

Image Classification with Neural Networks

- *Statistical Methods for Machine Learning* -

Nicolò Cavedoni

May 7, 2021

1 Assigned Project

Use Tensorflow 2 to train neural networks for the classification of fruit/vegetable types based on images from the dataset at <https://www.kaggle.com/moltean/fruits>. Images must be transformed from JPG to RGB pixel values and scaled down (e.g., 32x32). Use fruit/vegetable types (as opposed to variety) as labels to predict and consider only the 10 most frequent types (apple, banana, plum, pepper, cherry, grape, tomato, potato, pear, peach). Experiment with different network architectures and training parameters documenting their influence of the final predictive performance. While the training loss can be chosen freely, the reported test errors must be measured according to the zero-one loss for multiclass classification.

2 Dataset

The dataset comes with two main folders Training and Test, each one containing 131 sub-folders named after the fruit/vegetable categories. Every subfolder is filled with 100x100 .jpg images on a white background depicting different point of views of the same fruit, by rotating it. This is already sufficient material to train a neural network efficiently, but to follow the assignment, we need to remove each subfolder that doesn't fit the description. This is done manually. About half of the fruit/vegetables gets cut in the process, remaining with all the varieties of apples, bananas, plums etc... with a total of 62 categories.

We need to group all the varieties in the same class, but the filenames are overlapping so it's not wise to do it manually; luckily, the different subfolders have the class name in common (i.e "Apple 1", "Apple 2", "Apple 3"), so we can easily assign the same label to all the varieties of the same fruit by checking if the parent directory contains that fruit name. Figure 1 below shows the resulting division of the samples. We can immediately spot that the ratio between the training set and the test set is 3:1. Moreover, during data preprocessing the training set is split between training and validation sets with a ratio of 5:1.











APPLE (0)	BANANA (1)	CHERRY (2)	GRAPE (3)	PEACH (4)
				
6404 + 2134	1430 + 484	3444 + 1148	4401 + 1476	1722 + 574
PEAR (5)	PEPPER (6)	PLUM (7)	POTATO (8)	TOMATO (9)
				
5037 + 1689	2478 + 826	1767 + 597	1803 + 601	5103 + 1707

Figure 1: Classes with their numeric label. In blue, the number of samples in the format "training + test"

2.1 Preprocessing the Dataset

To manage the dataset we use the `tensorflow.keras.preprocessing` library. This allows us to load all the image files in a format fit for Tensorflow and operate on all of them easily. Method `image_dataset_from_directory` satisfies the requests of transforming a jpg into a 3-channel (RGB) matrix of pixels, and scales it down at will. We opted for a 32x32 resolution as it's the best compromise between our machine's computational power and a decent level of detail for the input. Smaller images would be processed faster, but it's not worth the drop of accuracy. Then we just follow the best practices for datasets to boost performance and avoid input bottlenecks, so:

`Dataset.prefetch()` prepares the next batch of images while processing the current one;

`Dataset.cache()` stores the images in memory after the first load.

These operations can be tuned manually, but we rely on the autotune option offered by the framework, that chooses the level of parallelism and the width of buffers depending on the context. The last piece of data preprocessing is the normalization of the data, transforming the [0-255] RGB range into [0-1]. This will be helpful during the gradient descent, leading to a faster convergence. By the way, we choose not to normalize all the images via function map, but to add a normalization layer in each model right after the input one.

3 The Base Model: ReLU CNN from Tensorflow guides

Since we selected *Tensorflow* as framework, it seemed natural to train its basic architecture for image classification on the Fruits-360 dataset. The model can be found at <https://www.tensorflow.org/tutorials/images/classification> and it's a *Convolutional Neural Network* (CNN) with *ReLU*s as activation functions. Figure 2 shows the starting architecture. Three convolutional layers of

