Project of Statistical Methods for Machine Learning

Classification of Cats and Dogs

Name: Giorgio Degl'Innocenti

MSc: Data science and Economics

giorgio.deglinnocenti@studenti.unimi.it

September 3, 2022

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.

Abstract

The goal of this brief project is to build a convolutional neural network (following CNN) to classify correctly photos of cats and dogs.
The project aims to look at the influence of different parameters on the accuracy of the neural network and to apply a cross-validation in order to compute the risk estimate. The study start from a NN with only 2 Convolutional layer (following CL) and after that it keeps implementing more layers and different architectures.
We see that the more precise CNN has been one of the last with four CLs which achieved an accuracy close to 90%.

Contents

[1] most of this section will quote liberally from the original paper[4]

1 Introduction

The goal of this brief study is to correctly implement a CNN that can classify photos of cats and dogs, to do that we used the know dataset Dogs vs. Cats containing 25.000 photos equally distributed between cats and dogs. In our research, we also implement a cross-validation to compute the risk estimate of a model but in order to save time we decided to implement it only for the CNN who would have the best performance.
In section 2 we introduce the CNN showing some of the features that distinguish it from the rest, how it works, and the use of the different layers. Section 3 show the different model, discussing the architecture, exposing the learning curves and discussing the results. Section 4 will briefly discuss the results of the study.

2 The Convolutional Neural Network

The Convolutional Neural Network is a deep learning class of algorithms, developed in the 80s and inspired by the visual cortex used mainly for tasks of classification and Computer Vision.
The CNN is a regularized version of the multilayer perceptron made necessary due to the ease with which these algorithms tend to overfit and to the greater efficiency of this type of neural network.
As we will see there are multiple stages in which a CNN is working; it will start with a convolutional layer (follow CL) and will end up with a fully connected one (follow FCL). We can see in figure 1 how usually a CNN is built. Now we will explore every layer of the CNN and explain the process which makes a CNN function.

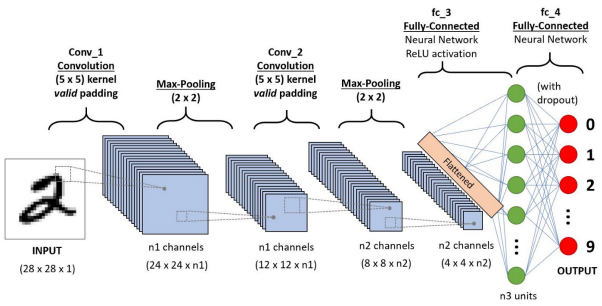

Figure 1: Typical architecture of a CNN

2.1 Input Image

The first, thing to understand on a CNN is what are the input data. A neural network can take a variety of input data, from audio to text to images, in our case we will focus on images since is the part our study is focused on.
A computer sees an image as a collection of pixels, we can see in Figure 2 how a computer picture an image, in this particular case we are looking at an RGB image, so every layer

will express a different color Red, Green and Blue and the image will have dimension 4x4x3.
We want to notice that this is not the only color space, there are several more as CMYK or grayscale (which have only dimension 1).

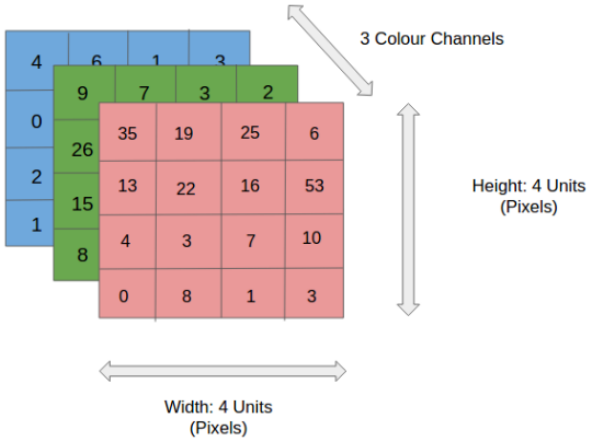

Figure 2: How does a computer see an image

2.2 Convolutional Layer

The convolutional layer is the most important block of a CNN, through this layer the neural network can extract all the most important features of an image. In order to do that the input data is passed through a filter (smaller than the input data) that applies a dot product through the width and height (and depth) of an image sliding from one point to another. The result of the dot product between the filter and the input map gives us a so called activation map (or output map as we can see in Figure 3).

