Homework 5

Ana Santos, 84364 Inês Branco, 81506 Vânia Nunes, 85235

31 May 2019

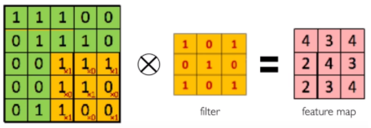
The link to the code: https://colab.research.google.com/drive/1qFXAGYkgg3EB1tD2aZGcH9mLYXufOlbF

1 Why to use CNN's

We have studied fully connected neural networks where every single neuron connects to every neuron in the subsequent layer. If we want to do image classification our 2D input image is transformed into a vector of pixel values and this vector is then fed into the network. By squashing the 2D matrix into 1D vector, we are loosing spatial information. Furthermore, by defining the network in this way we end up having enormous amounts of parameters (since every neuron in each hidden layer is connected to all neurons into the following layer). This make us conclude that training a neural network in this format for a task like image classification becomes unfeasible in practice (very time and computationally consuming). In order to immediately use the spatial structure, we will instead represent our 2D input image as an array of pixel values and we will connect patches of the input to neurons in the hidden layer, where each neuron in a hidden layer will only "sees" a particular region of what the input to that layer is, which allows to reduce dramatically the the number of weights in our model.

2 Basic idea of how CNN's work

In convolution neural networks, it is used the operation of convolution represented in the image below, where it is applied to the set of input images a filter with the traits we want our CNN to learn to recognize (the values/weights associated with this filter are the ones the CNN will learn).



1	1	1	0	0
0	1	1	1	0
0	0	1 ×1	1 ×0	1 ×1
0	0	1 ×0	1 ×1	0 ×0
0	1	1 ×1	0 ×0	0 ×1

⊗

1	0	1
0	1	0
1	0	1

filter

=

4	3	4
2	4	3
2	3	4

feature map

Figure 1

The calculation of convolution performed in CNN is given by the following equation:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \quad (1)$$

The steps of the general image classification and the ones also used in our Mobile Net implementation are described by the following image:

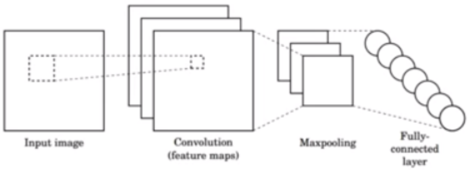


Figure 2

where the first step is the process of convolution. The result is a set of feature maps. It is then applied a non-linear activation function, after each convolutional layer since the image data is highly non-linear: $\sum_{i=1}^{N}\sum_{j=1}^{M} w_{ij}x_{i+p,j+q} + b$ (for a neuron (p,q) in a hidden layer). It was used a Rectified Linear Unit (ReLU) activation function: $g(z) = max(0, z)$. We can look at this function and interpret the negative values after convolution as a negative detection of that specific feature.

The third step is "pooling" which is a down-sampling operation to reduce the size of a map. In the code, we aplied the function *GlobalAveragePooling2D()* where the output tensor with dimensions hwd is reduced in size to have dimensions $11d$. GAP layers reduce each hw feature map to a single number by simply taking the average of all hw values.

The computation of the class scores and actually outputting a prediction for the class of an image is achieved by a fully connected layer at the end of the network. These fully connected layers can effectively output a probability distribution for the images membership over a set of possible classes and a common way that this is using a function called softmax (it is used in the last layer of the NN present implement in the code):

$$softmax(y_i) = \frac{e^{y_i}}{\sum_j e^{y_i}} \quad (2)$$

3 Pre-trained CNN's

Though training a CNN from scratch is possible for small projects, most applications require the training of very large CNN's and this takes extremely huge amounts of processed data and computational power.

In transfer learning, we take the pre-trained weights of an already trained model and use these already learned features to predict new classes.

The advantages of transfer learning are:

1: There is no need of an extremely large training dataset.

2: Not much computational power is required, as we are using pre-trained weights and only have to learn the weights of the last few layers.

The reason why it works so well is that, we use a network which is pretrained on the imagenet dataset and this network has already learnt to recognize the trivial shapes and small parts of different objects in its initial layers.

By using a pretrained network to do transfer learning, we are simply adding a few dense layers at the end of the pretrained network and learning what combination of these already learnt features help in recognizing the objects in our new dataset.

4 Data Set

The fruits dataset was downloaded from kaggle.

We used the following folders from the data set to get the classes asked for in the homework.

The Apple class contains the images in the following folders of the kaggle data set: Apple Braeburn, Apple Crimson Snow, Apple

Golden 1 , Apple Golden 2, Apple Golden 3, Apple Granny Smith, Apple Pink Lady, Apple Red 1, Apple Red 2, Apple Red 3, Apple Red Delicious, Apple Red Yellow 1 and Apple Red Yellow 2.

