# SPATIAL PYRAMID POOLING IN CONVOLUTIONAL NETWORKS

*Sofia Cannizzaro e Maddalena Pismataro*

**ABSTRACT-** The aim of this project is to use a pooling strategy, known as "spatial pyramid pooling" (SPP), to avoid the need of a fixed-size input image, since deep convolutional networks (CNNs) require it. A fixed-size may reduce the recognition accuracy for the images or sub-images of an arbitrary size/scale, since they need, for instance, to be cropped. Adding a spatial pyramid pooling layer also improves the percent confidence in predictions.
Furthermore we dealt with the problem of overfitting, and overcame it introducing a data augmentation and a drop out layer, obtaining better results in terms of loss and accuracy.

## 1 INTRODUZIONE

There is a technical issue in the training and testing of the CNNs: the prevalent CNNs require a fixed input image size (e.g., *180x180*), which limits both the aspect ratio and the scale of the input image. When applied to images of arbitrary sizes, current methods mostly fit the input image to the fixed size, either via cropping or via warping. But the cropped region may not contain the entire object, while the warped content may result in unwanted geometric distortion. Recognition accuracy can be compromised due to the content loss or distortion.

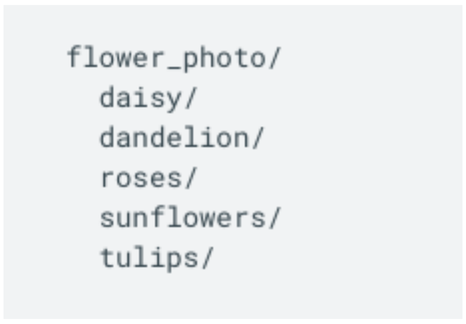
The fact that CNNs require a fixed input size is not related to the convolutional layers, since they can obtain feature maps from images of any size, but due to the fully connected layers which need a fixed-length input image size. To solve this problem, in this project we introduced the structure of spatial pyramid pooling that gives fixed output regardless of the input size. In this way it's possible to avoid cropping and wrapping, only using this kind of pooling in the deeper stage of the network.

Another advantage of SPP is using multi-level spatial bins, that, as the experiments can show, improves the accuracy.

In this project, to train our model, we considered images of 5 different kind of flowers. During the training we noticed, from the validation loss and accuracy values, a problem of overfitting. To solve this we added a Data Augmentation and a Dropout layers, that actually better the results.
Then, to test it, we used an ambiguous image of a red sunflower to check the goodness of our model.

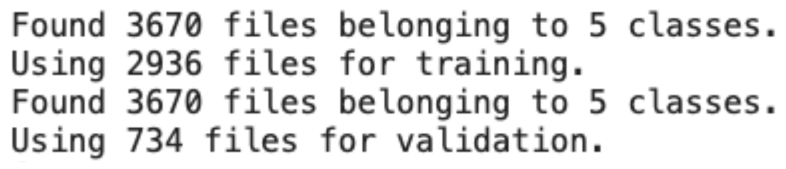
## 2 DATA UPLOADING AND PREPARATION

We considered a dataset composed by 3670 images of flowers, divided into 5 subdirectories depending on the kind of flowers they are:



```
flower_photo/
  daisy/
  dandelion/
  roses/
  sunflowers/
  tulips/
```

We downloaded it from "https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz", using the utility image_dataset_from_directory obtaing a td.data.Dataset.
The images are resized into a *180x180* images and the batch size is 32.
When training a machine learning model, we split our data into training and validation datasets. We will train the model on our training data and then evaluate how good the model is on unknown data - the validation set.
In our case the 80% of the images are used for the training and the remaining part for the validation.



```
Found 3670 files belonging to 5 classes.
Using 2936 files for training.
Found 3670 files belonging to 5 classes.
Using 734 files for validation.
```