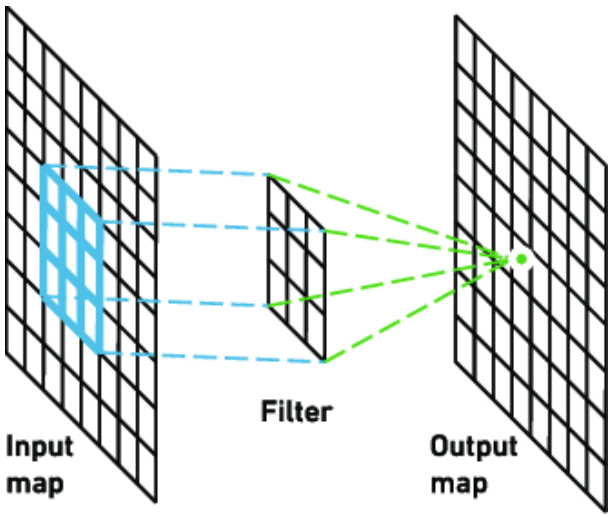

Figure 3: How a convolutional layer work

The kernel is no more than a matrix that slides through the image and since it is always the same it can "filter" the same features for the whole image (lines, etc...).

In a CNN we have several CLs where the first will collect low-level features (lines, edges, etc...) while the last ones will collect the high-level ones (shapes, objects, etc...).

2.3 Stride

We called stride the number of pixel the kernel will "jump" after the dot product. With a stride of 1, for example, the kernel will end up in the second column, while with a stride of 2 the kernel will end up in the third one. In our project, we used the default stride of 1.

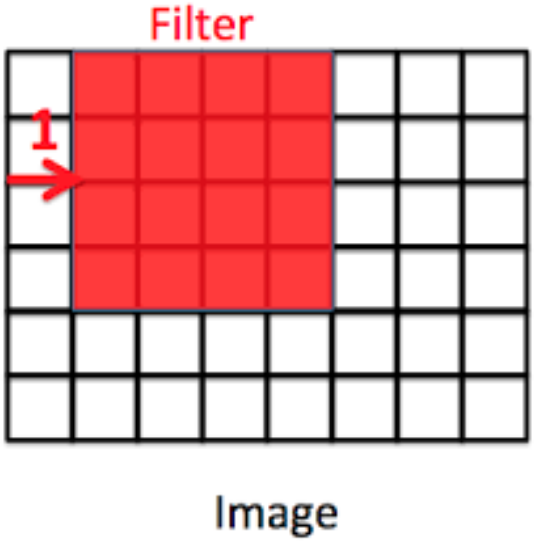

Figure 4: Stride of 1

Stride is mainly used as a way to down-sample an image (more in subsection 2.7) which in our model has been achieved through Pooling layers.

2.4 Activation Function

The activation function is that function that defines if a neuron will activate or not and transforms the summed weighted input into the output. This step of a neural network is rather important because it allows the neural network to learn more complex patterns introducing non-linearity in the model.
In our study we have decided to use the ReLU function (Rectified Linear activation function) this is one of the most used activation function, mainly for two reasons:

1. It is very simple to compute and very effective computationally involving only comparison, addition and multiplication.

2. It doesn't suffer from the vanishing gradient problem which occurs when the gradient decreases dramatically while propagating backward, adding small or no improvements to the model (this problem is especially true when using a model with many hidden layers)

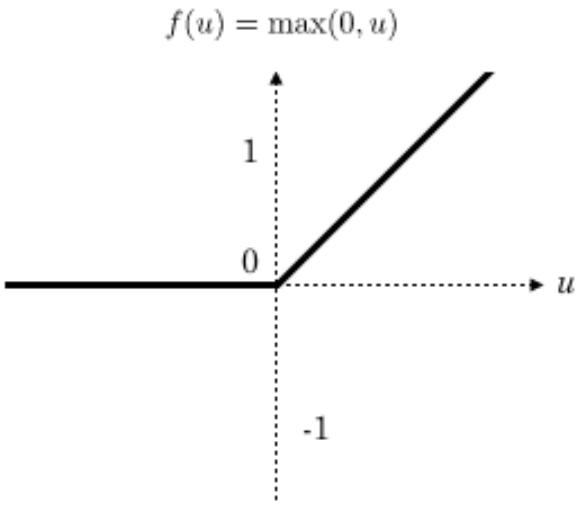

Figure 5: ReLU function

2.5 Batch size

At this point seems important to start to talk about how a CNN improve his model and why the gradient is so important(and why the vanishing gradient is a problem). A normal NN use the stochastic gradient descent optimization algorithm to train itself. What does it means is that the model makes a prediction than compare it with the expected value and then use the error to improve itself.
This gradient error is,therefore, used to updated the weights of the CNN and the number of samples used to estimate this gradient error is called batch size.
There is no magical number to decided what number a batch size should be, it has been shown that if we use too few examples to train our model the results will be very noisy estimate which will be strictly dependent of the examples took from the NN and therefore will lead to noisy update; large batch size (while being more fast and computationally efficient) have seen to produce worse model whit a lose in their ability to generalize (even if nowadays we still do not have a certain answer of why this happens [2].
Usually we distinct three type of batch size:

1. Batch Gradient Descent: Batch size is equal to the number of training examples
2. Stochastic Gradient Descent: Batch size is set to 1
3. Mini-batch Gradient Descent: Batch size is set between one and the number of training examples

[2]There has been some studies about this topic, a possible explanation has been brought from Keskar et al.[3] in their paper "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima" where a possible explanation has been that too large batch size convergence to sharp minimizers that gives rise to the poor generalization)

In view of these facts we opted for a batch size = 32, which have been seen to perform well in many experiments.

2.6 ADAM[3]

We have previously talk about the use of stochastic gradient descent optimization as the algorithm that CNN use to learn from a dataset, but in matter of true in our study we used a different optimization algorithm presented in 2015 by Kingma and Ba[4] in their paper "Adam: A Method for Stochastic Optimization".
Adam name comes from ADaptive Moment estimation, the algorithm is combine to take the advantages of two methods:

1. AdaGrad: which works well with sparse gradients
2. RMSProp: which works well in on-line and non-stationary settings

Adam has been shown to outperform others algorithms and in the last years has become the default for any deep learning problem, more that that it shows several advantages such:

1. the magnitudes of parameter updates are invariant to rescaling of the gradient
2. its stepsizes are approximately bounded by the stepsize hyperparameter
3. it does not require a stationary objective
4. it works with sparse gradients
5. it naturally performs a form of step size annealing
6. it is computationally efficient
7. it has little memory requirements

In light of this fact and the several advantages that this algorithm gave is natural for us to use this optimization scheme on our models.

2.7 Pooling Layer

Going back to the building block of the CNN one of the most important is the pooling layer, this type of layer is used for two reason:

1. Is a way of reducing the dimension of the feature map, diminishing the computational cost and the parameters of the model (helping therefore also in overfitting)
2. Most importantly, since that CL are location-dependent (therefore they associate features with a precise location in an image), we need it to down-sample the image in order to summarise better each feature and making them less position-dependent.

We can find several ways of pooling; the most used are max pooling (where we take the maximum value between a certain region) and average pooling (where we compute the average between all the values).
Both the methods are useful and their use differs while max pooling (the one used in

[3]most of this section will quote liberally from the original paper[4]

our study) tends to collect the brightest pixels and extract the most important features, average pooling tends to smooth the image and the features in it, putting a greater focus on the extent of the object.

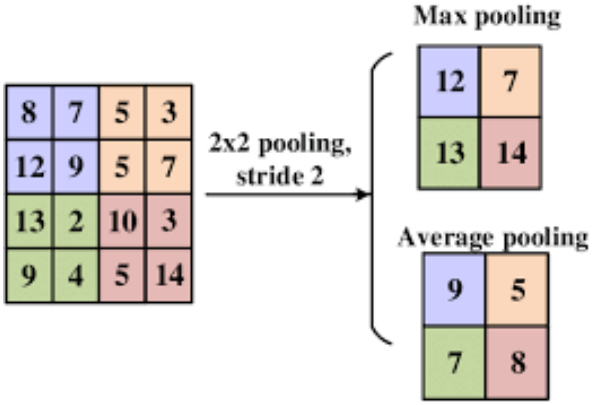

Figure 6: Max and Average Pooling

2.8 Dropout Layer

The dropout layer as we can understand from the name randomly drops a certain percentage of nodes after a hidden layer. This type of layer is a regularization technique used to prevent cases of overfitting.
The usual dropout values differ if the dropout is applied on a CL or on a FCL. In the original paper of 2012 "Improving neural networks by preventing co-adaptation of feature detectors"[2] the researcher suggests to apply the dropout layer on the FCL with a percentage $\simeq 0.5$, while further studies such as "Analysis on the Dropout Effect in Convolutional Neural Networks"[5] shows how this technique can be also be applied on the CL with smaller dropout percentage ($\simeq 0.1 - 0.2$).

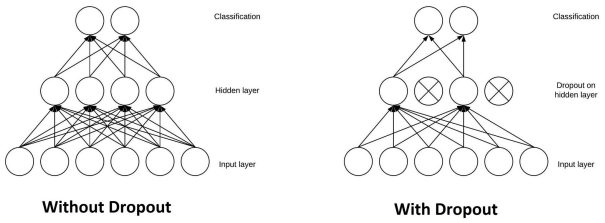

Figure 7: Dropout layer

2.9 Fully Connected Layer

As we have seen from Figure 1, the CNN ends up with one or more FCLs. What happens is that we pass our matrix through a flatten layer that flattens the input into a 1-D vector; after that we have the series of FCLs that try to correctly classify all the features extracted by the CLs.

