Computer Vision Assignment #4:

Facial Manipulation Detection

Yuval Katz 203678164

Yonatan Shafir 308570290

Chapter 1: Introduction:

1.1 – 1.8. Read and done.

Chapter 2: Build Faces Dataset :

1. We implemented the two methods: __getitem__ and __len__, both are attached at the code under face_dataset.py

   For the __getitem__ implementation, we decided that the real images are located at the first indexes and the fake are the the last.

2. Running plot_samples_of_faces_dataset.py, we derives the next plot:

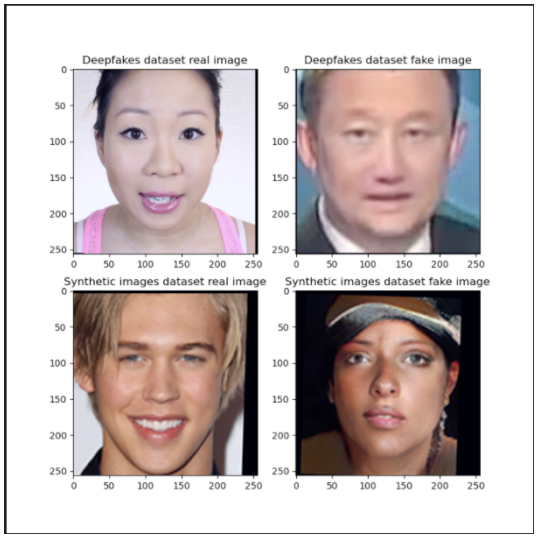

Figure 1: plot_samples_of_faces_dataset output

Chapter 3: Write an Abstract Trainer:

Our architecture is based on the following figure:

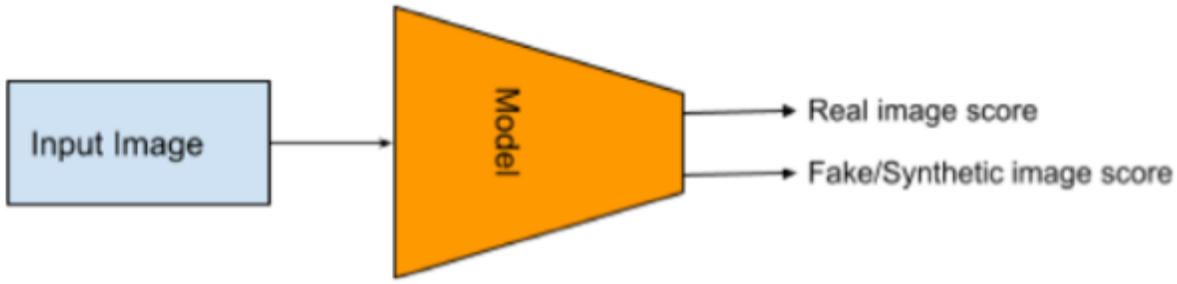

Figure 2:For each input image, the model outputs two scores: score for how real the image is and how Fake / Synthetic the image is. The classification is done by taking the argmax of the two scores.

3.1 Implement an Abstract Trainer

3. We implemented a function that trains the model for a single epoch, the function is placed under trainer.py
   The method that was implemented included a loop over all batches of the data loader.
   For each batch we did the following:
   1. Zero the gradients.
   2. Compute a forward pass.
   3. Compute the loss w.r.t to the criterion.
   4. Compute the back propagation.
   5. Step optimizer.
   6. Update the average loss and accuracy.

   The average loss and accuracy were calculated by the following form:

$$\text{Average loss} = \frac{\text{Accumulated loss}}{\text{Batch index} + 1} \quad where\ 0 \leq batch\ index \leq \#of\ batches - 1$$

$$Accuracy = 100 * \frac{accumulated\ correct\ labled\ samples}{accumulated\ number\ of\ samples}$$

4. We implemented a function that evaluates the model on a given dataset, the function is placed under trainer.py
   The method that was implemented included a loop over all batches of the data loader.
   For each batch we did the following:
   1. Compute forward pass under a torch.no_grad() context manager.
   2. Compute the loss w.r.t to the criterion.
   3. Update the average loss and accuracy.

3.2 Train a Deepfake Detection Classifier

5. We run the main training script: train main.py to train the SimpleNet architecture on the Deepfakes dataset. We used a learning rate of 1e − 3, batch size: 32, 5 epochs and the Adam optimizer as described in the document.

6. Using an IDE for opening the json created in the out directory as a result of running the script, we can see there a sort of a log file that collects all the data (accuracy and loss) from training, evaluating and testing our model. It does make a lot of sense; with that we can later plot our graphs.
7. We run the plot_accuracy_and_loss.py script to visualize the data held in the json.

