PRJ381: Description of modeling scripts and iterations

Kenneth Mc Ferrier 577503

Leon Botha 576378

Kealeboga Molefe 577482

Keneilwe Mpho Ramatsui 577669

Kaylee Carstens 577331

Table of Contents

[Initial scripts and Initial results: 2](#_Toc181045413)

[Download and organizing script: 2](#_Toc181045414)

[The libraries we used for this script: 2](#_Toc181045415)

[Steps in this script: 3](#_Toc181045416)

[Split data script: 4](#_Toc181045417)

[The libraries we used for this script: 4](#_Toc181045418)

[Steps in this script: 5](#_Toc181045419)

[What the different sets get used for: 5](#_Toc181045420)

[Initial Preprocessing script: 6](#_Toc181045421)

[The libraries we used for this script: 6](#_Toc181045422)

[Steps in this script: 8](#_Toc181045423)

[Initial Model training script: 9](#_Toc181045424)

[The libraries we used for this script: 9](#_Toc181045425)

[Steps in this script: 10](#_Toc181045426)

[Initial results: 11](#_Toc181045427)

[Second iteration: 12](#_Toc181045428)

[2nd Iteration of the Model training script: 12](#_Toc181045429)

[Different changes: 14](#_Toc181045430)

[Results of the 2nd iteration: 15](#_Toc181045431)

[Third Iteration: 17](#_Toc181045432)

[3rd Iteration of the Model Training script: 17](#_Toc181045433)

[Different changes: 19](#_Toc181045434)

[Results of the 3rd iteration: 19](#_Toc181045435)

[Fourth Iteration: 20](#_Toc181045436)

[2nd Iteration of the Preprocessing script: 20](#_Toc181045437)

[Different changes: 20](#_Toc181045438)

[Results of the 4th iteration: 21](#_Toc181045439)

[Final Model training and Preprocessing scripts: 22](#_Toc181045440)

[Model training script: 22](#_Toc181045441)

[Preprocessing script: 24](#_Toc181045442)

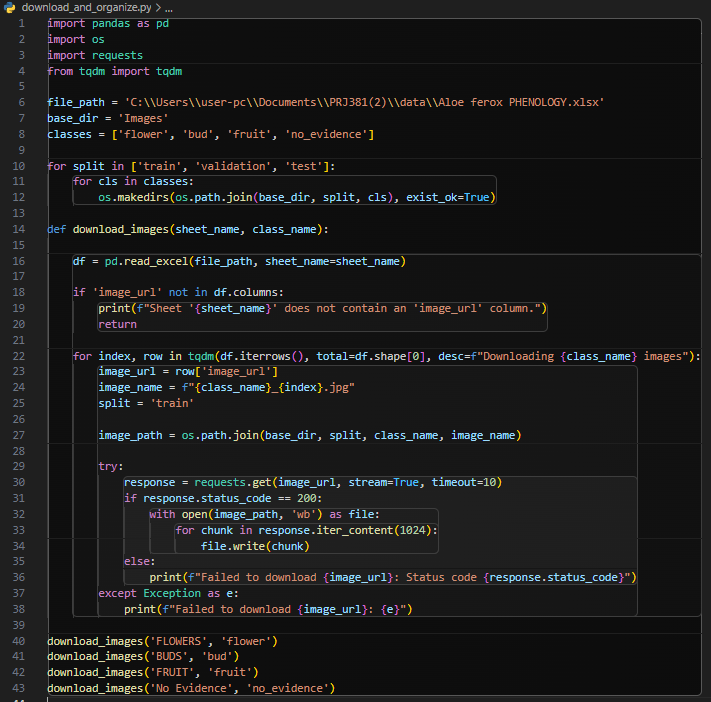
# Initial scripts and Initial results:

## Download and organizing script:

The main goal of this script is to download and organize the images into the appropriate folders. **(This script stayed the same throughout our model building process.)**

### The libraries we used for this script:

* **pandas:** This was used to read the data from the excel file.
* **requests:** Used to download the images from the URLs.
* **os:** To manage directories and file paths.
* **tqdm:** To show us a progress bar during the image downloading process.



### Steps in this script:

1. Define paths:

* Specifies the path to the excel file that was provided to us.
* It defines the “base\_dir” (Images) and classes (like “flower” and “bud”) to organize the images into different categories.

