## Project:

# Facial Manipulation Detection

Ben Solomon 204454615

Din Carmon 209325026

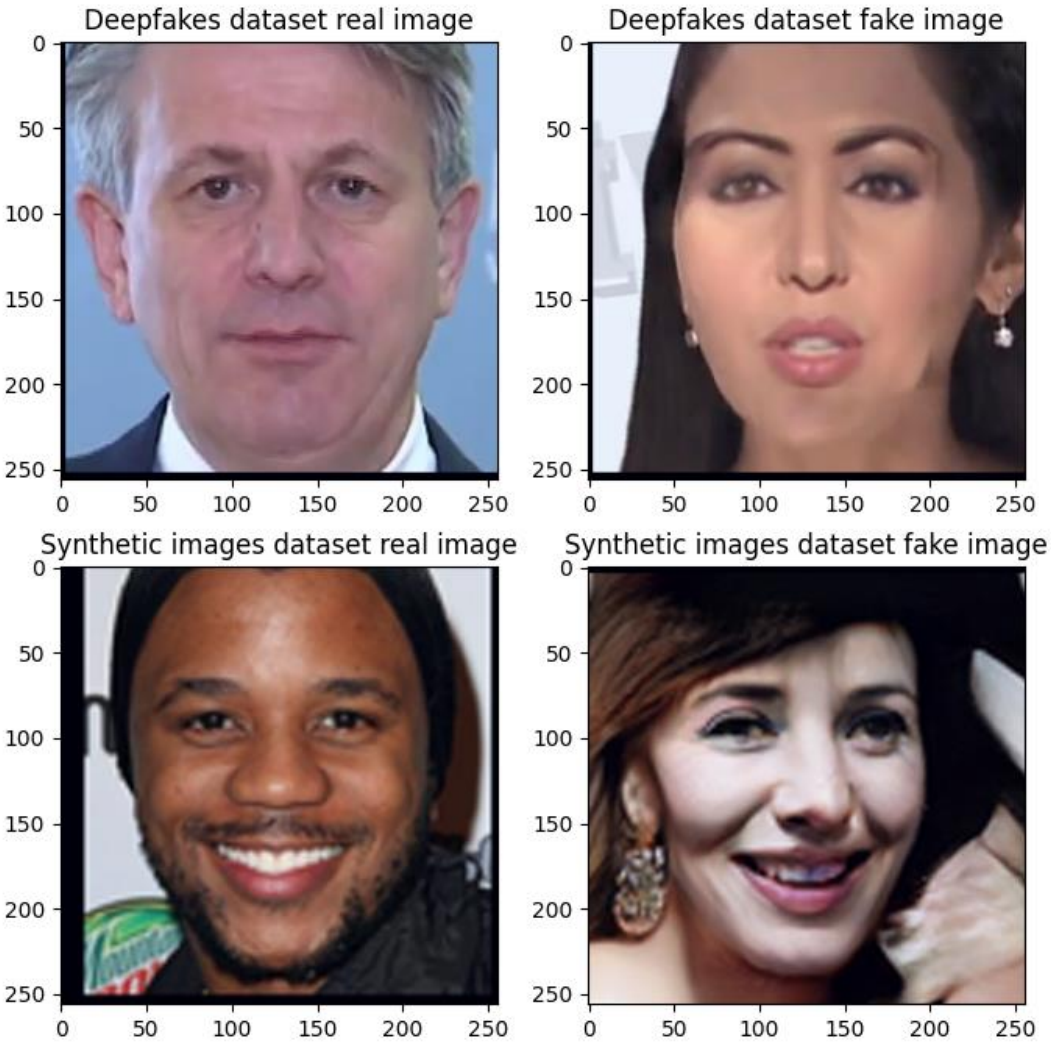
# Chapter 2: Build Faces Dataset

## Question 1:

__getitem__() and __len__() are implemented under faces_dataset.py

## Question 2:

The results of plot_samples_of_faces_dataset.py:



# Chapter 3: Write an Abstract Trainer

## Question 3:

train_one_epoch() is implemented under trainer.py

## Question 4:

evaluate_model_on_dataloader() is implemented under trainer.py

## Question 5:

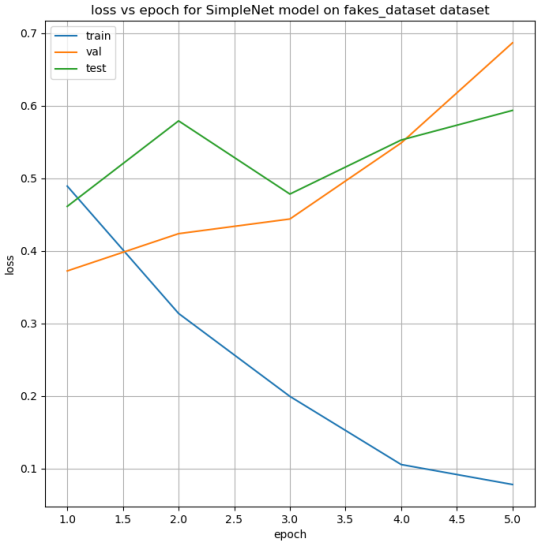
We trained the SimpleNet architecture on the Deepfakes dataset, with lr of 1e-3, batch size 32, 5 epochs and the Adam optimizer.