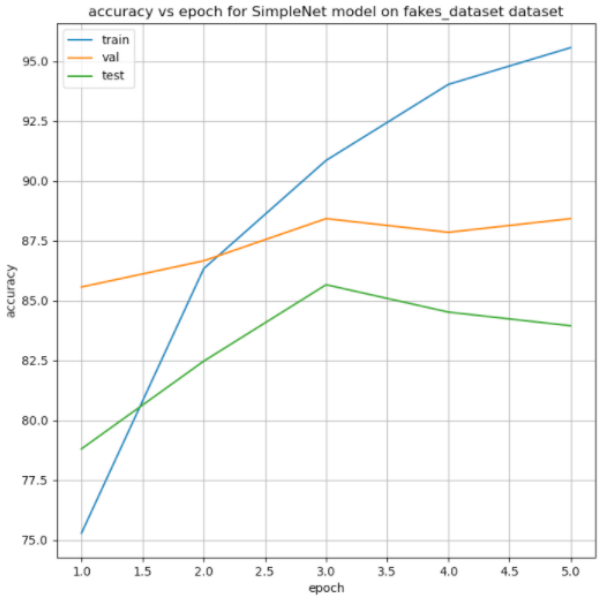

Figure 4: fakes_dataset_SimpleNet_accuracies_plot

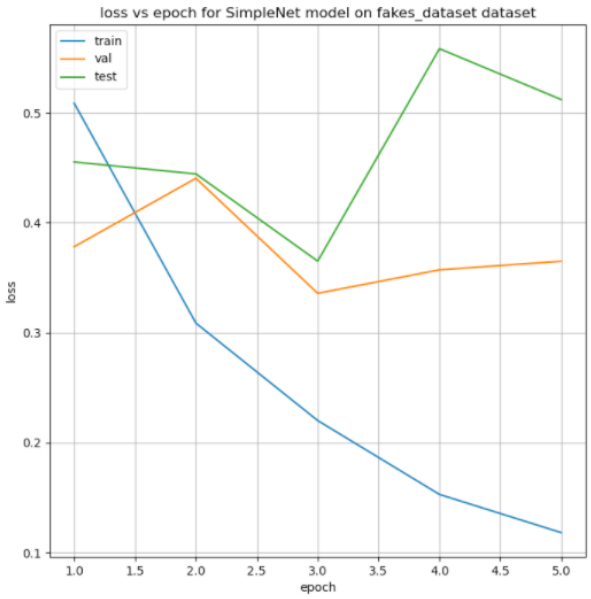

Figure 4:fakes_dataset_SimpleNet_losses_plot

Both in figure 3 and 4 we can notice an overfit. In figure 3 we can notice in a very noticeable way that as we continue train and lower the loss for the train dataset, we can see a growth in the loss for both the validation set and the test set.
Same thing can be seen in the accuracy graph, although we keep on getting better and the train accuracy is growing, we can see that for the validation and test datasets, we achieved the maximum accuracy, and we start on being less accurate as we keep training. This is what can be explained by overfitting to the training dataset.
In statistics, overfitting is "the production of an analysis that corresponds too closely or exactly to a particular set of data, and may therefore fail to fit additional data or predict future observations reliably". (Wikipedia).

8. The test accuracy corresponding to the highest validation accuracy we received is: 86.7 in test vs 90.2 in the validation set.
9. The proportion of fake images to real images in the test set is:
   700 fake images vs 1400 real images, thus the proportion is 1:2.

```
len(test_dataset.real_image_names)
1400
len(test_dataset.fake_image_names)
700
```

Figure 5: fake images vs real images in the test dataset

10.

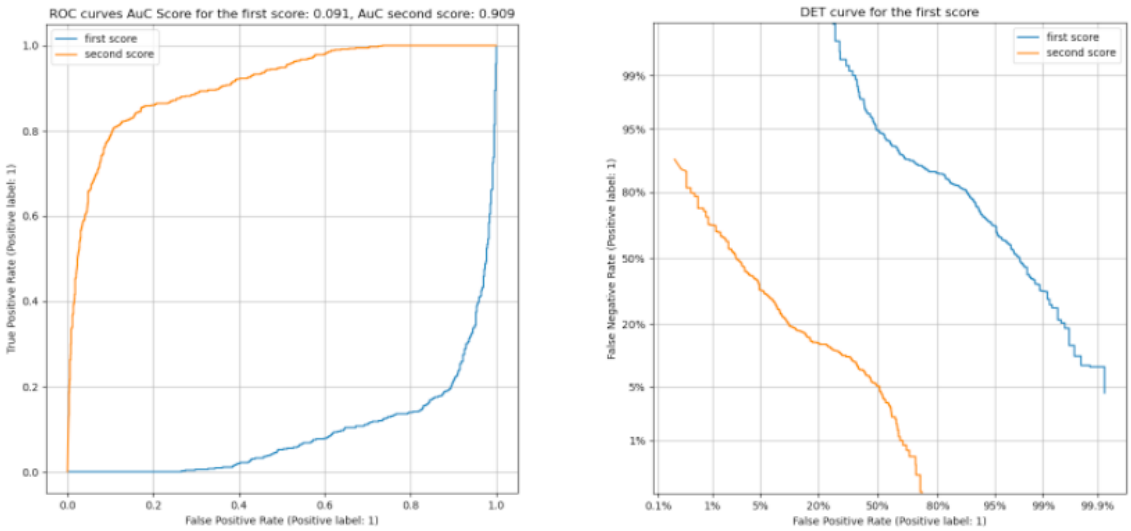

Figure 6: fake dataset roc and det curves

11. ROC – Receiver Operating Characteristic, is a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied.
    DET – Detection Error Tradeoff, is an alternative to the ROC curve which plots the false negative rate (missed detections) vs. the false positive rate (false alarms) on non-linearly transformed x- and y-axes.
    In ROC graph, what we see and does make sense, is two graphs(yellow and blue) of a binary classifier, the reason that we see two graphs is that usually a binary classifier assuming one output, where the label is supposed to be also binary (1 for true and 0 for false), in our case we have 2 binary outputs, where for each of the two we get 1 or 0 if the sample is real / fake in each neuron. Therefore, we see two mirrored graphs. In our case we can look only at the yellow graph.

