Bioinformatics

CNN project - 3.7 Growing DeepTree

Final Project Report

Politecnico di Torino

Referents: Prof. Ficarra Elisa

Alberto Benincasa [251415]
Marco Morandi[253488]
Chiara Penaglia [246289]

September 18, 2019

1 Introduction

This project wants to be a prototype of a tool able of receiving images of human tissues, and giving the classification of the pathology that has hit the tissue.

In particular the inputs are part of a medical dataset, composed by images from colon tissue of different patients. A CNN is used to distinguish the different tissues. Therefore the tissues are classified in five classes:

1. Adenocarcinoma, tumour (AC)
2. Healthy tissue, normal colonic glands (H)
3. Serrated Adenoma, precursive lesion of cancer that may turn into it (Serr)
4. Tubular adenoma, precursive lesion of cancer that may turn into it (T).
5. Villous Adenoma, precursive lesion of cancer that may turn into it (V).

The dataset is divided in folders by patient, and each image has this notation:

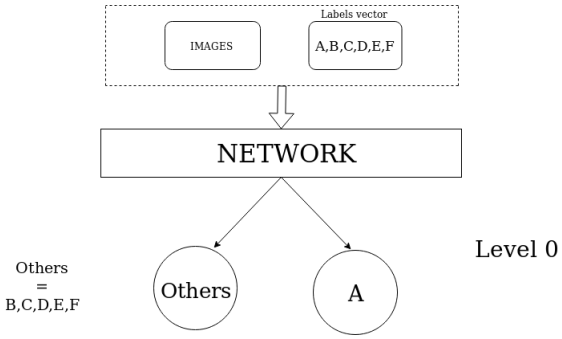
`{Patient_ID}_{Class_ID}_{Image_ID}.png`

To manage this dataset multiple functions are used, and they will be described later.

2 The concept

The group implements a growing DeepTree, which is able to classify images in a context of incremental learning. Each time a new class arrives, the network, after a training that involve only that specific class, should be able to recognize it. The result of this project is a tree on multi-level, where at each step the network is able to classify between a specific class and others. Where others is the collection of all the classes except for that one.

The main idea is represented in the following image.



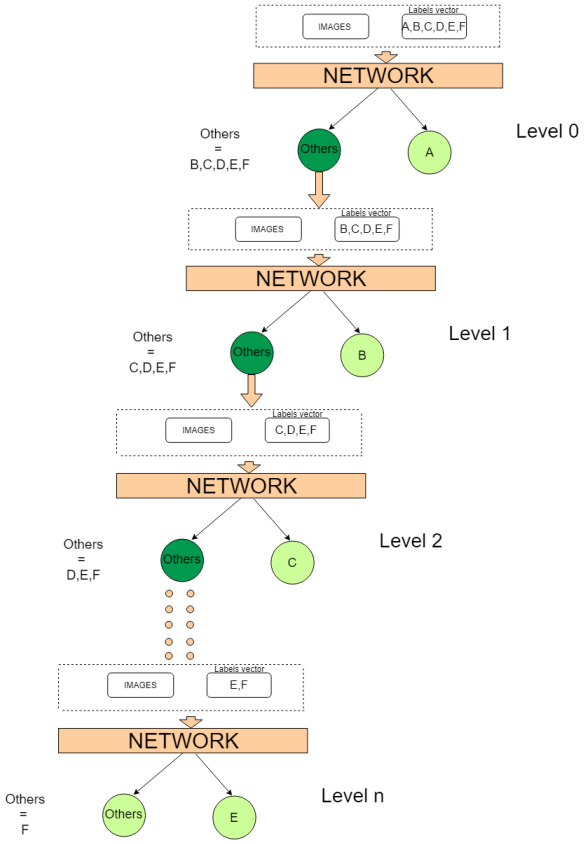


Figure 1: Original Tree

3 Convolutional Neural Network

Thanks to Deep Learning it has been possible remedy to some limitations of simple Neural Networks. In particular NNs have an high training cost due to the fact that a neuron may be considered as a regression algorithm, hence is very expensive train all the fully connected regressions. Furthermore training cost is directly proportional to the number of hidden layers because of backpropagation algorithm typically used in the training phase. DL overcomes these limitations introducing more layers to emulate the deep cognitive process of human brain. Humans first learn simpler concepts and then merge them in order to create a more complex and abstract macro concept DL try to seeks features giving them a sort of hierarchical representation through multiple learning stage. Each stage has features of higher level with respect to the previous one, moving toward the output layer each stage increase the level of abstraction and complexity. One powerful application of DL is Convolutional Neural Network CNN.

