Artificial Neural Network and Deep Learning

Homework 1: Image Classification

Andrea Seghetto, Francesco Pastore, Oswaldo Morales

November 28, 2022

Introduction

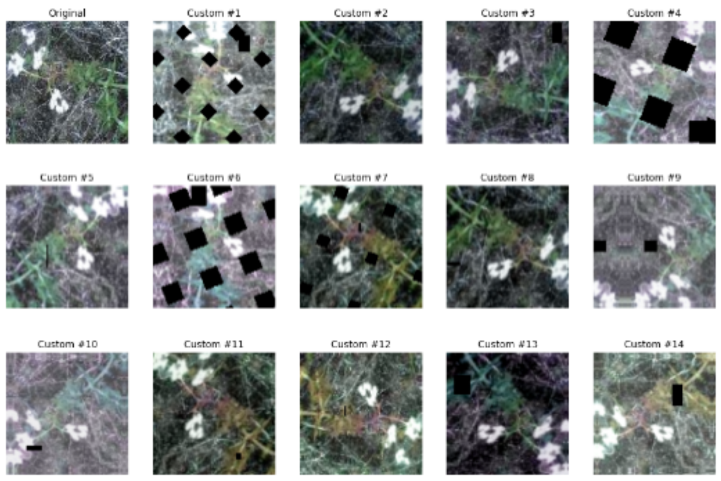
Our goal for the challenge was to design a neural network for image classification of different types of plants. The given dataset contains 3542 images divided in eight classes. We have tried standard CNN built by us and transfer learning with different models.

The zip delivered besides this report, contains three notebooks: one describing the first model implemented, one for the fine tuned EfficientNet model with data augmentation and one for the fine tuned EfficientNet without augmentation. The last two models are very similar architecturally but have different performances, more specifically the model without augmentation provides better results locally reaching a higher accuracy. However, on the test set the two models provide the same accuracy reported later.

Dataset analysis

The first problem that we identified was the imbalance between classes. Indeed, the first class has 186 images, the sixth 223, and the others around 500 images.

To overcome this issue we decided to implement oversampling by repeating the same images multiple times to reach a given amount. We executed many tests with different numbers of images. We also thought about undersampling but the dataset is so small that we found no advantage with it.



During training, we have added data augmentation to the model to avoid overfitting. We have tried different techniques by using the random functions available with Keras. The operations that we used are some basic ones like random zoom, flip, and changes in brightness and contrast. Also, we have tried with cutout, gridmask, hue, saturation, and the rand_augment provided by Keras.

Data augmentation

```
def custom_preprocessing(x):
    x = tf.image.random_hue(x, 0.2, seed=seed)
    x = tf.image.random_saturation(x, 0.8, 1.2, seed=seed)
    x = random_cutout(x, training=True)
    if random.random() <= 0.3:
        x = random_gridmask(x, training=True)
    if random.random() <= 0.3:
        x = rand_augment(x, training=True)
    return x

tf.keras.Sequential([
    layers.RandomFlip("horizontal_and_vertical", seed=seed),
    layers.RandomRotation(0.2, seed=seed),
    layers.RandomBrightness(0.2, seed=seed),
    layers.RandomContrast(0.2, seed=seed),
    layers.RandomZoom(0.1, seed=seed),
    layers.Lambda(lambda x: custom_preprocessing(x)),
])
```

	Specie 1	Specie 2	Specie 3	Specie 4	Specie 5	Specie 6	Specie 7	Specie 8
Original dataset	186	532	515	511	531	222	537	508
Training set + oversampling	168	479	464	460	478	200	484	458
Validation set 10%	18	53	51	51	53	22	53	50

We noticed very low performance during training like five or six minutes per epoch. So, we decided to apply some filters before training in order to have faster execution. During oversampling we applied the most expensive filters on the repeated images, while during training we used the lighter ones.

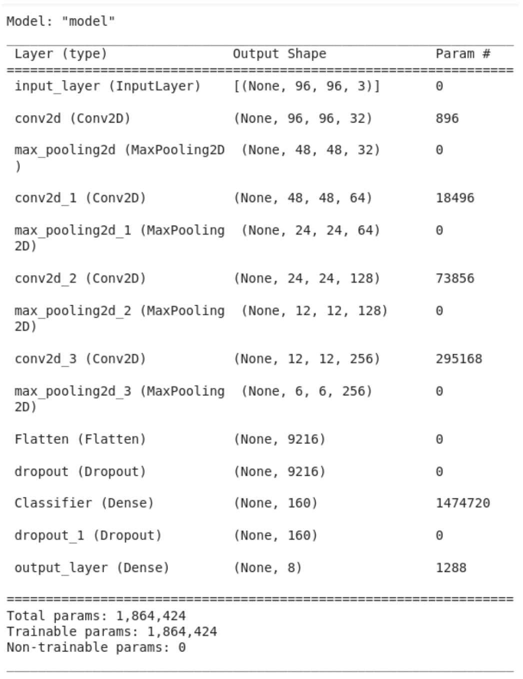
Furthermore, we had to split the dataset into a training set and a validation set. We decided to take 0.1 as the minimum amount that allowed us to obtain accurate results. We have tried also with 0.15 and 0.20 but we noticed worse performances, probably caused by the minor number of images available for training. We thought that the main reason for this problem was the small dimension of the dataset.

