Machine Learning project
Image classification with Neural Networks

Nicole Maria Formenti 941481
Università degli Studi di Milano - DSE

2020
October

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.

ABSTRACT

The project's, *image classification with Neural Networks*, central topic is that of building a Convolutional Neural Network architecture which is trained and tested on the *Fruits 360* dataset from *Kaggle*. It is a multi-classification problem for fruit and vegetable images. The main goal is to find a model with the lowest possible zero-one loss rate, by applying architectures and methods widely described and used in literature.
The models considered are the *VGG*, the *GoogleNet* with both naïve and standard inception modules, and the *Resnet*. Furthermore, the techniques that will be applied to improve the performance of the networks are the batch normalisation, the fine-tuning of the learning rate, the weights regularisation, the dropout, the early stopping, the image augmentation and the use of images in grayscale format.
In the end, the architectures that better perform in each group are, respectively, the three blocks *VGG* with 16.3% of errors, the two blocks *GoogleNet* with naïve inception module with an error rate of 3% and the two blocks *Resnet* with 3.8% of errors. In all the cases, batch normalisation, image augmentation and a learning rate of 0.00001 have improved the result, whereas, concerning the two last architectures, dropout also had a positive effect. In fact, they ended up being the two models with the best predictive capability.

INTRODUCTION

This report content is about the classification of images of fruit and vegetables based on their type, contained in the *Fruits 360* dataset stored on *Kaggle*. Specifically, it is a multi-classification problem relying on the usage of neural network architectures.
The original dataset contains 131 fruit and vegetables varieties, but for this analysis the type, which is the more general category, will be used instead. Moreover, it will be considered only the type of the 10 most frequent fruits and vegetables, namely *apple, banana, plum, pepper, cherry, grape, tomato, potato, pear, peach*.
The main aims of the analysis are, on the one hand, to obtain the best possible prediction by using neural network architectures, on the other, to comment the different architectures used and the process of experimentation, along with providing a theoretical background. The networks used are based on three different architectures which are among the top-performing ones for image classification, namely *VGG*, *GoogleNet*, with both naïve and bottleneck version of the inception module, and *ResNet*.

DATASET

The reduced version of the *Fruits 360* dataset contains 10 classes of fruit and vegetables which are the following: apple, banana, plum, pepper, cherry, grape, tomato, potato, pear, peach.
The dataset is split into training set, containing 32.607 images, and the test set, with 10.906 images. The original images have a shape of 100x100 pixels, which has been reduced to 32x32. In order to feed them to a convolutional neural network, they have to be represented in tensors format. Since the images are transformed using the RGB format, every image has three different dimensions or channels, each of them representing the intensity of one of the three primary colours red, green and blue, with a value between 0 and 255. Both the width and the height of the image have a value of 32, each scalar representing a single pixel.
The final tensor has four dimensions, one representing the number of instances in the dataset, one for the width and one for the height of the image, plus another one for the number of channels.

The last step is to rescale the pixels' value in an interval between 0 and 1. This process is called normalisation and it is useful so that the inputs have a similar distribution; as a consequence, the magnitude of the values that features can take is similar. Hence, all the weights are updated with a similar speed and convergence is easier and faster.
Another requirement is to provide the network with images' labels in one-hot encoded format, meaning that the one-dimensional vector containing a number between 1 and 10 associated to a certain category becomes 10-dimensional, where each position corresponds to a category. If the image belongs to a certain class, the corresponding position takes value 1, whereas the others have value 0.

Later, the dataset will be converted to grayscale to test whether the network performance improves. Transforming an image in grayscale format means to reduce the number of channels from three to one, where the pixels can take a value from 0 to 256, with 0 representing black and 256 white.

ARCHITECTURES

This analysis makes use of neural network architectures, in particular convolutional neural networks, being the best solution for image classification available so far.
The main difference with traditional neural networks is the presence of two additional types of layers, specifically convolutions and poolings.

Convolution operations are carried out by moving a filter or kernel with a certain fixed dimension along the image and performing a dot product between the kernel and the value of the matching grid. The number of the possible horizontal and vertical alignments with the original image or layer corresponds to the height and width of the next layer. Each filter must have the same number of channels as the layer it is applied to. By performing an inner product between the different channels, a scalar is then returned.
More than one filter can be used, thus the number of features of the next layer will be equal to the number of filters applied. What Kernels basically do is to map certain features in a specific spatial area, from the simplest ones such as edges and lines to the most complex ones, reaching a higher level of abstraction as the network deepens. That's why the output of a filter is also called feature map. Common filters' dimensions are 1x1, 3x3 or 5x5.
The sliding step of the filter is called stride. A higher stride reduces more the level of granularity of the convolution.
Applying a convolution with a kernel larger than 1x1 has the natural consequence of shrinking the image, moreover loosing precious information along the borders. In order to avoid it, the so-called padding can be used. This consists in edging the image with many 0 to keep the original dimension.

Next, pooling is an operation used to reduce the size of a layer, when the pooling dimension is greater than 1x1, producing for each feature map another feature map. The number of filters remains unchanged. It works by taking a region of a certain size in the original layer and returning its maximum value, in the case of max-pooling, or the average of all the values, in the case of average-pooling. This operation is repeated for every area of the layer and the step is defined again by the value of the stride.

From now on the discussion will focus on the features in common between all the architectures.
Regarding the assessing of the model's performance, the zero-one loss (1) is used in order to measure the ability of the network to generalise on the test set after having being trained on the training set. The zero-one loss is reported both in absolute and relative terms.
This loss is used for categorisation problems, assigning a value of 0 if the predicted label value is equal to the true value, 1 otherwise. The total loss is then calculated by summing the assigned values.

$$L(y, \hat{y}) = \begin{cases} 1, & \text{if } y = \hat{y} \\ 0, & \text{if } y \neq \hat{y} \end{cases} \tag{1}$$

For the sake of completeness, the Accuracy metrics is reported too since it corresponds to the opposite of the zero-one loss in relative terms.

