**Project:**

**Facial Manipulation Detection**

Ben Solomon 204454615

Din Carmon 209325026

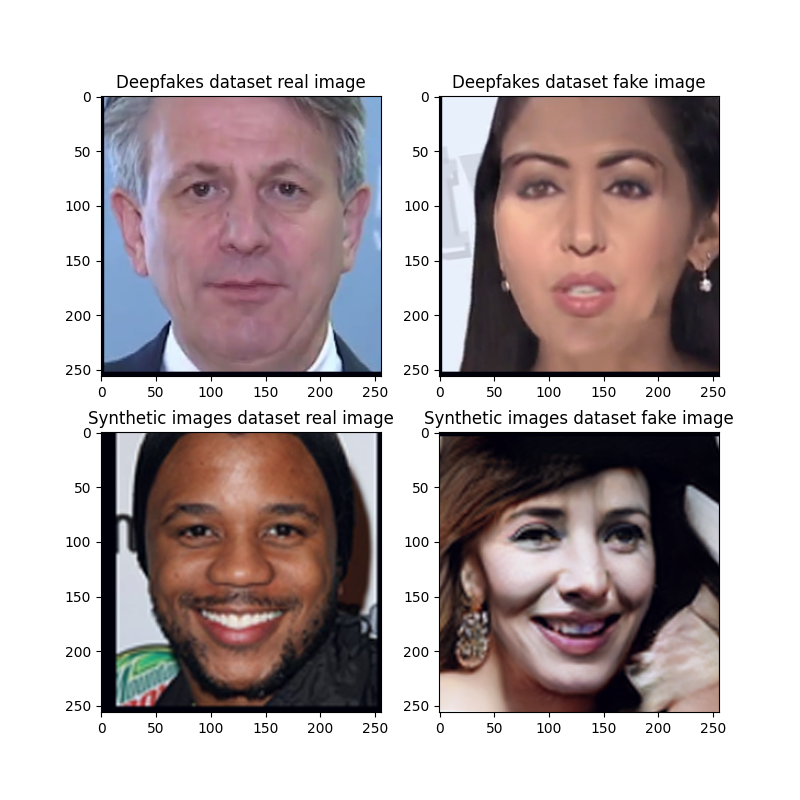
Chapter 2: Build Faces Dataset

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:

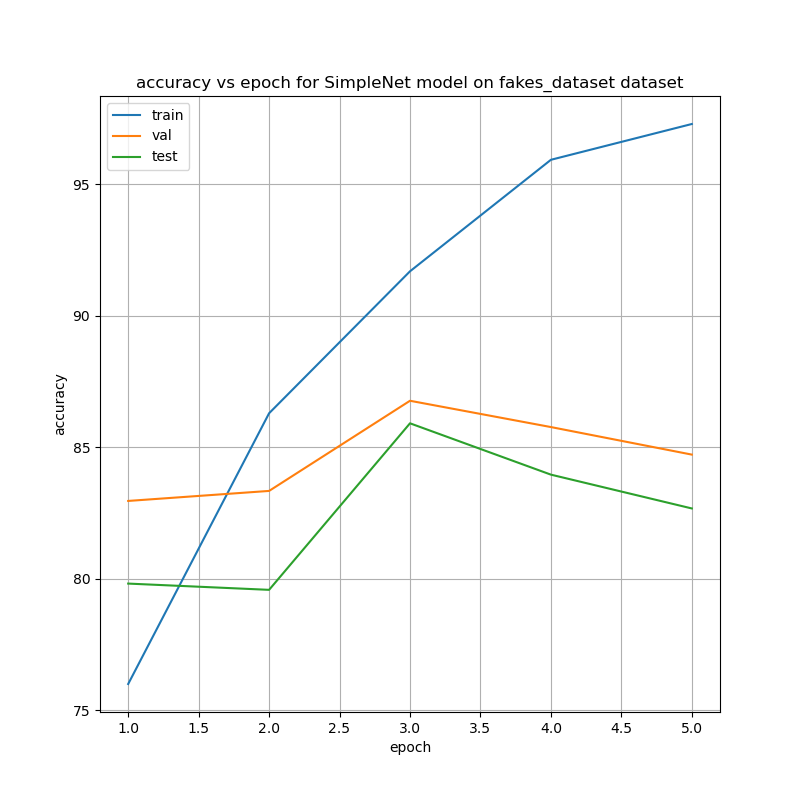
We trained the SimpleNet architecture on the Deepfakes dataset, with lr of 1e-3,

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:

תמונה שמכילה קו, תרשים, עלילה

התיאור נוצר באופן אוטומטיThe 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 Lossfollows 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).