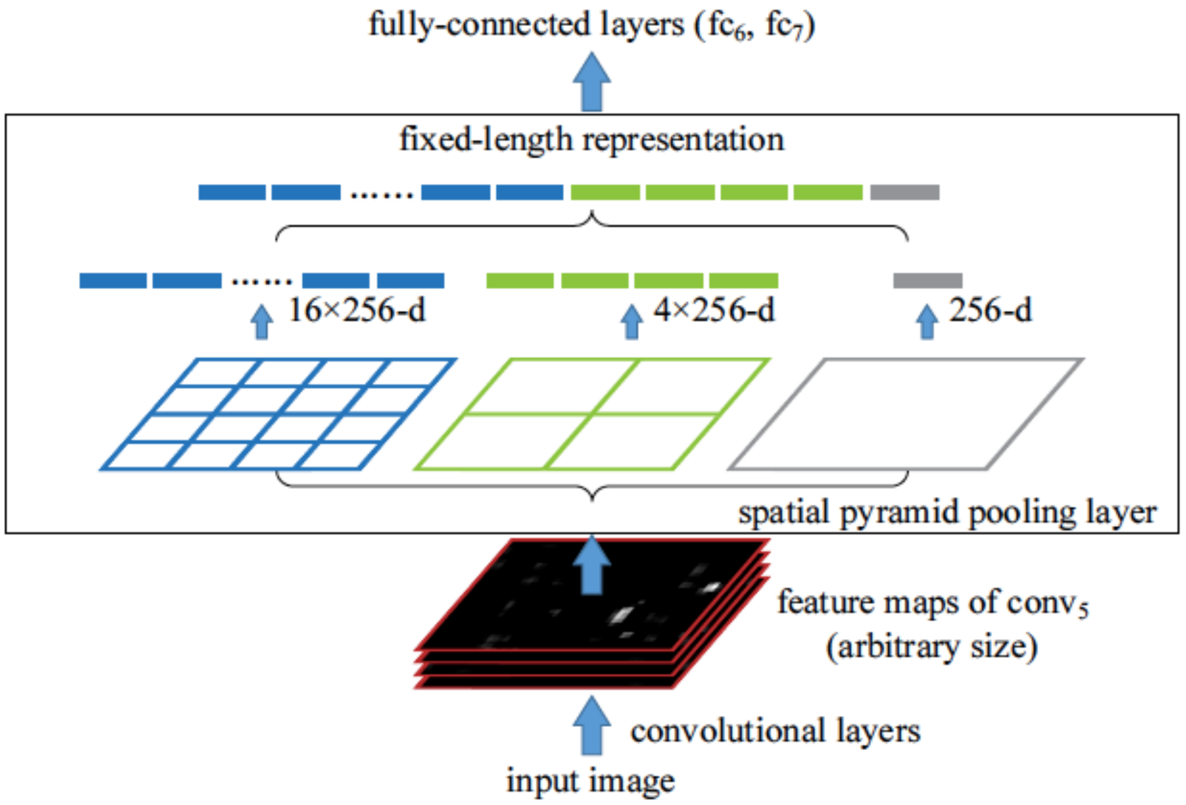
Image_batch is a tensor whose shape is (32, 180, 180, 3), that means 32 images of size 180x180x3 (the last dimension refers to the RGB color channels), while label_batch is a (32,) tensor that indicates the corresponding images labels.
It's important to notice that the labels are numbers between 0 and 4, indicating the different kinds of flowers, so it's better to use a Categorical Crossentropy as loss function (as we will discuss later on).

Usually the value of the RGB channels are between 0 and 255. This is not suitable for neural network hence it's advisable to reduce them. The most common choice is to normalize them in order to set values in [0,1]. This standardization can be obtained either through the map function or by adding a further layer (Normalization layer) in the model, as we did.

## 3 SPATIAL PYRAMID POOLING

Spatial pyramid pooling (known as SPP) is one of the most successful methods in computer vision and detection. That is because SPP is able to generate a fixed-length output regardless of the input size, while the sliding window pooling used in the convolutional neural network (only composed by convolutional, dense layers and "classical" pooling layers) cannot. Remember that pooling layers can also be considered as “convolutional”, in the sense that they are using sliding windows.

Thanks to the flexibility of input scales, spatial pyramid pooling can pool feature extracted at variable scales. For its properties, the spatial pyramid pooling layer can be added to a CNN replacing, for instance, a flatten layer, right before a fully connected one, as we did (see next chapter). This is because the convolutional layers accept arbitrary input sizes but they produce outputs of variable sizes, and the requirement of fixed sizes is only due to the fully-connected layers that demand fixed-length vectors as inputs.


fully-connected layers ($fc_6$, $fc_7$)
fixed-length representation
16×256-d
4×256-d
256-d
spatial pyramid pooling layer
feature maps of $conv_5$ (arbitrary size)
convolutional layers
input image

Supposing to have k filters for the output of the last convolutional layer, the outputs of the spatial pyramid pooling will be a kM-dimensional vectors, where M is the number of bins, which is fixed regardless of the image size. This is in contrast to the sliding window pooling, where the number of sliding windows depends on the input size.
Then for each filter we'll have to define M spatial bins that have sizes proportional to the image size. In each spatial bin, we pool the responses with maxpooling.

In our model we have 3 pyramid levels, as shown in the image, of respectively *1x1*, *2x2*, *4x4* spatial bins.

The fixed dimensional vectors becomes the input to the fully-connected layer.
It's important to notice that the advantages of SPP are independent of the convolutional network architectures used.

We first implemented a single-size training, considering a network taking a fixed-size input (resizing the images of the dataset into *180x180*). For an image with a given size, we can pre-compute the bin sizes needed for spatial pyramid pooling. If the feature map after the last convolutional layer has a size of *axa* and we have a  pyramid level of *nxn* bins,  then we can implement this pooling level as a sliding window pooling, where the window size is $win = \lceil a/n \rceil$ and stride $str = \lfloor a/n \rfloor$.

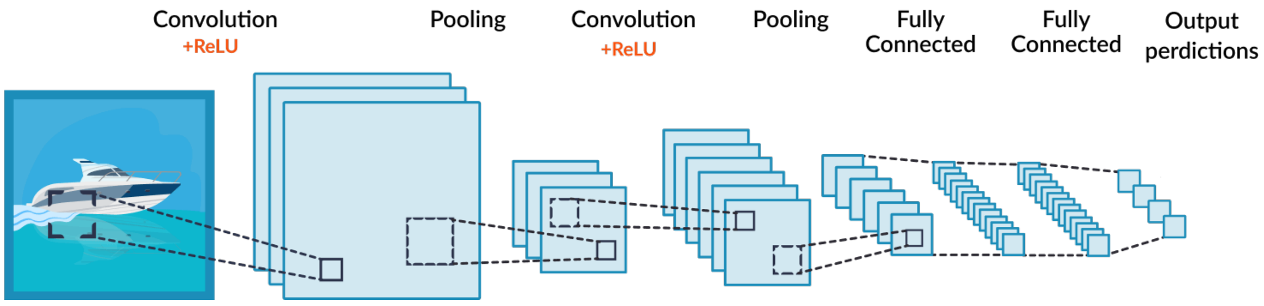
We implement this kind of layer as many times as the number of level-pyramid and then the dense layer will concatenate these outputs.
The main purpose of our single-size training is to enable the multi-level pooling behavior. As it can also be seen in the Result chapter, experiments show that this method allows a gain of accuracy.