As for the training of the model, the weight and biases have been optimised through the Adam optimisation algorithm. Adam stands for Adaptive Moment Estimation and it is based on stochastic gradient descent while adapting the parameter learning rates using an estimation of first-order (mean) and second-order moments (uncentered variance). Its main advantage is that of delivering quite good results while at the same time being computationally efficient. It works especially well when dealing with large datasets and many parameters. The loss function to be minimised by Adam in order to update the weights is the categorical cross-entropy (2), also called Softmax loss. It is the union of the Softmax activation and the Cross-Entropy loss, and it is specifically used for multi-class classification problems. It takes the labels in one-hot encoded format. The Softmax loss formula is the following:

$$L(y, \hat{y}) = -\log\left(\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}\right) \tag{2}$$

Where K is the number of classes, z_i is the input vector of the softmax function and $\sum_{j=1}^{K} e^{z_j}$ is the normalisation term. In order to minimise a loss function, one needs the gradient, which is the partial derivative of the loss with respect to the weights and biases. In the case of multi-layer neural networks, the gradient is learnt through the chain rule of differential calculus. That's the reason why non-differentiable functions, such as the zero-one loss, cannot be used to train neural networks.

The last point in common between all the architectures is the usage of the ReLU activation function (3), namely Rectified Linear Unit, after the convolutional layers and the final fully connected flattened layer. This function returns 0 if it receives a negative number and the value itself otherwise

$$f(x) = \begin{cases} 0, & \text{if } x < 0 \\ x, & \text{if } x \geq 0 \end{cases} \tag{3}$$

Furthermore, the last activation function used for classifying the 10 labels is the Softmax function (4). The Softmax function works by assigning the probability, from 0 to 1, that an instance has of belonging to each class, thus it is suited for a multi-class problem. In addition, all the probabilities sum up to 1.

$$f(y_i) = \frac{e^{y_i}}{\sum_{j=1} e^{y_j}} \tag{4}$$

Last but not least, instead of using the whole dataset to train the model all at once, smaller samples of data called mini-batches are created. Each time a single mini-batch is fed to the network, weights and biases are updated. The size of a mini-batch is set to 100. Given that the training set contains 32.607 instances, the total number of batches is about 326.
When the entire dataset, meaning all the mini-batches, has been used once to train the network, one says an epoch has passed. Hence, the number of mini-batches corresponds to the number of

iterations required to complete an epoch. Usually it is not enough to use only one epoch since it can lead to underfitting, so the number of epochs used for this analysis is 100.
It is also important to avoid overfitting due to an excessive number of epochs. It can be assessed by looking at how the test error changes through the different epochs, if it starts increasing after a certain number of epochs, there probably is overfitting. Vice versa, if the error converges in the long run.

After having presented the general features, an overview of the specific models considered will be provided next. All of them have been presented during the *ILSVRC* (ImageNet Large Scale Visual Recognition Challenge), which is a yearly event where different algorithms for object detection and image classification compete.
For the purpose of this project, a reduced version of them have been used for computational time reasons.

VGG

VGG has been presented in the 2014 *ISLVRC* edition. Although it wasn't the winner, it still provided a top-5 error rate of 7.3%, which is the percentage of classified test images for which the correct label doesn't appear among the 5 most probable labels. The idea behind it is to reduce the size of the filter by increasing the number of convolutions, thus the depth of the network. In this way the filters are able to capture more sophisticated features. Moreover, the net requires fewer total parameters and introduces more non-linearity due to the presence of more ReLU activation functions. The original version is composed by 16 layers.

All the filters of the convolutional layers have a dimension of 3x3, and the images required as inputs have a size of 224x224 pixels and are in RGB format.
The first two blocks are formed by two consecutive convolutions with stride and padding equal to 1, followed by a max-pooling layer of size 2x2. The last three blocks comprehend three convolutions with the same stride and padding as before, followed again by a max-pooling layer. The number of filters used starts with 64 in the first block and doubles after each block. At the end, three flattened fully connected layers are placed, the first two with 4096 nodes and the last one with 1000 nodes. After, there is the softmax layer for classification.

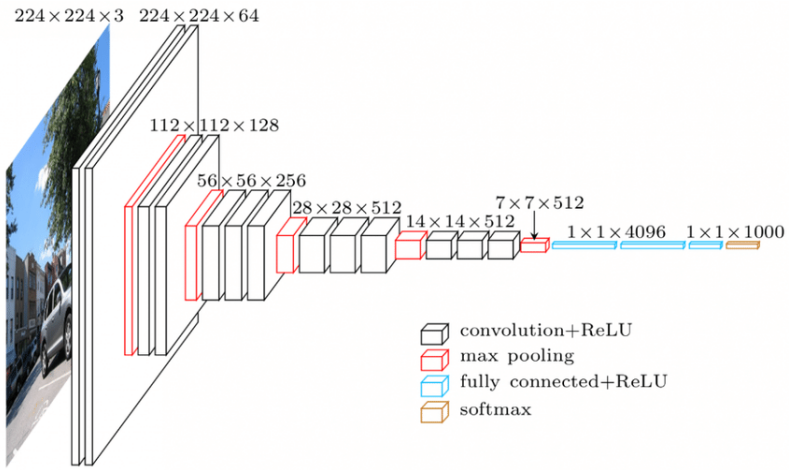


Figure 1: Full 16 layers *VGG* architecture

For this project, four different architectures inspired to *VGG* have been used.
All the filter used for convolutional layers have a size of 3x3 and the max-pooling layer of 2x2, as in the original network. A padding and stride of 1 is used. Moreover, the final layers are the same for all the models, with a fully-connected layer composed by 128 nodes and a final softmax layer. The four blocks *VGG* have, instead, 256 nodes in the final dense layer. No biases have been included.

The first architecture uses a single block with two convolutions, each with 32 filters, followed by a max-pooling layer. Then, in the second one another block of two convolutions with 64 filters is added. The remaining models are respectively a three blocks *VGG*, with other two convolutions having 128 filters, and a four blocks *VGG*, with an additional block composed once again by two convolutions with 256 filters. After each block a max-pooling layer is inserted. The total number of layers for each architecture is three for the former, five for the second one, seven for the third one and nine for the latter.
One should notice that the pooling and softmax layers are not considered when determining the depth of a network.

Afterwards, the second architecture, *GoogleNet*, will be introduced.