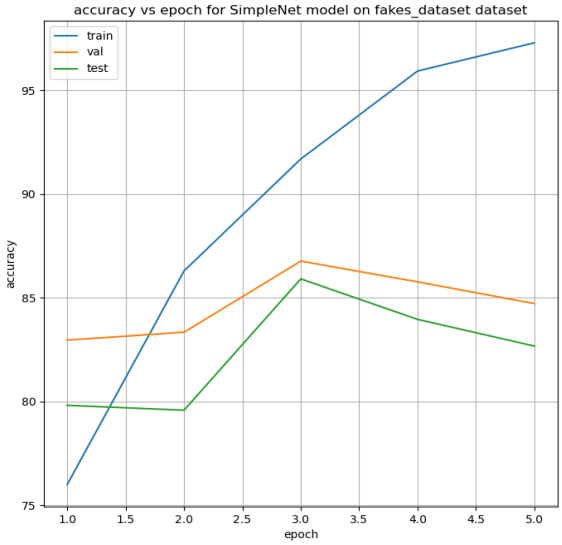
## Question 6:

The JSON contains the training, validation, and test results for a SimpleNet model trained on fakes_dataset using the Adam optimizer. It includes Optimizer settings (e.g learning rate and betas) and Train/Validation/Test Loss and Accuracy over 5 epochs. The results make sense to us because the model learns effectively on the training data (decreasing train loss and increasing train accuracy). However, validation and test losses increase after early epochs suggesting overfitting, meaning the model struggles to generalize.

## Question 7:

The results of plot_accuracy_and_loss.py:



Loss vs epoch:
Train Loss decreases consistently over epochs, suggesting the model is learning from the training data. Validation Loss increasing after the first epoch, indicating potential overfitting. Test Loss follows a similar trend as validation loss, with no consistent improvement.

Accuracy vs epoch:
Train Accuracy increases steadily, showing the model is improving its ability to correctly classify the training data. Test Accuracy and Validation Accuracy peaks at epoch 3, and fluctuates slightly afterward, suggesting the model might not generalize well unseen data – sign of overfitting.

## Question 8:

The highest validation accuracy is 86.76% (at epoch 3).
The corresponding test accuracy is 85.90%.
There is a good alignment between validation and test accuracy, but both are significantly lower than training accuracy, indicating overfitting.

## Question 9:

The proportion of fake images to real images in the test set of "fakes_dataset" is 1/2 (700 fake images and 1400 real images).

## Question 10:

Before viewing the graphs, we show here some definitions for the different concepts:

- True Positives (TP): The outcomes that are correctly predicted as positives.
- False Positives (FP): The outcomes inaccurately predicted as positives.
- True Negatives (TN): The outcomes that are correctly predicted as negatives.
- False Negatives (FN): The outcomes inaccurately predicted as negatives.

And with these concepts we can define the following:

- True Positive Rate (TPR) – True positive rate is the proportion of positive instances that are correctly classified by the model.
- False Positive Rate (FPR) – The false positive rate is calculated as the ratio between the number of negative events wrongly categorized as positive (false positives) and the total number of actual negative events (regardless of classification).
- False Negative Rate (FNR) — also called the miss rate — is the probability that a true positive will be missed by the test.

We wish for our classifier to reach a TPR close to 1, and an FPR, and FNR close to 0.

| Acquired Class \ True Class | T | F |
|---|---|---|
| Y | True Positives (TP) | False Positives (FP) |
| N | False Negatives (FN) | True Negatives (TN) |

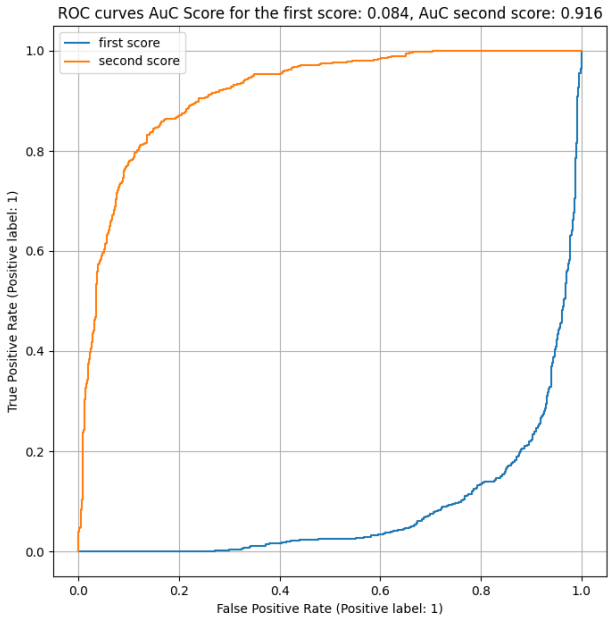
$$\text{True Positive Rate (TPR)} = \frac{TP}{TP + FN}$$

$$\text{False Positive Rate (FPR)} = \frac{FP}{FP + TN}$$

$$\text{Accuracy (ACC)} = \frac{TP + TN}{TP + FP + TN + FN}$$

The **ROC (Receiver Operating Characteristic) curve** is the plot of the TPR against the FPR at each threshold setting (The threshold at which above it we classify the prediction of the NN as 1, and below it we classify the prediction of the NN as 0).
The ROC **AUC score** is the area under the ROC curve. It sums up how well a model can produce relative scores to discriminate between positive or negative instances across all classification thresholds. The ROC AUC score ranges from 0 to 1, where 0.5 indicates random guessing, and 1 indicates perfect performance.