The Lemon class contains the images in the following folders of the kaggle data set: Lemon and Lemon Meyer.

The Orange class is the same as the one in the kaggle data set.

The Pear class contains the images in the following folders of the kaggle data set: Pear, Pear Abate, Pear Kaiser, Pear Monster, Pear Red and Pear Williams.

It is also very low maintenance thus performing quite well with high speed.

5 Training, Test and Validation Sets

The number of images for each of the classes for training and testing are in the following table:

	Training	Test	ratio
Apple	6416	2138	75.0 - 25.0
Lemon	982	330	74.8 - 25.2
Orange	479	160	75.0 - 25.0
Pear	2928	986	74.8 - 25.2

Table 1

So we have approximately a 75 to 25 percent ratio between the training and testing set.

To split this training set into training and validation data sets, we randomly chose 25%. So we end up with the following number of images for each set:

	Training	Validation	Test
Apple	4812	1064	2138
Lemon	736	246	330
Orange	359	120	160
Pear	2196	732	986

Table 2

6 Architecture and Parameters

In this first step we want to classify objects in images with nothing else into 4 classes. MobileNets is an efficient network architecture that uses depth-wise separable convolutions to build light weight deep neural networks. This is a low latency model that can be easily matched to design requirements for mobile and embedded vision applications. We decided to use the MobileNet model because it does not require large datasets, it is relativity fast without compromising in this trade accuracy by a large amount (accuracy vs. computational time).

Adding more layers will help us extract more features. But we can only do that to a certain extent. There is a limit. After that, instead of extracting features, we tend to 'overfit' the data.

7 Regularization

Regularization helps us tune and control our model complexity, ensuring that our models are better at making (correct) classifications — or more simply, the ability to generalize.

If we don't apply regularization, our classifiers can easily become too complex and overfit to our training data, in which case we lose the ability to generalize to our testing data (and data points outside the testing set as well).

	Loss_{train}	Acc_{train}	Loss_{val}	Acc_{val}
3layer	0.0073	0.9987	1×10^{-6}	1
6layer	0.1155	0.9607	0.0285	0.9942
1layer	0.0147	0.9966	8×10^{-7}	1
Glorot Uniform	0.0147	0.9966	8×10^{-7}	1
zeros	0.9596	0.6289	0.9477	0.6335
ones	15.2749	0.0523	15.2671	0.0528
Random Uniform	0.0146	0.9959	0.0018	0.9987
Random Normal	2×10^{-5}	1	3.7×10^{-7}	1
none	0.0147	0.9966	8×10^{-7}	1
l2(0.1)	0.0152	0.9999	0.0103	1
l1(0.1)	13.6654	0.9230	12.8618	0.9463
$\alpha = 1$	0.0147	0.9966	8×10^{-7}	1
$\alpha = 0.9$	3×10^{-5}	1	5×10^{-7}	1
$\alpha = 0.75$	7×10^{-6}	1	4.7×10^{-7}	1
$\alpha = 1.1$	0.0248	0.9913	0.2647	0.9215
$\eta = 0.001$	0.0147	0.9966	8×10^{-7}	1
$\eta = 0.0001$	7×10^{-6}	1	2×10^{-7}	1

Table 3: Tested parameters. The parameters not mentioned in the first column are the ones of default.

We used the class callback in the function *fit generator* in order to perform the number of epochs that allowed for the error loss associated with the weights to stabilize.

8 Results

In table 3 we have the values of accuracy and loss for the training and validation datasets for the best epoch of each training. In the first section we looked for the best number of fully connected layers that we train, we tested for 1, 3 and 6 layers. In the second section we looked for the best initialization of the weights of the added layer, we tested the default: Glorot Uniform, all zeros, all ones, a random normal distribution around zero and a uniform distribution. In the third section we looked into the regularization, and in the fourth into the values of the alpha parameter.

By looking at the results we decided to use 1 layer, no regularization in that layer, $\alpha = 0.75$ and the Random Normal initialization. This gives 0.032 of loss and 99.8% of accuracy, for the test data set. We also tested for $\alpha = 1$, and the default initialization and got 0.0006 of loss and 100% accuracy for the test set.

We observed that the results obtained for various combinations of initial parameters didn't stabilize around a final value and oscillated quite largely (the values for the loss for both the training set and the validation set seem to be oscillating instead of decreasing or increasing). We suspected that this lack of convergence was due to an overly large value for the learning parameter, so we decided to run the code again for a learning rate 10 times smaller. Both values of accuracy and error revealed little oscillation as the number of epochs increased, which revealed our guess was right and the group was using an overly large value of η. With this learning rate it was obtained 0.004 of loss and 99.83% of accuracy.