Same can be explained for the DET graph, what we see are two mirrored graphs due to the structure of the output (2 binary neurons).

12. We used train_main.py to train SimpleNet on the synthetic data.
13. We used the plot_accuracy_and_loss.py script to visualize the data held in the json.

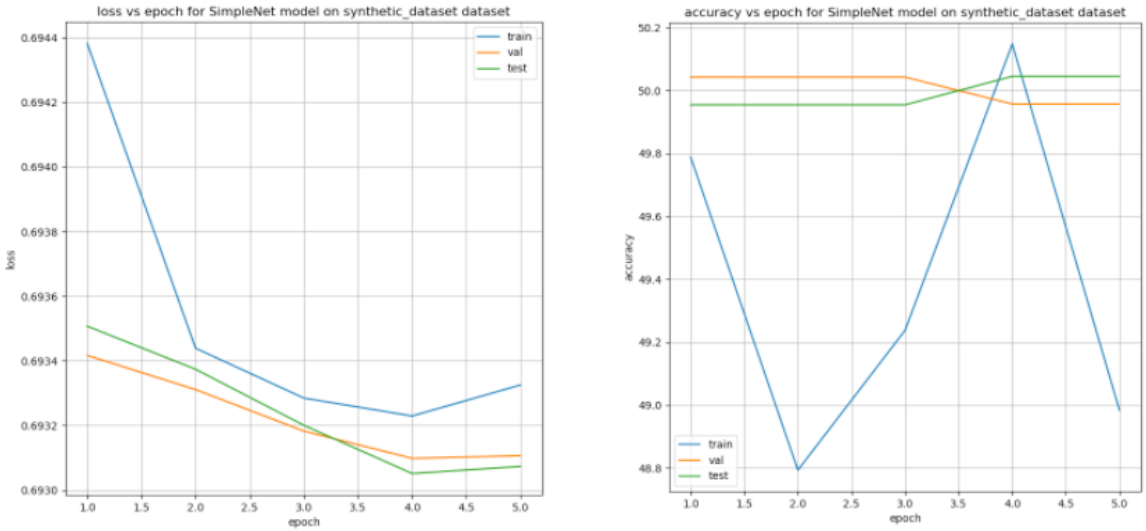

Figure 7:synthetic_dataset_SimpleNet accuracies_plot and loss plot

We can notice for this plot, that training SimpleNet on the synthetic data isn't efficient. Both test and validation accuracies are around the 50% which is more or like guessing. We can also notice that on the training dataset we don't get any good results.

14. The test accuracy corresponding to the highest validation accuracy we received is: 50 in test vs 50 in the validation set.
15. In the synthetic test set we have: Fake images – 552, Real images – 551. Meaning about 1:1 proportion.
16. We get a random classifier, which is guessing with 50% for each (real or fake).
17. Looking at samples from the synthetic Images dataset and the Deepfake dataset the result does make sense; it can be clearly visualized that the synthetic fake samples are much more visually realistic than those we have in the Deepfake dataset. Due to that it makes it much easier to classify fake images from the Deepfake dataset

Chapter 4: Fine Tuning a Pre-trained Model:

4.1 Intro

18. As we can read in the Xception class descriptor – "Xception was optimized for ImageNet dataset".
    The ImageNet dataset contains more than 14 million images classified into 1000 non-overlapping classes.

19. The basic building block of Xception is the depthwise separable convolution block.
    The depthwise separable convolution block has 2 main parts:
    - Depthwise convolution – referred to as filtering stage, in this stage the depth of the layers does not change, and a kernel convolution is used for each layer separately.
    - Pointwise convolution – this stage is meant to downscale the number of layers to a single layer. A filter with kernel size 1X1XLayers is applied.

    This convolution block is a smart way of performing depth convolution so that the number of multiplications is reduced and so the model more efficient.

20. Same question/answer as 18.

21. The input feature dimension for the final classification block ("fc") is of size 2048.
    In the original Xception architecture the final classification is done using a fully connected layer of input size 2048 and output the number of classes:

```
self.fc = nn.Linear(2048, num_classes)
```

22. Using the "get_nof_params" function we can check the number of parameters in the model and after running the function on Xception we get: 22,855,952 parameters (as written in the paper under "parameter count").

4.2 Attaching a new Head to the Xception backbone

23. Under the "get_xception_based_model" API we attached the new Head to the Xception backbone (using "build xception backbone" function) with the following MLP architecture:

```
XceptionBased.fc = nn.Sequential(nn.Linear(2048, 1000),
                        nn.ReLU(),
                        nn.Linear(1000, 256),
                        nn.ReLU(),
                        nn.Linear(256, 64),
                        nn.ReLU(),
                        nn.Linear(64, 2))
```

24. The full Xception model with the new MLP head includes 23,128,786 parameters, Subtracting from the parameter count int Q22 - 272,834 new learnable parameters are obtained.

25. Trained the new Xception-based model on synthetic dataset using:
    - Adam optimizer
    - Learning rate of 0.01
    - 5 epochs
    - Batch size of 32.

