Image classification of fruits and vegetables through deep learning

Pietro Prebianca

1 Introduction

The problem of image classification is a very well known topic and active line of research in machine learning field; the most notable approach to to tackle this problem are feedforward convolutional neural networks(CNN). Works such AlexNet[1], GoogleNet[9] and many others made become this class of neural networks, the state of the art for image classification.

But the application field of this tool is not limited on image classification; CNN are heavily used to solve tasks such as image semantic segmentation and object detection.

The key elements of these networks are the following:

- **Discrete convolutional layer:**

  Discrete convolutions layers allows to extract patterns that are present in images, thanks to matrix of weights (kernels). These kernels are generally 3x3x3, 5x5x3 matrices for RGB images, but the dimensionality of these is chosen according to the input.

  In mathematical terms, a convolution describes how the shape of a given function is modified by a second function; in image classification, this operation produces feature map.

- **ReLU activation function:**

  Sigmoid and hyperbolic tangent were very used in the past as activation functions, but they share commons problems; they tend to saturate or snap great input values to 1 and low value to -1 (hyperbolic tangent) or 0 (sigmoid). Furthermore, they suffer low sensitivity, which means they perceive changes around their mid-point of their input; these facts together causes the vanishing gradient problem.

  CNN generally uses ReLU for all internal nodes.

- **Pooling layer:**

  Pooling operations reduce the size of feature maps by applying functions to summarize subregions, such as average or maximum.

  Usually for image classification, the max pooling operation together with a convolution; the more deep the networks is, the more filters and pooling operations are applied; these two elements can highlight edges of objects depicted inside images.

However, pooling is not always performed, due to the information loss that it can generate.

- **Fully connected layer:**

  Convolution and Pooling can't classify samples by themselves; the classification happens through one or more fully connected layers, before the output.

2 About this report

This report contains some image classification results I've obtained with different CNNs, on a dataset of images of fruits and vegetables. All results were obtained with Tensorflow[6], an opensource machine learning platform; more specifically, Tensorflow is a wrapper of Keras[2], a machine learning framework. All tests have been ran on Google Colab, a cloud service which allows data scientist to build and share code and deep learning models.

The rest of the document is structured as follows: in section 3, it's described the dataset used for the for experiments treated in section 4, 5.

In each experiments, I tried different network structures with different combinations of hyper-parameters.

Finally in section 6, the document ends with a little declaration about the intellectual proprieties.

For each classification task, the best model were implemented in this Google Colab.

3 Dataset

The dataset used for each experiments is "Fruits-360"[7], which is available for download on Kaggle[4] and Github[3]. It is composed by 90483 images of fruits and vegetables, which are classified into 131 class of variety.

These dataset is splitted in:

- 67692 images of single fruits/vegetables for training
- 22688 images of single fruits/vegetables for testing
- 103 images of multiple fruits

Despite all these images are 100x100 pixels, all results shown in next sections have been obtained scaling all images to 32x32.

For further about all fruit and vegetable varieties, consults the previous mentioned references.

4 Classification based on fruit and vegetable types

4.1 Experiment description

The purpose this classification experiment were to find a model, which better recognizes types of fruit and vegetable belonging to this list:

- Apple
- Banana
- Cherry
- Grape
- Peach
- Pear
- Plum
- Pepper
- Potato
- Tomato

The original dataset weren't suited for this task, because images were organized by variety; so I extracted all variety related to those types, for training and test set, removing all other data. The resulting set were composed of 32607 images for training and 10906 for test; Table 1 shows the number of images for each class.

Table 1: Number of images for training and test sets for each class.

Type	**Training images**	**Test images**
Apple	6404	2134
Banana	1430	484
Cherry	3444	1148
Grape	3419	1146
Peach	1722	574
Pear	5037	1689
Pepper	2478	826
Plum	1767	597
Potato	1803	601
Tomato	5103	1707

4.2 Network design process

At first, I tackled this problem with a small CNN structured with 5 layers; the network structures is depicted in Table 2.

The hyper-parameters has been chosen in part manually according to common sense, while others (number of filters and neurons) through a grid search with a nested 5-fold stratified cross validation.

