Deep Learning

Convolutional Neural Network

Flowers 102 Dataset

Olga Valls Murcia

Introduction

The aim of this Laboratory is to configure and train a Convolutional Neural Network to be able to classify a set of images in their corresponding classes.

Dataset

The data set used is the Flowers102 dataset, available at http://www.robots.ox.ac.uk/~vgg/data/flowers/102/ , consisting of 8189 images of flowers commonly occuring in the United Kingdom of 102 different classes. Each class consists of between 40 and 258 images.

The resolution of the images is around 500px x 500px (RGB), as some of them are 500px height or 500px width.

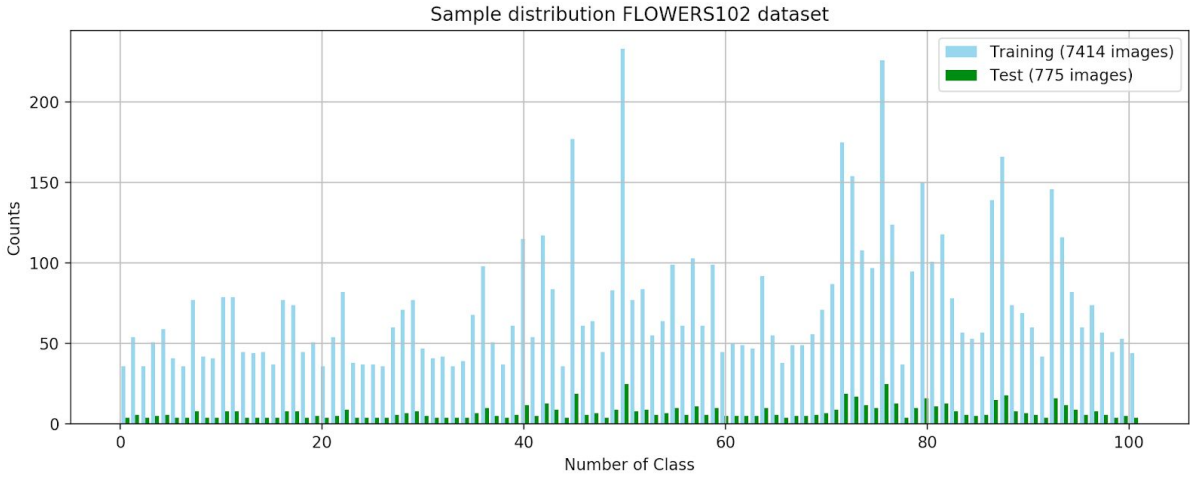
The images have large scale, pose and light variations. In addition, there are categories that have large variations within the category and several very similar categories.

The dataset also contains a .mat file with the labels of the class for each of the images.

Distribution of training and test sets

The networks needs to be trained with a subset of images, to later on test the accuracy of the system.
To do so, the images has been resized to 256px x 256px, and the 99% of the dataset (7414 samples), has been used as training set and the rest, 1%, (775 samples) has been used for the test set. No validation set has been used.

The sample distribution for the training and test set, for each of the classes is the following:



Pre-processing the dataset

3 scripts were created to pre-process the images from the original format to the desired format to be able to feed the network properly:

- *ResizeImages.m*: Matlab script to resize and crop the original images to 256px x 256px (RGB).
- *CreateSets.m*: Matlab script to create the train a test sets with their corresponding images, along with two .csv files, with the labels of the class for each of the images, one for train and one for test.
- *Subfolders.py*: Python script that, for the train subset, as well as for the test subset, creates one subfolder per class containing the corresponding images. It uses the previous generated csv files containing the labels to assign the images to their class folders. This folder structure is needed to be able to use the flow_from_directory Keras function that allows to feed the network with batches of images to avoid memory issues in the BSC cluster.