GoogleNet

GoogleNet or *Inception V1* was the winner of the same edition of the *ISLVRC* where the *VGG* has been first presented. It delivered a top-5 error rate of 6.7%, 0.6 percentage points below *VGG*.
The main feature of this architecture is the use in the intermediate layers of the so-called inception module.
When using larger filters, the information captured is less precise and it refers to a wider area, whereas with smaller filters, more detailed information about a specific region can be acquired. Since it is not possible to know a priori what level of granularity should be applied to every different area, an alternative solution to the use of sequential small filters and deeper net, such as in the case of *VGG*, is to apply different sized convolutions in parallel. The filters have a dimension of 1x1, 3x3 and 5x5. Afterwards, the resulting filters are concatenated and they are fed to the next layer. This has the advantage of providing a certain amount of flexibility without excessively increasing the depth of the network when it is not required.
There exist two different versions of the inception module, the naïve version and the version using bottlenecks, which provide a dimensionality reduction.
As for the former, three parallel convolutions are performed using different filter sizes, namely 1x1, 3x3, 5x5. There is also an additional max-pooling operation. In order to concatenate the resulting filters, they need to have the same shape; that's the reason why it is applied a padding that maintains constant the width and height of the input when being convolved. In addition, the stride is equal to 1.

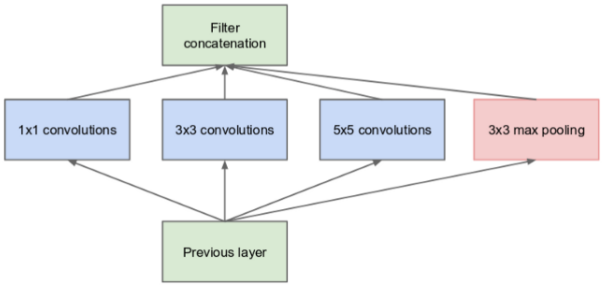


Figure 2: Inception module, naïve version

The main problem with this structure is that having to perform many convolutions using wide filters is computationally expensive. In order to reduce the depth of the input layer, namely the number of filters, a further convolution of size 1x1 is applied before the 3x3 and 5x5 convolutions and after the 3x3 max-pooling. Convolutions performed with a 1x1 filters are much faster than those that use bigger kernels. These additional blocks are referred to as bottleneck operations and help reducing the computational time required to train the network.

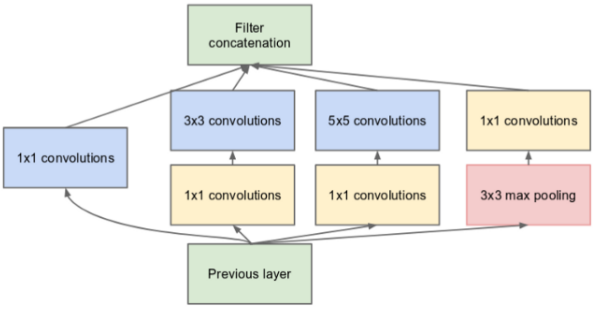


Figure 3: Inception module, bottlenecks version

The *GoogleNet* architecture is quite complicated, having 22 total layers.
The first few layers don't make use of inception module. A first convolution with a filter of 7x7 is followed by a 3x3 max-pooling, then other two convolutions, one with size 1x1 and another 3x3, are again followed by a max-pooling layer. Afterwards, 9 inception modules with dimensions reduction are used. The final block is formed by a 7x7 average-pooling and a fully-connected layer with 1000 nodes before the softmax layer.
An additional feature is the use of two so-called auxiliary classifier only for training, consisting in intermediate softmax branches, which loss is added to the final loss with a weight of 0.3. Their aim is that of reducing the vanishing gradient problem. The vanishing and exploding gradient problems are quite common when dealing with deep neural networks. They are due to the fact that, when backpropagating using the chain rule, the updates in the earlier layers can result in being too small or too large. The chain-products may exponentially increase or decrease as the path length increases.

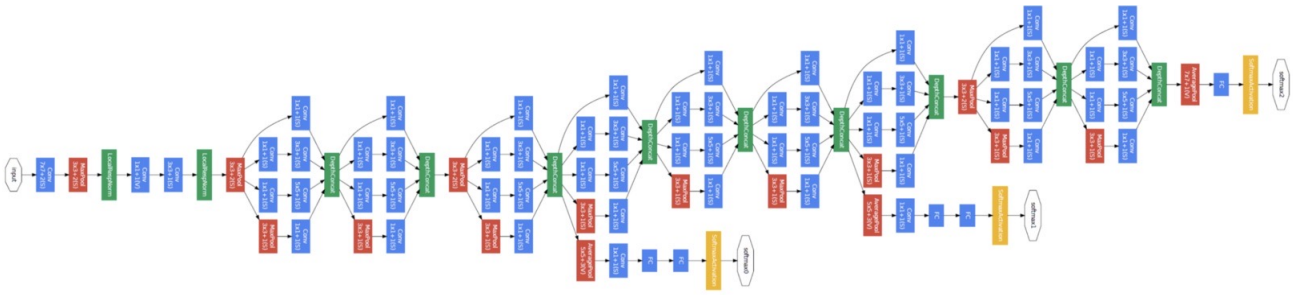


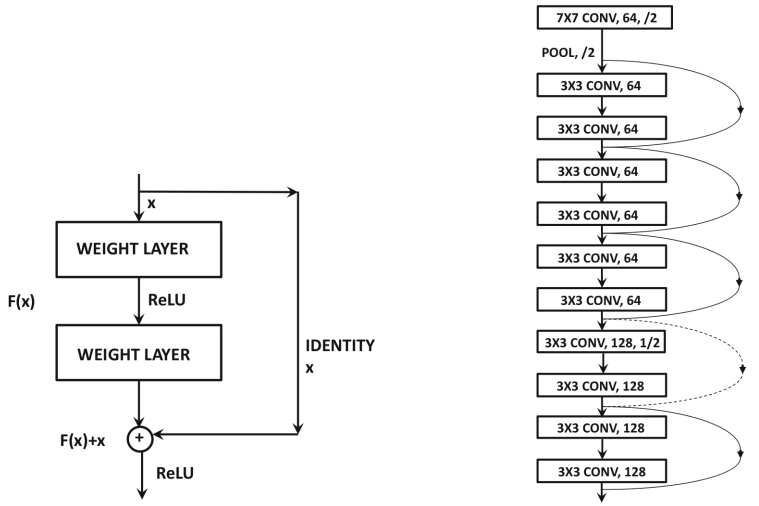
Figure 4: Full *GoogleNet / Inception V1* architecture