26. In this section we used ”plot_accuracy_and_loss.py” script in order to get a graph visualization of the models accuracy and loss during training:

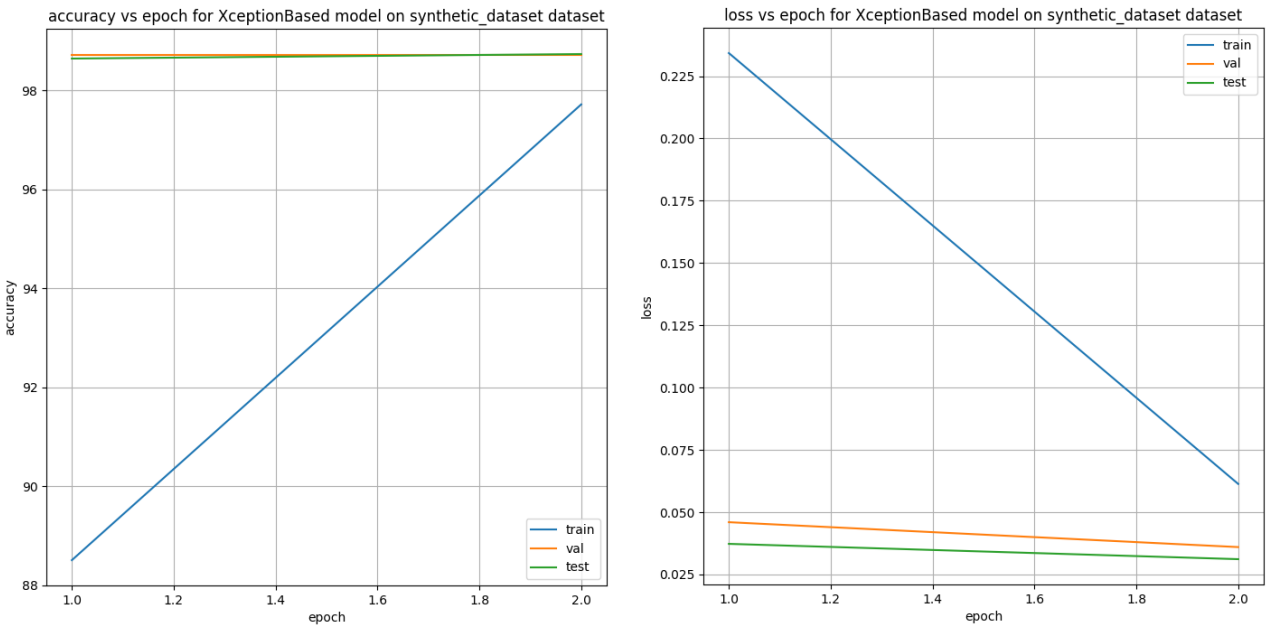

Figure 8: synthetic_dataset XceptionBased accuracies_plot and loss plot

27. As we can see in the accuracy graph in figure 8 and as written in the Json file the test accuracy corresponding to the highest validation accuracy we received is:
    - Validation accuracy – 98.71%
    - Test accuracy – 98.73%

28. In this section we used ”numerical analysis.py” script in order to get a graph visualization of the models results:

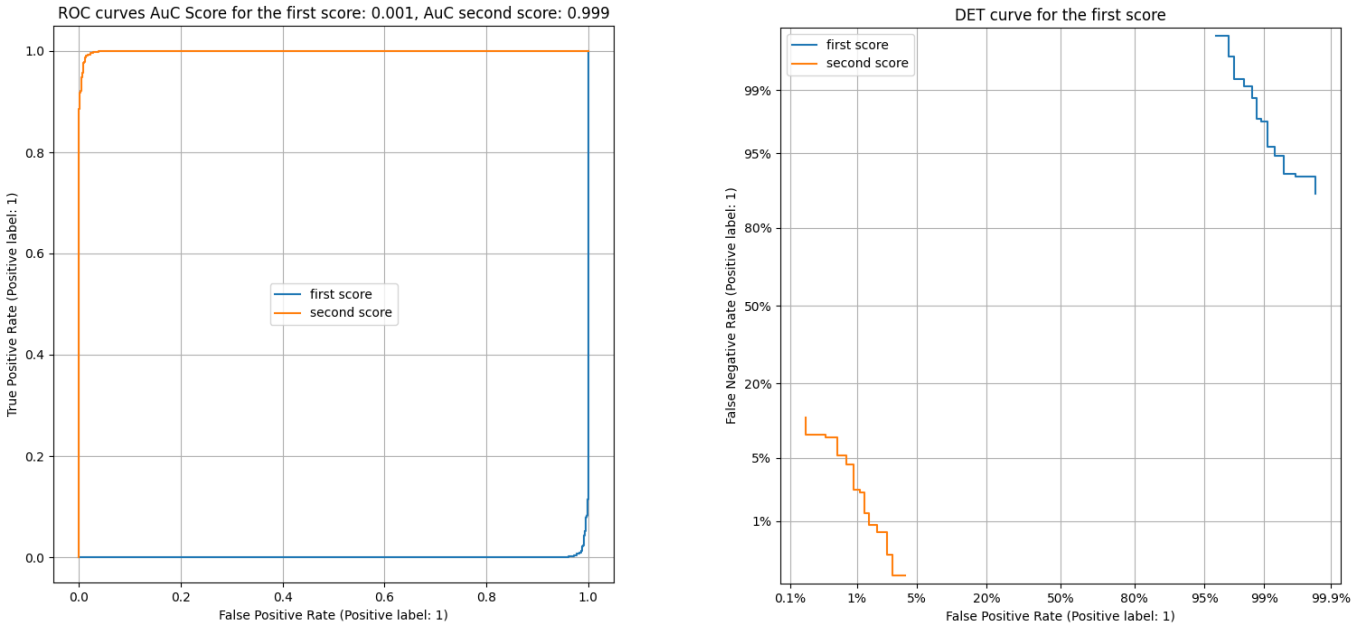

Figure 9: synthetic dataset XceptionBased ROC plot and DET plot

As we can see in the graphs the XceptionBased architecture with the MLP head gives very good results. For example, in the ROC graph’s Classifiers that give curves closer to the top-left corner (for TRUE value equals 1 – real images) indicate a better performance, as for fake images a classifier that is closer to the bottom- left indicate a better performance - in both cases we see this behavior in the XceptionBased ROC curve.