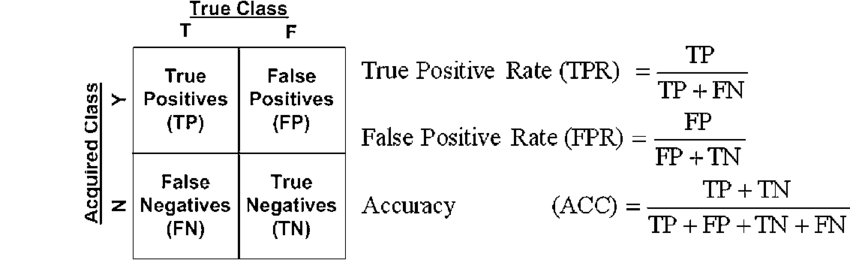
Question 10:

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:

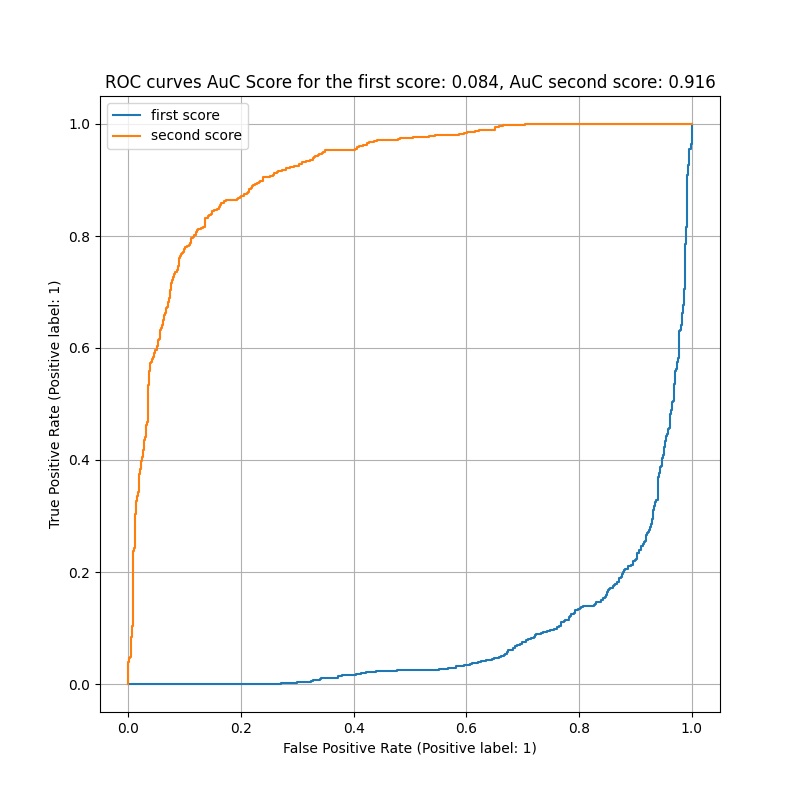
* True Positive Rate (TPR) – True positive rate is the proportion of positive instances that are correctly classified by the model.
* False Positive Rate (FPR) – The false positive rate is calculated as the ratio between the number of negative events wrongly categorized as positive (false positives) 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.

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.

