

References

1. Adnane Cabani, Karim Hammoudi, Halim Benhabiles, and Mahmoud Melkemi, "MaskedFace-Net - A dataset of correctly/incorrectly masked face images in the context of COVID-19", Smart Health, ISSN 2352-6483, Elsevier, 2020

2. A Style-Based Generator Architecture for Generative Adversarial Networks Tero Karras (NVIDIA), Samuli Laine (NVIDIA), Timo Aila (NVIDIA)

3. Kumar, N. (2019, December 17). Visualizing Convolution Neural Networks using Pytorch. Retrieved December 07, 2020, from https://towardsdatascience.com/visualizing-convolution-neural-networks-using-pytorch-3dfa8443e74e