Computer Vision Project

Tomer Geva Maor Naftali

January 2022

1 Introduction

1.1 The problem

Two datasets of real and fake images are given:

- Deepfake detection dataset ("fakes dataset")
- Synthetic image detection dataset ("synthetic dataset")

Each dataset contains a set of

- Real images - images of existing identities
- Fake images - created by planting one face into the context of another
- Synthetic images - created by GANs trained against a pristine set of images.

1.2 Goal

The goal is to build detection models which given an image, predicts if the image is real or not.

2 Build Faces Dataset

2.1 Introduction

Question 1

In this question we were asked to implement a basic pytorch dataset. Every mapping based pytorch dataset must have the following functions implemented

- __init__(self, <optional variables>)
- __len__(self)
- __getitem__(self, index)

The __init__ function was partially implemented, with an additional class parameters containing the name lists of all the real pictures ($real\_image\_names$) and fake pictures($fake\_image\_names$). An addition for the init method we made was to ass additional parameters holding the length of these lists ($real\_len$ and $fake\_len$ respectively). Therefore, the __len__(self) method returns the cumulative length of both real image name list and the fake image name list.

The __getitem__(self, index) method was implemented as follows

1. if $index < real\_len$
   (a) picture = $real\_image\_names$[index]
   (b) label = 0
2. else
   (a) picture = $fake\_image\_names$[index - $real\_len$]
   (b) label = 1
3. Load picture as PIL image
4. if transform is not None, perform transform
5. return picture, label

Question 2

In this subsection we ran the "plot_samples_of_faces_dataset.py" file to load real and fake images and plot them. The resulted plot is seen in Figure (1) below.

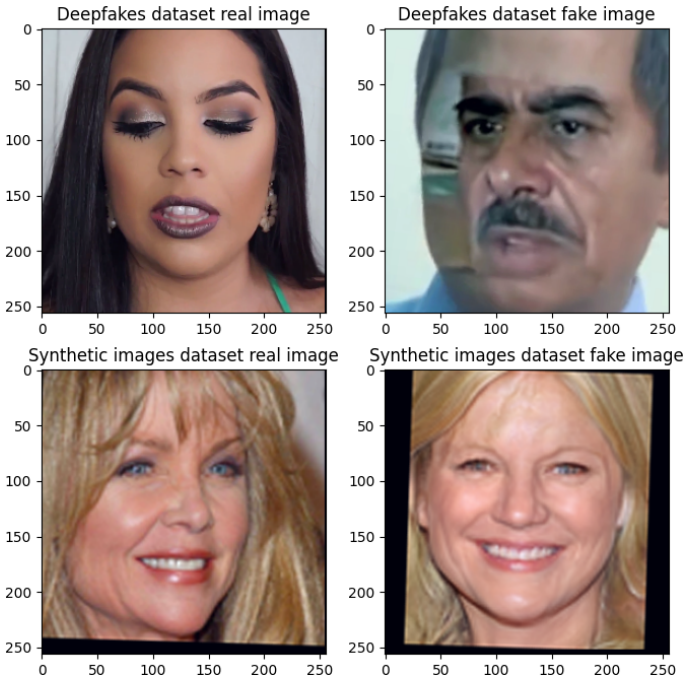


Figure 1: Dataset samples, script output

3 Abstract Trainer

3.1 Abstract Trainer Implementation

Question 3

In this subsection we were instructed to implement an abstract trainer function that trains a model for a single epoch. The trainer function was implemented under the "train_one_epoch" API under the Trainer class and consisted of the following steps

1. Load the dataloader with the wanted batch size
2. For every batch in the dataloader
   (a) Zero the gradients at the optimizer
   (b) Compute a forward pass of the model
   (c) Compute the loss w.r.t the criterion
   (d) Compute the backward pass
   (e) perform an optimization step using the optimizer
   (f) update the mean loss and accuracy
3. return mean loss, accuracy

Question 4

In this subsection we were asked to implement a basic method of evaluation the performance of the model. the evaluation was iimplemented under the "evalutae_model_on_dataloader" API under the Trainer class. Under a torch.no_grad() context manager, the API performs the following

1. Create a dataloader from the given dataset with the wanted batch size
2. Set the model to evaluation mode
3. For every batch in the dataloader
   (a) Compute a forward pass
   (b) Compute the loss w.r.t the criterion
   (c) Update the mean loss and accuracy
4. Set the model back to training mode
5. return mean loss, accuracy

3.2 Training a Deepfake Detection Classifier

Question 5

Using the "train_main.py" module we trained the "SimpleNet" architecture on the Deepfakes dataset. The training used the following hyper parameters

- Learning rate $1e-3$
- Batch size 32
- 5 epochs
- Adam optimizer

The training was initiated used the following command:
"python train main.py -d fakes_dataset -m SimpleNet –lr 0.001 -b 32 -e 5 -o Adam"

3.3 Deepfake Detection Classifier Analysis

Question 6

After finished, the script ran from Question 5 saved a json file. After inspecting the file all looks in tact and all the variables made sense.