The hyper-parameters looked by the grid search are:

- # Neurons: 512, 1024, 2048
- # Filters: 4, 8, 16, 32

The training have been done in 10 epochs, for each fold, for each model; as batch size, I preferred to start with a small value (16) and eventually increasing it, in successive tests.

The optimizer chosen is Adam[5], a stochastic gradient descent method that adapts the learning rate during the training, which is fast and works well in deep learning; Keras implementation of Adam uses default parameters.

The lost function used is the sparse categorical cross entropy loss function, which is very used for CNN.

Table 2: Network structure used for the tuning session.

Layer	**Output**
2D Convolution - Kernel Size (3x3)	(32, 32, # Filters)
Max Pooling - Pool Size (2x2)	(16, 16, # Filters)
Flatten	(16*16* # Filters)
Dense	# Dense
Dense	10

Strides for convolution and pooling were respectively (1, 1) and (2, 2).

Since each C class has a different number of samples N_c, I associated a weight W_c to each class, in order to not bias classes with less samples.

$$W_c = \frac{S}{N_c * 2}$$

Table 3 shown obtained results for each model.

Table 3: Average training and validation accuracy.

Filters	**Neurons**	**Train. Acc.**	**Val. Acc.**
4	512	99.99 %	99.76 %
4	1024	99.99 %	99.82 %
4	2048	99.99 %	100.0 %
8	512	99.99 %	99.85 %
8	1024	100.0 %	100.0 %
8	2048	100.0 %	99.91 %
16	512	100.0 %	99.92 %
16	1024	100.0 %	99.99 %
16	2048	99.99 %	100.0 %
32	512	99.99 %	99.99 %
32	1024	100.0 %	99.81 %
32	2048	100.0 %	100.0 %

From these results seem that all models generalize very well on unseen data, even with few filters, with no trace of overfitting, which is generally quite common in deep learning.

Since I couldn't identify, with these results, which model performed better, I picked all of them and evaluated against the test set 10 times each, shuffling training set each time and calculating average accuracy on training and test, with related standard deviation; I left epochs and batch sizes unchanged, since each models were performing well. The discrepancy between validation accuracy and test accuracy were not so big, as Table 4 shown.

Table 4: Average accuracy and standard deviation on training and test data.

Filters	**Neurons**	**Train. Acc.**	**Train. Acc. σ**	**Test Acc.**	**Test Acc. σ**
4	512	100.0	0.0	97.11	0.48
4	1024	100.0	0.0	96.60	0.48
4	2048	100.0	0.0	97.71	0.49
8	512	100.0	0.0	97.60	0.39
8	1024	100.0	0.0	97.83	0.35
8	2048	100.0	0.0	97.73	0.45
16	512	100.0	0.0	97.58	0.40
16	1024	99.99	0.01	97.91	0.27
16	2048	100.0	0.0	97.94	0.49
32	512	100.0	0.0	97.90	0.41
32	1024	100.0	0.0	98.02	0.30
32	2048	100.0	0.0	97.67	0.44

During training phase, I noticed that the training error in 10 epochs reaches values of order $\sim 10^{-5}$, so I decided fix it for next experiments. I didn't perform other CV sessions, since it didn't help to understand which model was better.

To increase performances of previous models, I thought to add a dropout layer, right after the first dense layer; dropout[8] is wide used regularization technique used in deep learning, which can help to reduce overfitting; practically this layer denies randomly the output of neurons, according to a probability threshold.

So I set this dropout layer with 50 % probability, obtaining Table 5 structure. I evaluated the previous models again 10 times each, shuffling training data, and testing the same combinations of filters and neurons; batch size, stride values, loss function and optimizer were unchanged.

Table 5: Network structure with dropout layer.

Layer	**Output**
2D Convolution - Kernel Size (3x3)	(32, 32, # Filters)
Max Pooling - Pool Size (2x2)	(16, 16, # Filters)
Flatten	(16*16*# Filters)
Dense	# Dense
Dropout (50%)	# Dense
Dense	10

Table 6: Average accuracy and standard deviation on training and test data, with one dropout layer (50 %).

