ANNDL2020_Homework1

The Deepover Club – Capaldo, Laffi, Pacelli

Introduction

In this homework we were asked to face an Image Classification problem.

Our strategy can be divided in two macro phases: firstly we have successfully built a hand-made CNN, then, adopting Transfer Learning technique, we have tuned some VGG and ResNet variants obtaining significant higher results.

In both cases, we have followed a similar approach: first attempts were aimed at choosing the most promising CNN architecture. Then, once the model was fixed, we performed a hyperparameters tuning phase in which we tried to change some values in order to achieve better results. The tuning phase was made possible by comparing the performance plots (training and validation) of the different models.

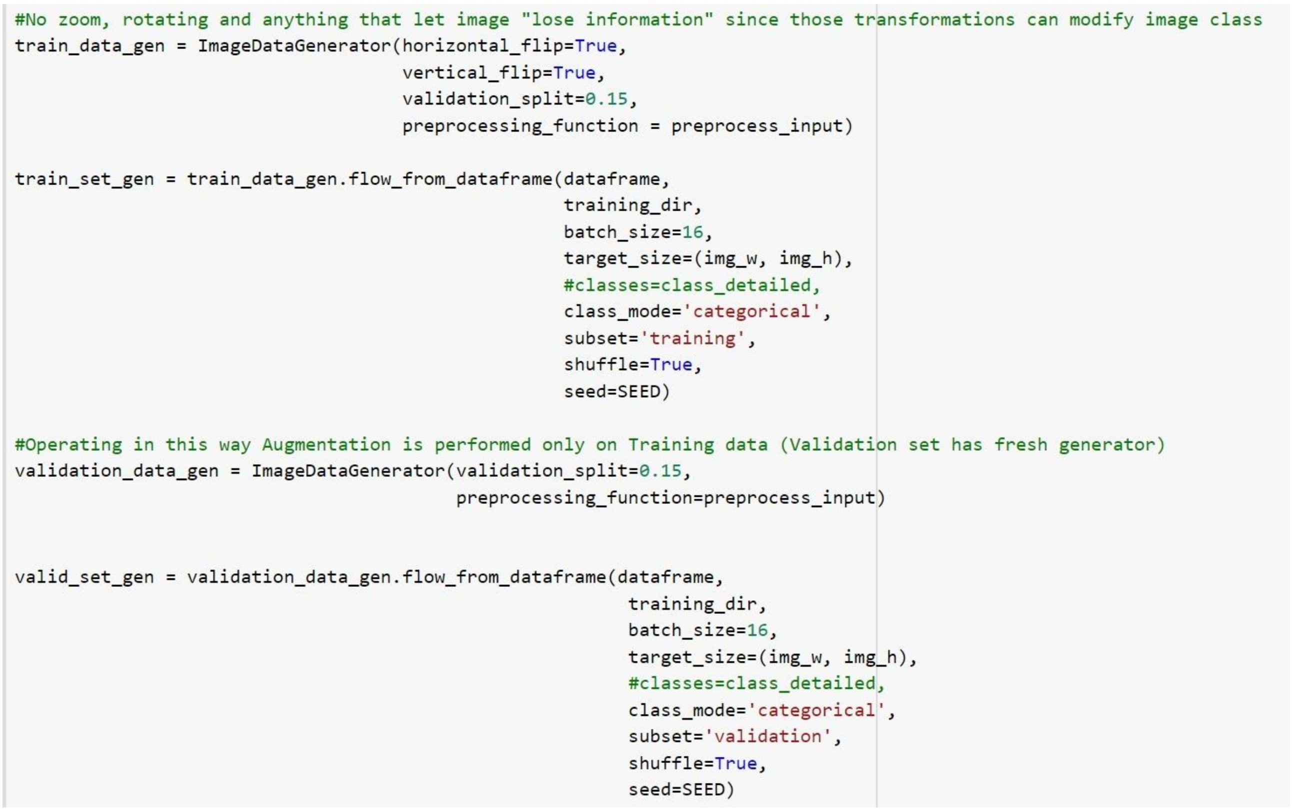
Base Settings

Our starting Dataset was divided only in two subsets: training and testing.
Then, to obtain more realistic results, we further splitted the training into training and validation sets. In order to do that, we imported *pandas* libraries that operate at folder-level (all files are on a shared Drive folder that is accessed through Colab).
Image-to-class correspondence has been loaded from dataset folder and imported:

```
with open(os.path.join(dataset_dir,"train_gt.json")) as train_label:
    train_dict = json.load(train_label)

dataframe = pd.DataFrame(train_dict.items())
dataframe.rename(columns = {0:'filename', 1:'class'}, inplace = True)
dataframe["class"] = dataframe["class"].astype(str)
dataframe = dataframe.sample(frac=1).reset_index(drop=True)
```

In the last row, `sample()` method shuffle data frame BEFORE performing splitting. This was added because, after a few tries, it has been noticed that inside the validation set, being the *.json* file ordered by class, all images were belonging to '1' class even if, later on in the code, though Keras built-in shuffling, the same operation is performed. The problem is that Keras shuffling is performed AFTER splitting operation.