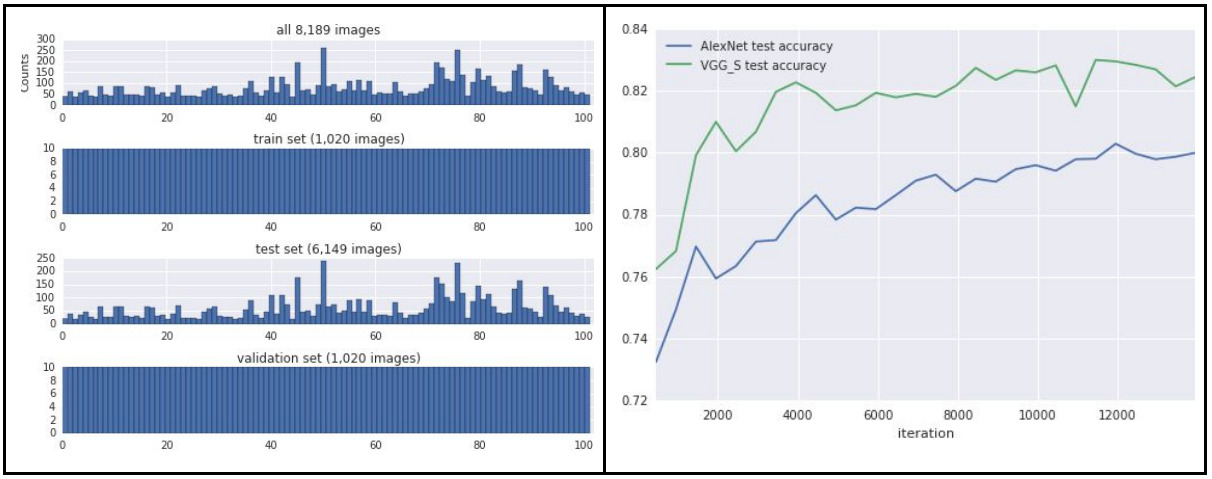
How this problem has been solved?

State of the art

Literature on this issue shows that several networks tried to solve this problem, being different versions of VGG the ones that got better results (91,60%), followed by different versions of Alexnet (85.20%).
https://www.researchgate.net/figure/Classification-performance-on-Oxford-102-flowers_tbl1_275669246

The information can be found at:
https://github.com/jimgoo/caffe-oxford102

The following table shows the distribution of the sets and the results:



Personal solution to this problem

Some experiments have been done to configure a network that can classify, which are explained in the next section.

Configuration of the Convolutional Neural Network

Base code

Using the Python base code provided, the first thing to do was to adapt the last dense layer to 102 neurons (number of classes). The same activation and loss functions have been used, this is also a classification problem.
The batch size used has been mainly 64 images, as it has been given better results than less batch size, and a higher number provoques memory issues.

As expected, as this is a more complex problem than mnist, the first results of acc and val_acc, were low (0.23 and 0.21).

First approach: add more convolutional layers

Little by little, more convolutional layers, with different number of filters, as well as more dense layers, has been added to the network.
The experiments have shown that more layers don't provide better results in accuracy, at least as what dense layers concern.
(Last dense layer - softmax - not considered in this work when talking about the number of dense layers).

- 3 convolutional layers + 1 dense layer: 0.31 in val_acc
- 4 convolutional layers + 2 dense layers: 0.15 in val_acc
- 4 convolutional layers + 1 dense layer: 0.25 in val_acc

Second approach: modify dense layer and change optimizer

As the dense layer recombines the results of the convolutional layers, a larger number of neurons for this layer is tested, finding that a medium number of neurons in the dense layer (512) gives better results.

Some literature recommends the optimizer adam as it gives better results:
https://shaoanlu.wordpress.com/2017/05/29/sgd-all-which-one-is-the-best-optimizer-dogs-vs-cats-toy-experiment/

The experiments show a huge increase in accuracy in training as well as test, given results of val_acc of 0.55 and acc of 0.93, much better than before.