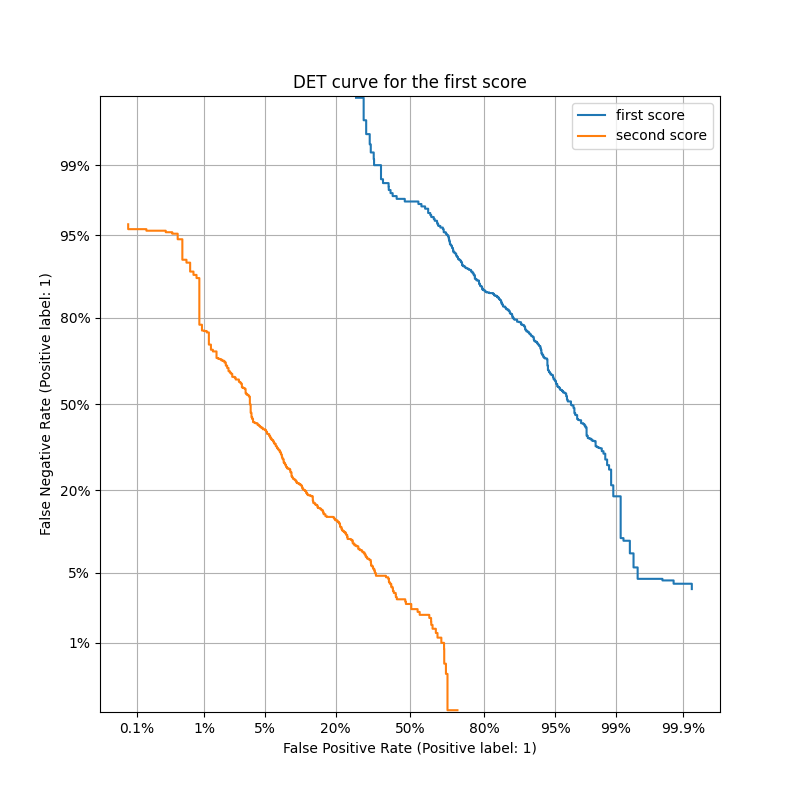
ROC curve graph:



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:

1. soft scores obtained from the first output of the network corresponding to the

"real" class score.

1. 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.

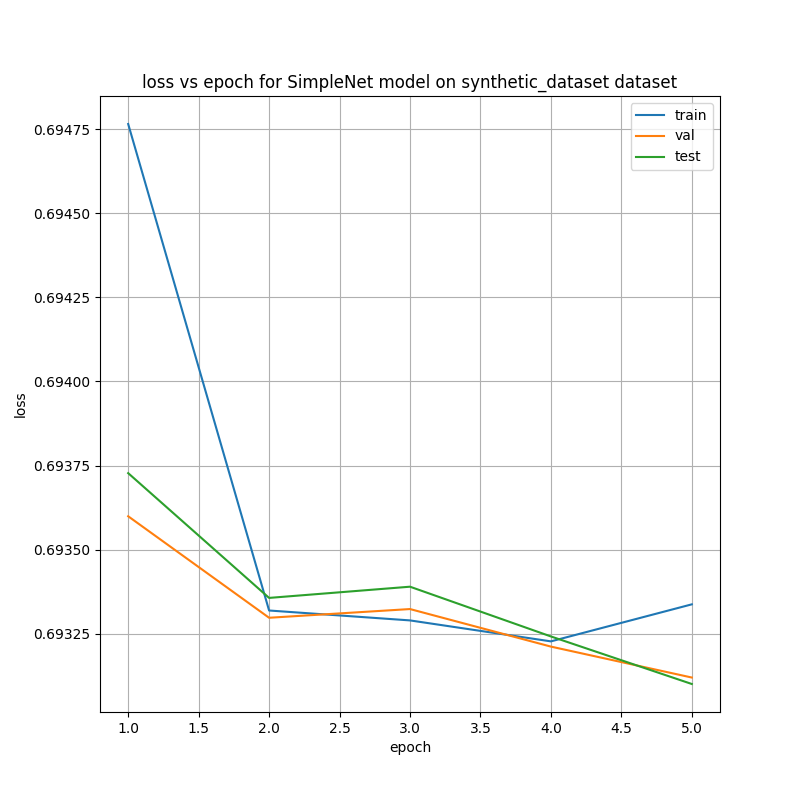
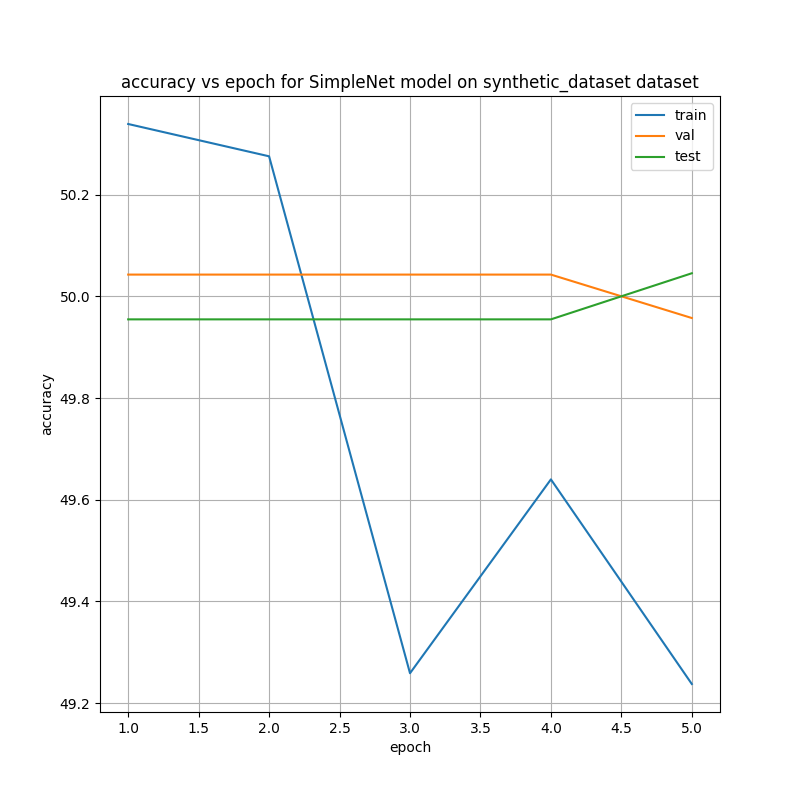
The same logic goes for the second curve of the DET curve. The **second curve is actually the TPR as a function of the TNR** if we use the second soft score to classify the data.