Another way to increase accuracy and to help with the issue of varying image sizes is to consider a set of different pre-defined sizes for the training. In our case we considered images of size *180x180* and *224x224*. We implemented two fixed-size-input-networks that share parameters. The output of the spatial pyramid pooling layer of the 180-network has the same fixed length as the 224-network. Thus this 180-network has exactly the same parameters as the 224-network in each layer.

We could have also tested a variant using random input of size *sxs*, where *s* is uniformly sampled from [180; 224] at each epoch. At the testing stage, the strength of this method is the possibility  to apply SPP-net on images of any sizes.

# 4 THE MODEL

Now we are ready to describe the overall architecture of our CNN.
First of all there are a Normalization layer and a Data Augmentation layer that will be described later on.
We first considered a model where the input images size are specified at the beginning of the network. This model is composed by three convolutional layers and the ReLU non-linearity is applied to the output of all of them. Each of these convolutional layers is followed by a pooling layer, that can also be considered as "convolutional", in the sense that they are using sliding windows. Then there is a Flatten layer and the last two layers are fully connected.



To improve the accuracy of the model we use an SPP layer to replace the Flatten layer. Above all, this allows us to avoid the image-size specification at the beginning.

## CONVOLUTIONAL LAYER

A CNN is a cascade of convolutional filter intermediated with activation functions. The role of the ConvNet is to reduce the images into a form that is easier to process, without losing features which are critical for getting a good prediction

The objective of the Convolution Operation is to extract the high-level features from the input image. To work correctly, ConvNets need not be limited to only one Convolutional Layer: the first ConvLayer is responsible for capturing the Low-Level features (such as edges, color, gradient orientation, etc.), while the further layers indentify the High-Level features as well, giving us a network which has the good understanding of images in the dataset.
In our model all the convolutional layers have a 3x3 kernel and same padding, this means that the image dimension does not change. The first layer has 16 channels, the second 32 and the third one 64.
It's important to notice that the convolutional layer is simultaneous mapping spatial correlation and cross-channel correlation: the filter operates in all input channels in parallel.

## MAXPOOLING LAYER

Similar to the Convolutional Layer, the Pooling layer is responsible for reducing the dimension of the image (still selecting the dominant features). This is to decrease the computational power required to process the data.
In our model we use Max Pooling, that is done by applying a max filter to the portion of the image covered by the Kernel.

## DENSE LAYER

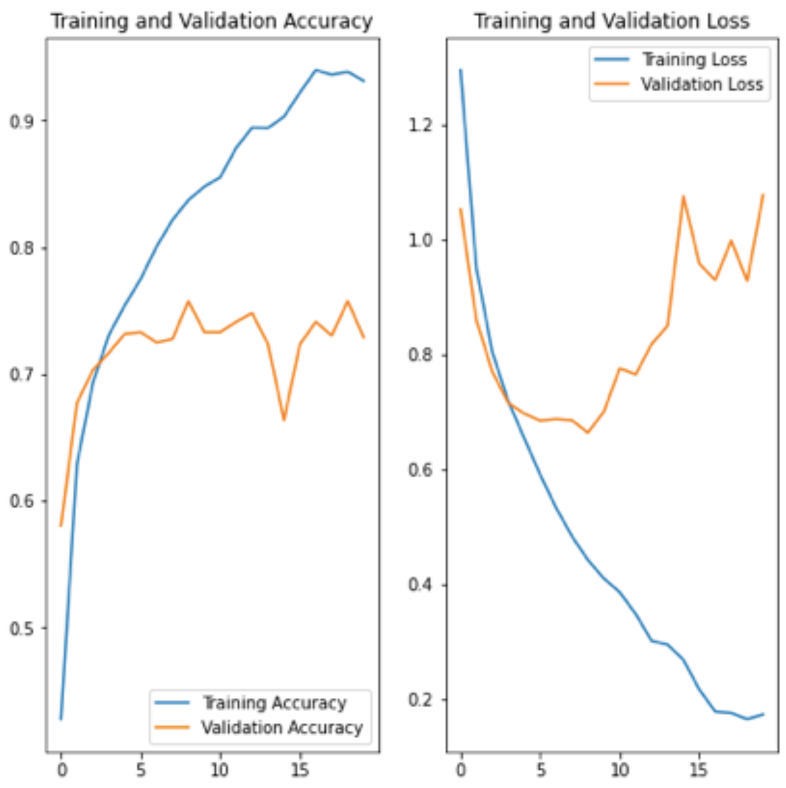
Fully connected layers are an essential component of Convolutional Neural Networks: they have been proven to be very successful in recognizing and classifying images. They are a way of learning non-linear combinations of the features represented by the output of the convolutional layers.
The fist dense layer of our network has 128 neurons and the last one has 5 neurons, one for each label, and it uses the softmax activation function (instead of ReLU) to get probabilities of the input being in a particular class.