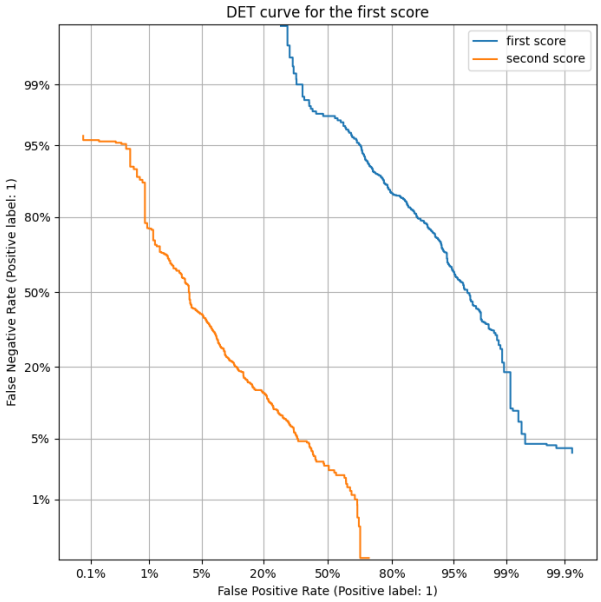
ROC curve graph:




The **DET (Detection Error Tradeoff) curve** is the plot of the FNR against the FPR at each threshold setting.
The x- and y-axes are scaled non-linearly by their standard normal deviates (or just by logarithmic transformation), yielding tradeoff curves that are more linear than ROC curve, and

use most of the image area to highlight the differences of importance in the critical operating region.

DET curve graph:




## Question 11:

Considering the two soft scores groups:

A. soft scores obtained from the first output of the network corresponding to the "real" class score.
B. soft scores obtained from the second output of the network corresponding to the "Fake/Synthetic" (not-real) class score.

The difference in results between (A) and (B) arises because the soft scores represent opposite class probabilities. For (A), the scores favor the "real" class, while for (B), they favor the "fake" class. This leads to mirrored curves in the ROC and DET graphs, as the thresholds adjust the tradeoff in opposite directions. in the ROC graph, the ROC curve approach the top-left corner, indicating high true positive rates with low false positive rates, and the second curve is mirrored diagonally. In the DET graph, for (A), as the threshold increases, false negatives will rise while false positives decrease, and the curve will trend towards the top-left corner. For (B), the opposite happens: as the threshold increases, false positives will rise while false negatives decrease, so the curve will trend toward the bottom-right corner.
In other words, for the second soft score, we wish to address the threshold the other way around (Below some threshold to classify the sample as fake, and above some threshold to label it as real). Therefore, each sample which is viewed as correctly classified is actually wrongly classified if we use the second soft score. Therefore **the second curve is actually the curve of the FNR as a function of the TNR**, if we use the second soft score to classify the data.

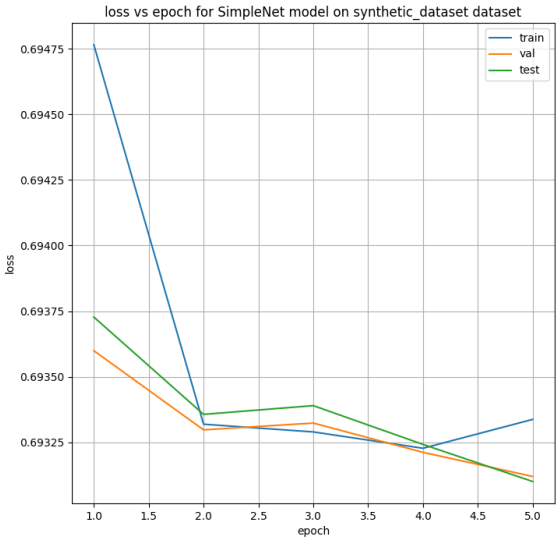
The same logic goes for the second curve of the DET curve. The **second curve is actually the TPR as a function of the TNR** if we use the second soft score to classify the data.