1. Creation of base directories:

* We create folders for training, validation and testing splits for each class.

1. Downloading the images:

* The excel file gets read through to check for the “image\_url” column and then downloads the images to the appropriate folders.
* If any of the images fails to download, it just prints an error message and then continues downloading the rest of the images.

## Split data script:

In this script the images we downloaded gets split into training, validation and test sets.

**(This script stayed the same throughout our model building process.)**

### The libraries we used for this script:

* **os and shutil:** For the file and directory operations.
* **random:** To shuffle the images for splitting.

**A screen shot of a computer program

Description automatically generated**

### Steps in this script:

1. Defining paths:

* Defines the paths for the training, validation and testing directories.

1. Creation of the directories:

* This ensures that the directories for the validation and test sets exist.

1. Proportioning the validation and test sets:

* This specifies that 15% of the images is for validation (val\_split) and 15% is for testing (test\_split).

1. Splitting the images:

* The images get shuffled in each class folder to ensure that the images are in a random order.
* The images then get split into validation and test sets and gets moved to their respective folders using shutil.move().

### What the different sets get used for:

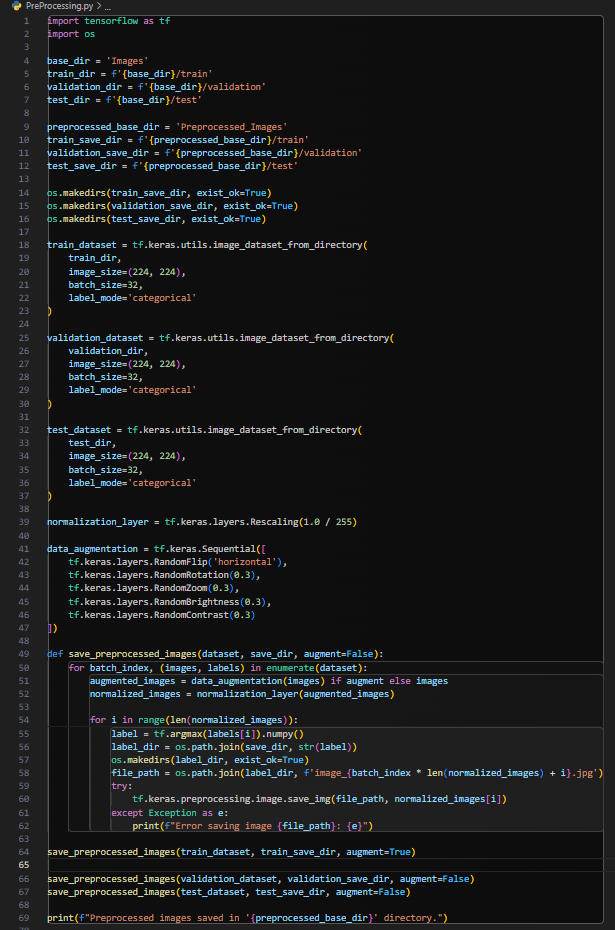
* Train set: Used for the model training.
* Validation set: This is used to evaluate the model while training to prevent overfitting.
* Test set: This is used for final evaluation to see how well the model generalizes to new , unseen data.

## Initial Preprocessing script:

This script loads the images, applies data augmentation, normalizes them, and saves the preprocessed images.

### The libraries we used for this script:

* **tensorflow:** To handle data loading, augmentation and processing.
* **os:** To handle the file paths.



### Steps in this script:

1. Defining the paths:

* Defines the paths for the train, validation and test directories.
* New directories for saving the preprocessed images gets created (Preprocessed\_Images).

1. Loading the datasets:

* The script uses ‘**image\_dataset\_from\_directory()’** to load the images from the train, validation and test directories.
* All the images are resized to (224, 224) pixels.
* The script loads the images in batches of 32 to efficiently train the model.

1. Normalization Layer:

* A rescaling layer gets applied to normalize the pixel values from [0, 255] to [0, 1] by dividing each value by 255.