For the classification of the *Fruits 360* dataset, both the naïve and the standard inception block have been applied.
In both cases, since it is a reduced version of *GoogleNet*, in the first block of the net there is only a single convolution with 64 filters of size 3x3 followed by a 3x3 max-pooling with stride 2 and a padding to maintain the original dimensions of the input. Furthermore, the final layers are the same for all the architectures, with a 3x3 average-pooling layer with stride 1 and no padding, followed by a single flattened fully-connected layer of size 256, expect for the three blocks net with 384 nodes. Lastly, the softmax function is applied in the softmax layer.
Concerning the naïve inception network, three different models have been trained with, respectively, one, two and three blocks. The first naïve inception block uses 64 filters for the 1x1 convolution, 128 for the 3x3 one and finally 32 for the 5x5 convolutional layer. The max-pooling layer has size 3x3. The number of filters applied increases in every new inception module added to the network.
As for the architectures using the inception modules with dimensionality reduction, three versions with the same number of blocks as before has been used. The number of filters used in the convolutions is the same as with the naïve net. The only difference is the presence of the 1x1 convolutional layers which first reduce the number of filters of the input layer.
Moreover, in the three-blocks version there is a 3x3 max-pooling layer after the first two blocks. That's because in the *GoogleNet* architecture max-poolings are placed between blocks near the beginning and near the end of the net.

The final architecture to be discussed will be the *ResNet* model.

ResNet

The *ResNet* model won the 2015 *ILSVRC* competition with a top-5 error rate of 3.6%, reached through an ensemble of nets. It is noteworthy for being the first classifier achieving a human-like performance.
This architecture is addressing the main problems caused by training deep networks. They are, on the one hand, the vanishing or exploding gradient, on the other, the excessively slow convergence of the learning process. The latter phenomenon is due to the fact that not all the concepts in the image require deep networks in order to be learnt. Instead, for the simplest ones, shallow net would be enough.
ResNet introduces skip connections between layers. In this way, not only layer i is connected with the subsequent one, which is $(i+1)$, but it is also linked to the layer $(i+n)$, with $n=2$. This unit is called residual module, which is shown schematically in the image below.



(a) Skip-connections in residual module (b) Partial architecture of *ResNet*

Figure 5: *ResNet* overview

The residual module works by adding the result of the layer i to the result of the layer $(i+2)$, by propagating it using either an identity shortcut or a projection shortcut.
In the case of the identity function, the propagated output remains the same. The size of each block, which is defined by the width and depth, doesn't change since a padding of 1 is used along with 3x3 convolutions. Furthermore, the number of filters of each block remains the same. When the skip connections are represented by dotted lines, the dimension is reduced due to the use of a 1x1 convolution with a stride of 2. This is an alternative solution to a pooling layer and it is called projection matrix. Moreover, at the same time, the depth is increased by doubling the number of filters applied.
Before the last fully connected node, an average-pooling is performed. Furthermore, before the main residual module blocks there is a single convolution with filter size 7x7 and a stride of 2, followed by a max-pooling.
The advantage obtained by using residual modules is that the gradient can be directly backpropagated through the skip connections, avoiding vanishing or exploding gradient and the problem of slow convergence. In this way multiple paths of different length are available for backpropagation, giving the algorithm the flexibility to choose the level of non-linearity to be used for learning certain features. Simpler features require less non-linearity, thus skipping many connections. Deeper paths are used only when required by more complex attributes.

For the purpose of this analysis, three different architectures inspired to *ResNet* have been applied.
Each residual module checks first if the number of filters of the layer i is the same as that of the layer $(i+2)$. If the depth is the same, then the identity connection is applied, otherwise the number of filters is changed through a 1x1 convolution.
The size of the image is maintained constant since the original images have a size of 32x32 and halving them each time the number of filters is increased, would reduce the dimensionality too much. Before the main body of the architecture, a first 3x3 convolution and a 3x3 max-pooling are applied, both with a stride of 2 and a padding to maintain the size of the image. Furthermore, the final block is composed by an average-pooling with size 3x3 and a stride of 1, in order

to reduce the image dimensionality, followed by a fully-connected layer with 256 units.
The first model includes a single residual module with 64 filters, while as for the second one, another residual block with 128 kernels is added. The last model contains an additional block using 256 filters. Hence, the depth of the layers is increased by a factor of 2 each time a new module is added.

IMPROVEMENTS

When dealing with neural networks, it is important to be aware of the existence of some issues strictly connected to the way in which the networks work. The three main problems that can be encountered are overfitting, the vanishing or exploding gradient and slow convergence.
Overfitting happens when there is a large gap between the performance of the net on the training set and on the test set, where the latter is significantly worse. This signal the fact that the model learnt too well the training set data and it is no more able to generalise on new data. It is often related to neural networks due to the fact that they are overparametrized and complex models. Moreover, since they have so many parameters, they require a large amount of data, which are not always available. That's one of the main reasons for favouring deeper network instead of broader ones, since the former uses less parameters.
The problem of vanishing or exploding gradient has been discussed before. To sum up, since the gradient used to learn the weights and biases is backpropagated using the chain rule through many layers, the earlier update can end up either being very small or very large. This is because chain products can either decay or increase exponentially.
The last issue, which is the difficulty in converging, is often related to the vanishing gradient. It occurs when the network optimisation function takes an unreasonable amount of time to find the optimum point, that are those values for which the error remains stable.
To address these issues and improve the network performance, several techniques can be applied. For this project, they have been considered seven different approaches which are batch normalisation, tuning of the learning rate, image augmentation, weights regularisation, dropout, early stopping and the use of images in grayscale format.
These methods will now be explained in more detail.