## LOSS FUNCTION

When labels are mutually exclusive of each other, that is when each sample will belong only to one class, and you have integer targets instead of categorical vectors as targets, you can use sparse categorical crossentropy.
While traditional categorical crossentropy requires that your data is one-hot encoded and hence converted into categorical format , sparse categorical crossentropy It’s an integer-based version of this loss function. It’s the one used to compile our model because the label are number between 0 and 4 representing the 5 different flower.

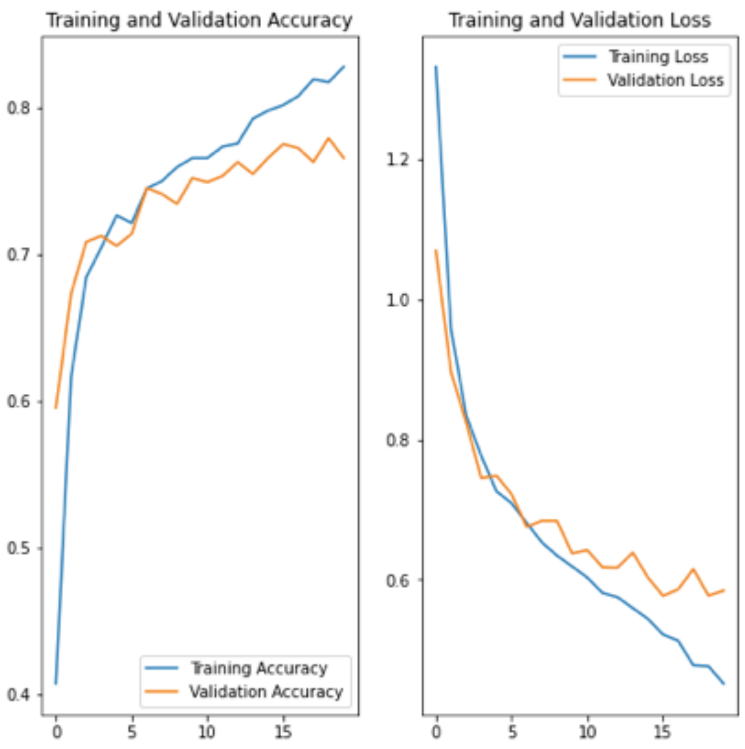
## 5 PROBLEM WITH OVERFITTING

At first, we trained the model without the data augmentation and dropout layers. After few performances, we noticed there was a problem of overfitting: while the train loss was actually decreasing, the validation loss was stacked into an high value. Plotting these values and the corresponding accuracies confirmed our suspect, as you can see below.



We found one of the reasons of overfitting in the size of our dataset: the number of images for the training was not sufficient. Thus, we looked for methods to increase the number of images, and we thought of data augmentation layer as the best solution. This layer creates new images starting from the existing ones, through random transformation as rotations and zooms.

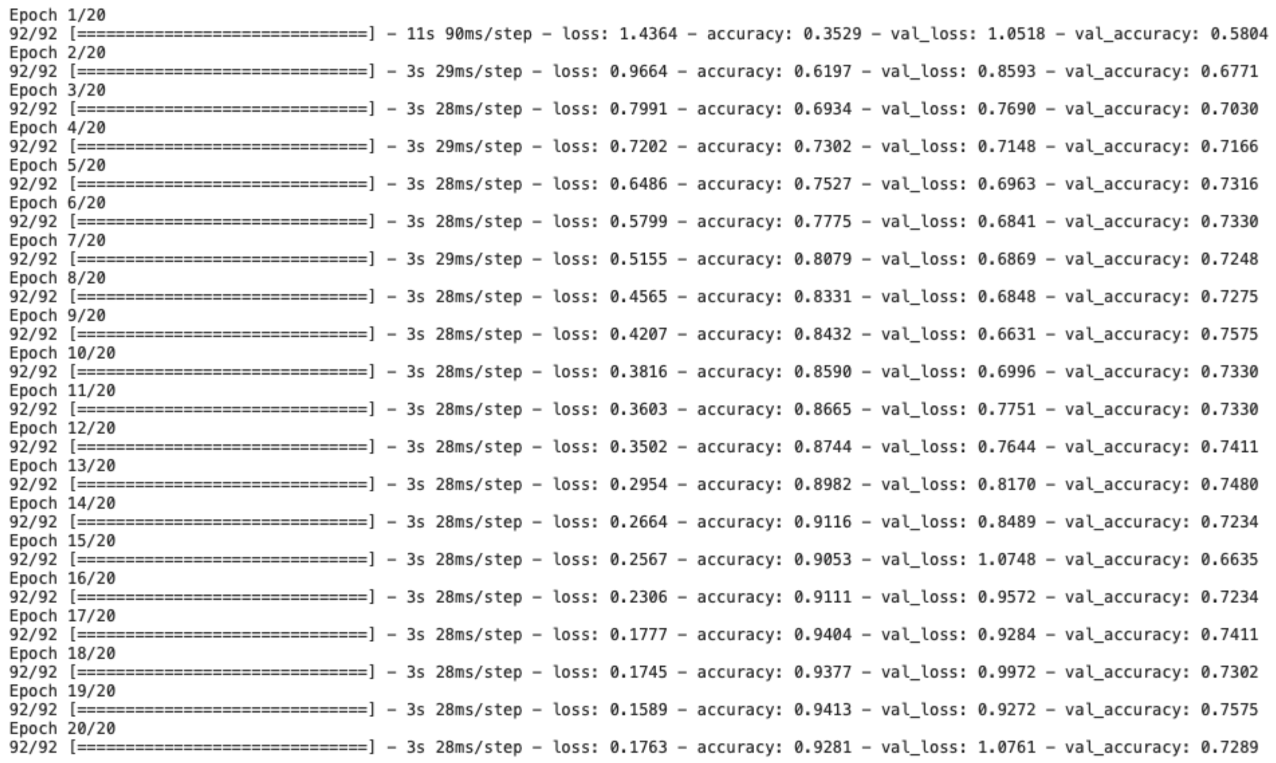
Another technique we decided to use in order to solve the problem of overfitting is using a Dropout layer. It's a regularization method that selects, randomly, a fraction of the outputs (20% in our case) and deletes it, or "drops it out". The aim is to make the training process noisy, forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs. This is the plot we obtained after using these techniques.