1. Data gets Augmented:

* The following augmentation techniques are used to artificially expand the dataset by introducing variations and helps the model generalize better:
* Random Flip
* Random Rotation
* Random Zoom
* Random Brightness
* Random Contrast

1. Save preprocessed images:

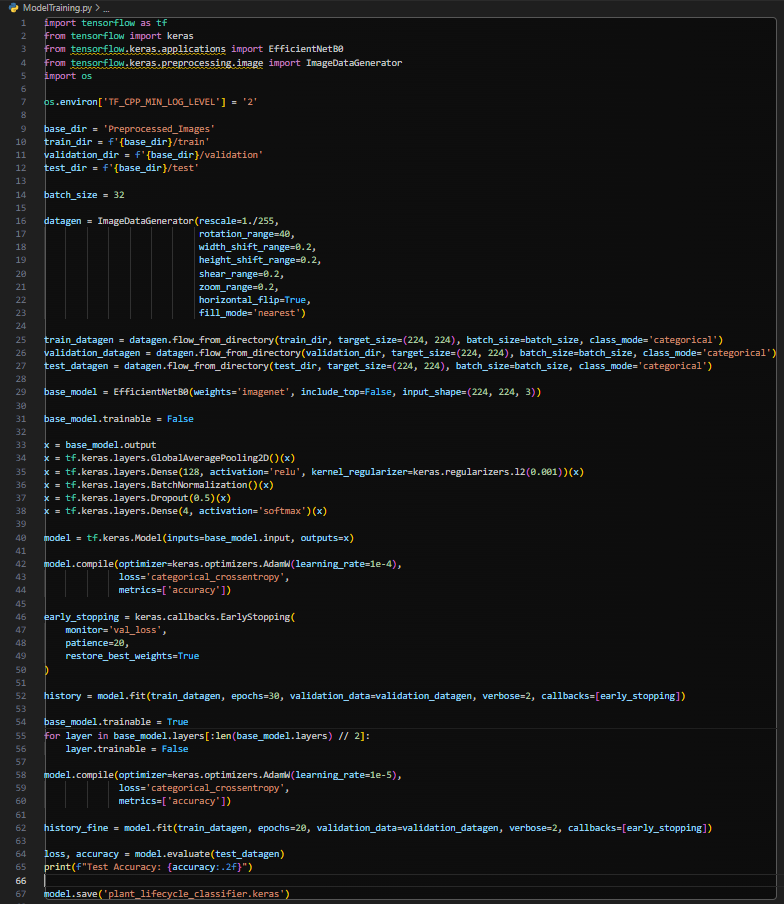
* The augmented images are saved in the “train” folder and the non-augmented images are saved in the “validation” and “test” folders.
* The script iterates through the dataset and saves each preprocessed image using ‘**tf.keras.preprocessing.image.save\_img()’.**

## Initial Model training script:

This script is responsible for creating the actual machine learning model using the preprocessed images. This script also evaluates the performance of the model. Transfer learning was used to leverage pre-trained features from EfficientNet, thereby speeding up and improving the model’s accuracy.

### The libraries we used for this script:

* **tensorflow, keras, EfficientNetB0, ImageDataGenerator:** For building, training and evaluating the model.



### Steps in this script:

1. The suppression of the warnings:

* During the installation of Tensorflow and Python something must have went wrong because we keep getting warnings when importing them.
* We wrote code to suppress the warnings to avoid excessive console logs.

1. Defining the paths:

* This script uses the preprocessed images from the “Preprocessed\_Images” folder.

1. Data generators:

* The script uses ImageDataGenerator to generate batches of image data with real time data augmentation.
* Random transformations like rotation, width/height shift, shear, zoom, etc. are used.

1. Loading EfficientNetB0:

* A pre-trained EfficientNetB0 model which has been trained on a large dataset (ImageNet) gets loaded. This model helps to quickly build an effective model sine it has already learned useful features.
* The “Include Top = False” part removes the classification layer, so we can add a custom head for our four classes.

1. Adding custom layers:

* We add a GlobalAveragePooling layer followed by several dense layers with dropout and batch normalization.
* The final layer has four output nodes with softmax activation to output probabilities for each of the four classes.

1. Compile the model:

* The script uses the AdamW optimizer with a learning rate of 1e-4.
* The loss function is categorical crossentropy, which is suitable for multi-class classification problems.

1. Early stopping callback:

* This callback stops training if the validation loss stops improving fro 20 epochs, preventing overfitting.

1. Training the model:

* The model gets trained for a total of 30 epochs using “train\_datagen” and evaluates it using “validation\_datagen”.