Custom model

Before testing a transfer learning model, we tried to implement a custom model to be trained from scratch. We tried with a model made at the beginning of one input layer followed by five pairs of convolutional layers and max pooling.

The first layer was composed of 32 filters, the second of 64, the third of 128, the fourth of 256 and the fifth of 512. Right after these blocks, we used a flatten layer the first time to train the model. Then we moved to the global average pooling layer because it provided better performance by reducing the number of parameters.

After these layers, we added one dense layer with a dropout layer before and after it. Lastly, we added an output layer for classification with softmax as activation function and eight units. We used Adam as an optimizer with the default learning rate of 0.001.



```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_layer (InputLayer)    [(None, 96, 96, 3)]       0

 conv2d (Conv2D)             (None, 96, 96, 32)        896

 max_pooling2d (MaxPooling2D  (None, 48, 48, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 48, 48, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 24, 24, 64)       0
 2D)

 conv2d_2 (Conv2D)           (None, 24, 24, 128)       73856

 max_pooling2d_2 (MaxPooling  (None, 12, 12, 128)      0
 2D)

 conv2d_3 (Conv2D)           (None, 12, 12, 256)       295168

 max_pooling2d_3 (MaxPooling  (None, 6, 6, 256)        0
 2D)

 Flatten (Flatten)           (None, 9216)              0

 dropout (Dropout)           (None, 9216)              0

 Classifier (Dense)          (None, 160)               1474720

 dropout_1 (Dropout)         (None, 160)               0

 output_layer (Dense)        (None, 8)                 1288

=================================================================
Total params: 1,864,424
Trainable params: 1,864,424
Non-trainable params: 0
_________________________________________________________________
```

After some tests we found out that the best dropout rate for both layers was 0.6 and that the best number of units for the hidden layer was 160. The model once uploaded scored ~0.5 while locally was around 0.6.

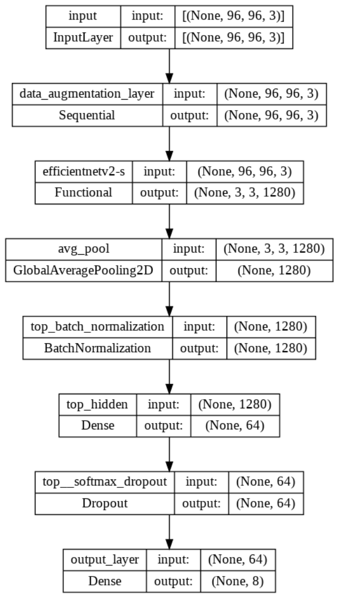
We have also tried to change the architecture of the model by adding or removing layers like the convolutional and the dense ones. We found out that the best performance was achieved by the architecture with only four convolutional and max pooling layers.

Transfer learning

We have tried different models for transfer learning to obtain better performances starting from VGG16 loaded with Imagenet weights. After some tests we scored around 0.65. Then, we tried other models like VGG19, MobileNet, Xception and finally EfficientNet that proved to be best for our dataset.

We have tried different versions of EfficientNet v1 and v2, and B0, B1 and B2 and also the S one. Eventually, we discovered that the second version of EfficientNet was better than the first one and among B0, B1, B2 and S, we chose B0 as the best compromise between time for training and the final accuracy.

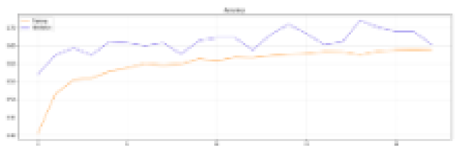
We have attempted to improve the performance during the transfer learning training by adding an additional dense layer before the output. We have made some tests with different numbers of units and with different dropout rates in order to find the best values for them.


input | input: | [(None, 96, 96, 3)]
InputLayer | output: | [(None, 96, 96, 3)]
data_augmentation_layer | input: | (None, 96, 96, 3)
Sequential | output: | (None, 96, 96, 3)
efficientnetv2-s | input: | (None, 96, 96, 3)
Functional | output: | (None, 3, 3, 1280)
avg_pool | input: | (None, 3, 3, 1280)
GlobalAveragePooling2D | output: | (None, 1280)
top_batch_normalization | input: | (None, 1280)
BatchNormalization | output: | (None, 1280)
top_hidden | input: | (None, 1280)
Dense | output: | (None, 64)
top__softmax_dropout | input: | (None, 64)
Dropout | output: | (None, 64)
output_layer | input: | (None, 64)
Dense | output: | (None, 8)

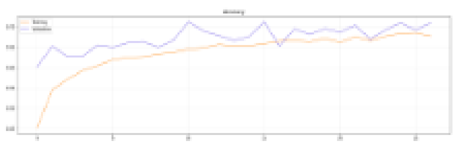
We discovered that running a second training just after the first one with a smaller learning rate was able to improve the accuracy over the dataset without overfitting it. The learning rates chosen were 0.001 and 0.0001 for the second one.