The FCL usually use as activation function the softmax activation function to estimate the probability of the input belonging to a particular class (in particular the softmax is suited for classification task to work with more than 2 classes) but in our study, since we had to implement only a binary classification, the sigmoid function worked perfectly.

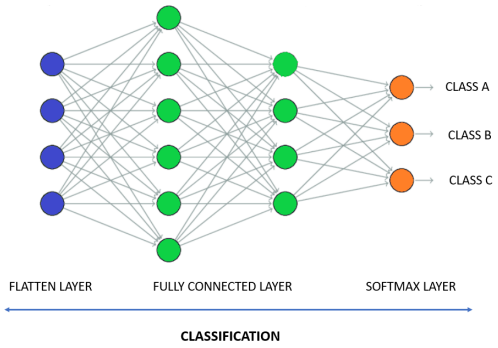

Figure 8: Flatten and FC layer

3 The Models

In our study we have tried 5 different architectures of our CNN, starting with simpler models and then adding layers and experimenting with different architectures.
Every model has been run for only 50 epochs due to computational cost, for sure more interesting results would have come out if we could have run for a higher number of epochs (eg. 100) but without a proper GPU, this has been unachievable.
The split of the dataset has been 16.000 images for the train set[4], 4.000 for the validation set and 5.000 for the test set.

3.1 Model 1: two CL

Our first model is composed of two different CL, we can see the structure of the model in the underlying Figure 9:

```
def create_model ():
    model=Sequential()
    model.add(Conv2D(32,(3,3),activation='relu',input_shape=(Image_Width ,Image_Height,Image_Channels)))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Dropout(0.2, seed = 42))
    model.add(Conv2D(64,(3,3),activation='relu'))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Dropout(0.2, seed = 42))
    model.add(Flatten())
    model.add(Dense(32))
    model.add(Dropout(0.5, seed = 42))
    model.add(Dense(2,activation='sigmoid'))
    model.compile(loss='binary_crossentropy',
    optimizer='Adam',metrics=['accuracy'])
    return model
```

Figure 9: Model 1 structure

[4] 15.998 due to two corrupted images

The architecture is composed of two different CL, whit kernels of size 3x3 and an increasing number of filters. [5]

Moreover, we used the dropout to regularize the model both on the Conv layers and after the FC layers as we discussed previously in 2.7.
We can see now from the underlying graphs that the model over-fit since the beginning even with the several dropout layers of the model, with the increasing of the epochs is very unlikely to see an improvement in the performance without a change in the architecture of the CNN[6].
In the end, the maximum accuracy reached by the model has been 78.06%.

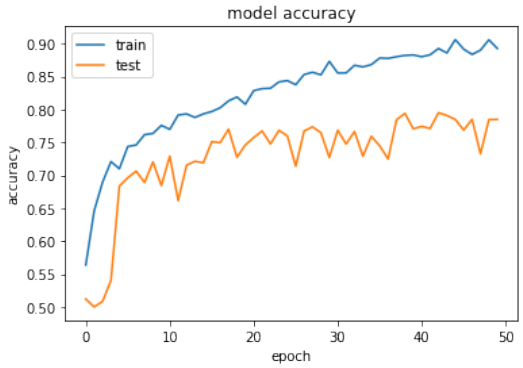

Figure 10: Model 1 accuracy

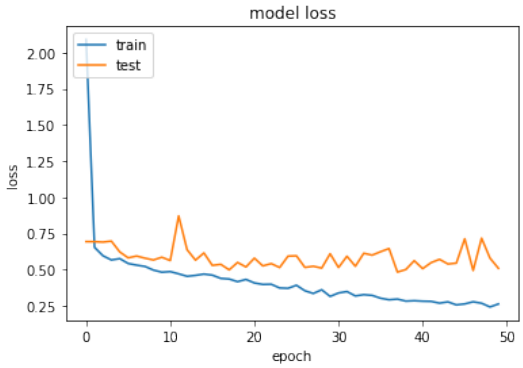

Figure 11: Model 1 loss

3.2 Model 2: three CL

The second model that we implement is characterized by three CL, we can see the structure in figure 12:

```
def create_model_2 ():
    model=Sequential()
    model.add(Conv2D(32,(3,3),activation='relu',input_shape=(Image_Width ,Image_Height,Image_Channels)))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Dropout(0.2, seed = 42))
    model.add(Conv2D(64,(3,3),activation='relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Dropout(0.2, seed = 42))
    model.add(Conv2D(128,(3,3),activation='relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Dropout(0.2, seed = 42))
    model.add(Flatten())
    model.add(Dense(32))
    model.add(Dropout(0.5, seed = 42))
    model.add(Dense(2,activation='sigmoid'))
    model.compile(loss='binary_crossentropy',
    optimizer='Adam',metrics=['accuracy'])
    return model
```