1. Fine-tuning the model:

* This unfreezes the base model layers, making them trainable.
* The model gets compiles with a lower learning rate (1e-5) to adjust the features learned by the EfficientNet base.
* The model gets fine-tuned for an additional 20 epochs.

1. Evaluating the model:

* The model gets evaluated on the test dataset and the accuracy gets printed.

1. Saving the model:

* The model gets saved to the “plant\_lifecycle\_classifier.keras” file.

## Initial results:

Initially the model the model started with a low accuracy of 23% for training. After a lot of epochs , the training accuracy did start to increase, but the validation accuracy remained stuck at around 31%. The official accuracy at the end of the initial run was 25%.

The issue we faced could have been because the model was underfitting, leading to the model not being able to learn enough from the data.

**Possible reasons for underfitting:**

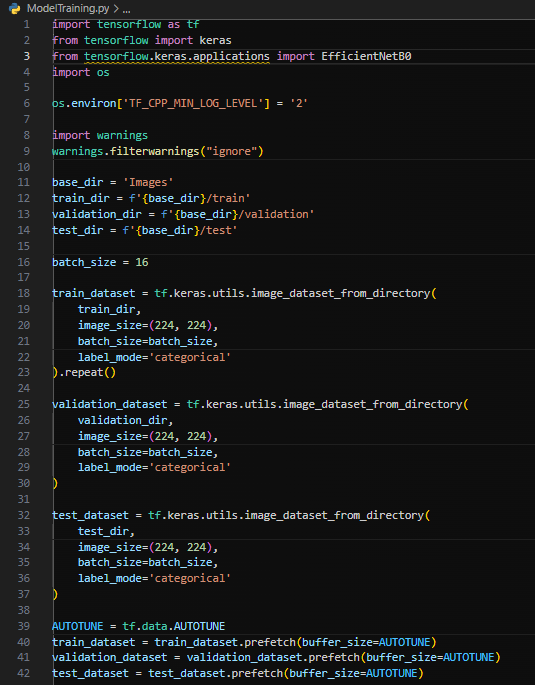
* The dataset size might be too small for a complex model like EfficientNet, which requires a lot of data.
* The data augmentation might have been too aggressive.

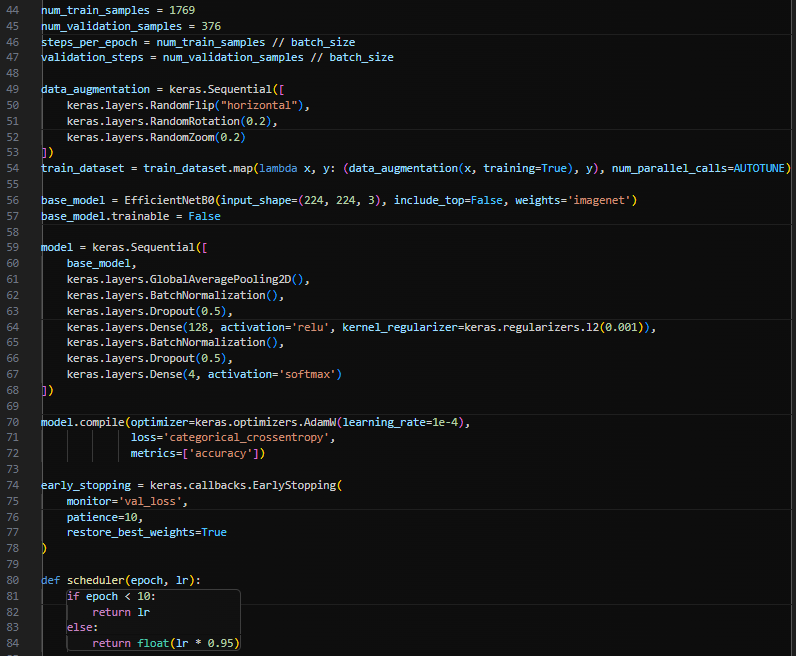
A screenshot of a computer program

Description automatically generated

# Second iteration:

## 2nd Iteration of the Model training script:





A screen shot of a computer program

Description automatically generated

### Different changes:

Here follows the changes we made in our second iteration of the machine learning model:

1. Data Augmentation changes:

* In the initial model we used the “ImageDataGenerator” to apply various types of augmentation, like shear, zoom and horizontal flipping. This led to more aggressive augmentation.
* In the second iteration we used custom “Keras Sequential” augmentation with RandomFlip, RandomRotation(0.2) and RandomZoom(0.2). This strategy of augmentation is less aggressive.