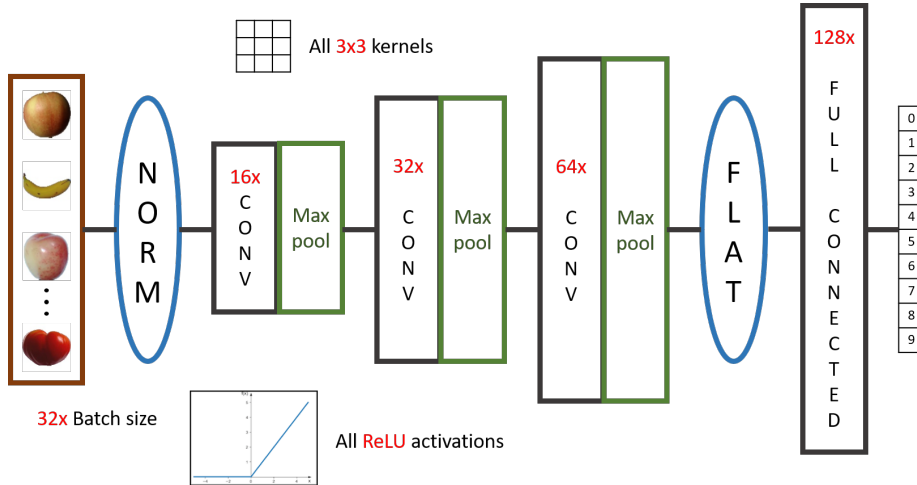


Figure 2: CNN with ReLUs. Parameters in red are to be tuned

increasing complexity are attached to three Max Pooling layers, and this is the most classical and immediate approach to down-sample the input, opposed to choosing a stride factor greater than 1. The increasing number of filters is because deeper layers produce more complex feature maps, so we want to be able to learn as many different complex patterns as we can, instead of focusing on less important features learned earlier. It's just a convention to use powers of two for the number of filters, but it provides no actual benefit to performance, so we could break that convention when we tinker with that parameter.

The first layer just normalizes the input, as we announced in the previous section. The flattening layer is needed to transform the output from the convolutions into a readable input for the dense layer, which is activated by ReLU as the others. In the context of multi-class classification, the output layer should implement the Softmax activation function, as it returns a probability distribution and not a matrix of marginal probabilities. Our experiments proved that activating the output layer with Softmax doesn't change the prediction accuracy, neither the training time. It's needed to evaluate the machine's predictions in term of probabilities instead of logits, but we're not going to need it for this work, so we prefer to keep the same activation function through all the layers.

3.1 Evaluating the Base Model

Specifics ask us to evaluate each model through the 0-1 loss for multiclass classification. Tensorflow offers a series of classic evaluation metrics, like Mean Squared Error, various Cross Entropies and a lot of metrics based on true/false positive and negative results. The zero-one loss corresponds to the metric called *accuracy*, which is nothing more than the number of correct classifications divided by the total of those classifications. This metric has to be specified inside of `model.compile`, along with the loss function and the optimizer (the learning algorithm). For the base model, the guide suggests to use the sparse categorical cross-entropy loss, i.e. the standard cross-entropy for multiple classes identified with integers; for the optimizer, we're adopting the Adam, a combination of Adaptive Gradient and Root Mean Square propagation, that's thought to be the most refined and modern learning algorithm.

After the compilation, the model is ready to be trained via `model.fit` on the Fruits-360 training set for 10 epochs. The number of epochs is arbitrary, but thanks to the preprocessing the only critical iteration is the first, so 10 epochs makes it easier to follow the training process without taking too much time. We're able to track the machine's accuracy and loss during training (both on training and validation set), and we expect very good results due to the optimal conditions of the dataset and the architecture being state-of-art, even if simple. As expected, our first model tested great on a well-built dataset. The average