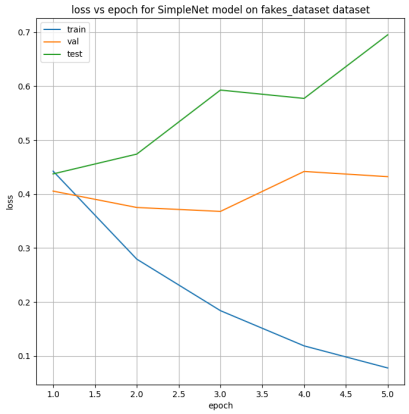
Question 7

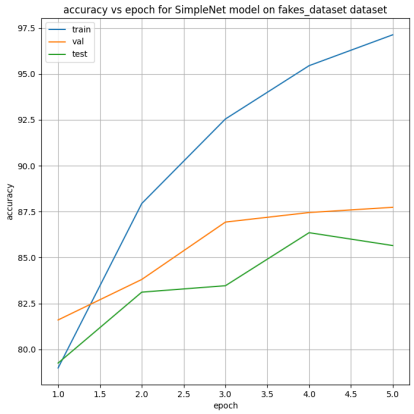
In this subsection we were requested to run the "plot_accuracy_and_loss.py" script to visualize the data held in the json file using the following command.
"python plot_accuracy_and_loss.py -m SimpleNet -j out/fakes_dataset_SimpleNet_Adam.json -d fakes_dataset"

The resulted visualization can be seen in Figure (2b) below. We can see the signs of significant over-fitting, which will be expanded upon in the following questions, specifically in Question 11.

Question 8

As can be seen in Figure (2b) below, the highest validation accuracy received was in epoch #5 and was a little over87.5%. The matching test accuracy was approximately 85% in epoch #5.





(a) Deepfake dataset SimpleNet losses plot *(b) Deepfake dataset SimpleNet accuracy plot*

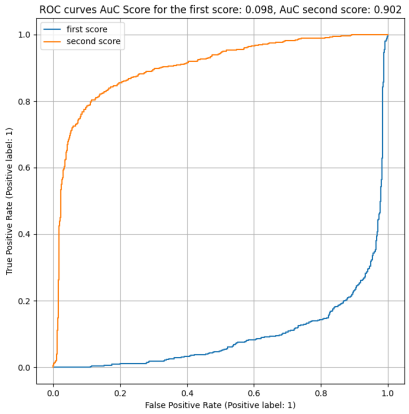
Figure 2: Deepfake dataset SimpleNet performance plots evolving with epochs

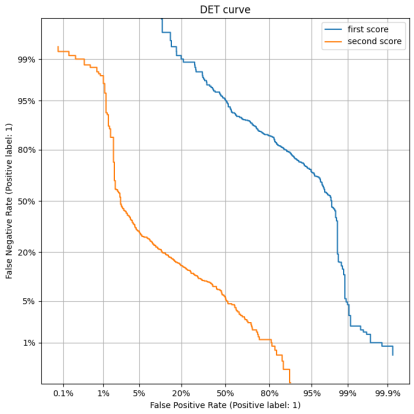
Question 9

In this subsection we were queried regarding the proportion of fake images w.r.t the real images in the testing database for the test set of the fake dataset. The fake dataset has 700 fake images and 1400 real images rendering the fake images to be 33.333% of the test set and the proportion of fake to real images is 1:2.

Question 10

In this question we were asked to implement a function which gathers the soft scores for both scores, i.e. the probability of the input being real with label 0 or the probability of the input being fake / synthetic. The function was implemented under the "get_soft_scores_and_true_labels" API which receives a dataset and a model and returns three lists holding the soft scores for the real score, fake/synthetic score and the true labels respectively. This API is then used to generate the Receiver Operating Characteristic (ROC) curve and the Detection Error Tradeoff (DET). The resulted plots using the SimpleNet model after 5 epochs of training and the fake dataset is seen if Figure (3) below.





(a) Deepfake dataset SimpleNet ROC curve *(b) Deepfake dataset SimpleNet DET curve*

Figure 3: Deepfake dataset SimpleNet numerical analysis plots

The ROC curve is a curve showing the performance of a classification model at different classification thresholds. This curve plots the True Positive Rate (TPR) in the y axis and the False Positive Rate (FPR) in the x axis. The TPR and the FPR are defined in equation (1) below.

$$TPR = \frac{TP}{TP + FN} \tag{1a}$$

$$FPR = \frac{FP}{FP + TN} \tag{1b}$$

$$TNR = \frac{TN}{TN + FP} \tag{1c}$$

$$FNR = \frac{FN}{FN + TP} \tag{1d}$$

Where TP, FN, FP, TN stands for True Positive, False Negative, False Positive and True Negative respectively. Lowering the classification threshold renders more classifications as positive thus increasing both TP and FP while increasing the threshold causes the opposite effect. From the ROC curve we can compute the Area Under the ROC Curve (AUC). the AUC provides a scale invariant aggregate measure of performance across all possible thresholds.