## 6 RESULT

### SPP WITH OVERFITTING

At first we trained the model without the Data Augmentation and Dropout layers. As mentioned before, it turned out that the loss was very low, which seemed to be a proof that the neural network was working correctly, but the validation loss was pretty high instead.

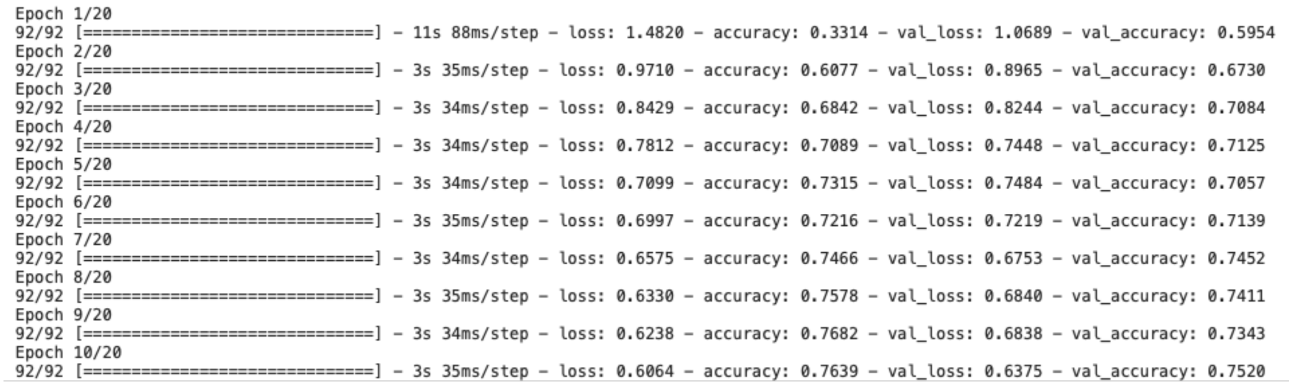


```
Epoch 1/20
92/92 [==============================] - 11s 90ms/step - loss: 1.4364 - accuracy: 0.3529 - val_loss: 1.0518 - val_accuracy: 0.5804
Epoch 2/20
92/92 [==============================] - 3s 29ms/step - loss: 0.9664 - accuracy: 0.6197 - val_loss: 0.8593 - val_accuracy: 0.6771
Epoch 3/20
92/92 [==============================] - 3s 28ms/step - loss: 0.7991 - accuracy: 0.6934 - val_loss: 0.7690 - val_accuracy: 0.7030
Epoch 4/20
92/92 [==============================] - 3s 29ms/step - loss: 0.7202 - accuracy: 0.7302 - val_loss: 0.7148 - val_accuracy: 0.7166
Epoch 5/20
92/92 [==============================] - 3s 28ms/step - loss: 0.6486 - accuracy: 0.7527 - val_loss: 0.6963 - val_accuracy: 0.7316
Epoch 6/20
92/92 [==============================] - 3s 28ms/step - loss: 0.5799 - accuracy: 0.7775 - val_loss: 0.6841 - val_accuracy: 0.7330
Epoch 7/20
92/92 [==============================] - 3s 29ms/step - loss: 0.5155 - accuracy: 0.8079 - val_loss: 0.6869 - val_accuracy: 0.7248
Epoch 8/20
92/92 [==============================] - 3s 28ms/step - loss: 0.4565 - accuracy: 0.8331 - val_loss: 0.6848 - val_accuracy: 0.7275
Epoch 9/20
92/92 [==============================] - 3s 28ms/step - loss: 0.4207 - accuracy: 0.8432 - val_loss: 0.6631 - val_accuracy: 0.7575
Epoch 10/20
92/92 [==============================] - 3s 28ms/step - loss: 0.3816 - accuracy: 0.8590 - val_loss: 0.6996 - val_accuracy: 0.7330
Epoch 11/20
92/92 [==============================] - 3s 28ms/step - loss: 0.3603 - accuracy: 0.8665 - val_loss: 0.7751 - val_accuracy: 0.7330
Epoch 12/20
92/92 [==============================] - 3s 28ms/step - loss: 0.3502 - accuracy: 0.8744 - val_loss: 0.7644 - val_accuracy: 0.7411
Epoch 13/20
92/92 [==============================] - 3s 28ms/step - loss: 0.2954 - accuracy: 0.8982 - val_loss: 0.8170 - val_accuracy: 0.7480
Epoch 14/20
92/92 [==============================] - 3s 28ms/step - loss: 0.2664 - accuracy: 0.9116 - val_loss: 0.8489 - val_accuracy: 0.7234
Epoch 15/20
92/92 [==============================] - 3s 28ms/step - loss: 0.2567 - accuracy: 0.9053 - val_loss: 1.0748 - val_accuracy: 0.6635
Epoch 16/20
92/92 [==============================] - 3s 28ms/step - loss: 0.2306 - accuracy: 0.9111 - val_loss: 0.9572 - val_accuracy: 0.7234
Epoch 17/20
92/92 [==============================] - 3s 28ms/step - loss: 0.1777 - accuracy: 0.9404 - val_loss: 0.9284 - val_accuracy: 0.7411
Epoch 18/20
92/92 [==============================] - 3s 28ms/step - loss: 0.1745 - accuracy: 0.9377 - val_loss: 0.9972 - val_accuracy: 0.7302
Epoch 19/20
92/92 [==============================] - 3s 28ms/step - loss: 0.1589 - accuracy: 0.9413 - val_loss: 0.9272 - val_accuracy: 0.7575
Epoch 20/20
92/92 [==============================] - 3s 28ms/step - loss: 0.1763 - accuracy: 0.9281 - val_loss: 1.0761 - val_accuracy: 0.7289
```