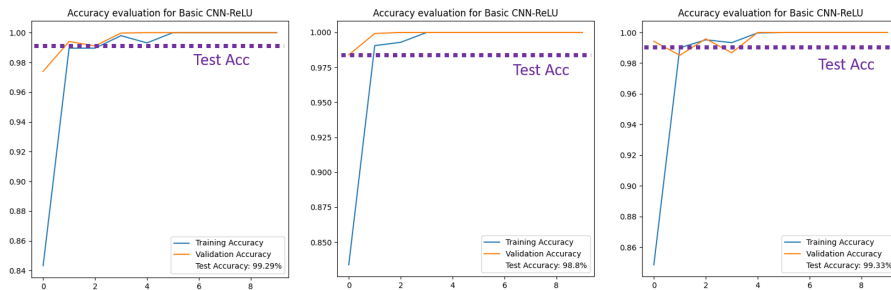


Figure 3: Base Model's measured accuracy

test accuracy is around 99% and it can probably be improved again if we provide more power to the architecture. We can see that there's a gap between the test accuracy and the validation accuracy, even if we're talking about a difference of 1% or lesser: This is because the Fruits-360 dataset comes already split into train and test portions, while the data used for cross-validation come only from the training set. This means that we can expect some slight divergence in the distributions of those two, that results in a slightly different performance on both. But still, the measured test accuracy is great, so we can conclude that the CNN model from Tensorflow guides is excellent to learn the Fruits-360 dataset.

4 Tinkering with the Base Model

Usually, neural networks need to be fine-tuned with some experiments, to find the optimal configuration to perform their task. Luckily (or not?) the first model we tried scored great, but we can still tinker with its hyperparameters to experience different results, mostly interested in bad performances rather than increasing the first's.

4.1 Number of Filters

In a CNN, every convolutional filter holds a feature map. This means that for every filter, the network is able to learn a specific pattern of pixels. There's no standard to choose the number of filters, because it heavily depends on the dataset used to train. Too few filters, and the CNN is missing important features; too many filters lead to overfitting, with the CNN learning non-generic details from the input or some noise in the images.

For the first experiment, we're going to check if this network can actually overfit the dataset, considering that the only prophylactic technique adopted for now is to shuffle the images when loading (no dropout, no regularization, no manual data augmentation). Figure 4 shows the comparison between three identical models which differ in the number of learning units implied in the process. As

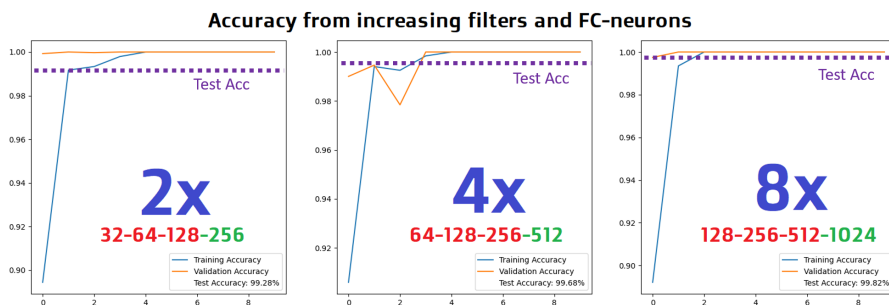


Figure 4: Accuracy when increasing number of units

expected, the model is more accurate when provided with more power, but it still shows no sign of overfitting. What really changes is the number of epochs needed to achieve the peak accuracy, or else, the number of samples required to fully learn the features of all the ten fruit classes. Choosing too many epochs is another way to obtain overfitting, and Figure 4 definitely shows that those machines don't need more than 4 epochs to peak. Even with too many neurons and too many epochs, there was no difference between train and test accuracy and the loss never showed to be increasing at all. Given that the 8x machine needed around 45 minutes to be trained (against the 10 minutes for the previous one), we decided to stop there and conclude that the missing overfitting is caused by the optimal conditions of the dataset, which is very rich, clean and variant.

Now we experiment with the inverse process, and see if cutting down units results in a worse performance: In reference to Figure 3 we can definitely confirm

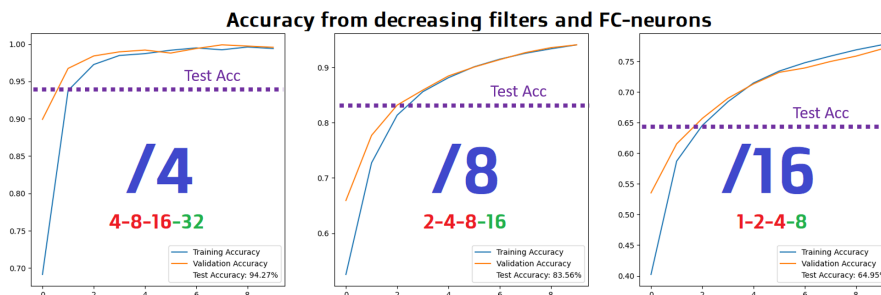


Figure 5: Accuracy when decreasing number of units

that decreasing filters and neurons harms the machine’s accuracy. Even with a minimal number of learning units, the CNN maintains a decent amount of accuracy: The 1-2-4-8 configuration can learn very few patterns, but seems that they’re still sufficient to classify around 60% of the samples. The explanation could be that fruits don’t have many traits to distinguish between them.