To begin with, batch normalisation is able to counteract the continuous change of the values of parameters through the different hidden layers, which is defined as covariate shift, thus stabilizing the learning process.
Some layers are added between the hidden layers, with the task of normalising the input of the previous layer by subtracting the batch mean and dividing by its standard deviation. Subsequently, the output is scaled, meaning that it is multiplied by a parameter γ and added to another one, called β. These two parameters are learnable and optimised during the training process. The final result is that the inputs for the different layers end up having a more similar distribution.
Normalisation layers can either be placed before or after the activation functions used when convolving.

Regarding the learning rate, it is an hyperparameter which controls the size of the step taken by the optimisation algorithm in the direction of the negative of the gradient. It takes a value between 0 and 1. A smaller learning rate means a smaller update of the weights, and vice versa. It basically controls the speed of the learning process.

In this case, different learning rates have been applied, including 0.01, 0.001 and 0.00001.

Another widely-used method, especially useful when having a small dataset, is data augmentation. It consists in using the original training set images to create new ones by transforming them. These modifications include image rotation, flipping, cropping, padding and resizing, along with change in the brightness, saturation or contrast. All of them can be randomly applied. The main advantage is that of increasing data variety, so that the network is able to better generalise when dealing with new data.
The *Fruits 360* dataset has been doubled in dimension by randomly applying, to the original images, four different types of image augmentation.

Next, weight regularisation helps in controlling overfitting by constraining the weights of the network to assume a small value. Having large parameters causes instability, since the output varies a lot due to a small change in the inputs. With regularisation, a penalty for having larger weights is added to the optimisation function. The penalty can either be $L1$ norm, which adds the sum of the absolute values of the weights, or $L2$ norm, which instead sums up the weights squared.
In this case, it has been applied the second type of weights regularisation, also called Ridge regression.

Early stopping also has a regularising effect. It works by stopping the training process when there is no further improvement of the performance on the test set, which means that the test set has saturated. Thus, it prevents overfitting, moreover reducing the number of epochs and so the training time.
It may happen that the test performance swifts or doesn't change for some epochs before starting to increase again. To overcome this problem, it can be set a number of epochs which indicates after how much the learning process can be stopped, if there is no improvement. This indicator is called patience.
The values of patience used are 30 and 60 epochs, and the total number of epochs to perform is 500. It has been applied only when deemed necessary.

The last regularising method is dropout. It is an alternative to using ensembles, that is a collection of different neural networks which results are averaged. This should reduce overfitting by reducing the prediction variability, however, it requires the fitting of many networks, which is computationally expensive.
By applying dropout, one obtains an approximation of an ensemble. When training the network, a certain percentage of nodes is randomly dropped out, meaning that they are not considered, along with their connections. In this way, the model copes with the problem of co-adaption. This phenomenon occurs when all the weights are learnt together and some connections are favoured due to their stronger predictive power, while others are not considered. In the end, only a small fraction of the network takes part in the training process and influences the final outcome.
Usually, the percentage of dropped nodes lies between 20% and 50%.
Dropout has been applied on the project models both on hidden layers and before the final fully-connected one.

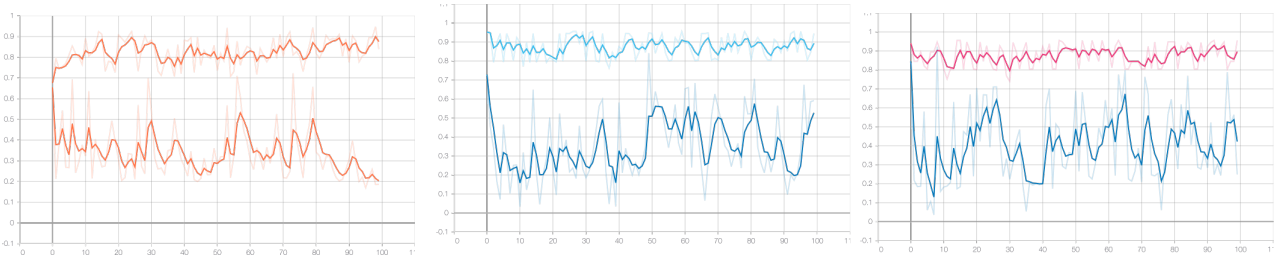
Finally, the networks are trained using images in grayscale format, with a depth reduced to one single channel. The reason why, is to assess whether employing images in RGB formats with

three channels have a significant effect on the final performance.

FINDINGS AND DISCUSSION

In this section, the analysis procedure and the main findings will be presented.
The approach used consists in, firstly, achieving a stable and good enough prediction performance for all the trained models. Afterwards, the best performing network in each of the three groups, namely *VGG*, *GoogleNet* and *ResNet*, are picked and further techniques to improve them are applied.
Three methods resulted to be successful in achieving a sufficient performance for all the nets. They are the batch normalisation, image augmentation and the tuning of the learning rate. They have been applied such that if one method turns out to be useful, it is maintained when implementing a new technique.

The effect of batch normalisation has been first tried on the smallest model of each group and, since it provided a noteworthy improvement, it has been applied by default on all the other architectures. For *VGG* models, batch normalisation has been used before the ReLU activation, while for the other models, after it.



[1] 1 VGG / validation (0.8756) [a]

[2] 1 VGG normalised/ validation (0.2033)

1 GoogleNet / validation (0.893)

1 GoogleNet normalised/ validation (0.5268)

1 ResNet / validation (0.8964)

1 ResNet normalised/ validation (0.4223)

[a] The value refers to the zero-one loss rate after 100 epochs

Figure 6: zero-one loss rate of 1 VGG, 1 GoogleNet and 1 ResNet block with and without normalisation

In all the cases, there has been a reduction of the zero-one loss between 40% and 70%.
The problem is that the performance is quite unstable, probably due to a problem of exploding gradient which prevents the optimisation algorithm to reach an optimum. Thus, the next improvement used is that of data augmentation, which should help the network to better generalise. It has been tested on some networks from all the groups, excluding the largest ones for avoiding excessive computations.