Then images, only on training set, were augmented adopting some flipping but not by translating/rotating them, because such operations in this context could make images change class that need to be categorically avoided.

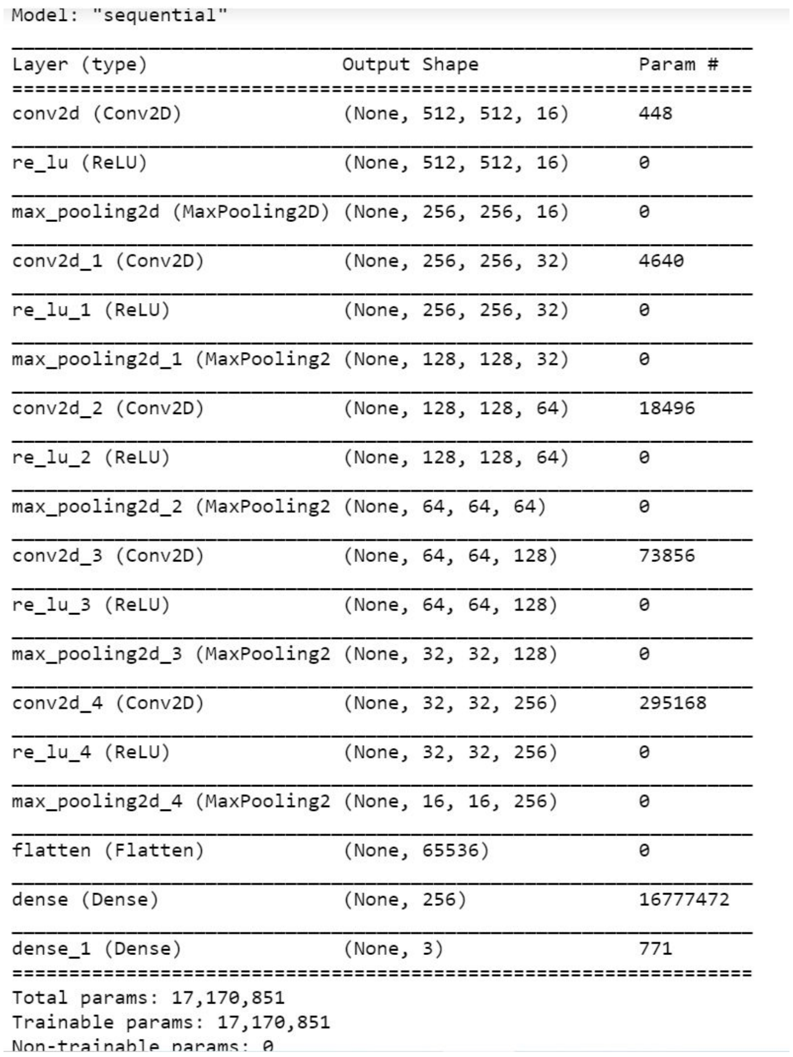


```
#No zoom, rotating and anything that let image "lose information" since those transformations can modify image class
train_data_gen = ImageDataGenerator(horizontal_flip=True,
                                    vertical_flip=True,
                                    validation_split=0.15,
                                    preprocessing_function = preprocess_input)

train_set_gen = train_data_gen.flow_from_dataframe(dataframe,
                                                   training_dir,
                                                   batch_size=16,
                                                   target_size=(img_w, img_h),
                                                   #classes=class_detailed,
                                                   class_mode='categorical',
                                                   subset='training',
                                                   shuffle=True,
                                                   seed=SEED)

#Operating in this way Augmentation is performed only on Training data (Validation set has fresh generator)
validation_data_gen = ImageDataGenerator(validation_split=0.15,
                                         preprocessing_function=preprocess_input)


valid_set_gen = validation_data_gen.flow_from_dataframe(dataframe,
                                                        training_dir,
                                                        batch_size=16,
                                                        target_size=(img_w, img_h),
                                                        #classes=class_detailed,
                                                        class_mode='categorical',
                                                        subset='validation',
                                                        shuffle=True,
                                                        seed=SEED)
```

Adam optimizer is used since it performs very well on classification tasks.