Chapter 5: Saliency Maps and Grad-CAM analysis

5.1 Intro

29. A saliency map is a method that gives us a visual understanding of how the models input contributes to its score output and to its classification. Given an image and its classification (I_0, c) and a ConvNet with class score function denoted as $S_c(I_0)$ as saliency map ranks every pixel in the image based on there influence on the final class score function.

    This is done by using a first order-Taylor approximation on the models score and computing the weight associated with each pixel:

    $$S_c(I) \cong w^T I + b \rightarrow w = \frac{dS_c}{dI}|_{I_0}$$

    Each wight can be calculated using the chain rule, and its magnitude corresponds with how much the pixel it multiplies affect the class score output.

30. Grad-Cam stands for – Gradient-weighted Class Activation Mapping and it is another method that can be used in order to visualize the contribution of each pixel to the output of a ConvNet classifiers decision.

    The methods work in the following manner for an input image and a specific class:

    - Identify the last CNN layer in a model.
    - Compute the gradients of the output score corresponding to the specific class for each of the wights in the final CNN denoted as $\{A_{ij}^k\}$, where k is the number of output layers in the final CNN - compute - $\frac{dy^c}{dA_{ij}^k}$.
    - Calculate to summation of all the gradients (normalized):

      $\alpha_k^c = \frac{1}{z}\sum_{ij}\frac{dy^c}{dA_{ij}^k}$, gives us the contribution of layer k to y^c.
    - Multiply each layer of the final CNN A^k by its assigned weight α_k^c and apply relu:

      $$L_{cam-grad}^c = Relu(\sum \alpha_k^c \cdot A^k)$$

- Increase $L^{c}_{cam-grad}$ resolution to image resolution.

The final result is a heat map assigning each pixel to its contribution on the final class score and to the final classification.

5.2 Saliency Maps

31. Implemented function “compute_gradient_saliency_maps” in file saliency maps.py

32. Using the saliency maps function implemented in Q31 we generated the saliency maps of the models XceptionBased and SimpleNet on the Synthetic faces datase, resulting in the following images.

    For the SimpleNet model:

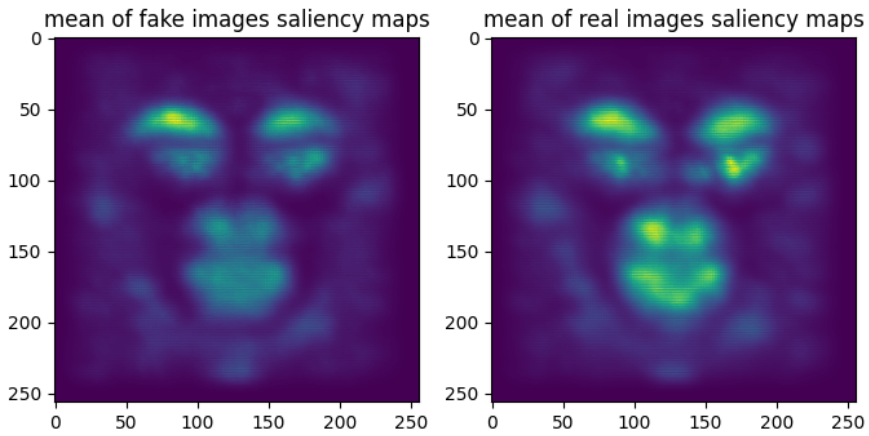

Figure 10: SimpleNet model Mean saliency maps real and fake

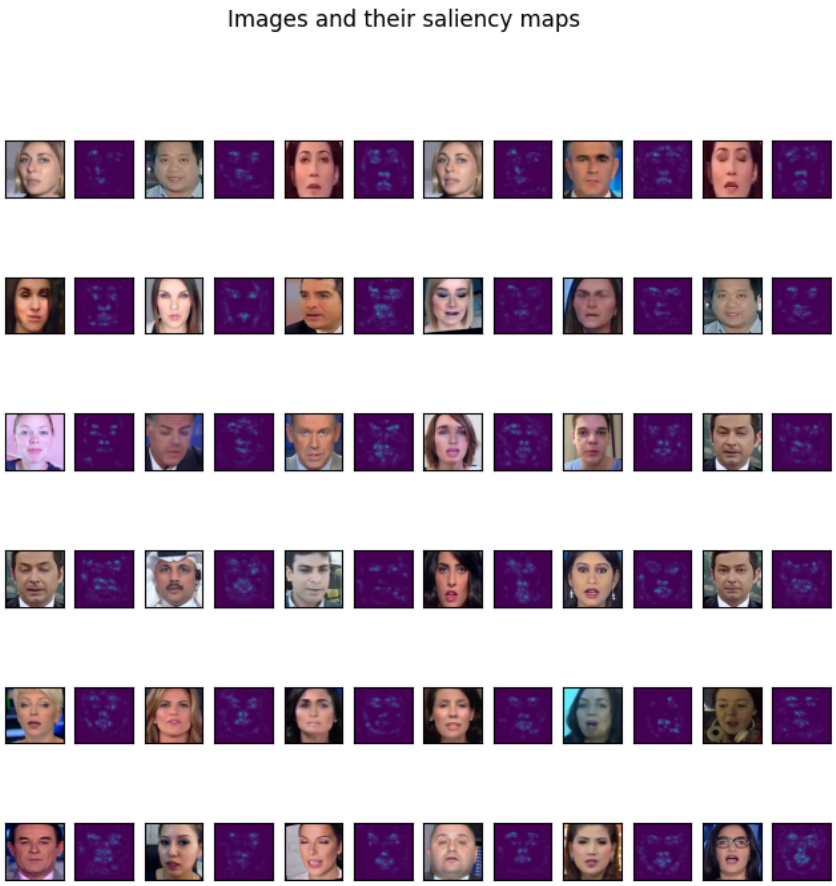

Figure 11: SimpleNet model images and their saliency maps

we can see in the saliency maps of the individual contributaion of each pixel in the image and the mean of saliency maps for real/fake images gives us a good understanding of what the model is looking at when deciding if an image is real or fake. We can see that the SimpleNet identifies specific features in the face such as eyebrows, eyes, mouth, nose, etc. another observation is that the model seems to concentrate on those key features mentioned mostly and not on the whole face – we can see that between key face features the pixels are black in the saliency map. This means the fake images aren't encouraged to try to emulate the real image in those areas.

Overall although the SimpleNet does seem to observe key features in the face, the overall resolution of the saliency map is low – this may indicate why the accuracy result isn't great.

For the XceptionBased model:

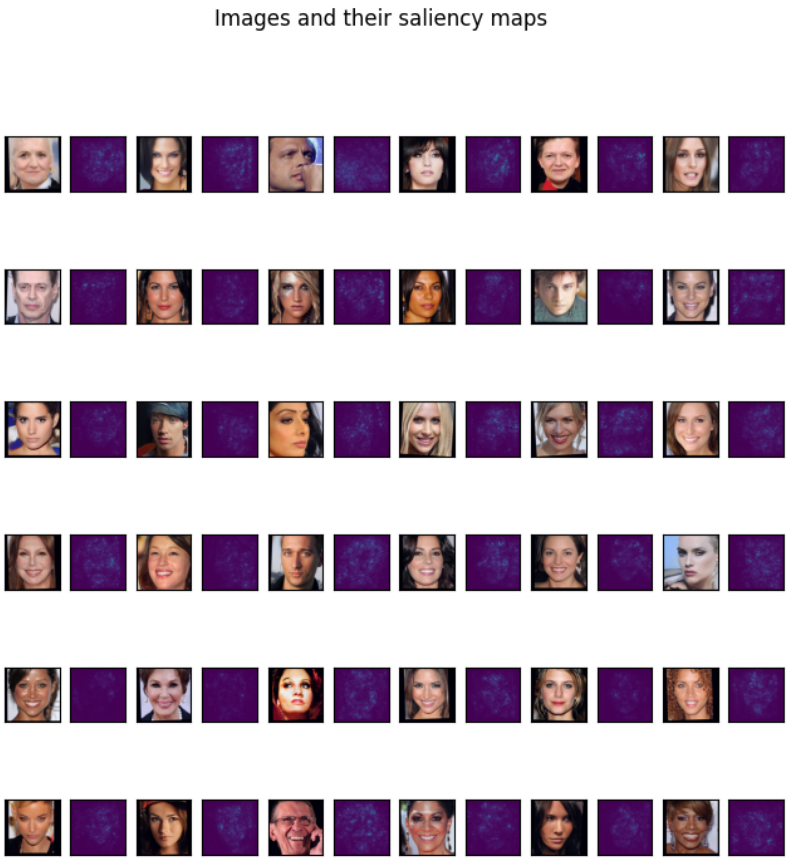

Figure 13: XceptionBased model Mean saliency maps real and fake

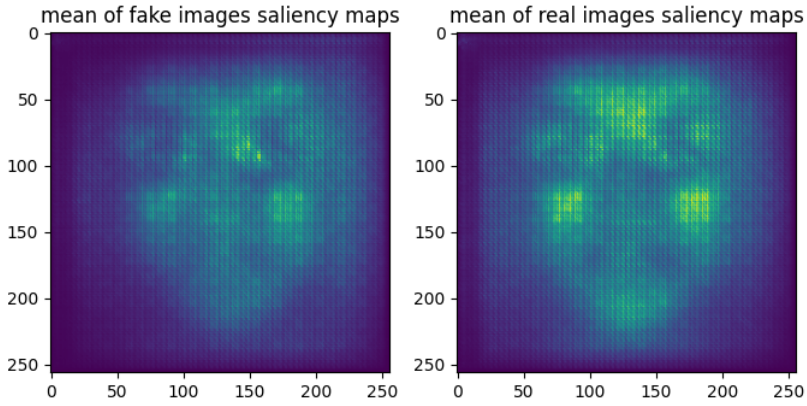

Figure 12: XceptionBased model images and their saliency maps

As we can see in the XceptionBased mean saliency maps the pixel that contribute to the models classification are mainly the pixel in the face area but the pixels around the face are also somewhat considered on the final result. Compared to the SimpleNet we can see that the saliency map is of greater resolution, which makes sense since the XceptionBased model results are much better.
Another observation is that compared to the SimpleNet results we can see that the model considers almost all the pixel in the face area and not only pixels that come from main face features (eyes, nose, mouth, etc.) this may be a reason why the XceptionBased model is harder to "fool" using fake images.