The DET curve is plotted in normal deviate scale and shows the FNR in the y axis and the FPR in the x axis, both defined in equation (1). For each score out of the two possible scores we need to choose a threshold. Assuming that the likelihood distributions for positive and negative results are normal with means μ_0, μ_1 and standard deviation σ_0, σ_1 respectively, an illustration of the FPR and FNR is seen in Figure (4). When looking at a DET curve, the closer the curves are to a straight line, the underlying likelihood distributions from the system are normal.

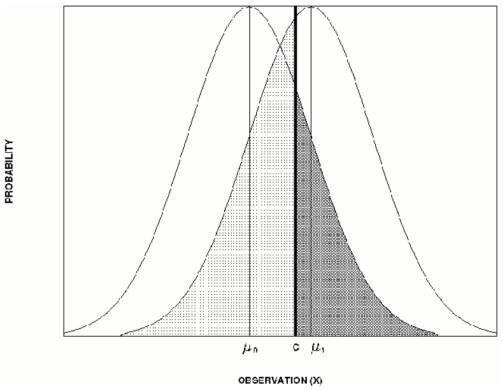


Figure 4: FPR (dark gray) and FNR (light gray) illustration for threshold c. [1]

Question 11

As the plots in Figure (3) show, the results for the first score(i.e. real score) are drastically different than the results for the second score (i.e. fake score). When looking at the ROC curve, the "no skill" line for a binary classifier that simply guesses the classification is the $TPR = FPR$ line, as the guess has 50% chance of being true. Thus, any classifier's ROC curve should reside above the $TPR = FPR$ line, otherwise a different classifier which simply guesses and ignores the input will out-perform the trained classifier. We note that the curve for the real criterion resides below the $TPR = FPR$ line even though the validation classification results after 5 epochs were 87.5%. This can be explained classification process. When training, 'real' label was set to 0 and 'fake' label was set to 1. This means that for the second score (the 'fake' score) the plot output is normal but for the first score (the 'real' score) the results are inverted. This means that when analyzing Figure (3a), as the wellness of the performance for the 'fake' score is measured by the distance of the AuC from 1 (meaning that the closer the AuC is to 1, the better the performance), the wellness of the performance for the 'real' score is measured by the distance of the AuC from 0. This can be verified by the AuC scores. since this is a binary classification problem, the classification can be either 'real' or 'fake' and the sum of the AuC values for both parameters should be 1, which is indeed the case.

The same explanation can be used to explain Figure (3b). The function computing the DET curve takes into consideration that the positive label is 1 and therefore the performance of the 'real' score is inverted.

3.4 Training a Synthetic Image Detection Classifier

Question 12

Using the "train_main.py" module we trained the "SimpleNet" architecture on the Synthetic faces dataset. The training used the following hyper parameters

- Learning rate $1e-3$
- Batch size 32
- 5 epochs
- Adam optimizer

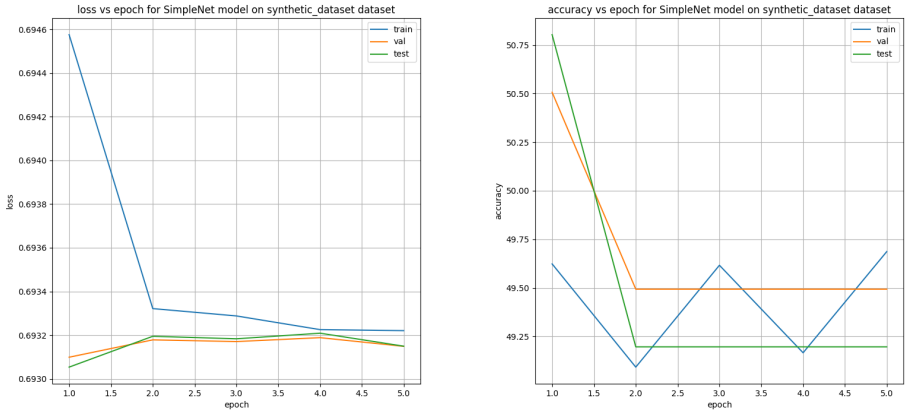
The training was initiated used the following command:
"python train main.py -d synthetic_dataset -m SimpleNet –lr 0.001 -b 32 -e 5 -o Adam"

3.5 Synthetic Image Detection Classifier Analysis

Question 13

In this subsection we were requested to run the "plot_accuracy_and_loss.py" script to visualize the data held in the json fileusing the following command.
"python plot_accuracy_and_loss.py -m SimpleNet -j out/synthetic_dataset_SimpleNet_Adam.json -d synthetic_dataset"

The resulted visualization can be seen in Figure (5) below. We can see the signs of significant over-fitting, which will be expanded upon in the following questions, specifically in Question 16.



(a) Synthetic images dataset SimpleNet losses plot

(b) Synthetic images dataset SimpleNet accuracy plot

Figure 5: Synthetic images dataset SimpleNet performance plots evolving with epochs

Question 14

As can be seen in Figure (5b), the highest validation accuracy received was in epoch #1 and was a little over 50.75%. The matching test accuracy was approximately 50.5% in epoch #1.