CNN's are composed of many blocks, each block may be made up of three important layers with different tasks, after the last block there could 2 or more fully connected layers in order to produce a final prediction. In the following, the blocks used are presented.

- Convolutional layer
- Pooling layer
- Non linear layer
- Fully connected layer

3.1 Function used in the CNN

3.1.1 Convolutional layer

The conv layer is the core layer of CNNs, it does most of the computation and it could be the bottleneck in network performance, in fact it has many tunable parameters. The CONV layer has a set of learnable filters. Every filter cover a small portion of the input data producing as output a feature map During the convolution, we shift each filter across the width and height of the input volume and compute dot products between the weights of the filter and the input at any position. The number of units we want shift the filter is called stride and it is one of the tunable parameters

In summary the output volume of each layer is affected by many parameters:

- Stride with which we will shift the filter, when the stride is 1 then we move the filters one pixel at a time, and so on.
- Number of filters determines the depth of the output volume.
-

3.1.2 Non Linear Layer

In order to add non linearity to the architecture is often applied a non-linear function to the output volume, the group chooses to use:

- f(x) = max(0, x) (ReLU)

3.1.3 Pooling Layer

Its aim is reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The CNN uses:

- Max pooling

3.1.4 Fully Connected Layer

Neurons in a fully connected layer have full connections to all activations in the previous layer, like regular Neural Networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset. So each neuron represent a different class and its output is the probability that input data belong to the that class. Usually in CNNs there are two or more fully connected-layers at the last step

4 Implementation

In this section the implementation is described using some reference to the code for a better explanation. The group decides to test the system, firstly using the dataset of cifar10 and subsequently using the the assigned dataset. As some purpose functions were made, they are presented in a first sub-section. After that will be presented as the group has approached the DeepTree.

4.1 Functions

As mentioned in the previous section, the following some main functions are reported. These first functions were used to treat CIFAR10.

4.1.1 *toDict (trainset)*

This function is made to translate the input in a different type of python variable, in particular a Dictionary. This step is done, because in some case the group needs to modify the training set of CIFAR10. Therefore labels and data are put in a dictionary. In the end a dictionary is created with the different classes as keys and all the data with that label as values.

4.1.2 *dropClass (dictInput, className)*

In this task a simple *pop* is performed. It takes as input a dictionary and a label of the class that should be popped-out. *DroopClass* returns the modified dictionary and the eliminated item.

4.1.3 *returnValues (dictInput)*

Simply function that takes in input a dictionary and returns two array:

- Labels
- Values

At the end of this function labels is composed by only two classes, named '0' (when it should be *others*) and '1' (when it should be the leaf). This step is done because having only two classes, the net gives as output only 0 or 1.

4.1.4 *datasetBalancing (originalDict, nClassesToDrop)*

This function has the purpose of balancing the dataset before using it in CNN. (In the next section it is seen the purpose of this function). Some of the previous functions are used.

---

tempDict = originalDict.copy()

print("Start␣rebalancing:␣classes␣to␣drop␣=␣" + **str**(nClassesToDrop))
for index **in range**(0, nClassesToDrop + 1):

```
    classToDrop = list(tempDict)[0]
    class0, tempDict = dropClass(tempDict, classToDrop)
    if len(tempDict.keys()) == 1:
      print("Last step: only 2 classes remaining")
      dictForCNN = {list(tempDict.keys())[0]:
       tempDict[list(tempDict.keys ())[0]], classToDrop: class0}
      values, labels = returnValues(dictForCNN)
      return values, labels
    print('Class ' + str(classToDrop) + ' dropped')

  class1 = tempDict.copy()
  #class0, class1 = dropClass(originalDict, classToDrop)

  elementsFromEachClass = int(round(len(class0) / len(class1.keys())))

  print('We will take' + str(elementsFromEachClass) + ' elements from each class ')

  class1_balanced = []
  for k in class1.keys():
      myClassElements = class1[k]
      random.shuffle(myClassElements)
      class1_balanced += myClassElements[:elementsFromEachClass]

  dictForCNN = {}

  dictForCNN.update({'others': []})
  dictForCNN.update({classToDrop: []})

  for i in class1_balanced:
      dictForCNN['others'].append(i)
  for i in class0:
      dictForCNN[classToDrop].append(i)

  values, labels = returnValues(dictForCNN)

  return values, labels
```