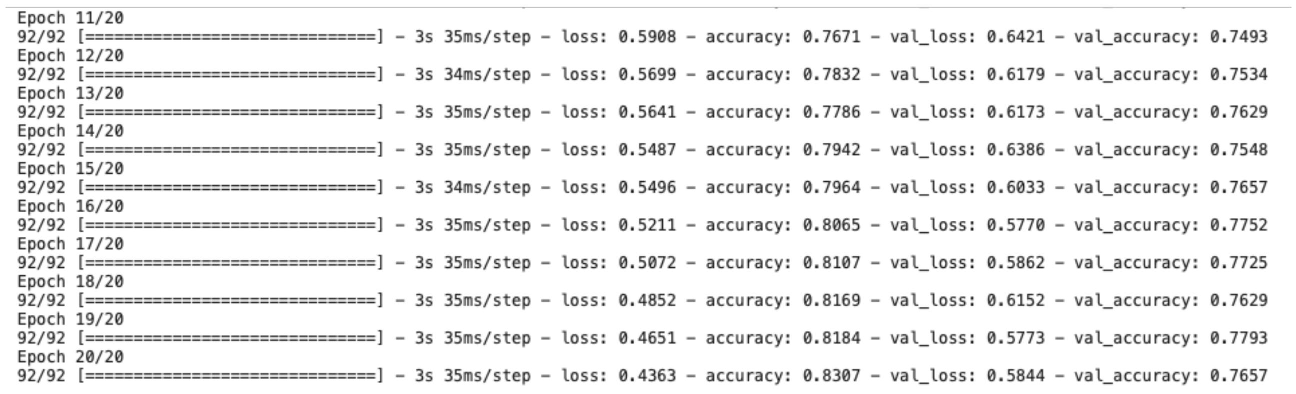
This means that, once the model is tested on an actual image, it predicts the kind of flower with an insufficiently good percent confidence.

### SPP

We then solved the problem of overfitting and thus obtained better results. Both the loss and, above all, the validation loss decreased:

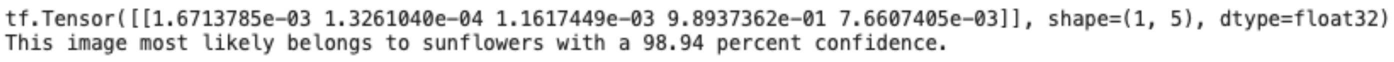


```
Epoch 1/20
92/92 [==============================] - 11s 88ms/step - loss: 1.4820 - accuracy: 0.3314 - val_loss: 1.0689 - val_accuracy: 0.5954
Epoch 2/20
92/92 [==============================] - 3s 35ms/step - loss: 0.9710 - accuracy: 0.6077 - val_loss: 0.8965 - val_accuracy: 0.6730
Epoch 3/20
92/92 [==============================] - 3s 34ms/step - loss: 0.8429 - accuracy: 0.6842 - val_loss: 0.8244 - val_accuracy: 0.7084
Epoch 4/20
92/92 [==============================] - 3s 34ms/step - loss: 0.7812 - accuracy: 0.7089 - val_loss: 0.7448 - val_accuracy: 0.7125
Epoch 5/20
92/92 [==============================] - 3s 34ms/step - loss: 0.7099 - accuracy: 0.7315 - val_loss: 0.7484 - val_accuracy: 0.7057
Epoch 6/20
92/92 [==============================] - 3s 35ms/step - loss: 0.6997 - accuracy: 0.7216 - val_loss: 0.7219 - val_accuracy: 0.7139
Epoch 7/20
92/92 [==============================] - 3s 34ms/step - loss: 0.6575 - accuracy: 0.7466 - val_loss: 0.6753 - val_accuracy: 0.7452
Epoch 8/20
92/92 [==============================] - 3s 35ms/step - loss: 0.6330 - accuracy: 0.7578 - val_loss: 0.6840 - val_accuracy: 0.7411
Epoch 9/20
92/92 [==============================] - 3s 34ms/step - loss: 0.6238 - accuracy: 0.7682 - val_loss: 0.6838 - val_accuracy: 0.7343
Epoch 10/20
92/92 [==============================] - 3s 35ms/step - loss: 0.6064 - accuracy: 0.7639 - val_loss: 0.6375 - val_accuracy: 0.7520
```



```
Epoch 11/20
92/92 [==============================] - 3s 35ms/step - loss: 0.5908 - accuracy: 0.7671 - val_loss: 0.6421 - val_accuracy: 0.7493
Epoch 12/20
92/92 [==============================] - 3s 34ms/step - loss: 0.5699 - accuracy: 0.7832 - val_loss: 0.6179 - val_accuracy: 0.7534
Epoch 13/20
92/92 [==============================] - 3s 35ms/step - loss: 0.5641 - accuracy: 0.7786 - val_loss: 0.6173 - val_accuracy: 0.7629
Epoch 14/20
92/92 [==============================] - 3s 35ms/step - loss: 0.5487 - accuracy: 0.7942 - val_loss: 0.6386 - val_accuracy: 0.7548
Epoch 15/20
92/92 [==============================] - 3s 34ms/step - loss: 0.5496 - accuracy: 0.7964 - val_loss: 0.6033 - val_accuracy: 0.7657
Epoch 16/20
92/92 [==============================] - 3s 35ms/step - loss: 0.5211 - accuracy: 0.8065 - val_loss: 0.5770 - val_accuracy: 0.7752
Epoch 17/20
92/92 [==============================] - 3s 35ms/step - loss: 0.5072 - accuracy: 0.8107 - val_loss: 0.5862 - val_accuracy: 0.7725
Epoch 18/20
92/92 [==============================] - 3s 35ms/step - loss: 0.4852 - accuracy: 0.8169 - val_loss: 0.6152 - val_accuracy: 0.7629
Epoch 19/20
92/92 [==============================] - 3s 35ms/step - loss: 0.4651 - accuracy: 0.8184 - val_loss: 0.5773 - val_accuracy: 0.7793
Epoch 20/20
92/92 [==============================] - 3s 35ms/step - loss: 0.4363 - accuracy: 0.8307 - val_loss: 0.5844 - val_accuracy: 0.7657
```