The problem that arises is overfitting, which need to be solved; otherwise the network learns very good for the training examples but it's not capable to generalize and be able to classify the test samples.

Third approach: dealing with overfitting

Two ways to deal with overfitting are:

- Regularization: lowers the weights. It stabilizes training and make tuning parameters easier, helping lower training time.
- Dropout: distribute the weights as random neurons become inactive during a single training element. It's a regularization technique, and a lower number is used for layers that have a lot of parameters, and a higher number (or 1) for layers with lower number of parameters.

https://www.youtube.com/watch?v=IFalw4AbiVk

Fourth approach: size of kernel in the first convolutional layer

For large images is recommended to use a higher kernel (than the usual 3x3) so that the network can learn larger features.
https://www.pyimagesearch.com/2018/12/31/keras-conv2d-and-convolutional-layers/

Putting together third and fourth approaches

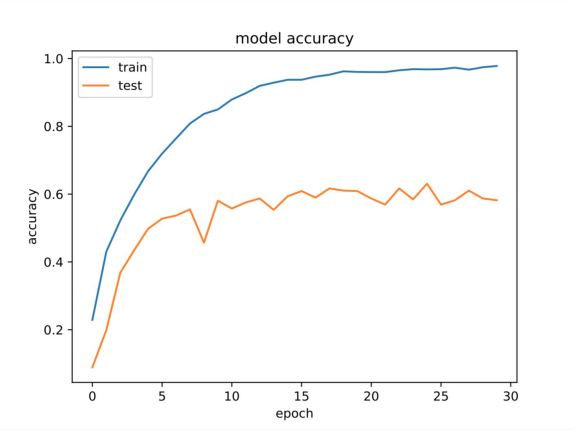
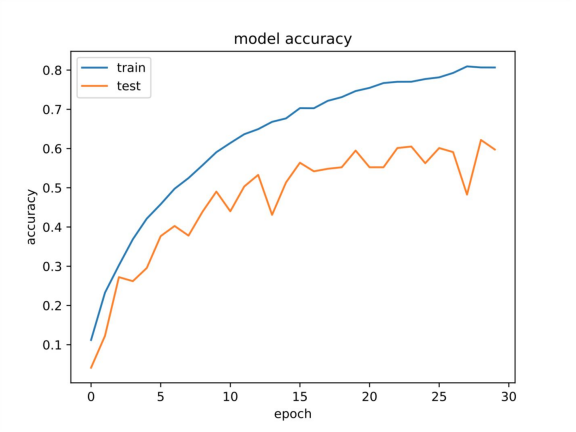
Those two last approaches has been tested together obtaining much betters results.
In the first case only BatchNormalization was tested. In the second case, Dropout was added to some layers:

The following table shows the architecture of the network for the two cases:

First case	Second case
Conv2D 128 7x7 stride 2x2 Conv2D 64 3x3 + BatchNormalization + MPooling2D Conv2D 32 3x3 + BatchNormalization + MPooling2D Conv2D 16 3x3 + BatchNormalization + MPooling2D Flatten Dense 512 relu + BatchNormalization Dense 102 softmax	Conv2D 128 7x7 stride 2x2 Conv2D 64 3x3 + BatchNormalization + MPooling2D Dropout 0.5 Conv2D 32 3x3 + BatchNormalization + MPooling2D Conv2D 16 3x3 + BatchNormalization + MPooling2D Flatten Dense 512 relu+ BatchNormalization Dropout 0.5 Dense 102 softmax

The results for the accuracy, as shown in the following tables, are:

- First case: val_acc of 0.6090 and acc of 0.9604
- Second case: val_acc of 0.5948 and acc of 0.7465

First case	Second case
	

We can see that Dropout has made a big impact preventing overfitting, and the val_acc hasn't drop almost anything.