Question 15

In this subsection we were queried regarding the proportion of fake images w.r.t the real images in the testing database for the test set of the fake dataset. The fake dataset has 552 fake images and 551 real images rendering the fake images to be 50% of the test set and the proportion of fake to real images is 1:1.

Question 16

As the plots in Figure (5) show, the loss for both the training, validation and the testing does not change during the training as well as the accuracy which remains 50% when observing at the accuracy plot in Figure (5b). When looking into the model outputs, we see that the 'real' scroe is always higher than the 'fake' score, meaning that the model ignores the output altogether and classifies all the input as 'real'. This yields 50% accuracy since the ratio of real and synthetic faces in this dataset is 1:1.

Question 17

Looking at the samples from both datasets, this results seems to make sense, since the fake dataset face quality is poor (as can be seen in the top right image in Figure (1)) and the synthetic face quality is remarkable and is indistinguishable for the naked eye. The "SimpleNet" is too simple to extract meaningful features from the synthetic faces and was unable to distinguish between real and synthetic results, therefore it classifies all the faces as 'real'.

4 Fine Tuning a Pre-trained Model

4.1 Introduction

We would like to obtain better performance measures on the Synthetic Images dataset. To do that, we will use a different model. We will take a pre-trained Xception backbone and change its fully-connected head to a Multi-Layered-Perceptron (MLP). Then, we will fine tune it on the task of Synthetic Image Detection.

Question 18

In this question we were asked on what database was the Xception architecture pre-trained on. In the original paper [4] the Xception model was trained on two datasets:

- ImageNet - Database of images belonging to 1,000 different classes
- JFT - Private Google dataset for large scale image classification, containing over 350 million high resolution images for 17,000 different labels

However, the pre-trained model which we will be using was trained on the ImageNet database.

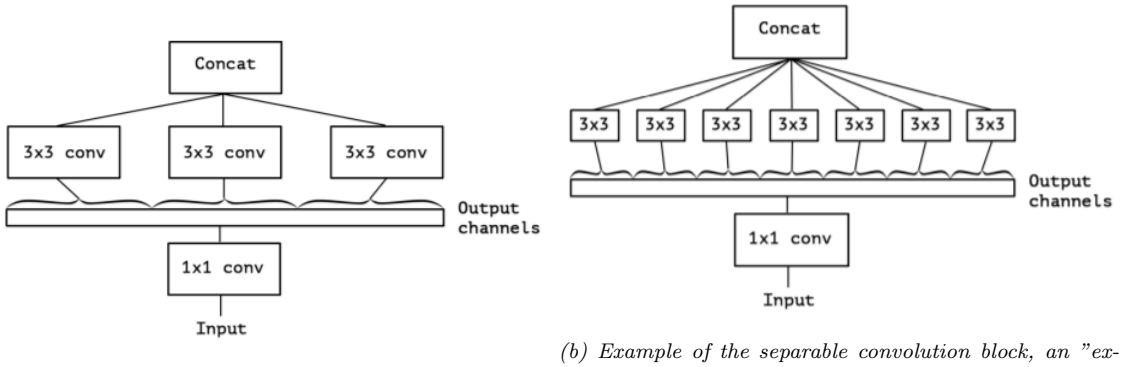
Question 19

In this question we were asked to expand upon the basic building blocks of the Xception architecture. The building block of the Exception architecture is the **separable convolution block**. The following explains the principles behind the separable convolution block. A convolution layer attempts to learn filters in 3 dimensional space composed of 2 spatial dimensions and a channel dimension. This means that the convolution layer is tasked to simultaneously map cross-channel correlations and spatial correlations.

The Xception architecture relies entirely on depthwise separable convolution layers. The main hypothesis behind of the Xception architecture is that the mapping of cross-channels correlations and spatial correlations in the feature maps of convolutional neural networks can be entirely decoupled. Thus, the name for this architecture stands for "Extreme Inception". This inception block performs a 1×1 convolution and then breaks the channel dimension to different groups, performing $n \times n$ convolution over each channel group followed by concatenation over the output channel dimension where n is the kernel size. This can be seen in Figure (6) below. The separable convolution block performs a 1×1 convolution to extract cross-channel correlations and then performs a $n \times n$ convolution over each channel separately, thus decoupling the cross-channel correlation and the spatial correlation mapping, as seen in Figure (6b).

Question 20

Similar question as question 18.



(a) Example if inception module

(b) Example of the separable convolution block, an "extreme" version of the inception module

Figure 6: Inception module (6a) and separable module (6b) [4]

Question 21

In this question we were asked regarding the feature size at the input of the final classification block, named "fc". The final classification block is a Fully Connected block (explaining the "fc" name) with an input vector with length of 2048.

Question 22

In this question we were queried regarding the number of parameters held in the Xception model by default. In the original paper [4], the number of learnable parameters was set to be 22,855,952 and using the "get_nof_parameters" API the result was the same.