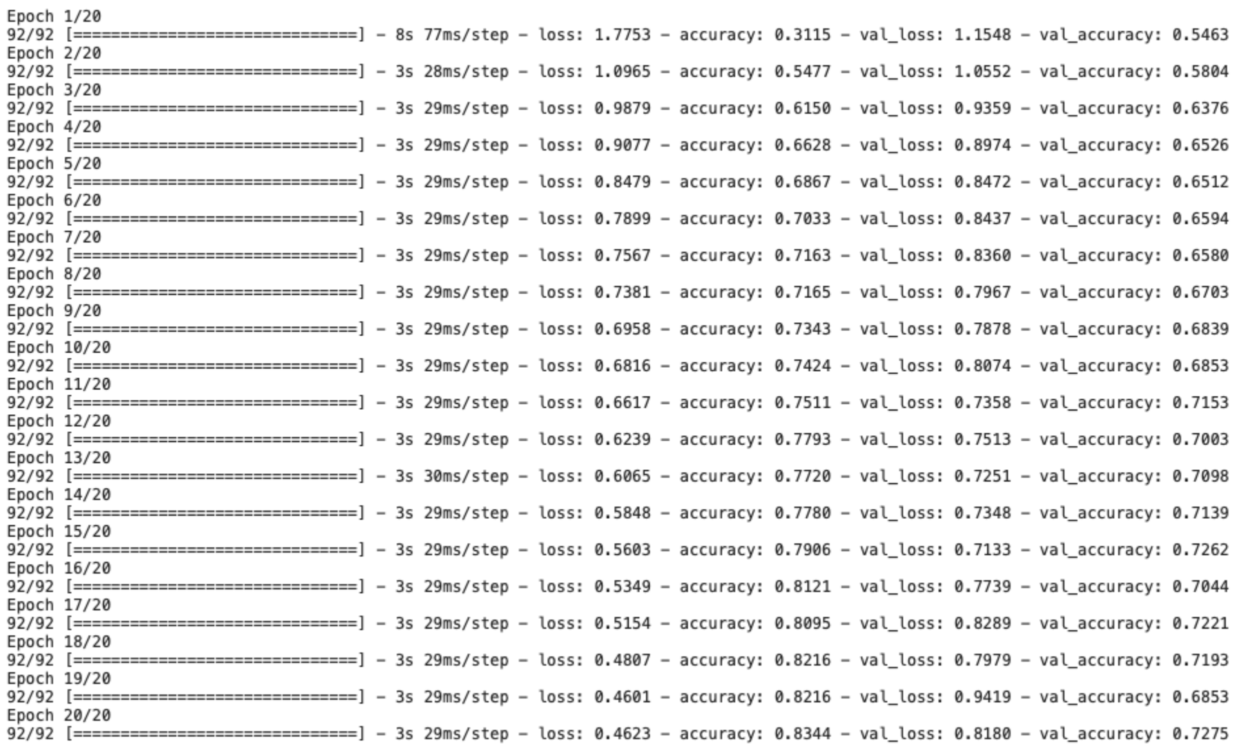
This result is confirmed once we test the model on a picture, for instance, of a red sunflower. It gives:



```
tf.Tensor([[1.6713785e-03 1.3261040e-04 1.1617449e-03 9.8937362e-01 7.6607405e-03]], shape=(1, 5), dtype=float32)
This image most likely belongs to sunflowers with a 98.94 percent confidence.
```