Figure 12: Model 2 structure

[5] This is because as we have seen filters capture features, and while in the first layers they capture simpler features (we talked about edges or such) going on with deeper layers they need to combine these features in more complex objects, this is such that there is a higher number of combination possible to capture and so the increasing number. Some paper regarding the topic is "The Impact of Filter Size and Number of Filters on Classification Accuracy in CNN" by Ahmed et al.[1]

[6] An idea if we don't want to put extra layers could be to apply some L1/L2 regularization or some kind of data augmentation

In this case, nothing changes much from the previous model except for the presence of a third layer with a higher number of filters in it.
Looking at the plot of accuracy and loss we notice this time a problem of unrepresentative train dataset, this particular case happens when the training set does not provide sufficient information compared with the test set. In this case, we see that the train and the test set keep improving but a gap remains between the two, another hint is the high variance of the test performance.
The accuracy of this model has been 81.52%.

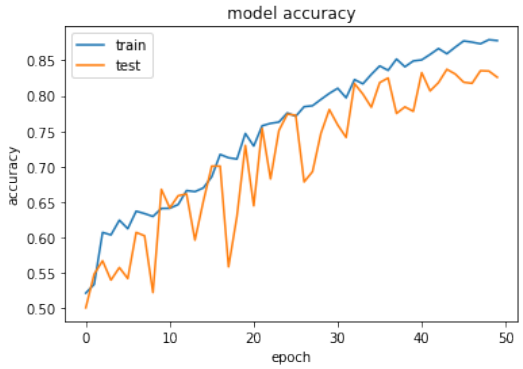

Figure 13: Model 2 accuracy

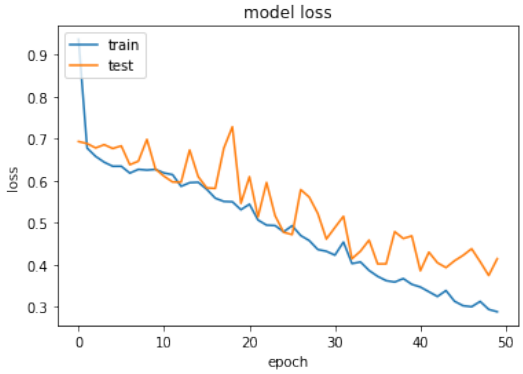

Figure 14: Model 2 loss

3.3 Model 3: four CL

Our third model is composed of four CL but it has some divergences from the typical architecture of a CNN. In this case after two normal layers (the first with batch normalization), it follows one layer with dropout but without a pooling layer, then the last layer similar to the first one.

```
def create_model_4 ():
    model=Sequential()
    model.add(Conv2D(32,(3,3),activation='relu',input_shape=(Image_Width ,Image_Height,Image_Channels)))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Dropout(0.2, seed = 42))
    model.add(Conv2D(32,(3,3),activation='relu'))
    model.add(Dropout(0.2, seed = 42))
    model.add(Conv2D(32,(3,3),activation='relu'))
    model.add(Dropout(0.2, seed = 42))
    model.add(Conv2D(32,(3,3),activation='relu'))
    model.add(Dropout(0.2, seed = 42))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Flatten())
    model.add(Dense(32))
    model.add(Dropout(0.5, seed = 42))
    model.add(Dense(2,activation='sigmoid'))
    model.compile(loss='binary_crossentropy',
    optimizer='Adam',metrics=['accuracy'])
    return model
```

Figure 15: Model 3 structure

Looking at the graph we see the effect of such a weird structure and even if the loss seems to behave nice, looking at accuracy we see a really high variance of the model, this case shows an even heavier unrepresentation of the train set, exacerbating, even more, the situation seen in 3.2.
The accuracy here doesn't go over 80.14% but still, this measure is really imprecise due to the variance of the model.