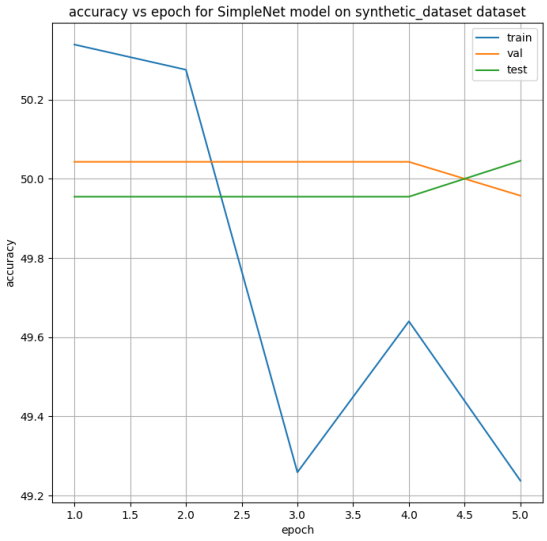
## Question 12:

We trained the SimpleNet architecture on the Synthetic faces dataset, with lr of 1e-3, batch size 32, 5 epochs and the Adam optimizer

## Question 13:

The results of plot_accuracy_and_loss.py:




Loss vs epoch:
Train, Validation, and Test losses stay near 0.693, which is typical for a binary classification model making random guesses (cross-entropy loss for 50% probability = $-\ln(0.5) = 0.693$)

Accuracy vs epoch:
Train, Validation, and Test accuracies stay around 50%, indicating the model fails to learn and is effectively guessing.

## Question 14:

The highest validation accuracy is 50.04%, which occurs at epochs 1-4.
The corresponding test accuracy is 49.95%.

## Question 15:

The proportion of synthetic images to real images in the test set of "synthetic_dataset" is 552/551 (552 fake images and 551 real images) – about 1 to 1.

## Question 16:

The classifier is essentially a random guesser, as it achieves around 50% accuracy across training, validation, and testing. This suggests it fails to learn from the data and performs no better than random chance for the classification task. This may suggest underfit.

## Question 17:

The Synthetic Images dataset is significantly more realistic compared to the Deepfake dataset, making it harder to differentiate between real and fake images. In contrast, the Deepfake dataset contains artifacts like sharp color differences and unnatural straight lines or squares on faces, making fake images easier to spot. Thus, the results in this section make sense: as explained in questions 7,8 the model learns effectively on the Deepfake training data compared to questions 13,14 where the model fails to learn the synthetic training data and make random guesses.

# Chapter 4: Fine Tuning a Pre-trained Model

## Question 18:

The file mentions that the weights were “ported from the Keras implementation.”
**Keras** is an open-source deep learning framework written in Python.
Keras includes many pre-trained models such as Xception.
The Xception pre-trained model is trained on the ImageNet dataset, as documented here: https://keras.io/api/applications/xception/.
In addition, it is stated in the Xception class documentation string:
"""
*Xception optimized for **the ImageNet dataset**, as specified in*
*https://arxiv.org/pdf/1610.02357.pdf*
"""

This strongly suggests the weights are based on the **ImageNet** dataset.

## Question 19:

The basic building blocks of the **Xception (Extreme Inception)** model are inspired by the **Inception architecture**, but they replace the original inception modules with **depthwise separable convolutions**. Below are the key building blocks of Xception:

1. **Depthwise Separable Convolutions**
   This is the central idea of Xception. It decomposes a standard convolution into two steps:
   1. **Depthwise Convolution**: Applies a single convolutional filter per input channel (spatially) – for example RGB channels.
   2. **Pointwise Convolution**: Uses a 1x1 convolution to combine the output of the depthwise convolutions across channels.

   This reduces the computational cost while maintaining performance, as it separates feature extraction (spatial processing) and feature combination (channel-wise processing).

**2. Entry Flow**

- The initial part of the model extracts low-level features (a simple fundamental pattern) from the input image.
- It consists of:
  - Standard convolution layers (e.g., 3x3) with stride 2.
  - Depthwise separable convolutions.
  - Residual connections (to enhance gradient flow) - a **residual connection** refers to a shortcut or skip connection that allows the input of a certain layer to bypass the next convolutional operations and be directly added to the output of the layer.

**3. Middle Flow**

- A series of identical blocks that process the feature maps at a constant spatial resolution.
- Each block consists of:
  - Three depthwise separable convolution layers.
  - ReLU activations and batch normalization (It normalizes the inputs of each layer to have a mean of 0 and a standard deviation of 1. This process helps to stabilize the learning process, improve convergence speed, and reduce the sensitivity to the initialization of weights).

- The middle flow is responsible for capturing intricate patterns in the data.

**4. Exit Flow**

- This section refines high-level features and prepares them for classification.
- Similar to the entry flow, but with additional depthwise separable convolutions and a global average pooling layer at the end.
- Features are flattened and passed to the fully connected layers (or a classification head).

**5. Residual Connections**

- Residual connections are used throughout the model to improve gradient flow and alleviate the vanishing gradient problem during training.