5.3 Grad-CAM

33. Installed grad-cam python package version 1.3.6 from git repository.

34. Implemented function "get_grad_cam_visualization" using the grad-cam package in file grade_cam_analysis.py.

35. Using the script in grade_cam_analysis.py we produced a visualization for SimpleNet model and Xception model classification

36. Grad-Cam SimpleNet model on fakes dataset:

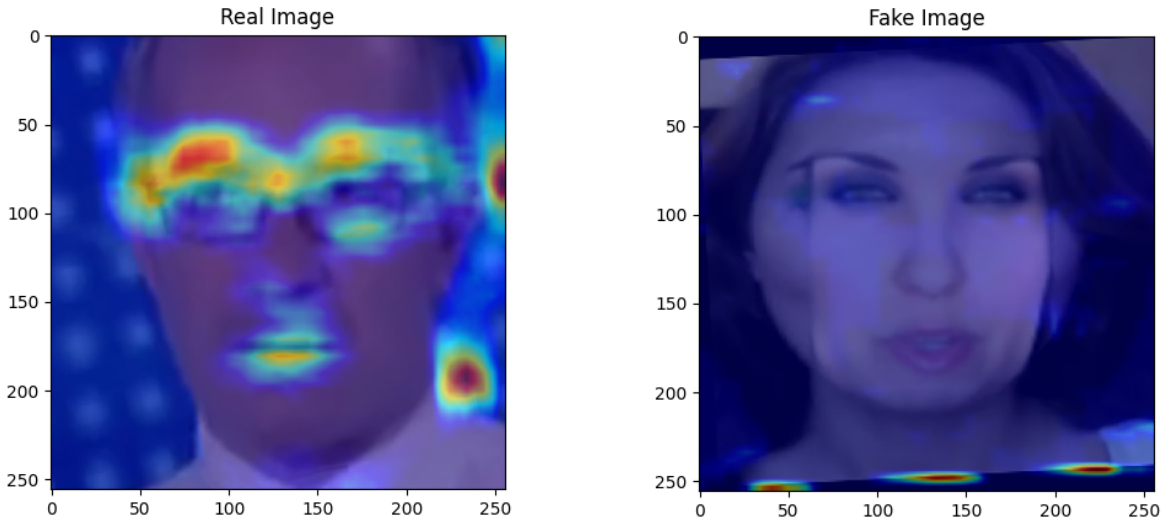

Figure 11: Grad-Cam SimpleNet model on fakes_dataset

We can see from these plots that the SimpleNet model's output regarding the real image takes into account the main features of the face in the image like the nose, mouth and eyebrows (as seen in the saliency map), on the other hand in the fake image the main pixels that are taken into account are the pixels in the bottom where the picture is cut - we can assume this unnatural feature is a sign for the model that the image is fake.

Grad-Cam SimpleNet model on synthetic dataset:

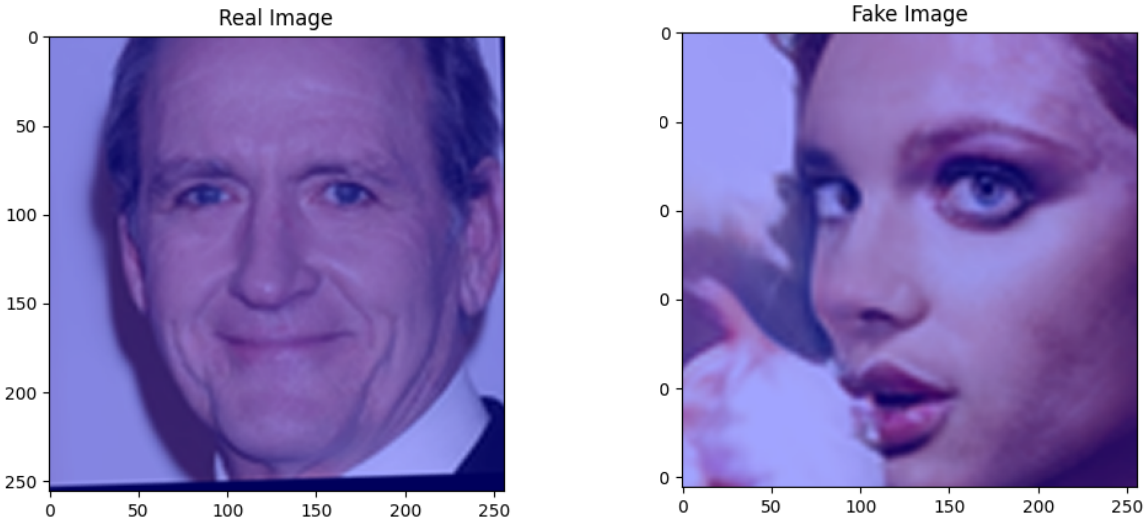

Figure 12: Grad-Cam SimpleNet model on synthetic dataset

As we can see here in both real and fake images no specific pixel contribute to the decision whether the image is real/fake. This makes sense since the model's result on this data set are not great – we can assume that because no specific pixel contributes to the decision the classification is highly random and thus wrong often.