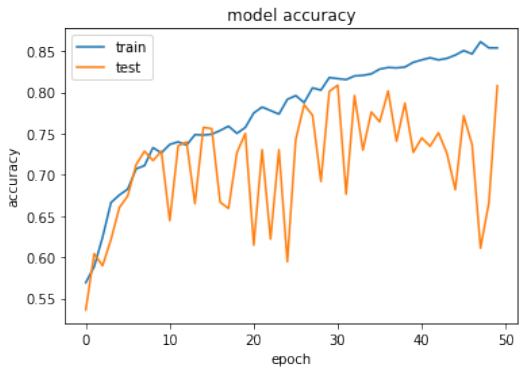

Figure 16: Model 3 accuracy

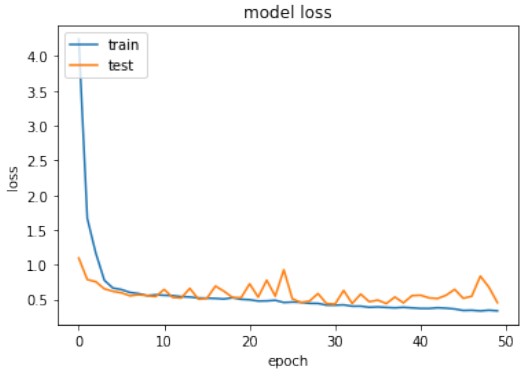

Figure 17: Model 3 loss

3.4 Model 4: four CL, filters = size

The following model has a more common structure with respect to the previous one, we still have four CLs but this time we used the usual sequence of layers (Conv → Pooling → Dropout).
The particularity of this mode is that this time we have always the same number of filters for all the layers equal to 32.

```
def create_model_5 ():
    model=Sequential()
    model.add(Conv2D(32,(3,3),activation='relu',input_shape=(Image_Width ,Image_Height,Image_Channels)))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Dropout(0.2, seed = 42))
    model.add(Conv2D(32,(3,3),activation='relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Dropout(0.2, seed = 42))
    model.add(Conv2D(32,(3,3),activation='relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Dropout(0.2, seed = 42))
    model.add(Conv2D(32,(3,3),activation='relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Dropout(0.2, seed = 42))
    model.add(Flatten())
    model.add(Dense(32))
    model.add(Dropout(0.5, seed = 42))
    model.add(Dense(2,activation='sigmoid'))
    model.compile(loss='binary_crossentropy',
    optimizer='Adam',metrics=['accuracy'])
    return model
```

Figure 18: Model 4 structure

Looking at the situation of the learning curves, even if it seems to get better respect than the previous models this one still suffers from a high variance (we can see around the 40 epochs a huge fall in the accuracy of the model), the situation is anyway improving since the model does not overfit, the accuracy of this model has reached 83.56%.

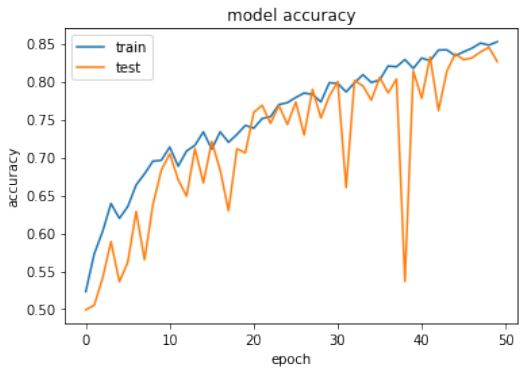

Figure 19: Model 4 accuracy

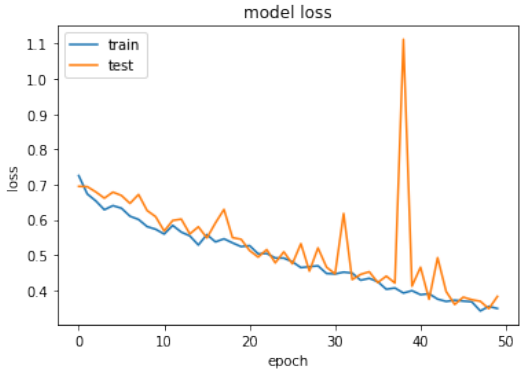

Figure 20: Model 4 loss

3.5 Model 5: four CL, filters ≠ size

The last model that we ran can be seen in the underlying picture 21, the particularity of this model is that it uses an increasing number of filters as the layers go deeper:

```
def create_model_6 ():
    model=Sequential()
    model.add(Conv2D(32,(3,3),activation='relu',input_shape=(Image_Width ,Image_Height,Image_Channels)))
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Dropout(0.2, seed = 42))
    model.add(Conv2D(64,(3,3),activation='relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Dropout(0.2, seed = 42))
    model.add(Conv2D(128,(3,3),activation='relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Dropout(0.2, seed = 42))
    model.add(Conv2D(256,(3,3),activation='relu'))
    model.add(MaxPooling2D(pool_size=(2,2)))
    model.add(Dropout(0.2, seed = 42))
    model.add(Flatten())
    model.add(Dense(32, kernel_regularizer=regularizers.l2(0.01)))
    model.add(Dropout(0.5, seed = 42))
    model.add(Dense(2,activation='sigmoid'))
    model.compile(loss='binary_crossentropy',
    optimizer='Adam',metrics=['accuracy'])
    return model
```