4.2 Attaching a New Head to the Xception Backbone

We would like to adjust Xception's head to our classification problem. Specifically, we would like to override Xception's "fc" block with the following MLP architecture:

- Fully-Connected 2048x1000
- ReLu
- Fully-Connected 1000x256
- ReLu
- Fully-Connected 256x64
- ReLu
- Fully-Connected 64x2

Question 23

In this question were asked to implement an API which is loading the pre-trained Xception architecture and replaces the last Fully Connected layer with the new Multi Layer Preceptron (MLP) head written above. This was implemented under the "get_xception_based_model" API under the "models.py" module.

Question 24

In this question we were asked how many additional learnable parameters were added from the addition of the new head. The total number of parameters in the new model is 23,128,786, thus the number of additional learnable parameters is 272,834.

4.3 Train and Evaluate the New Architecture

Question 25

Using the "train_main.py" module we trained the Xception based architecture on the Synthetic image dataset. The training used the following hyper parameters

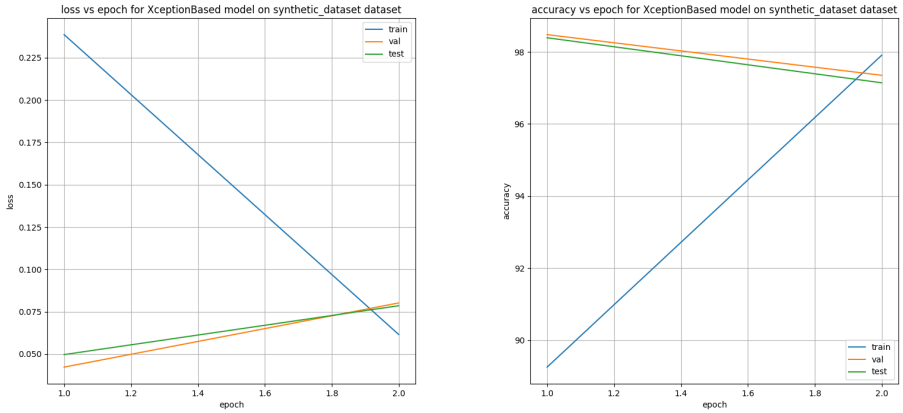
- Learning rate $1e-3$
- Batch size 32
- 2 epochs
- Adam optimizer

The training was initiated used the following command:
"python train main.py -d synthetic_dataset -m XceptionBased –lr 0.001 -b 32 -e 2 -o Adam"

Question 26

In this subsection we were requested to run the "plot_accuracy_and_loss.py" script to visualize the data held in the json fileusing the following command.
"python plot_accuracy_and_loss.py -m XceptionBased -j out/synthetic_dataset_XceptionBased_Adam.json -d synthetic_dataset"

The resulted visualization can be seen in Figure (7) below. We can see that the initialization of the weighted using the pre-trained Xception model bear fruit as the training shows close to perfect performance where the SimpleNet model performance matched a coin toss decision.



(a) Synthetic images dataset Xception based losses plot

(b) Synthetic images dataset Xception based accuracy plot

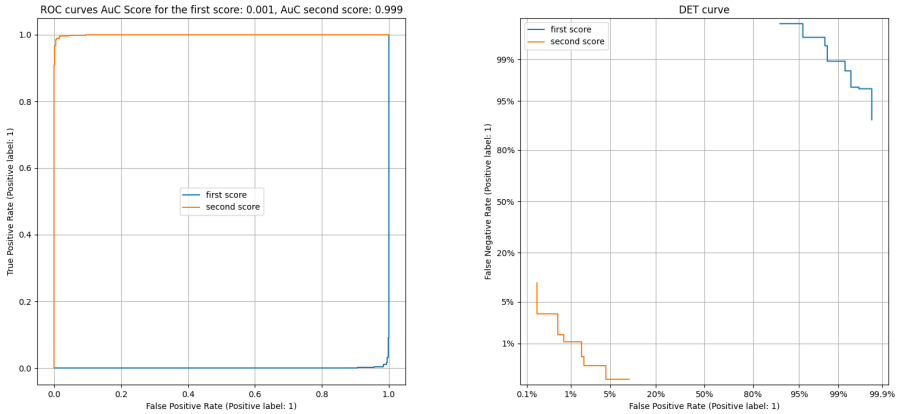
Figure 7: Synthetic images dataset Xception based performance plots evolving with epochs

Question 27

In this question we were asked to discuss the test epoch with the highest accuracy rate. As we can see in Figure (7b), the highest accuracy resullted at the end of the first epoch, with 98.4% accuracy. The corresponding training accuracy in the same epoch was 89.25%.

Question 28

In this question we were asked to run the similar numerical analysis done in question 10 for the Xception based model. the results can be seem in Figure (8) below. We can see that the classification is close to ideal, since the ROC curve (Figure (8a)) shows AuC vlaues very close to 0 and 1 for the 'real' and 'fake' scores respectively, as well as the DET curve (Figure (8b)) shows good behavior as the curves are at the top right and bottom left corners of the plot for the 'real' and 'fake' scores respectively.