Fifth and last approach: Maxpooling and nadam optimizer

MaxPooling downsizes the output of a layer and keeps the best answers. It helps us also to generalize, thus, preventing overfitting.

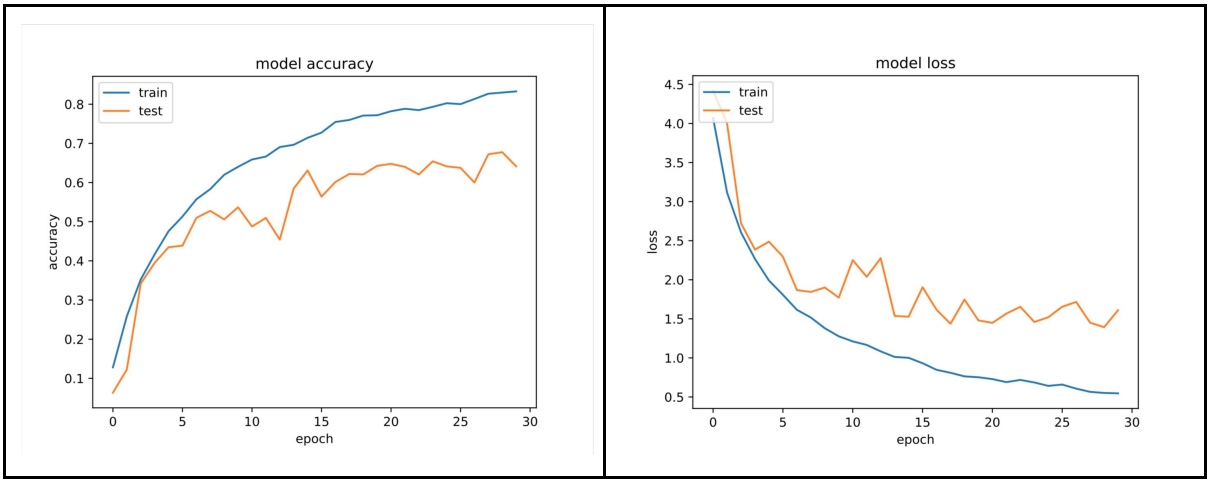
We have added MaxPooling after each convolutional layer but the first one, in which we apply a 7x7 kernel and a 2x2 stride.
In this case, after doing some tests, the results have shown that Maxpooling after this layer doesn't lead to better results, but a BatchNormalization does, which have been added to the last configuration.

Nadam optimizer comes from Nesterov Adam optimizer. Much like Adam is essentially RMSprop with momentum, Nadam is Adam RMSprop with Nesterov momentum.
https://keras.io/optimizers/

The results for this optimizer are a little bit better than for adam, with no detected penalization in time or resources; thus that was added to the final architecture of the network, which is the following:

Conv2D 128 7x7 stride 2x2 + BatchNormalization Conv2D 64 3x3 + BatchNormalization + MPooling2D Dropout 0.5 Conv2D 32 3x3 + BatchNormalization + MPooling2D Conv2D 16 3x3 + BatchNormalization + MPooling2D Flatten Dense 512 relu + BatchNormalization Dropout 0.5 Dense 102 softmax