Figure 21: Model 5 structure

We can now look at an example of the performance of the model, the model has performed so far the best one between the ones we experimented, therefore has been chosen to go through the all cross-validation process.[7]
Looking at the learning curves we finally have a good performance of the model with the validation curve that keeps following the train curve without any major gap.
In general, the performance of the model has been 87.86%(+/- 1.43%) again showing the good performance of the model also in the precision

[7]The other learning curves of the model can be found in the appendix5

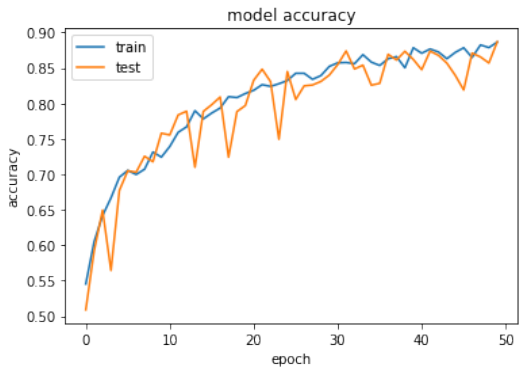

Figure 22: Model 5 accuracy

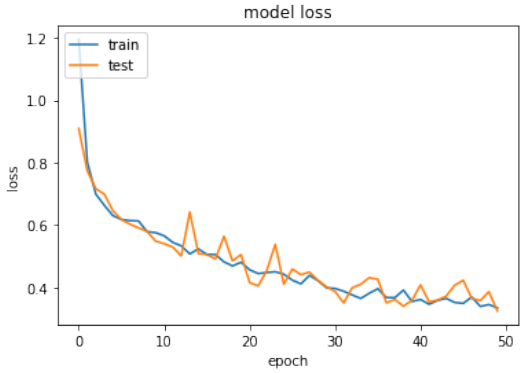

Figure 23: Model 5 loss

4 Results

In this brief study ,we have seen several different architectures of a CNN, we started with a more sparse architecture with few layers that suffered from overfitting; after that, we looked at more complex CNN stopping with the one with four layers and looking at the influence of the number of filters and the ordering of layer.
As we have seen a greater number of filters (as the number of layers rises up) produced greater stability and performance of the layer, which has worked the same with the increasing number of layers who brought to higher accuracy (at the expense, naturally, of a higher computational cost).
In the end, we saw that the last model we create achieved a good performance correctly classifying 9 images out of 10.

References

[1] Wafaa Shihab Ahmed and Abdul amir A. Karim. "The Impact of Filter Size and Number of Filters on Classification Accuracy in CNN". In: *2020 International Conference on Computer Science and Software Engineering (CSASE)*. 2020, pp. 88–93. DOI: 10.1109/CSASE48920.2020.9142089.

[2] Geoffrey E. Hinton et al. "Improving neural networks by preventing co-adaptation of feature detectors". In: *CoRR* (2012), p. 18.

[3] Nitish Shirish Keskar et al. "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima". In: *CoRR* abs/1609.04836 (2016). arXiv: 1609.04836. URL: http://arxiv.org/abs/1609.04836.

[4] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[5] Sungheon Park and Nojun Kwak. "Analysis on the Dropout Effect in Convolutional Neural Networks". In: *ACCV*. 2016.