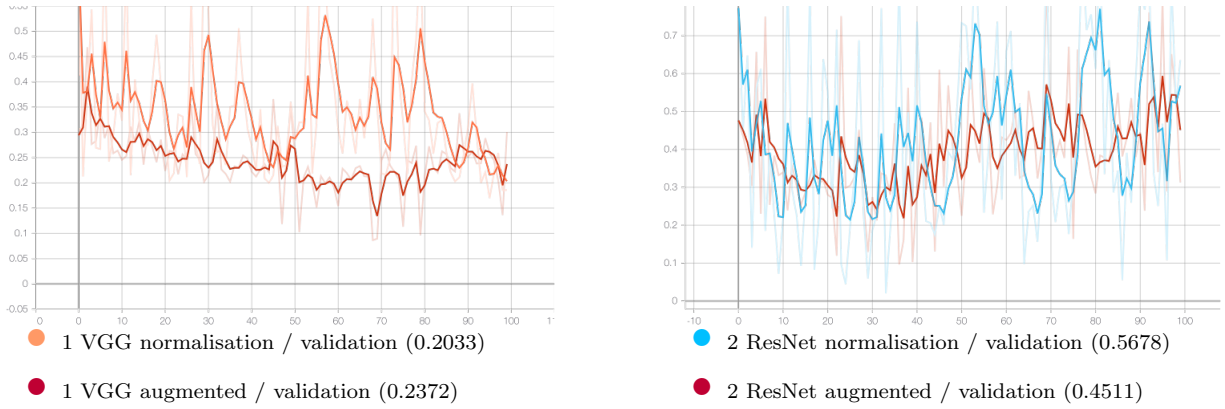


Figure 7: zero-one loss rate of 1 VGG and 2 ResNet with normalisation and augmentation

The final result was that of stabilising some of the models without increasing much the performance or even worsening it, while having no effect on others. However, they have been applied to all the networks by default since they may help reducing the error swinging.

In the end, the biggest contribution to the stabilisation and convergence of the network, is given by the fine tuning of the learning rate.
The learning rate applied by default is 0.001. For the sake of completeness, both a higher learning rate, equal to 0.01, and a lower one, which is 0.00001, have been used. As expected, the former didn't improve the model, while the latter had a positive impact, since unstable weights updates are caused by the fact that the optimisation algorithm is making too large steps, never converging towards an optimum.
The performance of all the models considerably improved by using the same learning rate of 0.00001.

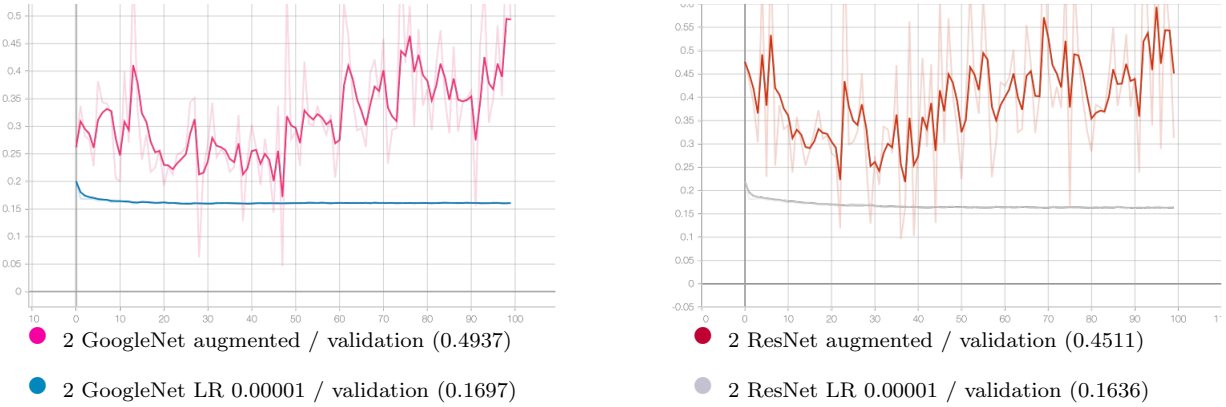


Figure 8: zero-one loss rate of 3 GoogleNet and 2 ResNet with augmentation and learning rate 0.00001

Along with an increased stability, as one can see in the above plots where the error line has been smoothed, the predictive capability has improved too. Now, all the architectures present a zero-one loss rate between 16% and 20%.
In order to choose the best model for each group, they need to be comparable, meaning to have

an enough stable and sufficiently good performance. This is ensured by the use at once of batch normalisation, the augmented dataset and a learning rate equal to 0.00001.

To begin with, the *VGG* models are considered.
By comparing them, it can be seen in the below graphs that their performance is pretty similar.

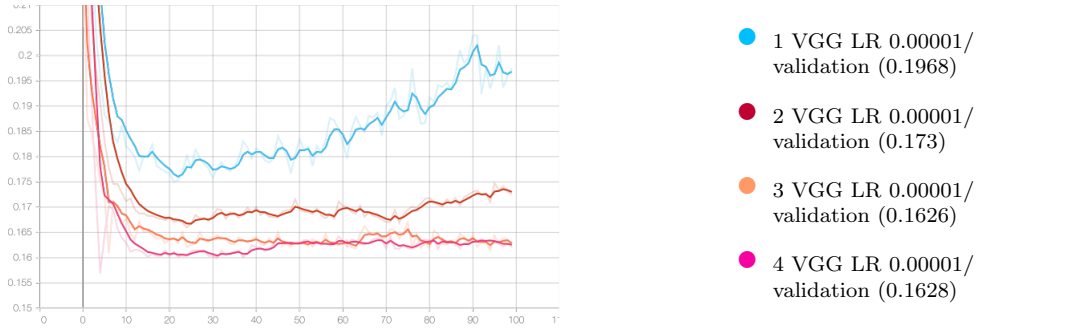


Figure 9: zero-one loss rate of all the VGG models with learning rate 0.00001

They can be ranked, from the worst to the best one, as follows: 1 block *VGG* (19.7% error), 2 blocks *VGG* (17.3% error) and, with nearly the same error rate, 3 blocks and 4 blocks *VGG* (16.3% error).
The architecture selected is the 3 blocks *VGG*, having a smaller computational cost.
Next, weights regularisation, dropout and the dataset in grayscale format are separately used on the 3 blocks *VGG*. Regarding the weights regularisation, both a value of 0.001 and 0.00001 are used. However, they provide the same results.