Filters	**Neurons**	**Train. Acc.**	**Train. Acc. σ**	**Test Acc.**	**Test Acc. σ**
4	512	100.0 %	0.0	96.21 %	0.78
4	1024	99.78 %	0.57	96.34 %	1.72
4	2048	99.97 %	0.02	96.39 %	0.82
8	512	100.0 %	0.0	96.97 %	1.06

Continued on next page

Filters	Neurons	Train. Acc.	Train. Acc. σ	Test Acc.	Test Acc. σ
8	1024	99.88 %	0.15	96.56 %	1.40
8	2048	99.86 %	0.2	96.60 %	1.11
16	512	100.0 %	0.0	97.51 %	0.45
16	1024	99.98 %	0.01	97.37 %	0.50
16	2048	99.86 %	0.33	97.08 %	0.81
32	512	100.0 %	0.0	97.61 %	0.43
32	1024	100.0 %	0.0	97.82 %	0.70
32	2048	100.0 %	0.0	97.52 %	0.88

Instead of improving results, there were a general worsening, of almost all tested models; I suppose that I was using a too high probability of dropout, so I tried again with a lower probability.

But with a lower probability, the results were worse than before.

Table 7: Average accuracy and standard deviation on training and test data, with one dropout layer (20 %).

Filters	Neurons	Train. Acc.	Train. Acc. σ	Test Acc.	Test Acc. σ
4	512	100.0 %	0.0	96.20 %	1.04
4	1024	99.96 %	0.04	96.16 %	0.83
4	2048	99.70 %	0.30	93.99 %	1.72
8	512	99.96 %	0.08	96.64 %	0.67
8	1024	100.0 %	0.0	96.97 %	0.83
8	2048	99.95 %	0.06	96.98 %	0.72
16	512	99.98 %	0.02	97.59 %	0.49
16	1024	99.98 %	0.03	96.97 %	0.74
16	2048	99.90 %	0.19	96.99 %	0.78
32	512	99.87 %	0.03	97.02 %	0.65
32	1024	99.95 %	0.06	97.11 %	0.72
32	2048	99.69 %	0.84	96.62 %	1.15

I noticed that in all previous experiments, there were no big differences between models with less filters and neurons, with those with many.

Since I didn't get any significatively improvements through dropout, I decided to increased network complexity, by adding one more dense layer; as a rule, the number of neurons decreases towards the end of the network, so I put one more dense layer, which has the half of neurons of the layer before.

The new structure is in Table 8.

Table 8: Network structure with a third dense layer.

Layer	Output
2D Convolution (3x3)	(32,32,# Filters)
Max Pooling (2x2)	(16,16,# Filters)
Flatten	(8*8*# Filters)
Dense	# Dense

Continued on next page

Layer	Output
Dense	(# Dense)/2
Dense	10

As I did before, I tested this structure with different combinations of neurons and filters, training it always in 10 epochs, with 16 as batch size and all other hyper-parameters unchanged; like before, each model were evaluated 10 times, calculating standard deviation for test and training accuracy.

But again, this introduction weren't a right choice, since it produced other more unstable models.

Table 9: Average accuracy and standard deviation on training and test data, with 3 dense layers.

Filters	Neurons	Train. Acc.	Train. Acc. σ	Test Acc.	Test Acc. σ
4	512, 256	99.70 %	0.47	96.05 %	1.74
4	1024, 512	100.0 %	0.0	96.73 %	0.75
4	2048, 1024	99.83 %	0.43	96.14 %	1.64
8	512, 256	99.87 %	0.38	96.99 %	1.08
8	1024, 512	100.0 %	0.0	97.82 %	0.38
8	2048, 1024	100.0 %	0.0	97.44 %	0.54
16	512, 256	99.89 %	0.30	97.47 %	1.33
16	1024, 512	100.0 %	0.0	97.54 %	0.66
16	2048, 1024	99.97 %	0.06	97.41 %	1.10
32	512, 256	100.0 %	0.0	97.89 %	0.36
32	1024, 512	99.98 %	0.04	97.60 %	0.80
32	2048, 1024	100.0 %	0.0	97.94 %	1.05

Then, instead of going deeper with dense layer, I though to add a further convolution and max-pooling layers, to see if I was able to achieve better results; in CNN, it's common to increase filter number as we go deeper in the network.

So I used the following model, with different combinations of filters and neurons:

Table 10: Network structure with 2 convolutions and 2 max pooling layers.