Starting from the input data, two distinct classes are generated. The first one will act as a leaf (called *class0*) and it is populated by one class of the original dataset. The second one is a "super" class containing all the elements belonging to the remaining classes (called *class1*). If the input of the function containg only two elements, it is excluded from the previous step. To keep the dataset balanced, the size of *class0* must be equal to the size of *class1*. In order to do that, the number of elements that have to be extracted from each "subclass" of *class1* is calculated based on the size of *class0* and the number of labels contained on *class0*.

To avoid to take always the same elements, the items of each labels are previously shuffled.

The *class0* is obtained using the function *dropClass*, then as already said, elements are balanced. At this point only one dictionary is generated, and two vectors are given as return using *returnValues* function, i.e *dictForCNN*.

4.1.5 *newclass(trainset_gold,extraclass)*

This task is used only when the net is fed with CIFAR10's dataset. It takes as input the original dictionary and the extra class that it will be added to the net, and returns as output two vectors generated by the function *returnValues*. Similarly to the function *datasetBalancing*, it merges two dictionaries, creating a unique one composed by only two labels '1' and '0'. Also in this case during the merge, the first dictionary is bigger then the second one, for this reason some elements of the first are randomically eliminated by each class that composed it.

Now the functions covered will be those used to modify the medical dataset.

4.1.6 *Balacing(set,j)*

As the group imports the dataset using the pytorch function *ImageFolder()*, the output of *torchvision.datasets* is different from the one obtained by CIFAR10. For this reason this function is used. *Balacing()* reads the vector of tuple, remove from the vector n-classes. Where n is the value of j. J is the loop's index described afterwards. J is also the class that will be the leaf in the net. As in the *datasetBalancing ()* the leaf and the superclass are managed, and the labels are re-named in '0' ans '1'. At this point a dictionary with only two keys is ready. The dictionary is now transformed in a vector of tuple, where the format is:

```
[(image , label),(image,label).....(image,label)]
```

4.1.7 *official_script.py*

Official_script is a python code used to transform the google drive dataset in a readable one. Thank's to google api we able to mount our gdrive folder into colab server file system. In order to group image files into their own class we create a folder for each class into drive. Then saving imgs original path into a dictionary where keys the classes we iterate over them in order to move the image into the right new folder using shutil library. After grouping all images into right folders we split dataset into test set taking just num_file samples from each class.

The test function will be presented at the end.

5 The Network

The DeepTree is designed as a tree composed by several small networks. Each network is able to recognize only two classes. The first one of these classes is a simple "mono-class" acting as the leaf of the network, the second one is a "super-class" composed by the all classes of the input vector with the exception of the leaf. Each network is a decision level. The level selects all the images belonging to the leaf of the network. The images not belonging to this level are classified as "others" and forwarded to the next level. The last network of the tree is composed by two leaves. At any time an extra class can be added to realize a Growing DeepTree. This is obtained by adding a new network to the tree. This new CNN is able to recognize only the new class from the totality of the old set. By adding a new net at the top of the tree, it is not necessary to train the old tree at every time, but only the new network.

5.1 Code explanation

As already mentioned, the group trains and tests the project using two different datasets. Therefore the code has been modified to adapt it to the sets.

At the first time the problem has been approached by the working group using CIFAR10, taking only five classes of ten. The trainset is modified several time in the code. To avoid to loose the original training set, it is downloaded two times, creating a not-touched "golden" version. This version is used as reference. As the previous section explains, the total network is designed as a tree. This tree is realized as a vector, where each trained net is stored. Each level of the tree is a cycle of a for loop. The particular type of the CIFAR's set, requires to change its format. Therefore at every time that the set is modified, it is transformed in a dictionary using *toDict*.

Looking the code, it can be seen that at each cycle *trainset.data* and *trainset.targets* are modified. In fact each cycle corresponds to a level of the tree. More in deep it goes less classes are contained in the class *others* of the corresponding network. *DatasetBalancing* takes care of it. So at each cycle it changes the objects of the trainset, removing classes and making the elements balanced. The images and the corresponding labels of the trainset are passed to one level of the tree. After five epoches used to train a single net, the trained CNN is appended to the tree vector. At the end the test phase starts, and the accuracy is calculated.