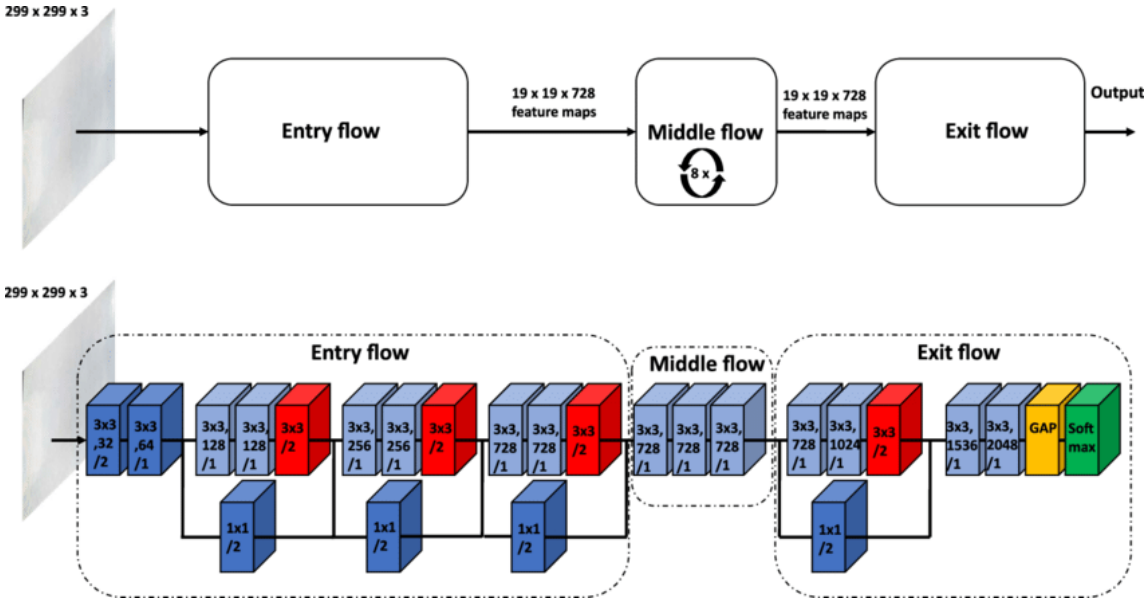
**6. Global Average Pooling**

- Xception uses **global average pooling** before the final classification layer. This operation averages the values across the entire spatial dimensions of each feature map, resulting in a single value per feature map. This reduces the spatial dimensions to a 1x1 size and significantly reduces the number of parameters in the final fully connected layers.

**Architecture Overview:**

- **Input** → Entry Flow → Middle Flow → Exit Flow → Fully Connected (FC) Layer → Output

The design philosophy of Xception is based on the **"extreme" version of Inception**, where each convolution is replaced with depthwise separable convolutions for efficiency and better feature extraction.
A design example for xception based architecture:




## Question 20:

Answered at question 18.

## Question 21:

The last final classification block is defined in this line:

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

Using pytorch documentation on the Linear class (https://pytorch.org/docs/stable/generated/torch.nn.Linear.html ), we know that the first argument is the dimension of the input – 2048.
In other words, in the last layer of the NN, we use 2048 extracted features, which goes through an affine linear transformation, for the final estimated predictions of the NN.

## Question 22:

A python script was implemented under the name q22_23.py which uses the function get_nof_parameters in utils.py to answer how much parameters can be trained in a default xception model: 22855952.
We can also calculate it from the class definition inner structure in xception.py:

- For a Conv2d layer:
  - $\#parameters = (in_{channels} * kernel_{height} * kernel_{width} * out_{channels}) + out_{channels} * (bias == true)$
- For a SeparableConv2D layer (from definition):
  - $\#parameters = (in^2_{channels} * kernel_{height} * kernel_{width}) + in_{channels} * (bias == true) + (in_{channels} * 1 * out_{channels}) + out_{channels} * (bias == true)$
- For a Linear layer:
  - $\#parameters = (in_{features} * out_{features}) + out_{features} * (bias == true)$
- For a block layer – The calculation is a bit more complex and shall not be written here.

The number of parameters can also be found at the end of page 6 in the referenced paper:

#### 4.5.2 Size and speed

Table 3. Size and training speed comparison.

| | Parameter count | Steps/second |
|---|---|---|
| **Inception V3** | 23,626,728 | 31 |
| **Xception** | 22,855,952 | 28 |

In table 3 we compare the size and speed of Inception

## Question 23:

The function get_xception_based_model was implemented. We used the script created at q22_23.py to verify that the number of parameters for the new model matches the hint.