Layer	Output
2D Convolution (3x3)	(32,32,# Filters)
Max Pooling (2x2)	(16,16,# Filters)
2D Convolution (3x3)	(16,16,# Filters*2)
Max Pooling (2x2)	(8,8,# Filters*2)
Flatten	(8*8*# Filters*2)
Dense	# Dense
Dense	10

All other hyper-parameters weren't unchanged.

Table 11: Average accuracy and standard deviation on training and test data, with a second convolution and max pooling layer.

Filters	**Neurons**	**Train. Acc.**	**Train. Acc. σ**	**Test Acc.**	**Test Acc. σ**
4, 8	256	100.0 %	0.0	97.32 %	0.5
4, 8	512	100.0 %	0.0	97.81 %	0.5
4, 8	1024	100.0 %	0.0	97.83 %	0.4
4, 8	2048	99.36 %	1.8	97.08 %	2.3
8, 16	256	100.0 %	0.0	98.76 %	0.48
8, 16	512	100.0 %	0.0	98.64 %	0.25
8, 16	1024	100.0 %	0.0	98.75 %	0.32
8, 16	2048	100.0 %	0.0	98.69 %	0.39
16, 32	256	100.0 %	0.0	99.04 %	0.27
16, 32	512	100.0 %	0.0	99.06 %	0.20
16, 32	1024	100.0 %	0.0	98.95 %	0.32
16, 32	2048	100.0 %	0.0	99.04 %	0.24
32, 64	256	100.0 %	0.0	99.18 %	0.22
32, 64	512	100.0 %	0.0	99.18 %	0.18
32, 64	1024	100.0 %	0.0	99.17 %	0.36
32, 64	2048	100.0 %	0.0	99.15 %	0.26

With this new introduction, some models were able to reach higher accuracies; all models with more that 4 and 8 filters achieved average accuracies on the test set between 98.00 % and 99.18 %, with low variance.

This trend of results made me think that maybe, increasing the number of filters were the good path.

Table 12: Average accuracy and standard deviation on training and test data, with 2 convolutions and 2 max pooling layers.

Filters	**Neurons**	**Train. Acc.**	**Train. Acc. σ**	**Test Acc.**	**Test Acc. σ**
64, 128	256	100.0 %	0.0	99.21 %	0.46
64, 128	512	100.0 %	0.0	99.11 %	0.15
64, 128	1024	100.0 %	0.0	99.11 %	0.16
64, 128	2048	100.0 %	0.0	99.20 %	0.25

And there were a very little improvement, but not so important.

Then I opted for looking for changes in other hyper-parameters, such as batch size; so I picked only the model which performed better until that moment, and tested with different batch sizes; I didn't tried models with different number of epochs, because as I mentioned earlier, the training error were already very low with 10 epochs. The model with best results in term of accuracy and stability is in Table 13.

Table 13: Model with 100.0 % (0.0 variance) on training data and 99.18 % (0.18 variance) on test data.

Layer	**Output**
2D Convolution (3x3)	(32,32,32)
Max Pooling (2x2)	(16,16,32)
2D Convolution (3x3)	(16,16,64)
Max Pooling (2x2)	(16,16,64)
Flatten	(8*8*64)
Dense	512
Dense	10

Table 14: Average accuracy and standard deviation on training and test data, on the best performing network, trained with different batch sizes.

Batch Size	**Train. Acc.**	**Train. Acc. σ**	**Test Acc.**	**Test Acc. σ**
32	100.0 %	0.0	99.06 %	0.24
64	100.0 %	0.0	98.86 %	0.23
128	100.0 %	0.0	97.77 %	0.39

As you can see, reducing the number of updates caused a worsening in performances.

As last experiment, I added a third convolution and a max pooling to the previous structure, and tried with different combination of filters and neurons.

Table 15: Network structure with 3 convolutions and 3 max pooling layers.

Layer	**Output**
2D Convolution (3x3)	(32,32,# Filters)
Max Pooling (2x2)	(16,16,# Filters)
2D Convolution (3x3)	(16,16,# Filters*2)
Max Pooling (2x2)	(8,8,# Filters*2)
2D Convolution (3x3)	(8,8,# Filters*3)
Max Pooling (2x2)	(4,4,# Filters*3)
Flatten	(4*4*# Filters*3)
Dense	# Dense
Dense	10

Table 16: Average accuracy and standard deviation on training and test data, after adding a second convolution, maxpooling layer.