4.2 Kernel Size

We used the term "filter" up until now to refer to the CNN’s basic learning unit, but it’s equally called *kernel*. The kernel is an $N \times N$ matrix which shifts around the image acting like a mask, and returning the dot product between the cell values and the underlying pixels. The developer can set the kernel size, but the actual content of the matrix are what’s learned by the network. The size of a filter determines how big are the details learned, along with their respective combinations (in the FC layer), so there are cases in which a machine performs better when focusing on large details. The most modern approach is to use small filters, but in larger quantity, in respect to the old-school big filters with few units.

We experimented on the Base Model by changing the kernel size of each convolutional layer and saw no difference at all, except for the training time which was faster with smaller kernels (2x2 and 1x1) and slower with bigger (4x4, 5x5, 7x7). The conclusion is that there’s no particular correlation between the kernel size and the classification accuracy.

Research says it’s not advised to use even-sized kernels like 2x2 or 4x4 because they can cause some distortion in the layer output, that can results shifted by half of a pixel. Also, 1x1 kernels are mostly useful to reduce the dimensions of a multi-channel input, when applying a number of filters lesser than the number of channels. But this was not the case of our Base Model, that felt no difference at all from the kernel size changes.

4.3 Number and Type of Layers

For this set of experiments, we added and removed core layers of the Base Model and tried to understand their impact on the machine’s performance. When removing layers, we kept the ones with more filters, to stay as much similar as possible to the original architecture. When adding layers, we follow the same reasoning and add some with few filters, but always in a pair ”Convolutional-MaxPooling”.

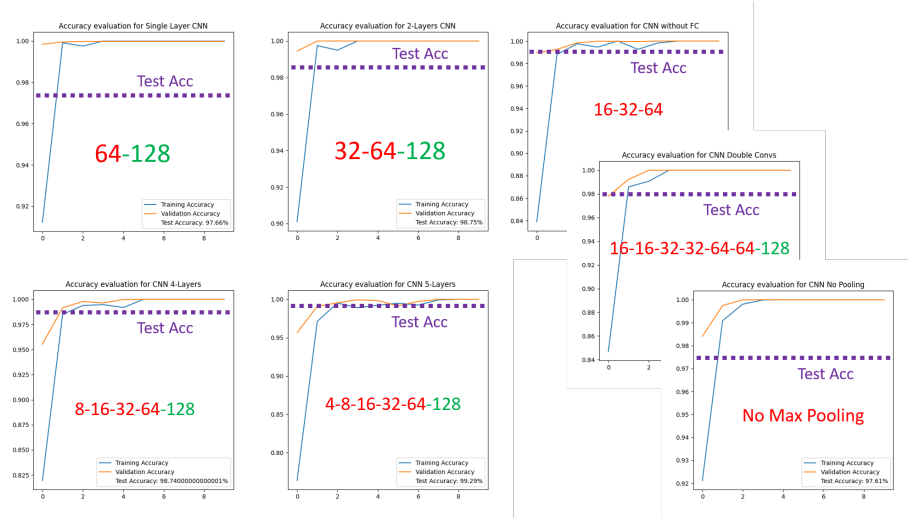


Figure 6: Accuracy when changing the layers

As Figure 6 shows, this kind of architecture on the Fruits-360 dataset is not that sensitive to the presence of 2, 3 or 4 layers of convolutions, most if the number of filters doesn’t scale much. What a layer does is to apply the filter while shifting it around the image, producing a feature map that contains the informations about some trait of the image. After all the filters have done this, the output is made non-linear (with ReLU in our case) and then reduced in its dimension by pooling. By doing this, we allow the next convolutions to be on a different image than the previous, specifically on a rescaled one, looking for smaller details. Having only one convolutional layer doesn’t allow to capture detailed features, but seems that in case of fruits this is not really relevant, except for just 1% of the samples tested (this could be overfitting, but its causes are unclear). The 5-layers architecture instead performed better, but required around 3 times more time to be trained. Notice that for a 32x32 input image, we are only able to do the pooling operation five times at maximum, with the sixth trying to downscale a single pixel and failing (at least when using a valid padding in the pooling layer).