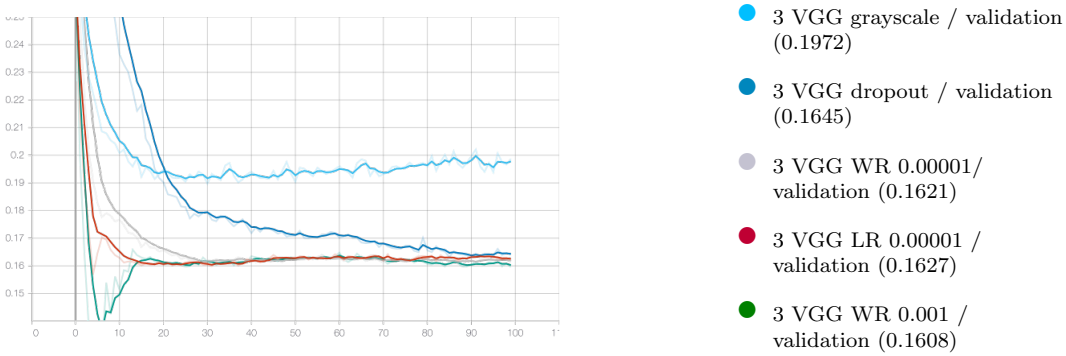


Figure 10: zero-one loss rate of all 3 VGG models

As one can see, there are no relevant improvements from the previous situation. There is a slight increase in the error rate (19.7%) when using grayscale images.
In this case, the smallest zero-one loss rate that can be obtained with the models from the *VGG* group is around 16%.
Since the error rate tends to converge early, it is useful to apply early stopping in order to reduce the number of epochs. By using a patience of 30, the optimal number of epochs selected is 52 with an error rate of 16.3%, which is half of the original one but still having the same performance.

To continue, the models in the *GoogleNet* set will be examined, both those using the standard inception module and those using the naïve inception one.

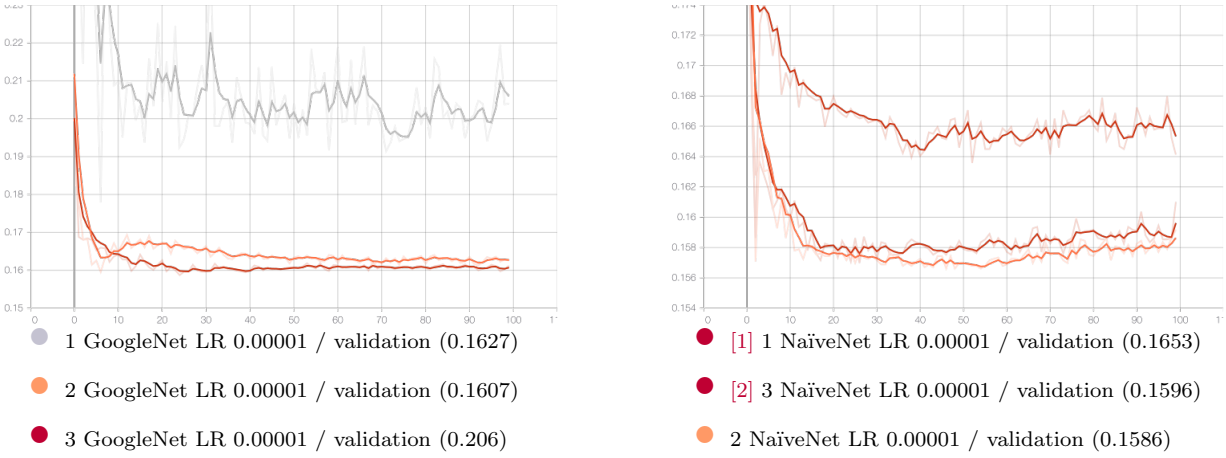


Figure 11: zero-one loss rate of all GoogleNet with naïve and standard inception module models with learning rate 0.00001

In order of performance, regarding the nets with regular inception module, there are the 3 blocks (20.6% error), the 1 block (16.3% error) and the 2 blocks (16.1% error) architecture. As for the networks with naïve inception module, there are the 1 block (16.5% error), the 3 blocks (16% error) and the 2 blocks (15.9% error) models.
The final model that has been picked is the 2 naïve blocks architecture, since it is slightly better than the 2 standard blocks net, moreover it is simpler and, hence, it is faster to train.
Then, they have been re-trained using weights regularisation, dropout and a learning rate of 0.00001.

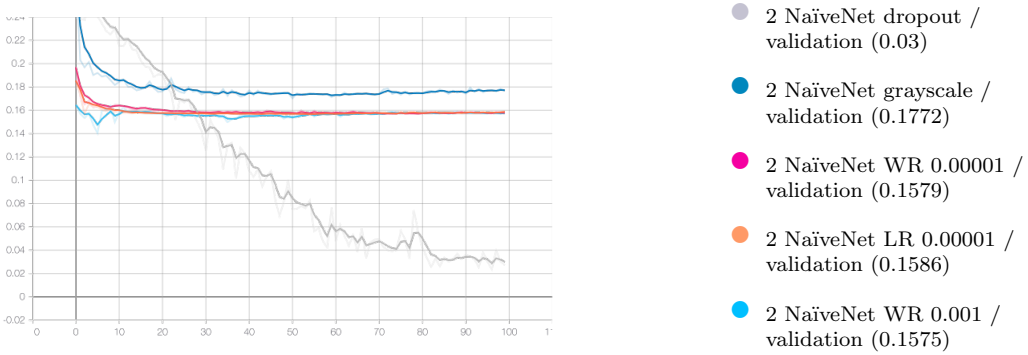


Figure 12: zero-one loss rate of all 2 GoogleNet models with naïve inception modules

Both weight regularisation with a rate of 0.001 and 0.00001 have no significant effect, while the use of the grayscale dataset provided an error rate of 17.7 %, 2 percentage points higher than with RGB format.
The technique that gave a real boost to the network performance is the dropout, which made it possible to reach a zero-one loss rate of 3%. In this case, 100 epochs look like they're enough for allowing the model to improve its performance and to converge, without adding too much useless training time. In fact, the net error starts to flatten around 85 epochs. Hence, it won't be much helpful to apply early stopping. Moreover, since the error rate trend is jiggly, the training

would be probably stopped too soon, in correspondence of a spike.
The architecture has also been trained with 300 epochs in order to be sure that the error rate remains stable, converging to a value around 3%.