## Question 24:

The amount of parameters we added by adding the MLP on top of the original Xception:

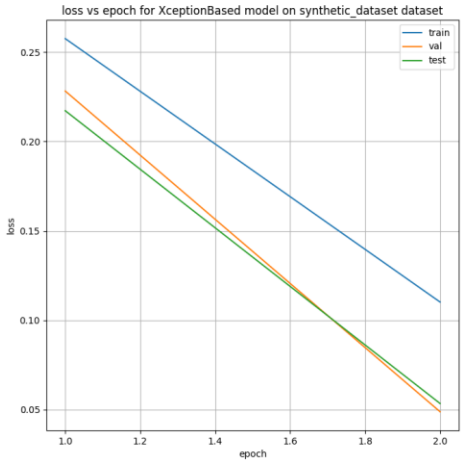
- 23128786 – 22855952 = 272834

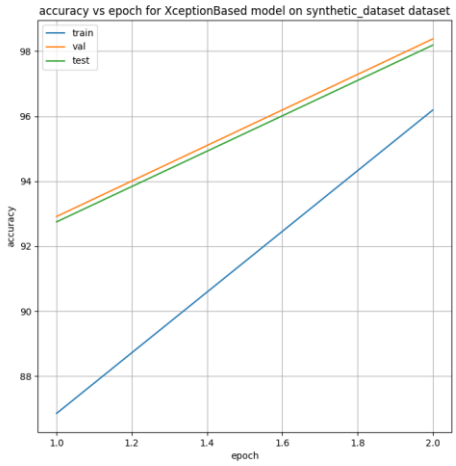
## Question 25:

The xception-based model was trained according to the instructed configuration. The json file documenting the results of the model along training is attached under the out folder.

## Question 26:

The accuracy and loss of the xception-based trained model are depicted here:



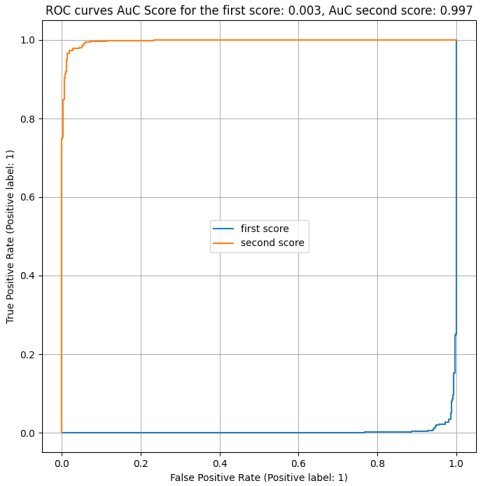

It is clear that the results are far better than using the simple-net model after only one epoch (The loss is smaller for the evaluation and tests data sets, and the accuracy is bigger) – by comparing to the graphs found at the answer to question 13.

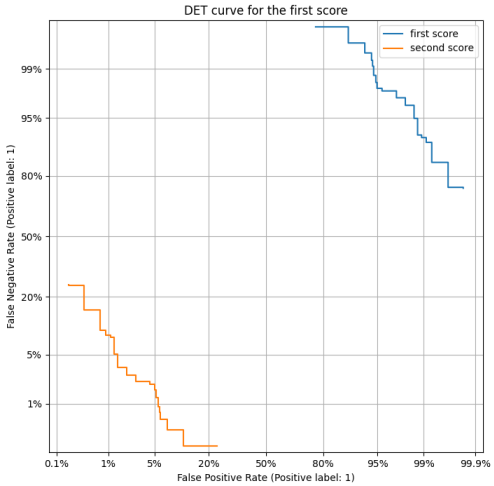
## Question 27:

The highest validation accuracy is 98.63%, which occurs at epochs 1.
The corresponding test accuracy is 98.19%.

## Question 28:

The ROC and DET curves:




Comparing to the graphs found at answer to question 10 (The comparison is somewhat lacking, since the dataset, in question, is different. In q10, the graphs is for the FakeData set, and here it is for the Synthetic data set. However, as we saw the synthetic data set is much more hard to predict correctly, thus only stretnthening our claim), Clearly, the ROC curve shows better results folr the Xception based solution, with a bigger AuC score of 0.997 compared to 0.916 beforehand.

# Chapter 5: Saliency Maps and Grad-CAM analysis

## Question 29:

A **Saliency Map** (also called Feature Attribution) is a method to visually highlight / represent the input features which were "relevant" for a certain input to be classified as it was classified.

In other words, it is a visual explanation for what region and attributes are used in the picture for the model to classify.

What makes a good visual explanation?

(a) class discriminative (i.e. localize the relevant features in the image)

(b) high-resolution (i.e. capture fine-grained detailed features).

Image specific class saliency visualizations (Also called Pixel Attribution) is a specific case of a saliency map where the input is an image. These are methods which are used to visualize which pixels were relevant for the image classification.

There are many approaches to calculate the saliency map. There are 2 different categories:

- **Occlusion- or perturbation-based**: Methods like SHAP and LIME manipulate parts of the image to generate explanations (model-agnostic).
- **Gradient-based**: Many methods compute the gradient of the prediction (or classification score) with respect to the input features. They tell us whether a change in a pixel would change the prediction. Examples are Vanilla Gradient and Grad-CAM. The interpretation of the gradient-only attribution is: If I were to increase the color values of the pixel, the predicted class probability would go up (for positive gradient) or down (for negative gradient). The larger the absolute value of the gradient, the stronger the effect of a change of this pixel. The gradient-based methods (of which there are many) mostly differ in how the gradient is computed.
    - For example, the vanilla gradient approach computes the saliency map by a "simple pure" gradient calculation as is done in the backpropagation algorithm. This means that if a relu function at some node got as an input a negative value, the backpropagation shall yield in a 0 gradient back propagating from that node. One can oppose, and claim that it is wrong to state that the nodes contributing to the negative input of these relu activation function, have a relevance in zeroing the output node. In other words, relevance is an ambigous term, and thus the vanilla gradient approach has its limits, and lacks some of the meaning of relevance in its algorithm.

