Flower Recognition Model

By Russie Shishmanov and Rangel Plachkov

Abstraction

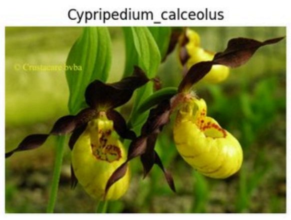
In this document, we will take you through how we made and perfected our flower recognition model, detailing various problems we encountered and how we solved them. The main objective is to develop a model that surpasses average human performance and implement it in a game where children and adults compete against the trained algorithm to see who can beat it. Throughout this process, we will explain how we addressed bias in our dataset, avoided overfitting, and expanded our data through augmentation techniques. We'll also cover our approach to training a CNN and customising VGG16 for flower recognition, along with the results of our efforts.

Introduction

Our aim is to create a machine learning model that is able to classify flowers, the threshold we started with was the average human accuracy which was estimated to be between 60% and 70%. We planned on starting with keras sequential model then testing a CNN model and finally VGG16. After comparing the results we will choose the best model for our use. We plan on using the models to make a fun and educational game, where users compete in recognizing flowers against the models and earn points if they outperform the model.

Data Set

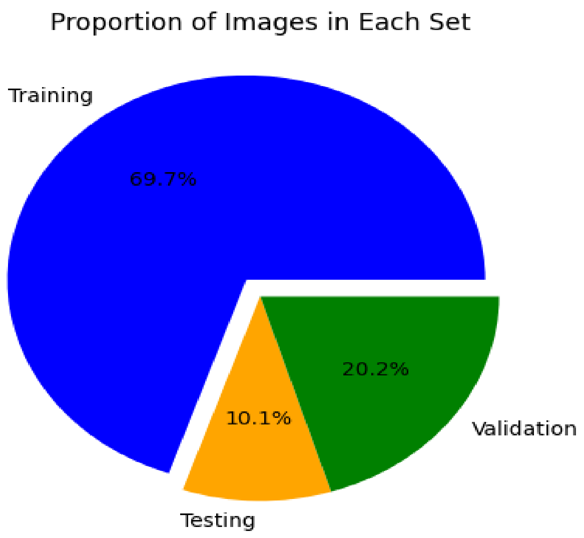
We use this data set, it contains about 1500 flower photos divided in 5 classes :



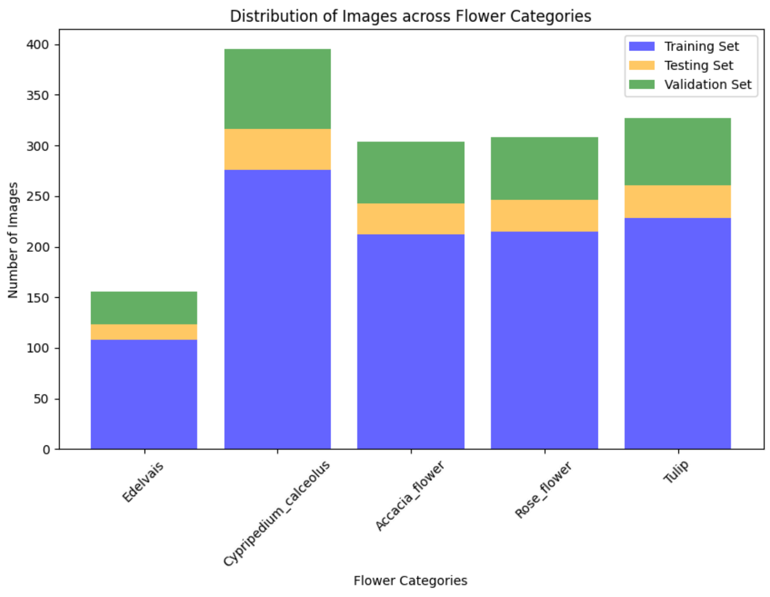
Possible problems in the data set are the under representation of edelweiss, and the not so big data set, which can cause overfitting. We can solve them using augmentation.