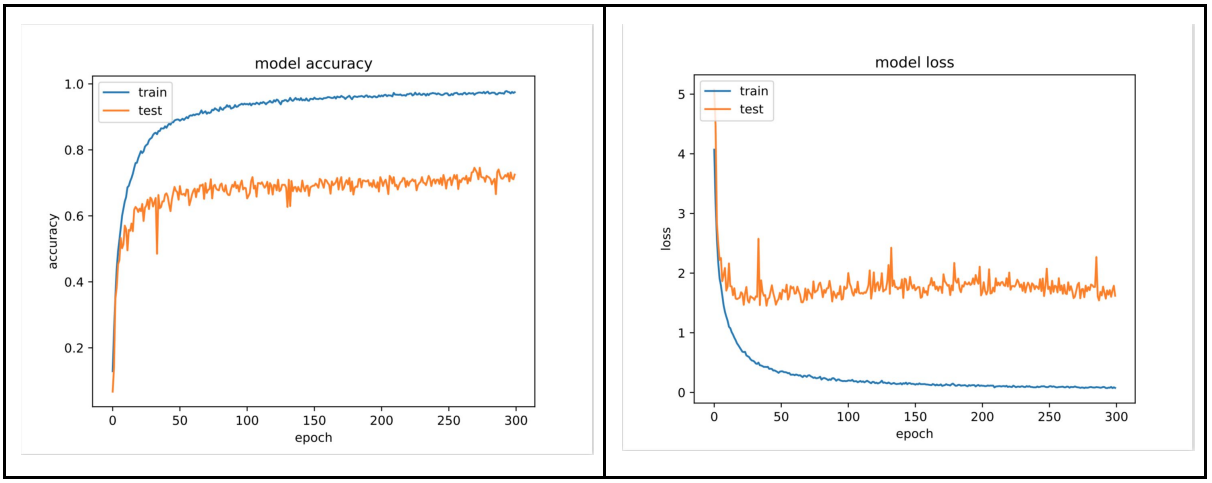
The plots for accuracy and loss show that, at 20 epochs, the values for val_acc and acc are 0.6427 and 0.7717 respectively, and the values for val_loss and loss are 1.4796 and 0.7519, which are better than the previous ones.



Final Results

Using the configuration that has given the best results, the last experiment consists on testing the network with a larger number of epochs (300) to see when it stabilizes and stop learning, and which is the classification for the test sample.

As we can see in the following plots the val_acc stabilizes around 40 epochs, giving the very best result in epoch 268 with val_acc 0.7316, which is a small increase with the results obtained for 20 epochs, given the amount of time to process the data in the network for 300 epochs (more than 6 hours of execution time).



The plots also show that at around 50 epochs algo the acc and the losses stabilizes, which tells that the network is not learning anymore.

The **final results** of accuracy and loss for the 300 epochs are as follows:

- val_acc: 0.7252
- val_loss: 1.6204
- acc: 0.9749
- loss: 0.0759

Despite the high result in accuracy, the analysis of results shows that only very few images where classified in its corresponding class, which are the ones shown below:

	precision	recall	f1-score	support
class_006	0.17	0.25	0.20	4
class_009	0.25	0.25	0.25	4
class_017	1.00	0.12	0.22	8
class_042	0.08	0.08	0.08	13
class_045	0.06	0.05	0.06	19
class_050	0.03	0.04	0.04	25
class_059	0.09	0.10	0.10	10
class_064	0.15	0.20	0.17	10
class_081	0.11	0.09	0.10	11
class_101	0.00	0.00	0.00	4
avg / total	0.02	0.01	0.01	775

Conclusions

The classification is not as good as it seems seeing the val_accuracy value. The model learns the training set but is not able to classify correctly the 775 test images.

Future Work

Some research on classification problems should be done to find how to fill the gap between the train acc plot and the val_acc plot.

From the conclusion, some questions arise, such as: Is CNN the proper Network to classify images? Is there a combination of different Networks the solution to this classification problem? Is only a problem of proper combination of convolutional and dense layers?

Git Repository

In https://github.com/ovalls/mai_dl you can find the code for the CNN along with some scripts:

- `flowers102_cnn.py`: code for the configuration of the Convolutional Neural Network.
- `ResizeImages.m`: script for pre-processing (explained in Pre-processing section).
- `CreateSets.m`: script for pre-processing (explained in Pre-processing section).
- `Subfolders.py`: script for pre-processing (explained in Pre-processing section).
- `Imagelabels.mat`: original labels for the flowers 102 dataset.
- `labels_train.csv`: labels for the images of the training set.
- `labels_test.csv`: labels for the images of the test set.