It should also be noted, that the term saliency map is also used in natural / biological vision to state the regions of the image where the attribute's eyes focus first. The saliency maps engineered in computer vision are typically not the same as of human eyes.

In addition, saliency map extraction can be thought of as a method of image segmentation.

## Question 30:

Grad-CAM stands for Gradient-weighted Class Activation Map. And, as the name suggests, it is based on the gradient of the neural networks. Unlike other gradient-based methods, the gradient is not backpropagated all the way back to the image, but (usually) to the last convolutional layer to produce a coarse localization map that highlights important regions of the image.
Pixel-space gradient visualizations such as Guided Backpropagation and Deconvolution are high-resolution and highlight fine-grained details in the image but are not class-discriminative.
In contrast, localization approaches like CAM are highly class discriminative, but do not hold high resolution of the features used.
To combine the best of both worlds, Grad-CAM was designed. As a result, important regions of the image which correspond to any decision of interest are visualized in high-resolution detail.
The output of the Grad-CAM algorithm is a heatmap that highlights regions of interest.
The steps of the Grad-CAM algorithm:

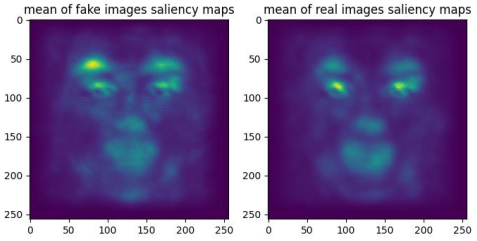
- Compute the gradients of each feature after the first convolutional layer
- Using the gradients, a weighted sum of the feature map is calculated, along with a ReLU activation to retain only the regions which positively influence the classification.
- The values are resized, and normalized for a visual overlay over the original image.

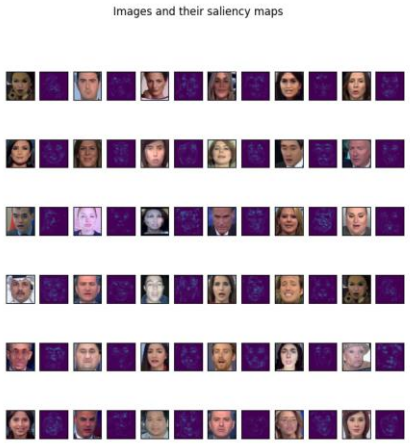
## Question 31:

The method compute_gradient_sailiency_maps was implemented.

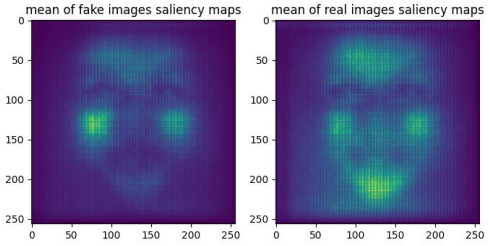
## Question 32:

The sailency map computed for the simple-net model:




The saliency map computed for the xception-based model:





It can be seen that the 2 models use different regions of the face for classification.

The simple-net model uses mainly the eyes and eyebrows to detect a real face. Threfore a synthetic face where this region is not corrupted, shall be classified incorrectly by the simple net model.
The xception-model uses mainly the cheeks to detect a synthetic image. In addition, to classify an image as a real image, it goes over “all” the features in the face to “approve” all of them. This can be seen, by the bright region being all of the face in the real image saliency map for the xception model

## Question 33:

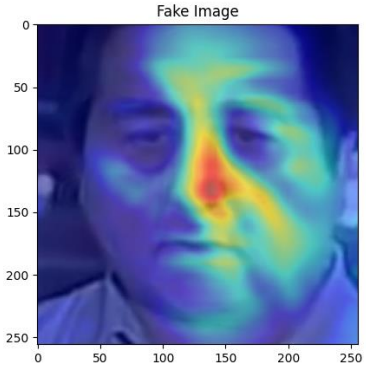
Using the installation command at the attached link (pip install grad-cam), the package was installed.

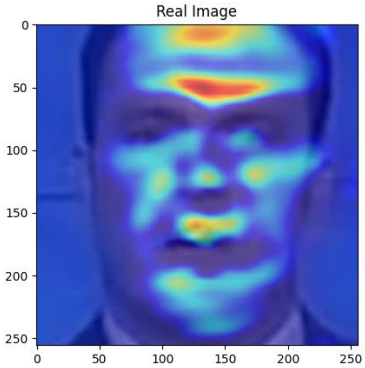
## Question 34:

The minimal changes were made at the function get_grad_cam_visualization().