We have tried other models, like adding an extra hidden layer on the top block and checking its performance with different numbers of neurons on it (32, 64, 96, 128, and 256 neurons). We have also made some tests by adding L2 regularization on the loss function with different values for the regularization factor, but the results did not improve that much.

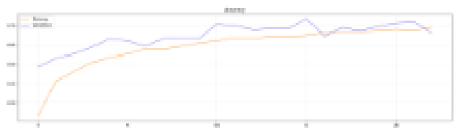
1x32 dense layer



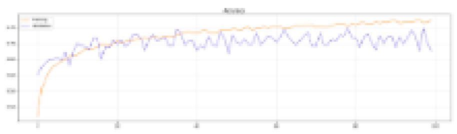
1x64 dense layer



1x96 dense layer



1x128 dense layer



Fine tuning

For the fine tuning phase we unfreeze some of the last layers of the model while we kept frozen the batch normalization layers and the previous ones. We tried with different numbers of layers and different learning rates still with Adam as optimizer.

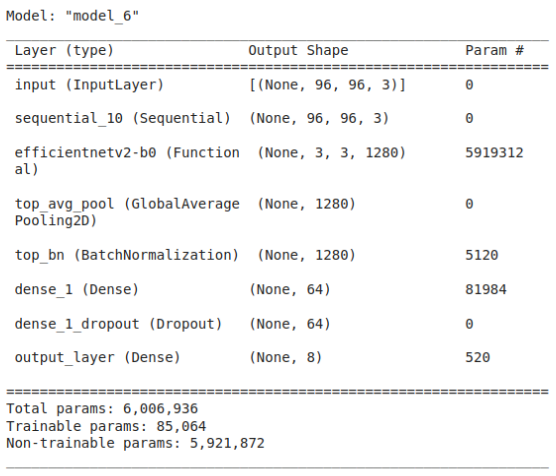
In the end the optimal value was 0.0001 for the learning rate and last 25 layers for the training. We have also discovered that by doing a second fine tuning with a smaller learning rate (0.00001) and a bigger number of layers unfrozen (35) the performance improved as it happened for the transfer learning phase.

After all these attempts we finally got to our best score that was ~0.75 on the test set while locally it reached an accuracy of ~0.8.

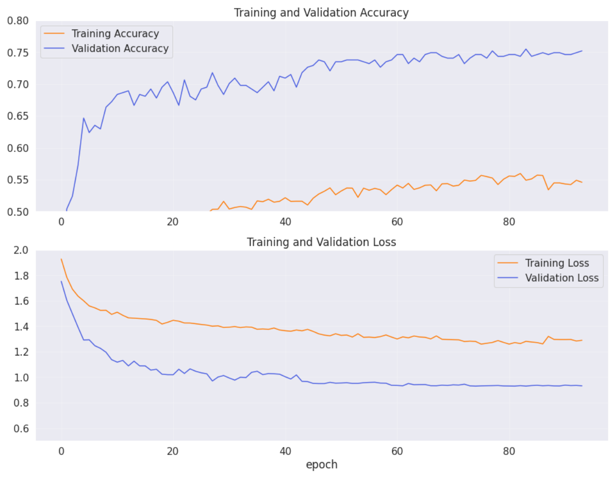
Conclusion

We found out that having more parameters and more complex models does not always mean reaching better performances. In fact, simple models can provide good results on small datasets while avoiding overfitting.

We figured out how important data exploration is as well as data preprocessing and data augmentation. Without this process the resulting model is not able to generalize well enough.



```
Model: "model_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input (InputLayer)          [(None, 96, 96, 3)]       0

 sequential_10 (Sequential)  (None, 96, 96, 3)         0

 efficientnetv2-b0 (Function  (None, 3, 3, 1280)       5919312
 al)

 top_avg_pool (GlobalAverage  (None, 1280)             0
 Pooling2D)

 top_bn (BatchNormalization)  (None, 1280)             5120

 dense_1 (Dense)             (None, 64)                81984

 dense_1_dropout (Dropout)   (None, 64)                0

 output_layer (Dense)        (None, 8)                 520

=================================================================
Total params: 6,006,936
Trainable params: 85,064
Non-trainable params: 5,921,872
_________________________________________________________________
```


Training and Validation Accuracy
Training Accuracy
Validation Accuracy
Training and Validation Loss
Training Loss
Validation Loss
epoch