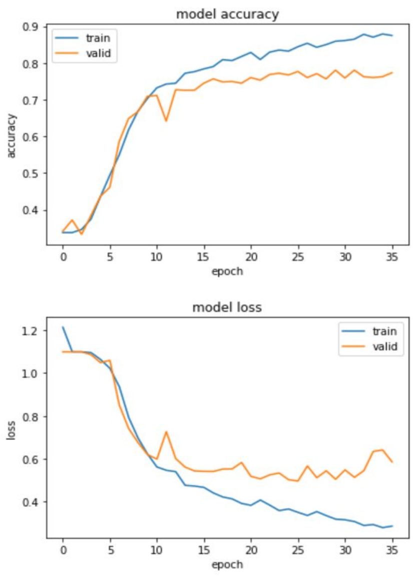
Hand-Made CNN

After some tries, we came up with this structure for our hand-made CNN: 15 convolutional layers with a “top” made of 2 FC layers.
The “convolutional part” (the one related to the feature extraction) is generated through a *for* loop: the macro block includes a Conv2d layer using 3x3 filters, a RuLU activation and a MaxPooling2d layer, repeated 5 times.
After building CNN, has been added the top classifier, which includes a first Dense layer with 512 neurons and another with a Softmax activation function which actually performs the classification task (3 neurons, one for each class).
Among these 2 parts we can find a flatten layer that makes the connection possible.



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d (Conv2D)              (None, 512, 512, 16)      448       
_________________________________________________________________
re_lu (ReLU)                 (None, 512, 512, 16)      0         
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 256, 256, 16)      0         
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 256, 256, 32)      4640      
_________________________________________________________________
re_lu_1 (ReLU)               (None, 256, 256, 32)      0         
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 128, 128, 32)      0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 128, 128, 64)      18496     
_________________________________________________________________
re_lu_2 (ReLU)               (None, 128, 128, 64)      0         
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 64, 64, 64)        0         
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 64, 64, 128)       73856     
_________________________________________________________________
re_lu_3 (ReLU)               (None, 64, 64, 128)       0         
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 32, 32, 128)       0         
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 32, 32, 256)       295168    
_________________________________________________________________
re_lu_4 (ReLU)               (None, 32, 32, 256)       0         
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 16, 16, 256)       0         
_________________________________________________________________
flatten (Flatten)            (None, 65536)             0         
_________________________________________________________________
dense (Dense)                (None, 256)               16777472  
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 771       
=================================================================
Total params: 17,170,851
Trainable params: 17,170,851
Non-trainable params: 0
```

Structure of the NN is summarized in the pic above. This gives us not so good performance (a test accuracy around 0.52 even if on training do not perform so bad).



We tried a bunch of different hyperparameter and at the most promising ones are:

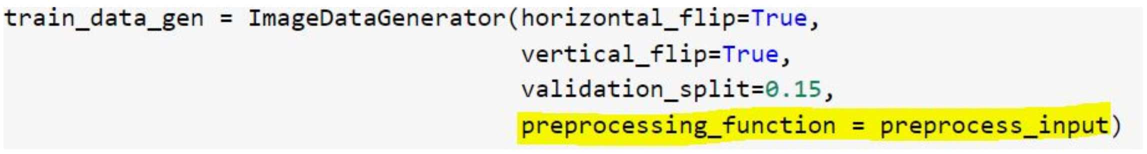
● Dense layer neuron #:	256
● Batch size:	64
● Learning rate:	1E-3

We tried both EarlyStopping and Dropout + Batch Normalization callbacks to face the overfitting problem and we noticed that EarlyStopping worked better.
At the end the best value for our hand-made CNN was about 0.65 of accuracy.

Transfer Learning approach

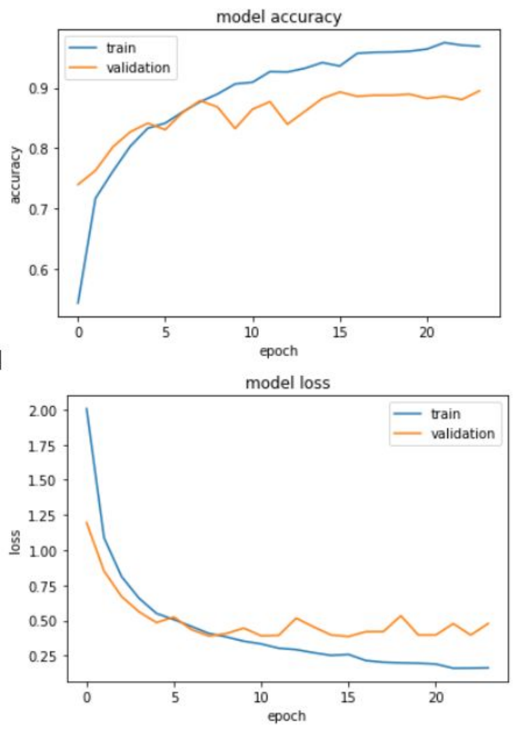
In order to achieve better performance, we moved on adopting Transfer Learning technique. So we searched online pre-trained models which could guarantee a good accuracy in performing a classification task.

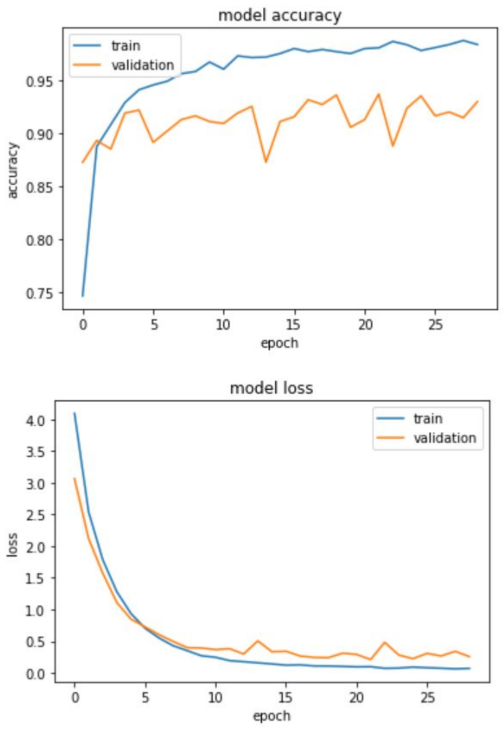
First of all, to implement Transfer Learning we had to preprocess images belonging both to the training and validation set.