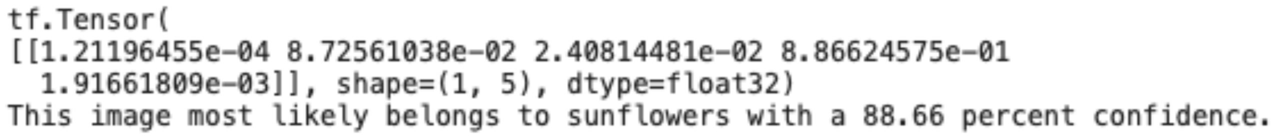
**WITHOUT SPP**

We first implemented the model using a flatten layer between the convolutional layers and the dense ones, instead of the spatial pyramid pooling layer. We observed that the validation loss was higher:



```
Epoch 1/20
92/92 [==============================] - 8s 77ms/step - loss: 1.7753 - accuracy: 0.3115 - val_loss: 1.1548 - val_accuracy: 0.5463
Epoch 2/20
92/92 [==============================] - 3s 28ms/step - loss: 1.0965 - accuracy: 0.5477 - val_loss: 1.0552 - val_accuracy: 0.5804
Epoch 3/20
92/92 [==============================] - 3s 29ms/step - loss: 0.9879 - accuracy: 0.6150 - val_loss: 0.9359 - val_accuracy: 0.6376
Epoch 4/20
92/92 [==============================] - 3s 29ms/step - loss: 0.9077 - accuracy: 0.6628 - val_loss: 0.8974 - val_accuracy: 0.6526
Epoch 5/20
92/92 [==============================] - 3s 29ms/step - loss: 0.8479 - accuracy: 0.6867 - val_loss: 0.8472 - val_accuracy: 0.6512
Epoch 6/20
92/92 [==============================] - 3s 29ms/step - loss: 0.7899 - accuracy: 0.7033 - val_loss: 0.8437 - val_accuracy: 0.6594
Epoch 7/20
92/92 [==============================] - 3s 29ms/step - loss: 0.7567 - accuracy: 0.7163 - val_loss: 0.8360 - val_accuracy: 0.6580
Epoch 8/20
92/92 [==============================] - 3s 29ms/step - loss: 0.7381 - accuracy: 0.7165 - val_loss: 0.7967 - val_accuracy: 0.6703
Epoch 9/20
92/92 [==============================] - 3s 29ms/step - loss: 0.6958 - accuracy: 0.7343 - val_loss: 0.7878 - val_accuracy: 0.6839
Epoch 10/20
92/92 [==============================] - 3s 29ms/step - loss: 0.6816 - accuracy: 0.7424 - val_loss: 0.8074 - val_accuracy: 0.6853
Epoch 11/20
92/92 [==============================] - 3s 29ms/step - loss: 0.6617 - accuracy: 0.7511 - val_loss: 0.7358 - val_accuracy: 0.7153
Epoch 12/20
92/92 [==============================] - 3s 29ms/step - loss: 0.6239 - accuracy: 0.7793 - val_loss: 0.7513 - val_accuracy: 0.7003
Epoch 13/20
92/92 [==============================] - 3s 30ms/step - loss: 0.6065 - accuracy: 0.7720 - val_loss: 0.7251 - val_accuracy: 0.7098
Epoch 14/20
92/92 [==============================] - 3s 29ms/step - loss: 0.5848 - accuracy: 0.7780 - val_loss: 0.7348 - val_accuracy: 0.7139
Epoch 15/20
92/92 [==============================] - 3s 29ms/step - loss: 0.5603 - accuracy: 0.7906 - val_loss: 0.7133 - val_accuracy: 0.7262
Epoch 16/20
92/92 [==============================] - 3s 29ms/step - loss: 0.5349 - accuracy: 0.8121 - val_loss: 0.7739 - val_accuracy: 0.7044
Epoch 17/20
92/92 [==============================] - 3s 29ms/step - loss: 0.5154 - accuracy: 0.8095 - val_loss: 0.8289 - val_accuracy: 0.7221
Epoch 18/20
92/92 [==============================] - 3s 29ms/step - loss: 0.4807 - accuracy: 0.8216 - val_loss: 0.7979 - val_accuracy: 0.7193
Epoch 19/20
92/92 [==============================] - 3s 29ms/step - loss: 0.4601 - accuracy: 0.8216 - val_loss: 0.9419 - val_accuracy: 0.6853
Epoch 20/20
92/92 [==============================] - 3s 29ms/step - loss: 0.4623 - accuracy: 0.8344 - val_loss: 0.8180 - val_accuracy: 0.7275
```

This means that the model is able to predict with a lower percent confidence. In fact, testing it on the same picture we used before, it returns:



```
tf.Tensor(
[[1.21196455e-04 8.72561038e-02 2.40814481e-02 8.86624575e-01
  1.91661809e-03]], shape=(1, 5), dtype=float32)
This image most likely belongs to sunflowers with a 88.66 percent confidence.
```

## 7 REFERENCES

*[1]* Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, *Spatial Pyramid Pooling in Deep Convolutional Network for Visual Recognition,* arXiv_1406.4729v4 [cs.CV], 23 April 2015

[2] Alex Krizhevsky Ilya Sutskever Geoffrey E. Hinton, *ImageNet Classification with Deep Convolutional Neural Networks*, January 2012