1. Loading datasets:

* The second iteration uses “image\_dataset\_from\_directory” from tensorflow to load the datasets instead of the “ImageDataGenerator”. This function creates “tf.data.Dataset” objects directly, which can be augmented, prefetched and mapped with greater control.

1. Prefetching for performance:

* In the initial version we did not use prefetching.
* In the second iteration we used prefetching so that the CPU can prepare the next batch while the GPU processes the current batch. This sped up the training time.

1. Batch size changes:

* We changed the batch size from 32 to 16. The reason for this is so the model can be a bit more stable when training.

1. Model architecture and changes to layers:

* In the second iteration we added an additional Batch Normalization layer before the output layer
* The second iteration still kept it’s dropout rate of 0.5.
* We also added more layers and dropout to increase the complexity.

1. Fine Tuning:

* In the initial version we fine tuned all of the layers of the base model after the initial training.
* With the second iteration we just fine tuned the last half of the base model layers by unfreezing them.
* This change helped balance the use of pre-trained knowledge with some more specific tuning tailored for our dataset.

1. Early stopping and learning rate scheduler:

* In the second iteration we reduced the early stopping patience to only 10 epochs to prevent overfitting. This means that the training will stop earlier if the model is not improving.
* We added a learning rate scheduler that reduced the learning rate by multiplying by 0.95 after 10 epochs. This helps improve convergence and slows down the learning rate as the training progresses, which helps the model settle into local minima more effectively.

1. Steps per epoch calculation:

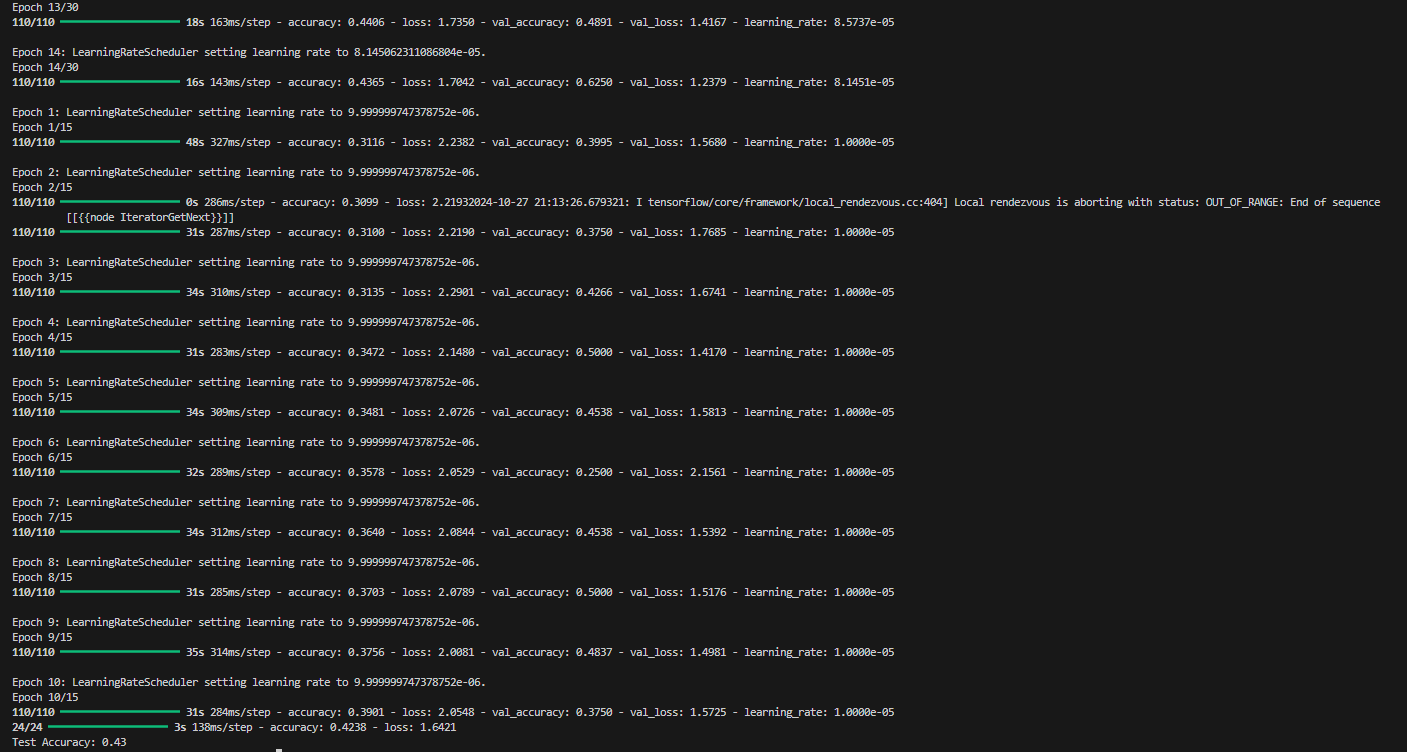
* In the initial version we used batches based on “ImageDataGenerator”.
* In the second version we defined “steps\_per\_epoch” and “validation\_steps” manually by calculating based on the number of samples and the batch size.
* We tried this approach to ensure that the entire dataset is utilized consistently. This can especially be useful if the dataset is not perfectly divisible by the batch size.