(a) Synthetic images dataset Xception based ROC curve

(b) Synthetic images dataset Xception based DET curve

Figure 8: Synthetic images dataset Xception based numerical analysis plots

5 Saliency Maps and Grad-CAM Analysis

5.1 Introduction

In this part developed tools to understand what the classifiers were trained to "see". We used two tools: First we asked what was the contribution of each pixel to the true class soft score and answered this question with Saliency Maps. Second we showed Class Activation Maps (CAM), specifically Grad-CAMs.

Question 29

In this question we were asked to explain what Image-Specific Class Saliency Visualization is. The article [3] discusses two main visualization methods. The first method is Class Model Visualization and the second is Image-Specific Class Saliency Visualization. In short, Image-Specific Class Saliency Visualization is a method which shows the contribution of each pixel in the input image w.r.t a queried class c.

Given an image I_0, we would like to allocate grades to each pixels as a function of their contribution for the score of a specific class c. In the case of a linear score model for a given class c.

$$S_c(I) = \omega_c^T I + b_c \tag{2}$$

Where ω_c are the weights and b_c is the bias used for the score. for this simple linear score model, the magnitude of the elements of ω defines the importance of the corresponding pixels of I for class c. Nevertheless, the score of a specific class $S_c(I)$ is a highly non-linear function of the input image I. Thus, given an image I_0, a linear approximation of $S_c(I)$ can me done used first-order Taylor expansion

$$S_c(I) \approx \omega^T I + b \tag{3a}$$

$$\omega = \nabla_I \left(S_C(I_0)\right) \tag{3b}$$

Question 30

In this section we were asked to explain what Grad-CAMs are. CNN architecture consists of a convolutional part followed by a second part weather it be a fully-connected set of layers (MLP), RNN or something else entirely. The Grad-CAM uses the gradient information from the last convolutional layer of the CNN, assigning importance values to each neuron w.r.t a particular decision. Using these gradient values, a visualization of the importance for each pixel in the original image regarding a decision of class-specific decision can be made. These in turn can be used to create a visualization of the importance of each pixel for the chosen classification.

5.2 Saliency Maps

Question 31

In this question we were asked to implement a methos which is inputted by a model, samples, their labels and outputs the saliency maps for each sample. the method was implemented under the 'compute_gradient_saliency_maps" API, using the following methodology shown in the saliency map article [3].

1. Perform forward pass of the image through the model
2. Compute the derivative ω via back-propagation
3. Since our images have multiple channels, we took the maximum magnitude of ω across all color channels, i.e. for pixel (i, j) in the saliency map value was set to be $\max_c |\omega_{h(i,j,c)}$ where $h(i, j, c)$ is the index of the gradient ω matching the (i, j) pixel

Question 32

In this question we were asked to run the "saliency_map.py" script to generate saliency maps for two cases.

For SimpleNet results using the Deepfakes dataset we used the following command:
python saliency_map.py -m SimpleNet -cpp checkpoints/fakes_dataset_SimpleNet_Adam.pt -d fakes_dataset
The resulted saliency maps can be seen in Figure (9) below.

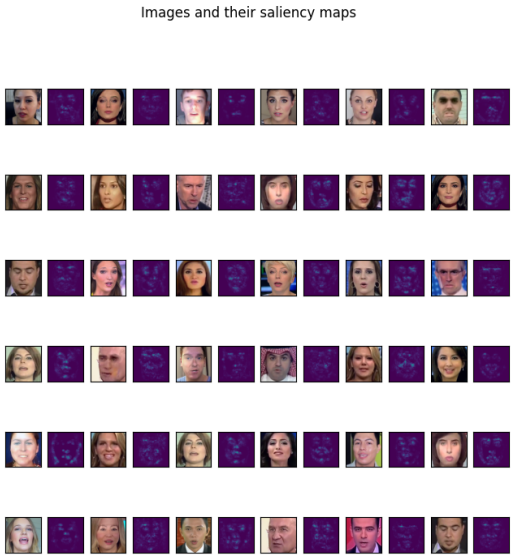


Figure 9: Deepsfakes dataset saliency maps and image pairs. The maps were extracted using a single back-propagation pass through SimpleNet.

To gain meaningful insights, a zoomed-in version of Figure (9) is seen in Figure (10). From the single pass results we can see that the face features seem to contribute the most to the recognition of the face itself and to the decision of 'real' or 'fake'. After running the saliency single run over the entire test database and averaging out the saliency maps to create a mean saliency maps, the results are seen in Figure (11) below. We ccan see that both for the real images and the fake images, the saliency maps looks similar, which explains the miss-classification of SimpleNet. Nevertheless, the real images' saliency map draws more meaning from the facial parts than the fake classification. This correlates with the results where the SimpleNet's test accuracy was 85%.