Sequential model

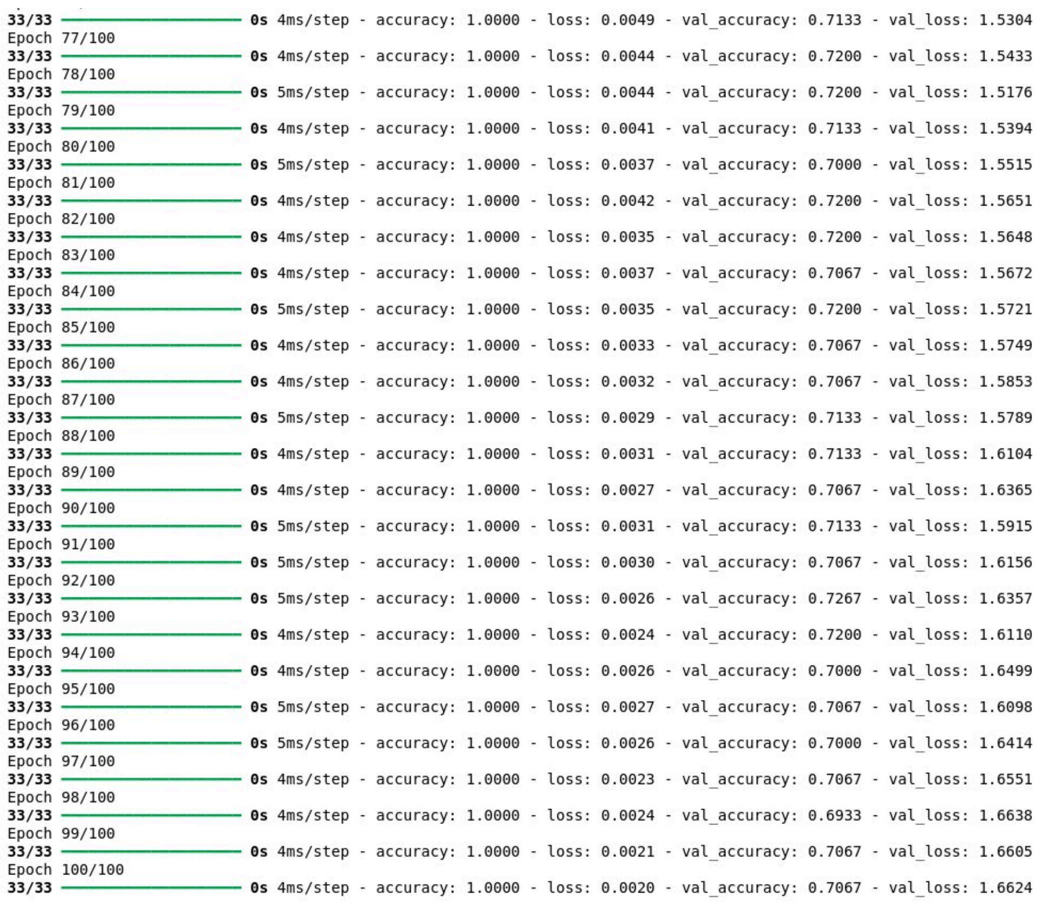
Starting with the RawData we first need to divide it into three groups. The first group contains the training data, the model will use this data to train on it, and make a better fit. The second group has the validation data, this will be used to fine tune and evaluate the model during development. And finally the test data, after completing the model we test it with this data, that it has never seen in order to see the models true accuracy. We decided to split the data like this:



And in each class we have :



Here we can clearly see the under representation of Edelvais. We train the model and after 100 epoch we see this:

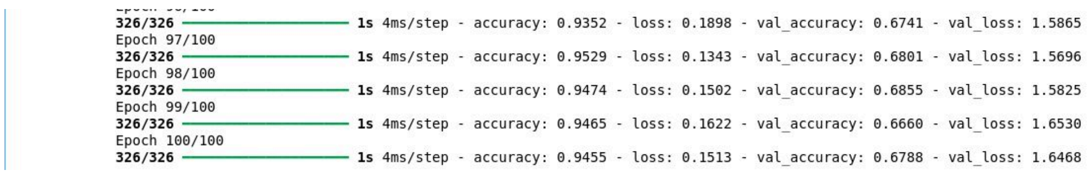


```
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss: 0.0049 - val_accuracy: 0.7133 - val_loss: 1.5304
Epoch 77/100
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss: 0.0044 - val_accuracy: 0.7200 - val_loss: 1.5433
Epoch 78/100
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss: 0.0044 - val_accuracy: 0.7200 - val_loss: 1.5176
Epoch 79/100
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss: 0.0041 - val_accuracy: 0.7133 - val_loss: 1.5394
Epoch 80/100
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss: 0.0037 - val_accuracy: 0.7000 - val_loss: 1.5515
Epoch 81/100
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss: 0.0042 - val_accuracy: 0.7200 - val_loss: 1.5651
Epoch 82/100
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss: 0.0035 - val_accuracy: 0.7200 - val_loss: 1.5648
Epoch 83/100
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss: 0.0037 - val_accuracy: 0.7067 - val_loss: 1.5672
Epoch 84/100
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss: 0.0035 - val_accuracy: 0.7200 - val_loss: 1.5721
Epoch 85/100
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss: 0.0033 - val_accuracy: 0.7067 - val_loss: 1.5749
Epoch 86/100
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss: 0.0032 - val_accuracy: 0.7067 - val_loss: 1.5853
Epoch 87/100
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss: 0.0029 - val_accuracy: 0.7133 - val_loss: 1.5789
Epoch 88/100
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss: 0.0031 - val_accuracy: 0.7133 - val_loss: 1.6104
Epoch 89/100
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss: 0.0027 - val_accuracy: 0.7067 - val_loss: 1.6365
Epoch 90/100
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss: 0.0031 - val_accuracy: 0.7133 - val_loss: 1.5915
Epoch 91/100
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss: 0.0030 - val_accuracy: 0.7067 - val_loss: 1.6156
Epoch 92/100
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss: 0.0026 - val_accuracy: 0.7267 - val_loss: 1.6357
Epoch 93/100
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss: 0.0024 - val_accuracy: 0.7200 - val_loss: 1.6110
Epoch 94/100
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss: 0.0026 - val_accuracy: 0.7000 - val_loss: 1.6499
Epoch 95/100
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss: 0.0027 - val_accuracy: 0.7067 - val_loss: 1.6098
Epoch 96/100
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 1.0000 - loss: 0.0026 - val_accuracy: 0.7000 - val_loss: 1.6414
Epoch 97/100
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss: 0.0023 - val_accuracy: 0.7067 - val_loss: 1.6551
Epoch 98/100
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss: 0.0024 - val_accuracy: 0.6933 - val_loss: 1.6638
Epoch 99/100
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss: 0.0021 - val_accuracy: 0.7067 - val_loss: 1.6605
Epoch 100/100
33/33 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 1.0000 - loss: 0.0020 - val_accuracy: 0.7067 - val_loss: 1.6624
```

Here we can see that we have about 70% accuracy but we have an overfit because we have 100% accuracy in our training data set. We need to fix this!