## Results of the 2nd iteration:

The overall accuracy of the second iteration did improve quite a bit with the accuracy standing at 43% now.

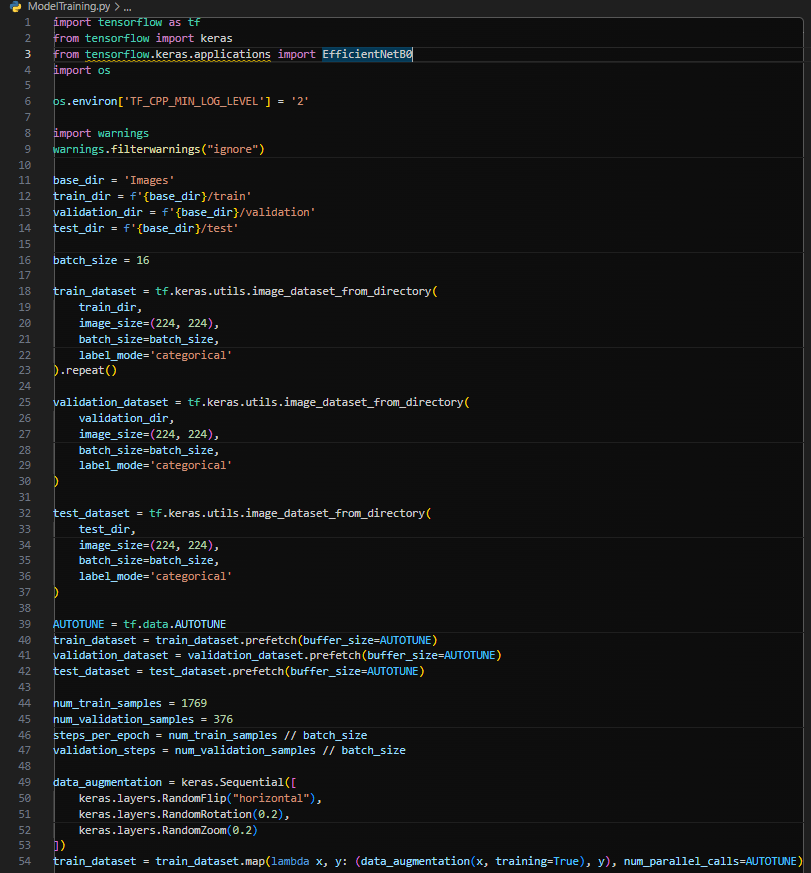
Possible reasons for this inaccuracy:

* The model could possibly be overfitting. We can make this assumption because the validation accuracy fluctuated a lot and did not consistently improve.
* The reduced learning rate decrease might have been too aggressive during the fine tuning. This could be preventing the model from making substantial progress.