Filters	**Neurons**	**Train. Acc.**	**Train. Acc. σ**	**Test Acc.**	**Test Acc. σ**
8, 16, 32	256	100.0 %	0.0	98.74 %	0.50
8, 16, 32	512	100.0 %	0.0	98.67 %	0.40
16, 32, 64	256	100.0 %	0.0	99.17 %	0.32
16, 32, 64	512	100.0 %	0.0	99.35 %	0.29
32, 64, 128	256	100.0 %	0.0	99.34 %	0.31

Continued on next page

Filters	**Neurons**	**Train. Acc.**	**Train. Acc. σ**	**Test Acc.**	**Test Acc. σ**
32, 64, 128	512	100.0 %	0.0	99.56 %	0.24

Except for one model, there were almost no differences between models with 2 convolution, max pooling layer and 3; this is because pooling operations (with an even pool size) reduce output size, which means that more the more we apply them, the more information we (pontentially) throw away; furthermore, pooling operations can lose relations between part of an image. So it's quite normal that there weren't big improvements; maybe it's possible to achieve higher accuracies on these data, with further tests without pooling operations.

In lights of these experiments, the model which best classify types of fruit and vegetable, with an accuracy of 100.0 %(0.0 variance) on training set, and 99.56 % (0.24 variance) on test set(10 epochs, 16 batch size, (1, 1) and (2, 2) as strides for convolution and max pooling), among all evaluated network is the following:

Table 17: Network structure with the highest accuracy on the test set.

Layer	**Output**
2D Convolution (3x3)	(32,32,32)
Max Pooling (2x2)	(16,16,32)
2D Convolution (3x3)	(16,16,64)
Max Pooling (2x2)	(8,8,64)
2D Convolution (3x3)	(8,8,128)
Max Pooling (2x2)	(4,4,128)
Flatten	2048
Dense	512
Dense	10

5 Classification based on fruit and vegetable varieties

5.1 Experiment description

This classification task involved 131 varieties of fruit and vegetable, from the most common to exotic ones.

I used the entire dataset mentioned earlier, that is composed of 67692 images for training, 22688 for test.

5.2 Network design process

At first, I picked as starting point, the network which worked better in the type classification experiment, with same hyper-parameters(batch size, epochs, strides, optimizer, loss function); I ran a first a grid search with a nested 5 fold stratified cross-validation, trying to figure out which combinations of filters and neurons generalizes better.

The set of hyper-parameters used in by the grid search is the following:

- # Neurons: 512, 1024, 2048
- # Filters combinations: (16, 32, 64), (32, 64, 128), (64, 128, 256)

Table 18: Network structure used for the tuning session.

Layer	**Output**
2D Convolution (3x3)	(32,32,# Filters)
Max Pooling (2x2)	(16,16,# Filters)
2D Convolution (3x3)	(16,16,# Filters*2)
Max Pooling (2x2)	(8,8,# Filters*2)
2D Convolution (3x3)	(8,8,# Filters*3)
Max Pooling (2x2)	(4,4,# Filters*3)
Flatten	(4*4*# Filters*3)
Dense	# Dense
Dense	131

The only difference is just the number of neurons in the output layer. I weighed again each class during the training, as every class were composed by a different number of samples, with the same method I used in the precedent section.

Table 19: Average training and validation accuracy.

Filters	**Neurons**	**Train. Acc**	**Val. Acc.**
16, 32, 64	512	99.98 %	99.97 %
16, 32, 64	1024	99.99 %	99.98 %
16, 32, 64	2048	99.99 %	99.97 %
32, 64, 128	512	99.98 %	99.97 %
32, 64, 128	1024	99.99 %	99.98 %
32, 64, 128	2048	99.85 %	99.82 %
64, 128, 256	512	99.99 %	99.98 %
64, 128, 256	1024	99.97 %	99.96 %
64, 128, 256	1024	99.96 %	99.94 %

The grid search observed high accuracies, over all folds and for all models, making not clear which combinations of filters and neurons were better. Hence, I evaluated all these models on the test set, in the same fashion I did before (10 evaluations, shuffling training data each time, for each model).