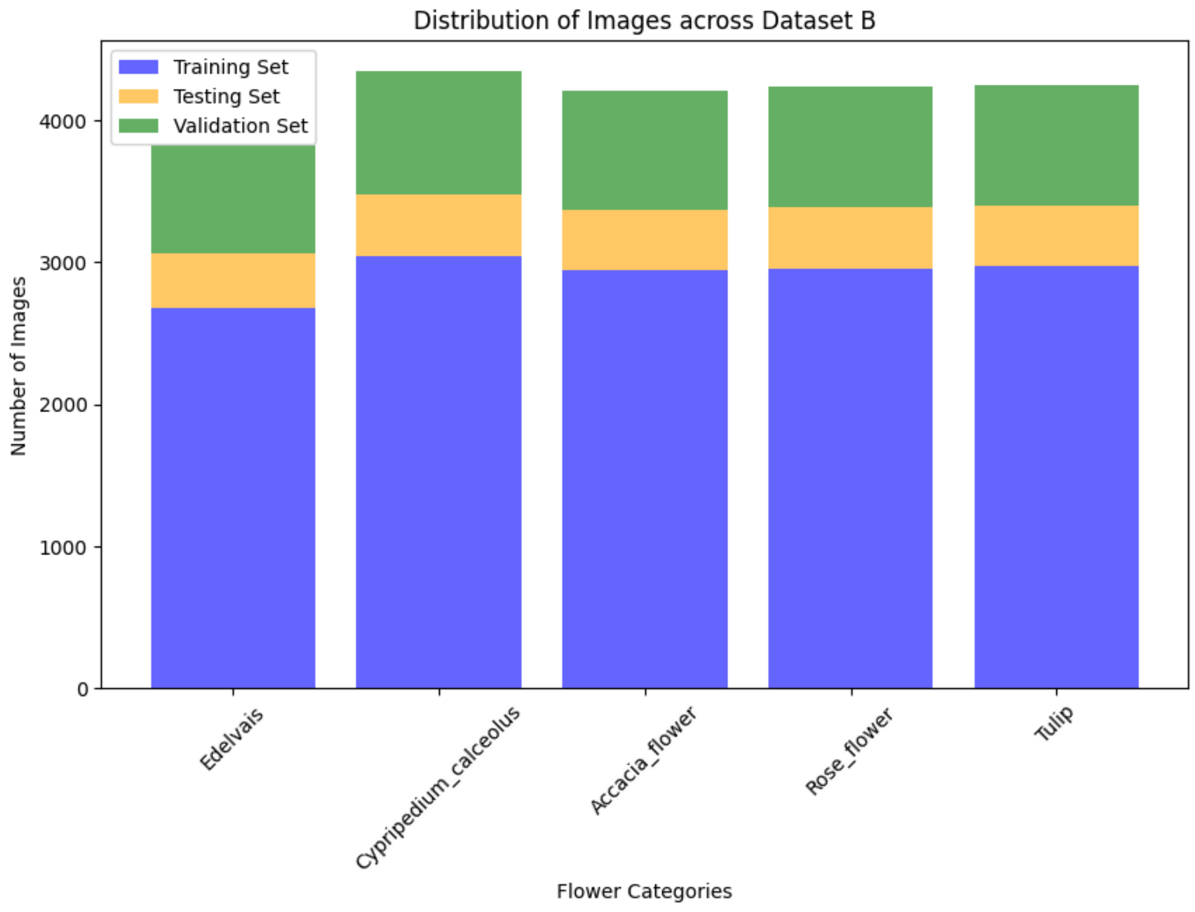
Battling Overfit Augmentation

We have to make our data set bigger, we can choose the easy path, blame the data set and go find a bigger one, but in some cases this can be very costly or even impossible, so we need to find a better way of fighting it. We will use augmentation, augmentation is when we purposely rotate, scale, flip(mirror), change brightness and introduce little noise in the data to get more data. So after creating 10 augmented version of each photo we now have a data set of about 15000 images, so lets try our model:

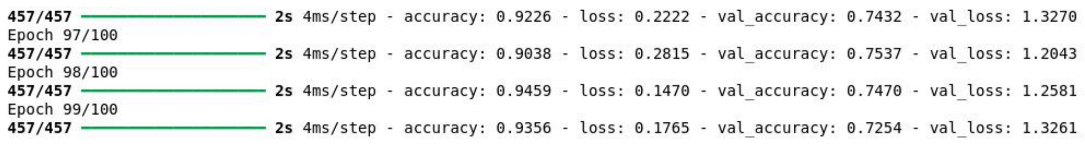


```
326/326 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9352 - loss: 0.1898 - val_accuracy: 0.6741 - val_loss: 1.5865
Epoch 97/100
326/326 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9529 - loss: 0.1343 - val_accuracy: 0.6801 - val_loss: 1.5696
Epoch 98/100
326/326 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9474 - loss: 0.1502 - val_accuracy: 0.6855 - val_loss: 1.5825
Epoch 99/100
326/326 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9465 - loss: 0.1622 - val_accuracy: 0.6660 - val_loss: 1.6530
Epoch 100/100
326/326 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9455 - loss: 0.1513 - val_accuracy: 0.6788 - val_loss: 1.6468
```