```
train_data_gen = ImageDataGenerator(horizontal_flip=True,
                                    vertical_flip=True,
                                    validation_split=0.15,
                                    preprocessing_function = preprocess_input)
```

The line highlighted specifies preprocessing function of that specific NN, needed since CNN imported is pre-trained on images with that characteristics.
The first CNN we set up was **VGG16** which gave us a maximum of 0.76 of accuracy on the test set that was better (sadly for us) than ALL our Hand-Made CNN so we decided to move to a more powerful pretrained model.
Another improvement had been reached by adopting **VGG19** which had an accuracy of 0.86 (test set). After we tried **Inception_ResNet_v2** and **ResNet_50v2**, reaching an accuracy of 0,81 and 0,91 respectively.





Results: VGG19 *Results: ResNet_50v2*

The best results were reached using the **ResNet152_V2**. Here the snippet that shows how this model was loaded.

```
# Load pre trained model

pre_trained_model = ResNet152V2(input_shape=(img_w,img_h,3), pooling='avg', include_top=False, weights="imagenet")
```

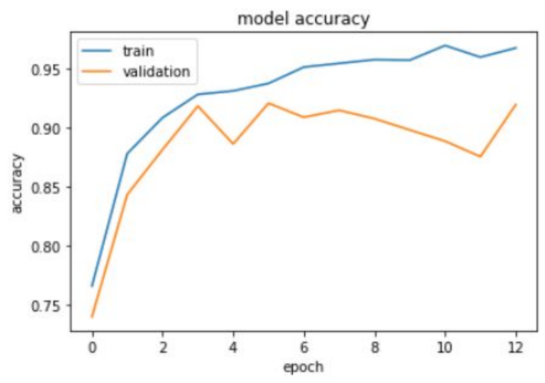
We took the pretrained weights from the “imagenet” dataset that nowadays is the biggest image dataset in the NN field. We prevented CNN from retraining the wights by freezing the first 23 layers of the NN.

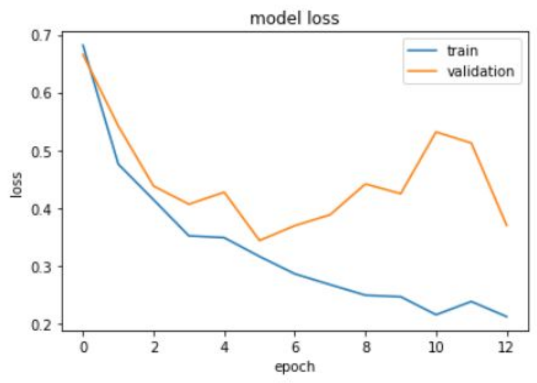
```
finetuning = True

if finetuning:
    freeze_until = 23 # layer from which we want to fine-tune

    for layer in pre_trained_model.layers[:freeze_until]:
        layer.trainable = False
else:
    pre_trained_model.trainable = False
```

Again after tried different hyperparameter, here we list most promising ones:

- 1st Dense layer: 256 neurons with ‘L2’ regularizer
- 2nd Dropout layer (rate of 0.6)
- Batch size: 16
- Learning rate = 1E-4

With this last NN we have reached an accuracy on test set of **0.93111**.


model accuracy
train
validation
accuracy
epoch


model loss
train
validation
loss
epoch