Table 20: Average accuracy and standard deviation on training and test data.

Filters	**Neurons**	**Train. Acc.**	**Train. Acc. σ**	**Test Acc.**	**Test Acc. σ**
16, 32, 64	512	99.76 %	0.40	95.55 %	1.10
16, 32, 64	1024	99.85 %	0.18	96.04 %	0.99
16, 32, 64	2048	98.77 %	1.5	93.44 %	2.84
32, 64, 128	512	99.81 %	0.37	96.25 %	1.15
32, 64, 128	1024	99.98 %	0.80	95.33 %	2.18
32, 64, 128	2048	99.87 %	0.27	95.70 %	1.01
64, 128, 256	512	99.25 %	1.41	94.66 %	2.44
64, 128, 256	1024	99.70 %	0.46	95.18 %	1.70
64, 128, 256	2048	99.37 %	0.7	94.60 %	1.90

All results have good accuracy, but also high standard deviation, which means that they are sensitive to training data order; this is normal, because we are dealing with more samples and more classes.

I decided to not perform other CV sessions, because weren't useful to identify good models. To decrease these fluctuations, I added a dropout layer before the output node, with a probability of 50.0 %, to the previous network.

Table 21: Network structure with dropout layer.

Layer	**Output**
2D Convolution - Kernel Size (3x3)	(32, 32, # Filters)
Max Pooling - Pool Size (2x2)	(16, 16, # Filters)
2D Convolution - Kernel Size (3x3)	(16, 16, # Filters*2)
Max Pooling - Pool Size (2x2)	(8, 8, # Filters*2)
2D Convolution - Kernel Size (3x3)	(8, 8, # Filters*3)
Max Pooling - Pool Size (2x2)	(4, 4, # Filters*3)
Flatten	(4*4*# Filters*3)
Dense	# Dense
Dropout (50%)	# Dense
Dense	131

Thus I evaluated again this new architecture, testing same values for filters, neurons and the remaining hyper-parameters.

Table 22: Average accuracy and standard deviation on training and test data, with one dropout layer (50%).

Filters	**Neurons**	**Train. Acc.**	**Train. Acc. σ**	**Test Acc.**	**Test Acc. σ**
16, 32, 64	512	99.88 %	0.20	96.61 %	0.65
16, 32, 64	1024	99.81 %	0.27	96.02 %	0.73
16, 32, 64	2048	99.67 %	0.36	95.37 %	0.90
32, 64, 128	512	99.89 %	0.09	96.45 %	0.79
32, 64, 128	1024	99.57 %	0.44	95.57 %	1.40
32, 64, 128	2048	99.66 %	0.27	95.15 %	1.07
64, 128, 256	512	99.86 %	0.12	96.43 %	0.9
64, 128, 256	1024	99.80 %	0.23	96.07 %	1.03
64, 128, 256	2048	99.66 %	0.28	96.94 %	0.98

The dropout reduced these fluctuations, but results were still very unstable.

Even with a lower probability (20%), results didn't change that much.

Table 23: Average accuracy and standard deviation on training and test data, with one dropout layer (20%).

Filters	**Neurons**	**Train. Acc.**	**Train. Acc. σ**	**Test Acc.**	**Test Acc. σ**
16, 32, 64	512	99.83 %	0.14	96.11 %	0.62
16, 32, 64	1024	99.75 %	0.18	95.56 %	0.46
16, 32, 64	2048	99.76 %	0.23	95.53 %	0.94

Continued on next page

Filters	**Neurons**	**Train. Acc.**	**Train. Acc. σ**	**Test Acc.**	**Test Acc. σ**
32, 64, 128	512	99.84 %	0.19	96.34 %	0.55
32, 64, 128	1024	99.57 %	0.39	95.29 %	1.38
32, 64, 128	2048	99.73 %	0.25	95.32 %	1.11
64, 128, 256	512	99.75 %	0.26	95.61 %	1.30
64, 128, 256	1024	99.55 %	0.13	95.47 %	1.17
64, 128, 256	2048	99.65 %	0.38	95.49 %	1.01

Models with a lower dropouts probability seems to be less stable, especially when the number of filters and neurons is higher.