Then we tried some other peculiar configuration like:

- Removing the Dense layer before the output. This operation prevents the network to learn the non-linear combinations of the feature maps, but it didn't affect the predictive accuracy at all. Probably the single feature maps are enough to correctly classify fruits on this dataset.
- Doubling the convolutions before every pooling. This is a more modern approach that aims at extracting the most features before a pooling, and then resize and repeat the process at a lower resolution. The result is worse than the Base Model by a 1%, probably due to a little overfitting.
- Removing the pooling layers at all, and using only convolutional ones. Max pooling just takes the maximum values from groups of features, and this operation keeps the relevant features for the next layer and discards the possible noise. By removing them, all the noise in the training set is learned as a feature, and overfitting comes along. In practice, the dataset is very clean and refined, so there isn't much noise to inadvertently learn, but the graph is pretty clear in showing signs of overfitting, even if the general accuracy remains high.

4.4 Activation Functions

As last experiment with the Base Model, we tried to replace the activation functions of the Conv layers with others. The base model features the ReLU, the most used activator for convolutional layers since some year, which is a simple $\max(0, x)$. Its success is due to its simplicity both in activation and during gradient computation, and mostly because it doesn't suffer from vanishing gradients like other functions we're going to see.

- *Sigmoid* ($\frac{e^x}{1+e^x}$): The sigmoid activation is the one that gave the worst results. The average accuracy dropped by roughly 10%, even though testing and validation were able to reach the peak. Apart from the big overfitting gap, what's curious to notice is the shape of the accuracy curve during training. The network needed more time to get to 100% validation, and behaved like the bad versions of it, when were removed lots of units (but still scoring better results).

- *Hyperbolic Tangent* ($\frac{e^x - e^{-x}}{e^x + e^{-x}}$): The same machine behaved diametrically opposite when activated with this function. Weights and biases needed just 1 epoch to be perfectly tuned, and then it stood still at the peak accuracy for the rest of the training. We tried this configuration three times and always got the same behaviour, to be sure that it wasn't an outlier. The accuracy was not affected (staying around 99%), but we can state that the Hyperbolic Tangent added some benefit compared to the ReLU.

- *Softplus* ($\ln(1 + e^x)$): Softplus is a famous variant of the ReLU, and became popular as it derives into a sigmoid. The so-called "smooth ReLU" is also more sensitive around zero and is continuously differentiable compared to the basic version (which it's not in zero). Although this works in theory, the experiment showed worse results with Softplus than with ReLU, scoring only 95% accuracy. This is not strange at all, many studies proved that Softplus is not that advantageous in practice, most when the task is image classification.

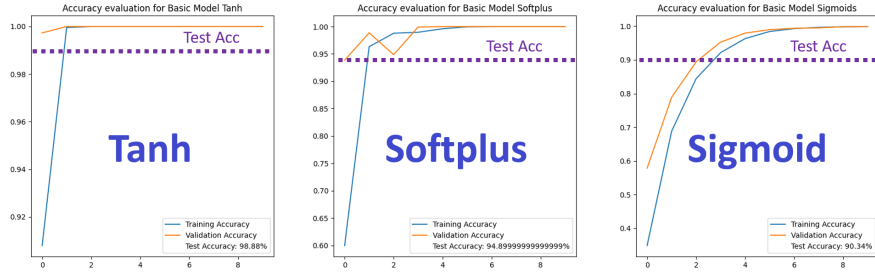


Figure 7: Accuracy when changing the activation functions

5 Multilayer Perceptron

The MLP model can be considered "legacy", compared to the newest approach to classification with convolutional networks. But adopting an MLP can still be a reasonable choice, mostly because it's way faster to train. Since we know an MLP will perform worse than the CNN Base Model, we're interested in doing some tests on its architecture and check which configuration produces the best results compared to the ones we've already seen.