# Third Iteration:

## 3rd Iteration of the Model Training script:



A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

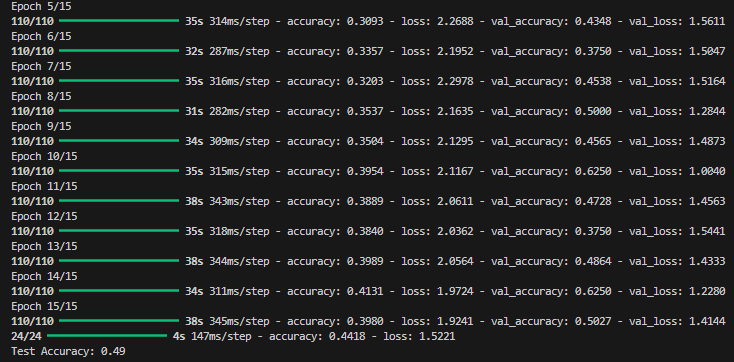
### Different changes:

1. Learning rate scheduler:

* In the third iteration we completely removed the learning rate scheduler. We did this to make the learning process a bit simpler, because the learning rate decrease in the second iteration might have been too aggressive.

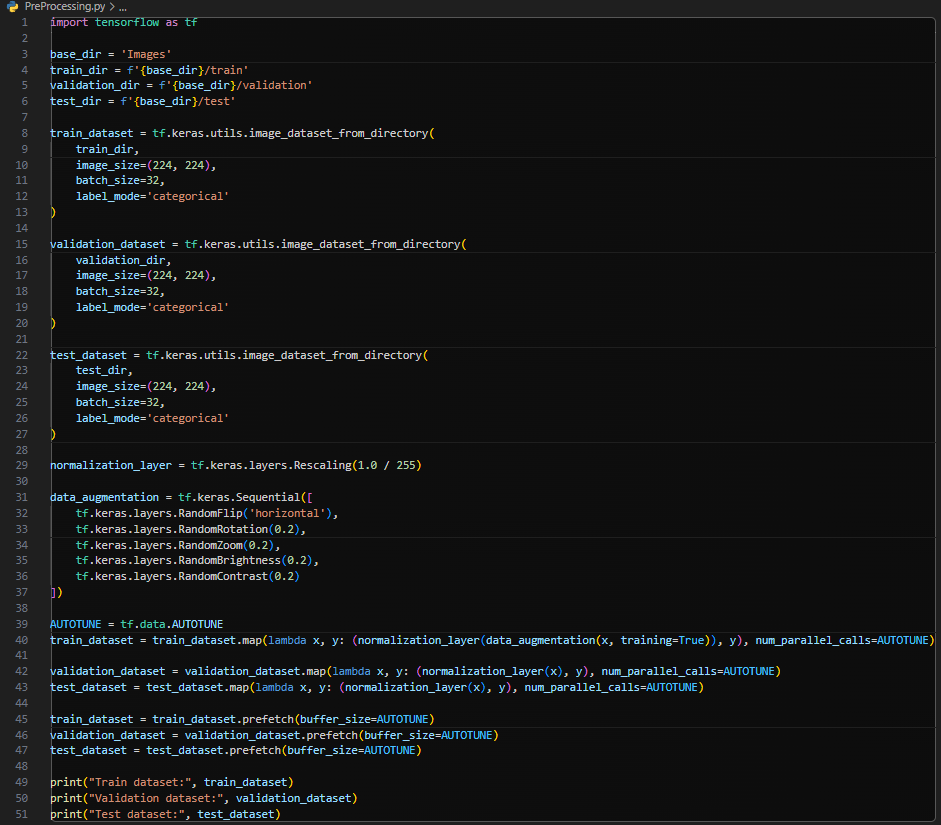
## Results of the 3rd iteration:

After removing the learning rate scheduler the model got an accuracy of 49%. Due to the model only improving with very minimal margins we decided to take a different approach and try to improve the preprocessing script.



# Fourth Iteration:

## 2nd Iteration of the Preprocessing script:



### Different changes:

1. The saving of the preprocessed images:

* In the second iteration we decided to not save the images to the disk. We added this change to improve efficiency, thereby speeding up the overall workflow.