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 = )

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 an 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**

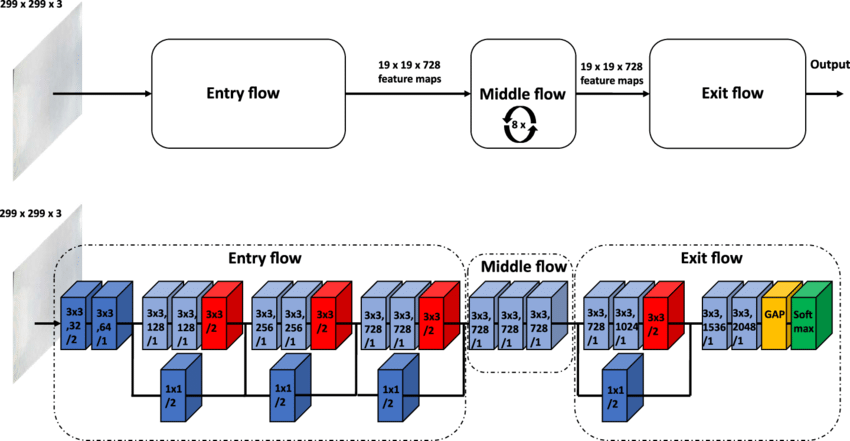
* Xception uses **global average pooling** before the final classification layer. This operation averages the values across the entire spatial dimensions of each feature map, resulting in a single value per feature map. This reduces the spatial dimensions to a 1x1 size and significantly reduces the number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

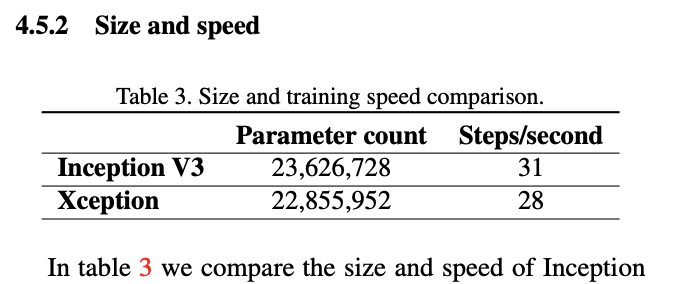
Question 22:

A python script was implemented under the name q22.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:
* For a SeparableConv2D layer (from definition):
* For a Linear layer:
* 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:



Question 23:

A python