The final set of models to be assessed are those in the *ResNet* group.

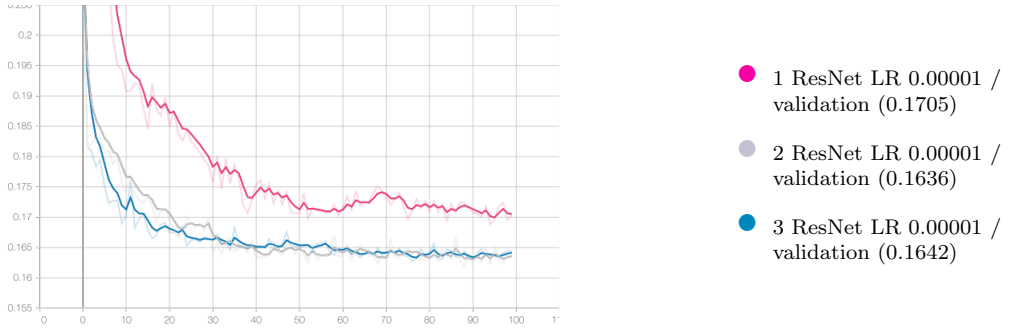


Figure 13: zero-one loss rate of all ResNet models with learning rate 0.00001

The worst performing model is the 1 block net (17% error), while the 2 and 3 blocks architecture have a similar performance (16.4% error). Since the 2 blocks *ResNet* is smaller and has a lower computational cost, it is selected to be used for further trials.
As before, the effect of applying weights normalisation, dropout and the dataset in grayscale format are compared.

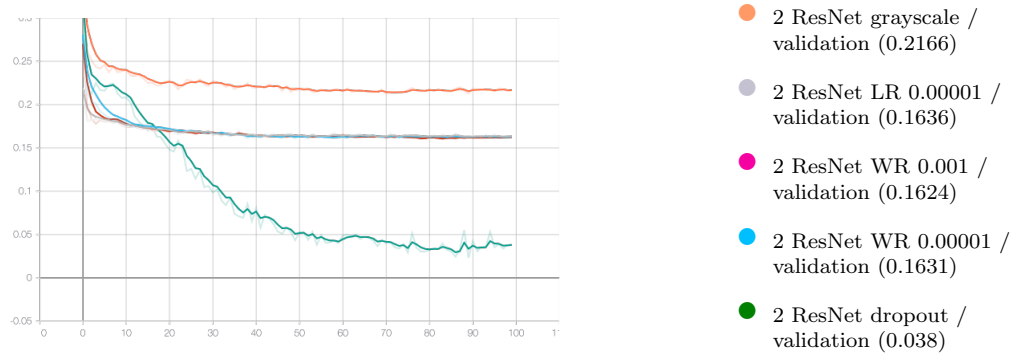


Figure 14: zero-one loss rate of all 2 ResNet models

The changes in the error rate are similar to those obtained with the *GoogleNet* architecture. The weights regularisation provided no improvements, whereas the use of the grayscale dataset worsened by 5 percentage points (21.7% error) the performance.
Regarding the dropout, the zero-one loss rate considerably fell, having a final value of 3.8%. However, the error rate becomes flat after 60 epochs, oscillating between 3% and 4%. This suggests that the training could be stopped before, so that to save some computations. Early stopping is then applied, both with a patience of 30 and of 60. The former stops the training process after 45 epochs ending with an error rate of 6.7%, while the latter after 82 epochs, providing an error rate of 3.8%. Thus, the second option is the best one. In this case, early stopping worked well, possibly due to the fact that the error rate trend is rather smooth.

CONCLUSION

From the previous analysis, the models in each group that better classify the images in the *Fruits 360* dataset are, respectively, the three blocks *VGG* (16.3% error), the two blocks *GoogleNet* with naïve modules and dropout (3% error), and the two blocks *ResNet* with dropout and early stopping at 82 epochs (3.8%).
It is noteworthy that all the architectures reached a similar and sufficiently good zero-one loss rate, between 16% and 20%, by applying the same three improvements. Namely, they are batch normalisation, data augmentation doubling the dataset and a fine-tuned learning rate of 0.00001.
It was not possible to further improve the *VGG* model, as opposed to *GoogleNet* and *ResNet*, which before the application of dropout had a performance similar to the former (16% and 15.9%). In their case, dropout was able to prevent overfitting, thus providing a regularising effect, in contrast with weights regularisation.
Furthermore, early stopping reduced the number of epochs needed to train the *ResNet* and *VGG*. However, it didn't provide a fundamental contribution.
In all the cases using a grayscale version of the data worsened the performance, meaning that having images encoded in RGB colour values is important for the models to correctly learn.

In conclusion, even without using the full architectures, which are the 22 layered *GoogleNet* and the 256 layered *ResNet*, it has been possible to obtain a good predictive power from shallower networks. Both the two blocks *GoogleNet* and two blocks *ResNet* correctly classified around 97% of the instances. Moreover, they considerably reduced the computational time needed for training.
The fact that small architectures predicted well the images in the *Fruits 360* dataset is probably due to the structure of the data themselves. The dataset contains images representing a single fruit or vegetable, which shape and colours are quite simple and easily recognisable. Furthermore, their background is plain, without further elements that can introduce noise or uncertainty in the training process. The classification task is rather easy, since there are only 10 classes of fruits.

There exist several consolidated techniques to improve the predictive capability of convolutional neural networks, however, whether or not they work and how much they are of help, depends on the features of each specific dataset. Training neural networks properly is an empirical process, meaning that it requires a trial and error procedure.
However not exhaustive, this report has introduced and explored several techniques and architectures with the aim to find the best suited solution to deal with the classification task at hand.