We see that we no longer have an overfit which is good but our model accuracy is lower at around 68%, this is because after augmenting we now have an even bigger underrepresentation of Edelvais so let's fix that, by first augmenting edelweiss so we have about equal representation and that scaling the whole data set. After doing this the date looks like this:

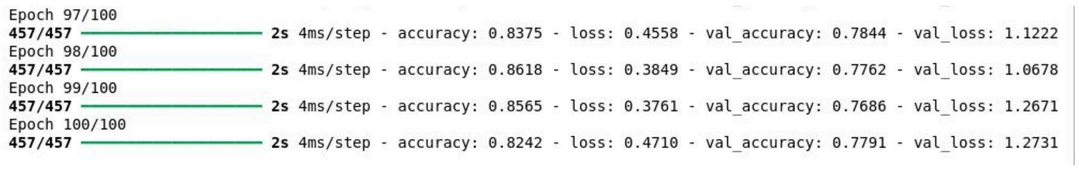


Now we have no bias so lets try our model again ot the new data :



```
457/457 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9226 - loss: 0.2222 - val_accuracy: 0.7432 - val_loss: 1.3270
Epoch 97/100
457/457 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9038 - loss: 0.2815 - val_accuracy: 0.7537 - val_loss: 1.2043
Epoch 98/100
457/457 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9459 - loss: 0.1470 - val_accuracy: 0.7470 - val_loss: 1.2581
Epoch 99/100
457/457 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9356 - loss: 0.1765 - val_accuracy: 0.7254 - val_loss: 1.3261
```

We can see that we now have about 75% accuracy which is better, so in all models we used 3 layer, what if we use more, after using more layers we managed to get the best results with 5 layers and a dropout layer :

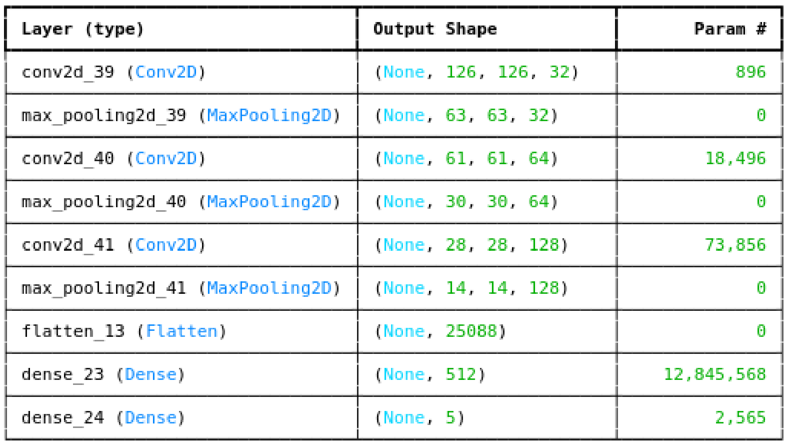


```
Epoch 97/100
457/457 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.8375 - loss: 0.4558 - val_accuracy: 0.7844 - val_loss: 1.1222
Epoch 98/100
457/457 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.8618 - loss: 0.3849 - val_accuracy: 0.7762 - val_loss: 1.0678
Epoch 99/100
457/457 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.8565 - loss: 0.3761 - val_accuracy: 0.7686 - val_loss: 1.2671
Epoch 100/100
457/457 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.8242 - loss: 0.4710 - val_accuracy: 0.7791 - val_loss: 1.2731
```

We have about 77% accuracy, and this is the best accuracy we managed to get with the sequential model, this good we managed to increase the accuracy with 7% and avoid overfitting.