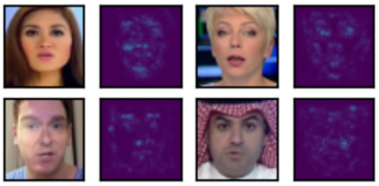


Figure 10: Zoom in of Figure (9)

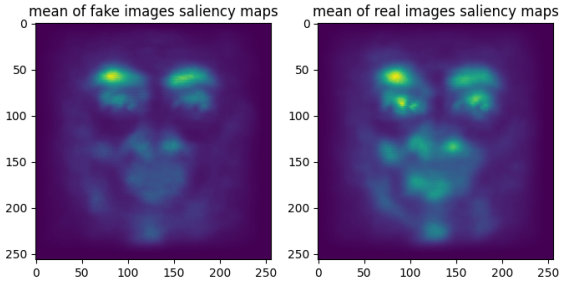


Figure 11: Deepfakes dataset mean saliency maps for 'real' and 'fake' labels using SimpleNet.

For the Xception based architecture with the synthetic dataset we used the following command:
python saliency_map.py -m XceptionBased -cpp checkpoints/synthetic_dataset_XceptionBased_Adam.pt -d synthetic_dataset
The resulted saliency maps can be seen in Figure (9) below.

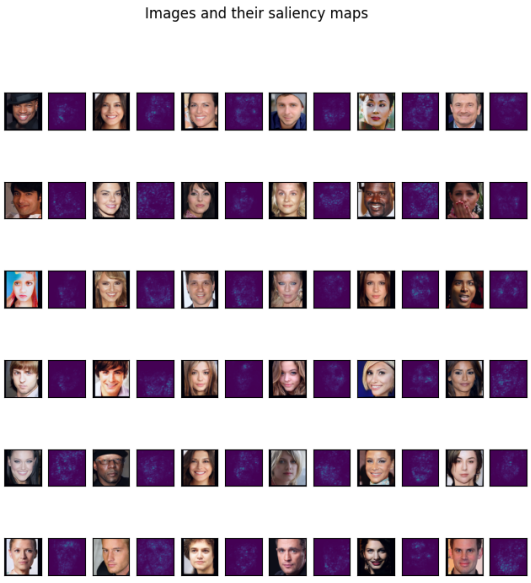


Figure 12: Synthetic dataset saliency maps and image pairs. The maps were extracted using a single back-propagation pass through XceptionBased model.

To gain meaningful insights, a zoomed-in version of Figure (12) is seen in Figure (13). When inspecting the saliency maps in Figure (13) in comparison with Figure (10), we can see that the XceptionBased model captures the facial features better than the SimpleNet model. In addition, we can see that the XceptionBased model extract finer facial features than the SimpleNet model. After running the saliency signle run over the complete test database and averaging the resulted maps, we get the mean saliency map for each criterion, seen in Figure (14) below. The mean saliency maps show that the XceptionBased model successfully isolated the facial features and extractes more meaningful facial data than the equivalent SimpleNet model. Where the SimpleNet model was focusing on the main features of a face such as eyes, eyebrows, mouth etc, the XceptionBased model perferred to give more attention to the general outline of a face as well as the general area surrounding the face itself, as in the mean map, almost all of the face shows high importance, whereas in the SimpleNet's mean saliency map (Figure (11)) only the main features of a face shows high importance.

Figure 13: Zoom in of Figure (12)

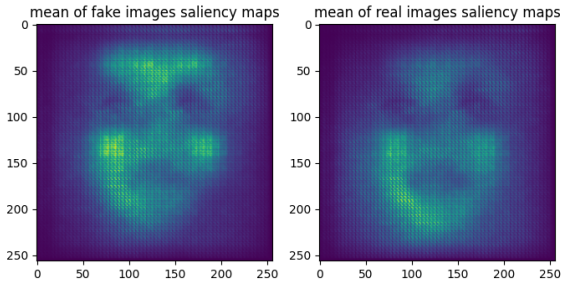


Figure 14: Synthetic dataset mean saliency maps for 'real' and 'fake' labels using XceptionBased model.

5.3 Grad-CAMs

Question 33

In this question we were asked to download the Grad-CAM python package using the installation line from the Github repository. Using the following installation line, we installed the Grad-CAM package in version 1.3.6:
pip install grad-cam==1.3.6

Question 34

In this question we were aked to implement aa method which is inputted a dataset and a model and computes the Grad-CAM for a random image w.r.t the "model.conv3" target layer. the method was implemented under the "get_grad_cam_visualization" API. The API returns the Grad-CAM of the image as well as the label matching the image. A brief explenation of the Grad-CAM operation is given below, summerized from [2].

As depicted in the Grad-CAM paper [2], in order to obtain the class-discriminative localization map $L^c_{Grad-CAM} \in \mathbb{R}^{u \times v}$ for class c with width u and height v, the gradient of the score y^c should be computed w.r.t the feature map activations of the last convolutional layer A^k. Global mean pooling of these gradients produces the neuron importance weights α^c_k

$$\alpha^c_k = \frac{1}{Z} \sum_i \sum_j \frac{\partial y^c}{\partial A^k_{ij}} \tag{4}$$

The class-discriminative localization map $L^c_{Grad-CAM}$ is them obtained via a linear combination of the neuron importance weights and the feature map activations of the last convolution.