5 Appendix

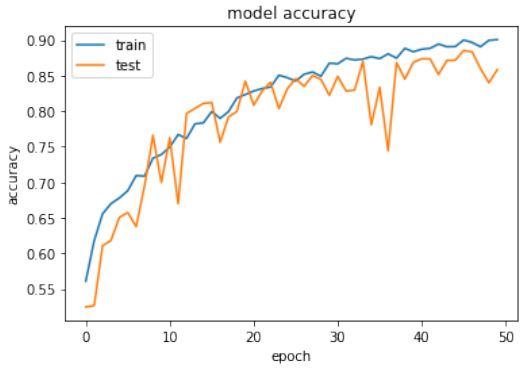

Figure 24: Model 5.2 accuracy

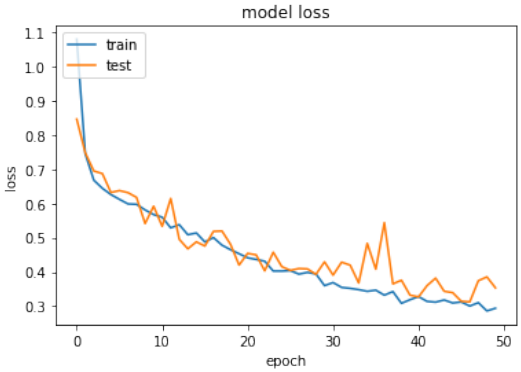

Figure 25: Model 5.2 loss

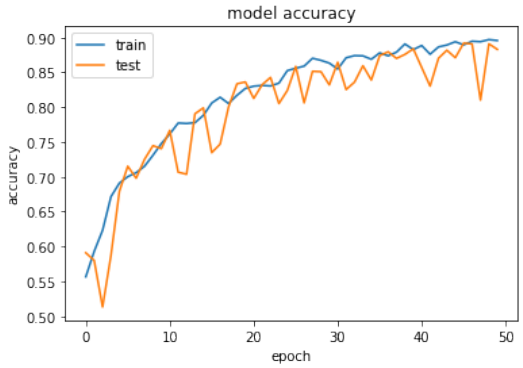

Figure 26: Model 5.3 accuracy

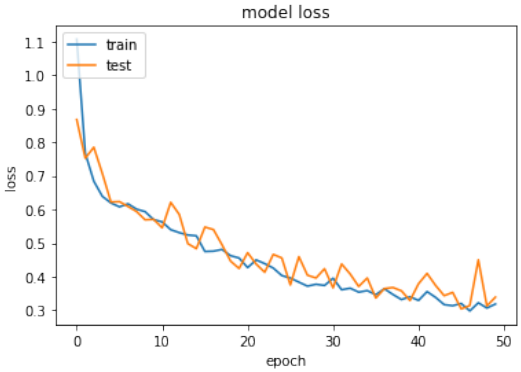

Figure 27: Model 5.3 loss

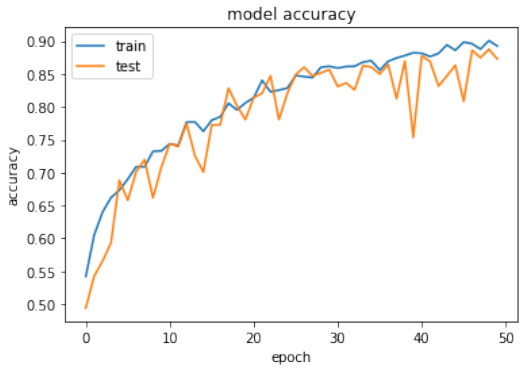

Figure 28: Model 5.4 accuracy

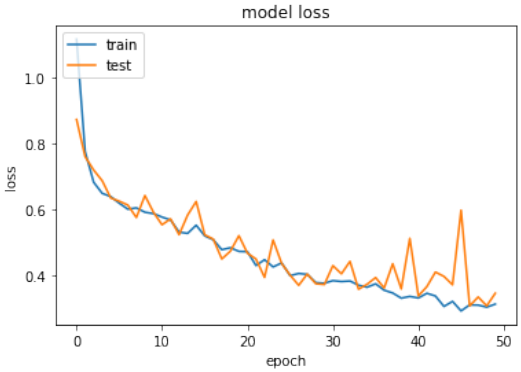

Figure 29: Model 5.4 loss

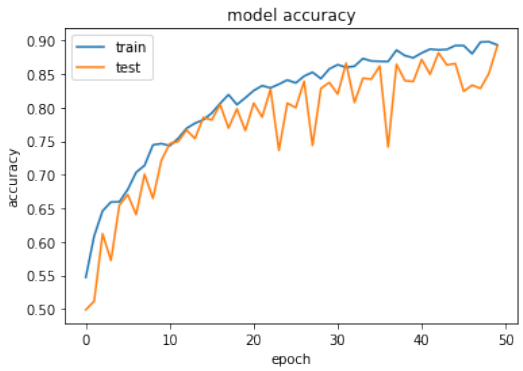

Figure 30: Model 5.5 accuracy

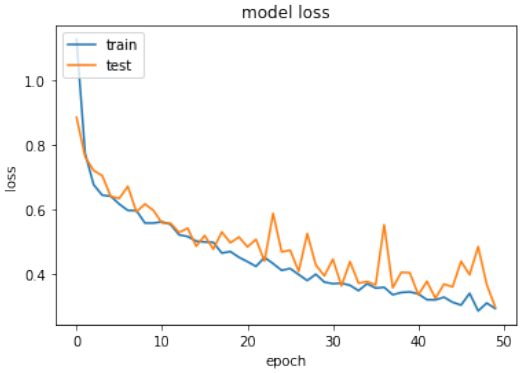

Figure 31: Model 5.5 loss