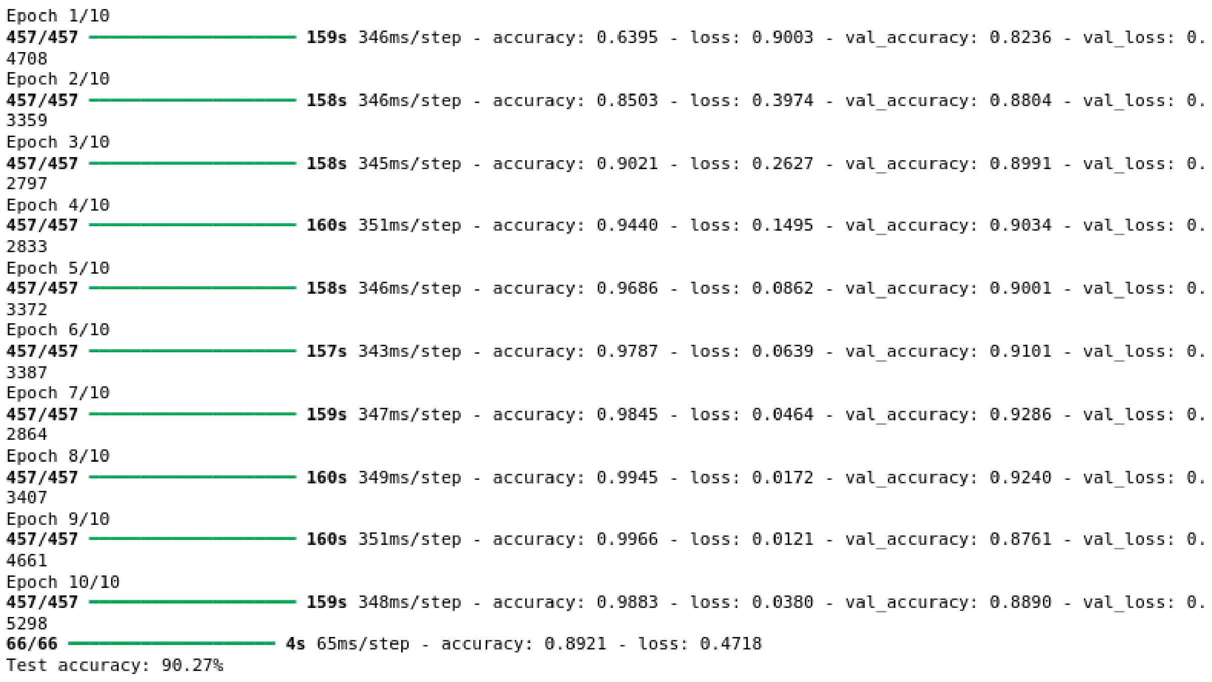
CNN model

After the sequential model, we decided to use a CNN model to see if we can increase our performance. The first and most basic CNN we test consists of: Conv2D layer followed by a MaxPooling2D layer, this construction repeats 3 times, and after that a flatten layer and one dense layer, and in the end the classification layer.



Layer (type)	Output Shape	Param #
conv2d_39 (Conv2D)	(None, 126, 126, 32)	896
max_pooling2d_39 (MaxPooling2D)	(None, 63, 63, 32)	0
conv2d_40 (Conv2D)	(None, 61, 61, 64)	18,496
max_pooling2d_40 (MaxPooling2D)	(None, 30, 30, 64)	0
conv2d_41 (Conv2D)	(None, 28, 28, 128)	73,856
max_pooling2d_41 (MaxPooling2D)	(None, 14, 14, 128)	0
flatten_13 (Flatten)	(None, 25088)	0
dense_23 (Dense)	(None, 512)	12,845,568
dense_24 (Dense)	(None, 5)	2,565

After running the CNN model on the first dataset we found some interesting results that indicate we have a data leak. Meaning that we augment images and then we split them in training, testing and validation. So when we validate and test the model, it has already seen some very similar images in the training set.