1. Applying augmentation and normalization:

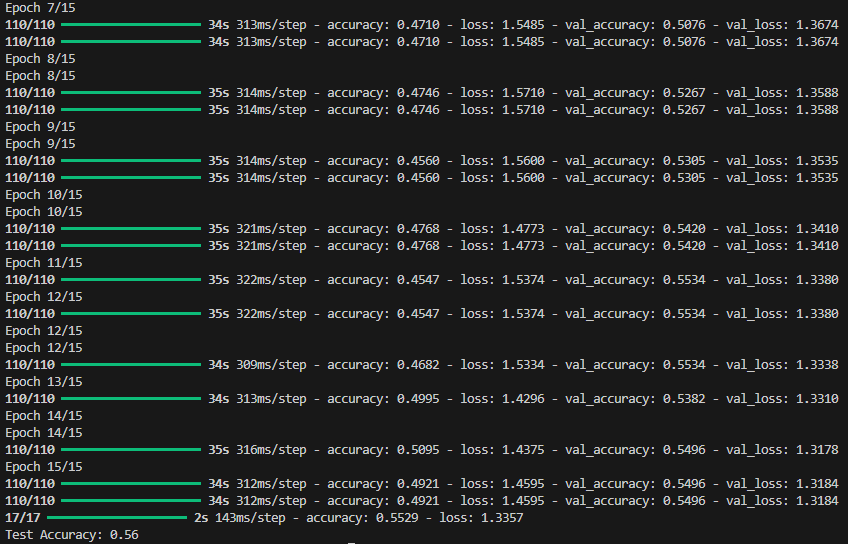
* In the initial script data augmentation was only applied when saving the training images, and the images were only normalized after augmentation.
* However in the second script we applied dynamic augmentation and normalization. This augmentation style is model flexible, providing different augmented samples during each training epoch.
* We reduced the augmentation parameters to be less aggressive:
* RandomRotation(0.2)
* RandomZoom(0.2)
* RandomBrightness(0.2)
* RandomContrast(0.2)

1. Dataset prefetching:

* We added prefetching in the second iteration to improve the training speed, which ultimately enhanced the model’s performance.

## Results of the 4th iteration:

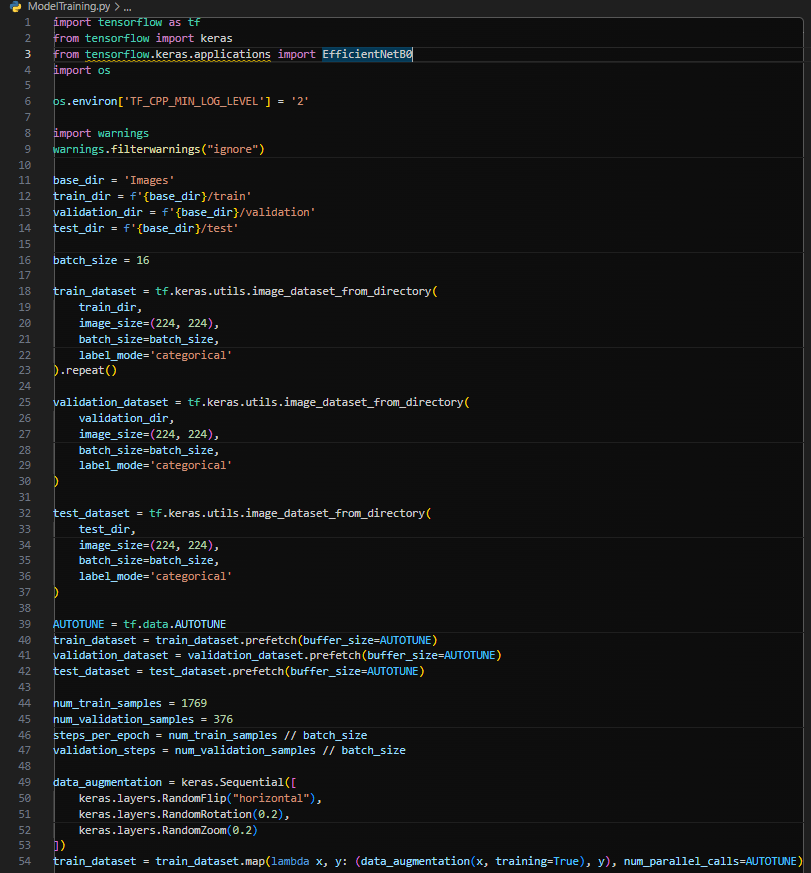
Applying a few changes to the preprocessing script did help improve the model’s accuracy to 56%.



# Final Model training and Preprocessing scripts:

After implementing a couple of other techniques, we were unable to produce a model that is more accurate than this one. This model gave us an accuracy of 56% and the proof of the results are mentioned above in the 4th iteration results.

## Model training script:



A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

## Preprocessing script:

A screen shot of a computer program

Description automatically generated