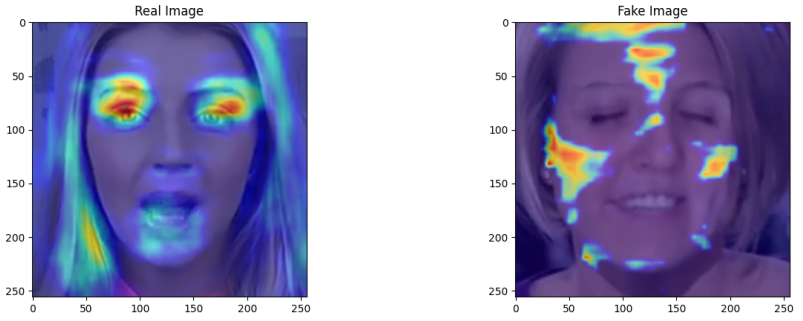
$$L^c_{Grad-CAM} = ReLU\left(\sum_k \alpha^c_k A^K\right) \tag{5}$$

Question 35

In this question we were asked to run the "grad_cam_analysis.py" script to generate Grad-CAM visualization for the following:

- SimpleNet trained on Deepfakes
- SimpleNet trained on Synthetic Faces
- Xception-Based trained on Synthetic faces

To generate the Grad-CAM visualization over the SimpleNet trained on Deepfakes, the following command was used:
python grad_cam_analysis.py -m SimpleNet -cpp checkpoints/fakes_dataset_SimpleNet_Adam.pt -d fakes_dataset
The resulted Grad-CAM visalizations can be seen in Figure (15) below. We can see that the real image (Figure (15a)) is identified using the major facial components, i.e. the eyes and mouth, whereas the SimpleNet model mainly foucused on the perimiter the the face when inputted the fake image (Figure (15a)), at the edge of the two patches.



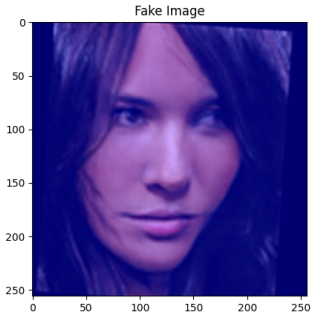
(a) Deepfakes dataset SimpleNet Grad-CAM result - real image

(b) Deepfakes dataset SimpleNet Grad-CAM result - fake image

Figure 15: Grad-CAM results, Deepfakes dataset using SimpleNet model

To generate the Grad-CAM visualization over the SimpleNet trained on Synthetic database, the following command was used:
python grad_cam_analysis.py -m SimpleNet -cpp checkpoints/synthetic_dataset_SimpleNet_Adam.pt -d synthetic_dataset
The resulted Grad-CAM visalizations can be seen in Figure (16) below. As the analysis in Question 16, the Grad-CAM results of the SimpleNet model trained on the synthetic database shows that the model completely ignores the input, since in both the real image Grad-CAM and the fake image Grad-CAM results all pixels have similar importance w.r.t to the classification, a low importance. This strengthens the claims from Question 16 that the network outputs 'real' classification for all inputs.

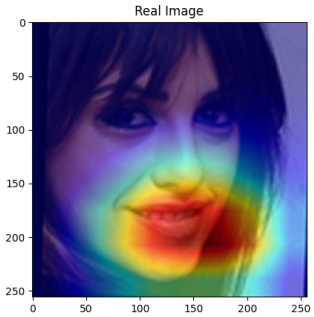
(a) Synthetic image dataset SimpleNet Grad-CAM result - real image



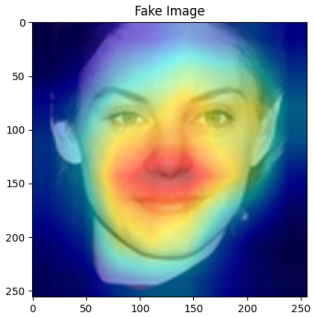
(b) Synthetic image dataset SimpleNet Grad-CAM result - fake image

Figure 16: Grad-CAM results, synthetic image dataset using SimpleNet model

To generate the Grad-CAM visualization over the XceptionBased model trained on Synthetic database, the following command was used:
python grad_cam_analysis.py -m XceptionBased -cpp checkpoints/synthetic_dataset_XceptionBased_Adam.pt -d synthetic_dataset
The resulted Grad-CAM visalizations can be seen in Figure (17) below. We can see that for the XceptionBased model, the output is heavily dependant on the input (which was expected when having 98% accuracy). The synthetic images are of very high quality and even though the naked eye is not able to distinguish between the real and synthetic images, the XceptionBased model finds dissimilarities and is able to successfully distinguish between them.



(a) Synthetic image dataset XceptionBased model Grad-CAM result - real image



(b) Synthetic image dataset XceptionBased model Grad-CAM result - fake image

Figure 17: Grad-CAM results, synthetic image dataset using XceptionBased model

References

[1] Martin et al. "The DET Curve in Assessment of Detection Task Performance". In: (1997).

[2] Selvaraju et al. "Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization". In: (2019).

[3] Simonyan et al. "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps". In: (2014).

[4] Francois Chollet. "Xception: Deep Learning with Depthwise Separable Convolutions". In: (2017).