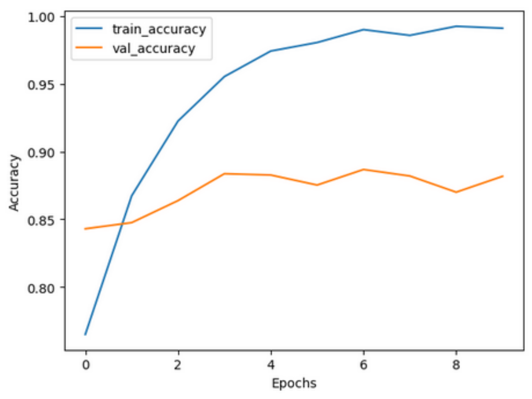
```
Epoch 1/10
457/457 ━━━━━━━━━━━━━━━━━━━━ 159s 346ms/step - accuracy: 0.6395 - loss: 0.9003 - val_accuracy: 0.8236 - val_loss: 0.
4708
Epoch 2/10
457/457 ━━━━━━━━━━━━━━━━━━━━ 158s 346ms/step - accuracy: 0.8503 - loss: 0.3974 - val_accuracy: 0.8804 - val_loss: 0.
3359
Epoch 3/10
457/457 ━━━━━━━━━━━━━━━━━━━━ 158s 345ms/step - accuracy: 0.9021 - loss: 0.2627 - val_accuracy: 0.8991 - val_loss: 0.
2797
Epoch 4/10
457/457 ━━━━━━━━━━━━━━━━━━━━ 160s 351ms/step - accuracy: 0.9440 - loss: 0.1495 - val_accuracy: 0.9034 - val_loss: 0.
2833
Epoch 5/10
457/457 ━━━━━━━━━━━━━━━━━━━━ 158s 346ms/step - accuracy: 0.9686 - loss: 0.0862 - val_accuracy: 0.9001 - val_loss: 0.
3372
Epoch 6/10
457/457 ━━━━━━━━━━━━━━━━━━━━ 157s 343ms/step - accuracy: 0.9787 - loss: 0.0639 - val_accuracy: 0.9101 - val_loss: 0.
3387
Epoch 7/10
457/457 ━━━━━━━━━━━━━━━━━━━━ 159s 347ms/step - accuracy: 0.9845 - loss: 0.0464 - val_accuracy: 0.9286 - val_loss: 0.
2864
Epoch 8/10
457/457 ━━━━━━━━━━━━━━━━━━━━ 160s 349ms/step - accuracy: 0.9945 - loss: 0.0172 - val_accuracy: 0.9240 - val_loss: 0.
3407
Epoch 9/10
457/457 ━━━━━━━━━━━━━━━━━━━━ 160s 351ms/step - accuracy: 0.9966 - loss: 0.0121 - val_accuracy: 0.8761 - val_loss: 0.
4661
Epoch 10/10
457/457 ━━━━━━━━━━━━━━━━━━━━ 159s 348ms/step - accuracy: 0.9883 - loss: 0.0380 - val_accuracy: 0.8890 - val_loss: 0.
5298
66/66 ━━━━━━━━━━━━━━━━━━━━ 4s 65ms/step - accuracy: 0.8921 - loss: 0.4718
Test accuracy: 90.27%
```

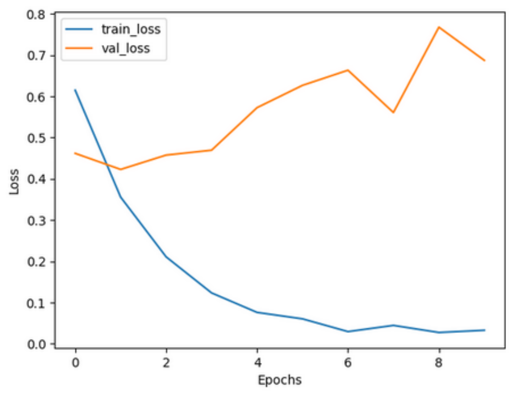
After fixing the data and making a new dataset which we first separate and then augment, meaning we no longer have a data leak. We get this results:

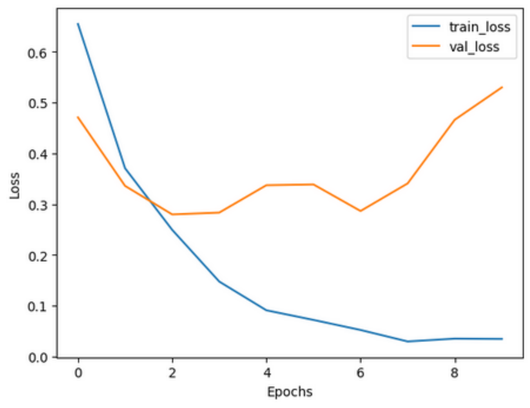
```
Epoch 1/10
457/457 ━━━━━━━━━━━━━━━━━━━━ 158s 344ms/step - accuracy: 0.6850 - loss: 0.8196 - val_accuracy: 0.8430 - val_loss: 0.
4617
Epoch 2/10
457/457 ━━━━━━━━━━━━━━━━━━━━ 156s 341ms/step - accuracy: 0.8586 - loss: 0.3791 - val_accuracy: 0.8476 - val_loss: 0.
4227
Epoch 3/10
457/457 ━━━━━━━━━━━━━━━━━━━━ 158s 346ms/step - accuracy: 0.9193 - loss: 0.2208 - val_accuracy: 0.8638 - val_loss: 0.
4575
Epoch 4/10
457/457 ━━━━━━━━━━━━━━━━━━━━ 159s 347ms/step - accuracy: 0.9533 - loss: 0.1312 - val_accuracy: 0.8837 - val_loss: 0.
4693
Epoch 5/10
457/457 ━━━━━━━━━━━━━━━━━━━━ 157s 344ms/step - accuracy: 0.9777 - loss: 0.0662 - val_accuracy: 0.8827 - val_loss: 0.
5726
Epoch 6/10
457/457 ━━━━━━━━━━━━━━━━━━━━ 159s 347ms/step - accuracy: 0.9871 - loss: 0.0441 - val_accuracy: 0.8753 - val_loss: 0.
6267
Epoch 7/10
457/457 ━━━━━━━━━━━━━━━━━━━━ 155s 340ms/step - accuracy: 0.9910 - loss: 0.0273 - val_accuracy: 0.8868 - val_loss: 0.
6634
Epoch 8/10
457/457 ━━━━━━━━━━━━━━━━━━━━ 157s 344ms/step - accuracy: 0.9925 - loss: 0.0249 - val_accuracy: 0.8820 - val_loss: 0.
5608
Epoch 9/10
457/457 ━━━━━━━━━━━━━━━━━━━━ 156s 342ms/step - accuracy: 0.9918 - loss: 0.0270 - val_accuracy: 0.8700 - val_loss: 0.
7678
Epoch 10/10
457/457 ━━━━━━━━━━━━━━━━━━━━ 156s 342ms/step - accuracy: 0.9949 - loss: 0.0171 - val_accuracy: 0.8817 - val_loss: 0.
6873
66/66 ━━━━━━━━━━━━━━━━━━━━ 4s 64ms/step - accuracy: 0.8073 - loss: 1.1088
Test accuracy: 81.13%
```

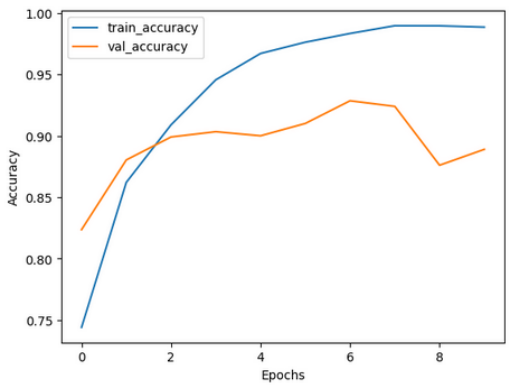
We can see that we have a very good validation result, and since this is our final model we run the test, and get an 81% accuracy which is significantly more than 66% we set to beat. So success!

Bonus the comparison between the data leaked model and the fix data model:


train_accuracy
val_accuracy
Accuracy
Epochs


train_loss
val_loss
Loss
Epochs


train_loss
val_loss
Loss
Epochs


train_accuracy
val_accuracy
Accuracy
Epochs