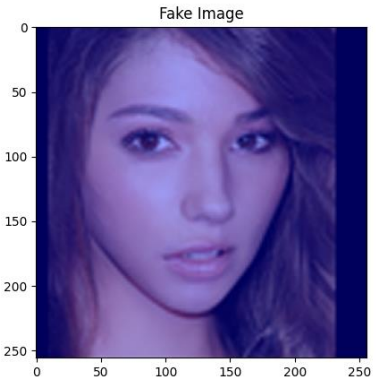
## Question 35:

The grad-cam visualization for the simple-net model with the fakes_dataset:

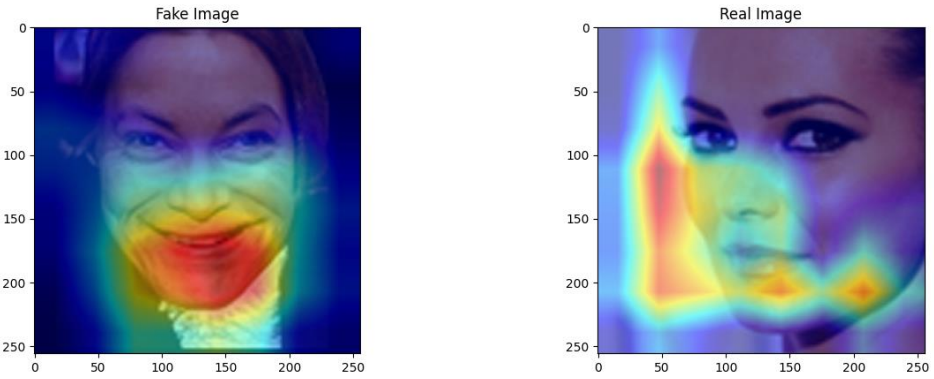


The grad-cam visualization for the simple-net model with the synthetic_dataset:




The grad-cam visualization for the xception-base model with the synthetic_dataset:




# Chapter 6: Bonus Part

We evaluated three different CNN architectures for the classification problem: **MobileNetV2**, **ResNet18**, and **SqueezeNet**. Each of these models has different trade-offs between accuracy, efficiency, and parameter count. We have trained the models on the train dataset of the Deepfakes and examined the result on the test dataset.

We did not try **EfficientNet-B0**, **RegNetX-400MF**, or **MobileNetV3**, all of which could have been strong candidates for this classification task. This is because the environment yaml provided is limited to **torchvision 0.10.0**, which does not include these models.

The models that we have evaluated are:

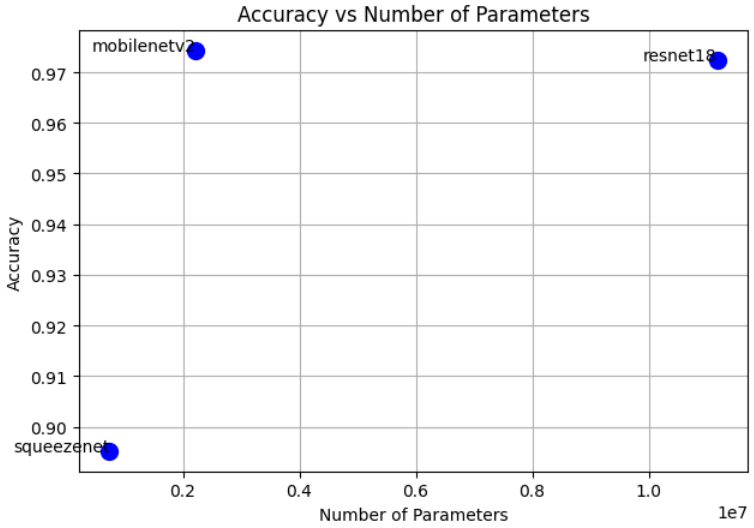
1. **MobileNetV2**
    - A lightweight CNN optimized for mobile and edge devices.
    - Uses depthwise separable convolutions to reduce computational cost.

- Achieved the highest accuracy (97.43%) with only 2.2M parameters, making it the most efficient model.

2. **ResNet18**
   - A deeper network using residual connections to improve training stability.
   - Has 11.2M parameters, significantly more than MobileNetV2.
   - Achieved 97.24% accuracy, slightly lower than MobileNetV2 but with much higher complexity.
3. **SqueezeNet**
   - An ultra-compact CNN optimized for minimal parameters.
   - Uses Fire modules (1x1 and 3x3 convolutions) to reduce computation.
   - Has only 736K parameters but also the lowest accuracy (89.52%).

**Results Summary**

| Model | Accuracy (%) | Parameters | Notes |
|---|---|---|---|
| MobileNetV2 | 97.43 | 2.2M | Best trade-off between accuracy & efficiency |
| ResNet18 | 97.24 | 11.2M | Slightly lower accuracy, much larger model |
| SqueezeNet | 89.52 | 0.74M | Smallest model but much lower accuracy |




Conclusion: The best model for our task is **MobileNetV2.**

- It achieves the highest accuracy (97.43%) while maintaining a low parameter count (2.2M).
- ResNet18 has similar accuracy but requires 5x more parameters, making it less efficient.
- SqueezeNet is extremely small, but its accuracy drops significantly, making it unsuitable.