Grad-Cam Xception model on synthetic dataset:

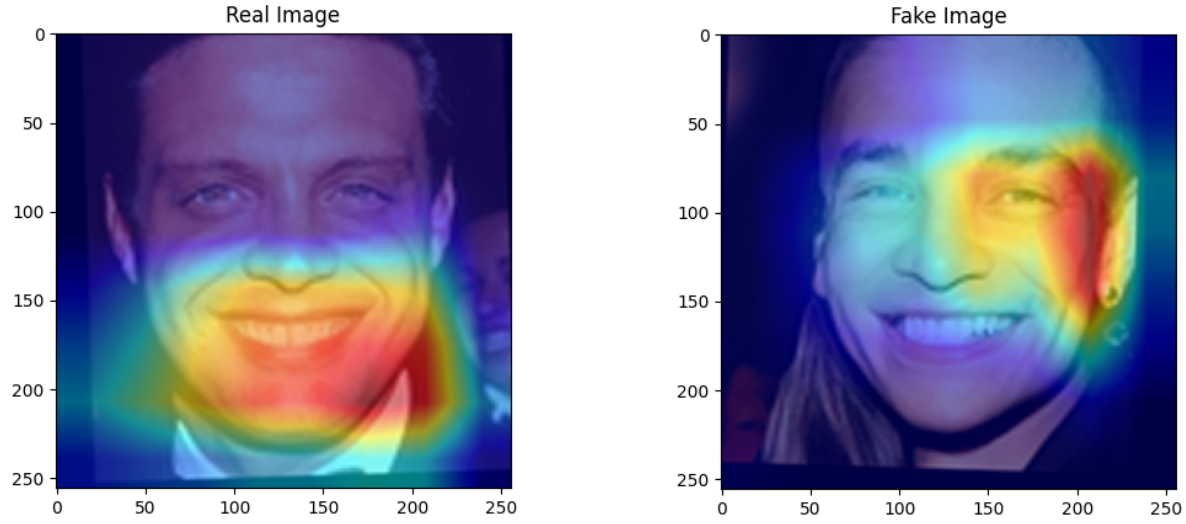

Figure 13: Grad-Cam Xception model on synthetic dataset

in these plots we can see that the result of the Xception model classification on these images in highly dependent on the image itself (in contrast to figure 13). We can see that the model focuses on specific parts on the face in the image and in each image the part is different.

Bonus Part

in this section we were asked to build and train a model on Deepfakes dataset while taking into account the models size encapsulated in the number of parameters in the model.

The model we designed is built on “mobilenetV3 small” model with a different classifier at its head. When designing this model, we thought taking a similar approach we took for implementing the XceptionBased model, essentially taking a pre-trained model on image classification and changing its head (in a similar manner as we changed in Xception) and finetuning it by training the whole model on our data.
The model we took is based on the following article: mobilenetV3

We chose mobilenetV3 small because the design of mobilenet is such that it is supposed to run in embedded systems therefore its size and number of parameters is small compared to other models. The models number of parameters in 1,091,298, which is more than 10 times smaller than Xception and 16 times smaller than SimpleNet and yet giving better results than SimpleBet.

in addition, we used the loss and accuracy plot methods to follow the model's capability:

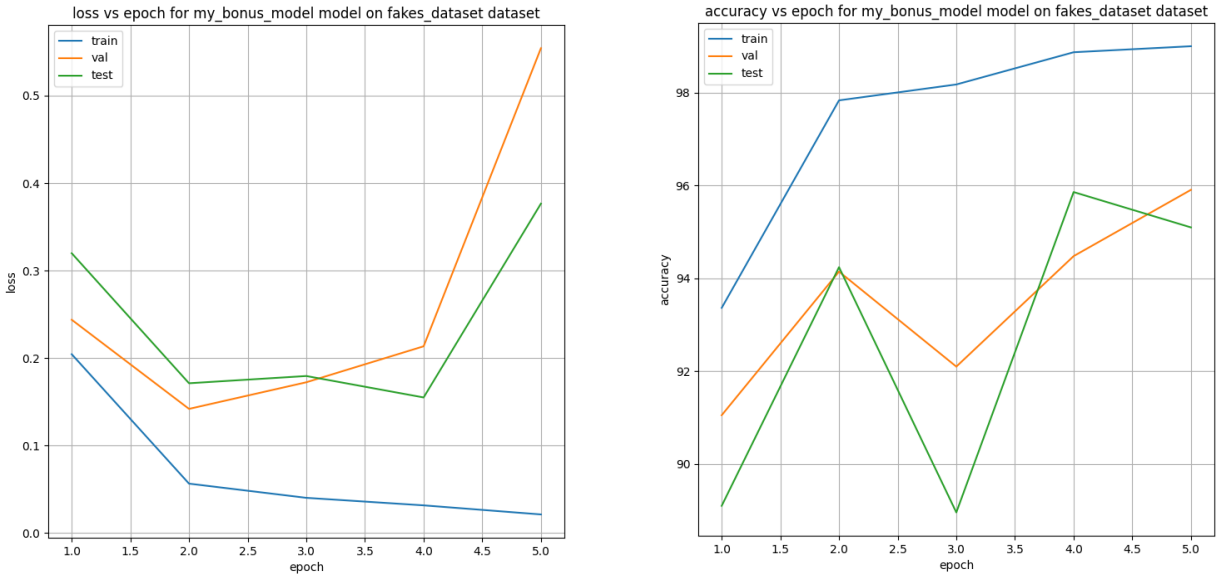

Figure 14: bonus model on Deepfakes dataset loss and accuracy

as we can see in the plot's the models highest validation accuracy is 95.9% with corresponding test accuracy of 95.09%. although it is noticeable that the model does perform overfitting to the dataset, considering the limited number of parameters and data we are allowed to train on, these results are very good.