As I said in the previous section, max pooling layers can generate information loss; since we are dealing with more classes and more samples, this could be a problem so, I removed just one convolution and max pooling layers.

Table 24: Network structure with two convolutions and max pooling layers.

Layer	**Output**
2D Convolution - Kernel Size (3x3)	(32, 32, # Filters)
Max Pooling - Pool Size (2x2)	(16, 16, # Filters)
2D Convolution - Kernel Size (3x3)	(16, 16, # Filters*2)
Max Pooling - Pool Size (2x2)	(8, 8, # Filters*2)
Flatten	(8*8*# Filters*2)
Dense	# Dense
Dropout (50%)	# Dense
Dense	131

Table 25: Average accuracy and standard deviation on training and test data, with 2 convolution and 2 max pooling layers.

Filters	**Neurons**	**Train. Acc.**	**Train. Acc. σ**	**Test Acc.**	**Test Acc. σ**
16, 32	512	99.94 %	0.07	96.35 %	0.70
16, 32	1024	99.96 %	0.03	96.48 %	0.53
16, 32	2048	99.87 %	0.18	95.83 %	0.78
32, 64	512	99.99 %	0.08	96.84 %	0.60
32, 64	1024	99.90 %	0.12	96.50 %	0.63
32, 64	2048	99.88 %	0.10	95.93 %	0.62
64, 128	512	99.97 %	0.02	96.95 %	0.41
64, 128	1024	99.96 %	0.05	96.55 %	0.62
64, 128	2048	99.85 %	0.23	95.71 %	0.79

The differences with previous models are little, which means those 2 layers were extra useless complexity.

I noticed during training phase that training error, after 10 epochs were high (~ 5.0); in order to adjust this, I increased number of epochs from 10 to 20 and evaluated again the previous networks.

This helped to bring down training error near to 1.0 during training, nevertheless, just few models gained some benefits.

Table 26: Average accuracy and standard deviation on training and test data obtained with 20 epochs of training.

Filters	**Neurons**	**Train. Acc.**	**Train. Acc. σ**	**Test Acc.**	**Test Acc. σ**
16, 32	512	99.96 %	0.05	96.99 %	0.35
16, 32	1024	99.97 %	0.01	96.73 %	0.51
16, 32	2048	99.86 %	0.31	96.01 %	1.02
32, 64	512	99.97 %	0.03	97.21 %	0.27
32, 64	1024	99.93 %	0.07	96.41 %	0.36
32, 64	2048	99.83 %	0.41	96.06 %	1.33
64, 128	512	99.97 %	0.02	97.12 %	0.49
64, 128	1024	99.96 %	0.04	96.67 %	0.61
64, 128	2048	99.91 %	0.07	96.00 %	0.60

After these experiment, I conclude saying that the network which classified better the test set, with an accuracy of 100.0% on training set(0.0 variance) and 97.21% on test set(0.27 variance)(20 epochs, 16 batch size, (1, 1) and (2, 2) as strides for convolution and max pooling) is this:

Table 27: Network structure with the highest accuracy on the test set.

Layer	**Output**
2D Convolution - Kernel Size (3x3)	(32, 32, 32)
Max Pooling - Pool Size (2x2)	(16, 16, 32)
2D Convolution - Kernel Size (3x3)	(16, 16, 64)
Max Pooling - Pool Size (2x2)	(8, 8, 64)
Flatten	(8*8*64)
Dense	512
Dropout (50%)	512
Dense	131

6 About the previous content

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work.

I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: (2012).

[2] Francois Chollet et al. *Keras*. 2015. URL: `https://github.com/fchollet/keras`.

[3] *Fruits 360 - Git Hub*. URL: `https://github.com/Horea94/Fruit-Images-Dataset`.

[4] *Fruits 360 - Kaggle*. URL: `https://www.kaggle.com/moltean/fruits`.

[5] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: (2017).

[6] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[7] "Horea Muresan and Mihai Oltean". *Fruit recognition from images using deep learning*. URL: `https://www.researchgate.net/publication/321475443_Fruit_recognition_from_images_using_deep_learning`.

[8] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: (2014).

[9] Christian Szegedy et al. "Going Deeper with Convolutions". In: (2015).