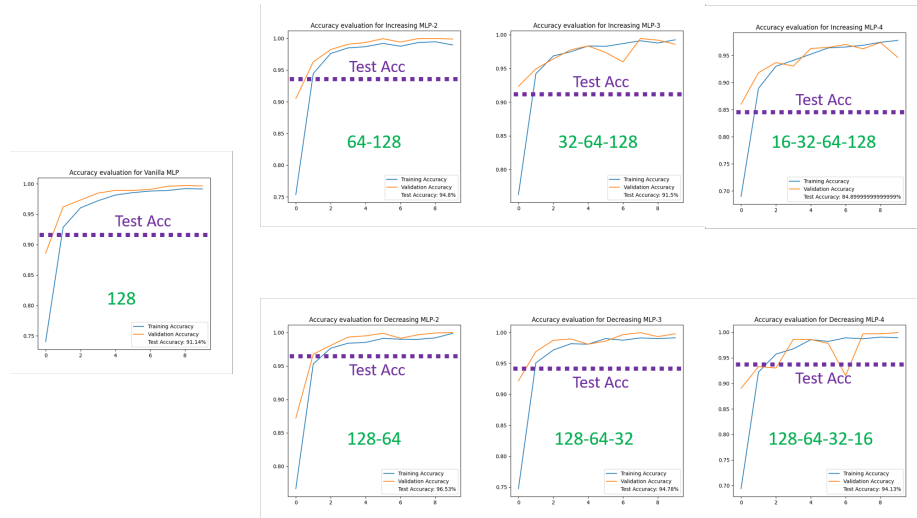


Figure 8: Accuracy in different MLP architectures

For this comparison we only tried different numbers of hidden layers and neurons, and it's necessary to say that every configuration was quite inconstant and needed some tries to find an average value. This is mostly due to the number of epochs needed to tune the weights, that was probably insufficient for the

networks to fully converge. We kept the number of epochs fixed at 10 to achieve a clearer comparison with the Base Model.

Figure 8 shows the results of the experiments on the multilayer perceptron. The number of neurons was kept as similar as possible to the compared one (although the CNN has "filters" that are not really neurons), and we tried to see if the network was better when the neurons in each layer were increasing or decreasing. Referring to this dataset, we can state that a decreasing number of neurons in each layer performs better than the opposite. Furthermore, the more hidden layers we added, the worse results we obtained, but this was not true for every case: In fact, the simplest MLP model with only 1 hidden layer scored a lower prediction accuracy compared to the 2-layers one.

As a final experiment with the MLP, we picked the top-scorer (128-64 architecture) and tried giving it more units to see if it could reach the 99% accuracy of the CNN. As the neurons multiplied by 2, 4 and 8, the test accuracy was pretty much the same, with the training time increasing exactly by the same multiplicative factor. There is surely some combination of parameters able to score better than that on the Fruits-360 dataset, but at the end a CNN is always powerful enough to avoid all the refinement process on an MLP.

6 Some Conclusive Words

Bouncing back to our original assignment, we were asked to try different neural network architectures on this dataset, and it could be pointed out that the CNN is just a modern and richer version of an MLP (it also features an actual perceptron in the last hidden layer). We chose to analyze those two models because it's widely known they're suitable for image classification, while other networks like *Recurrent* networks or *Long-Short Term Memories* are better for analyzing texts, sounds, temporal sequences.

This paper aimed at retracing some of the core concepts of image classification, through an analysis of the predictive performance of some models that apply these concepts versus some which don't. The Fruits-360 dataset we worked on is very well-made, and I think it's probably not the best for experimental purposes. While some little variance and overfitting in the predictions are always expectable, it's not easy to understand the impact of small improvements when the loss and accuracy stay pretty much the same, even though in theory they should improve as well. Our conclusion is that the CNN model from the Tensorflow tutorial (the so-called Base Model) is amazing in classifying this data, and it can be improved very little by boosting up the number of filters, at the expense of a huge slow-down in training time. One peculiarity is that the Hyperbolic Tangent activation, usually considered worse than the ReLU, converged faster and stayed at peak without overfitting, and it could deserve some additional attention in the future.

7 Declaration

I/We declare that this material, which I/We now submit for assessment, is entirely my/our 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/our work. I/We 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/us or any other person for assessment on this or any other course of study.