An extra class is defined. *Newclass(trainset_gold,extraclass)* this function is used to create a two-classes dictionary balanced. A simple training CNN is used to recognize the new class from the old set. After the net is trained, it is inserted in the first position of the tree vector. The other networks are shifted of one position inside the vector. The test phase starts again and the accuracy in calculated again.

For simplicity the group decides to explain how it works only at this point.

5.1.1 *test_accuracy (tree, dataloader, originalDict):*

This task calculates the accuracy. As it can be seen, it receives as input:

- Tree, that contains all the trained classes.
- Dataloader is the testset.
- OriginalDict is used to know how many classes are present inside the testset, and the order of the tree's leaves.

The same classes excluded for the training set are excluded also for the testset using a simple if statement. The group decides to use a simple test function used for a generic CNN, but with some changes. A loop is created. It is used to pass a single image through the various levels. At the beginning of the loop the label to check is stored in *labels_renamed* . If the label to check corresponds to the leaf of that particular level, *labels_renamed* is overwrite with '1', '0' otherwise. This step is performed because each cell of the tree vector is formed by a network of two classes, so the choice is '0' or '1'. At this point the image is passed to a network of the tree pointed by the loop's index. If the output's prediction of *torch.max* is '1', the fairness of the output is checked. If the hypothesis is true, correct is incremented. Correct is the variable that keeps the bill of the right prediction. If the output's prediction of *torch.max* is '0', the image is passed to the next level of the tree. In the end the accuracy is calculated.

As already said the groups decide to train and test the Growing Deeptree using also a different dataset. To do this some functions are added, described below.

6 Results

In a first part of this section the results of CIFAR10 are presented, and in the second part the results of the other dataset are presented.

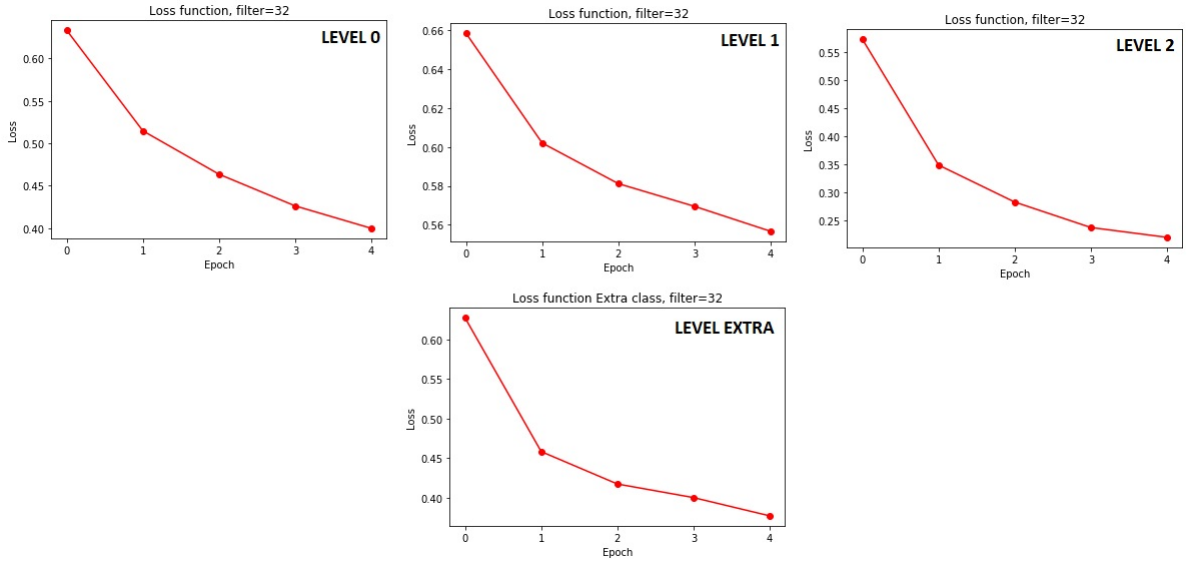


Figure 2: Loss with five epochs and filter size 32

The group changes the number of epochs and the size of the filter for both the two dataset to improve the accuracy.

In figure 2 a first result is shown. The image is composed by four graphs. Each graph of the first three represent the loss of the three levels. The last graph is the loss obtained by the network used to distinguish the extra class from the others. As the image shown, at each epoch the loss decreases, because the net is learning.

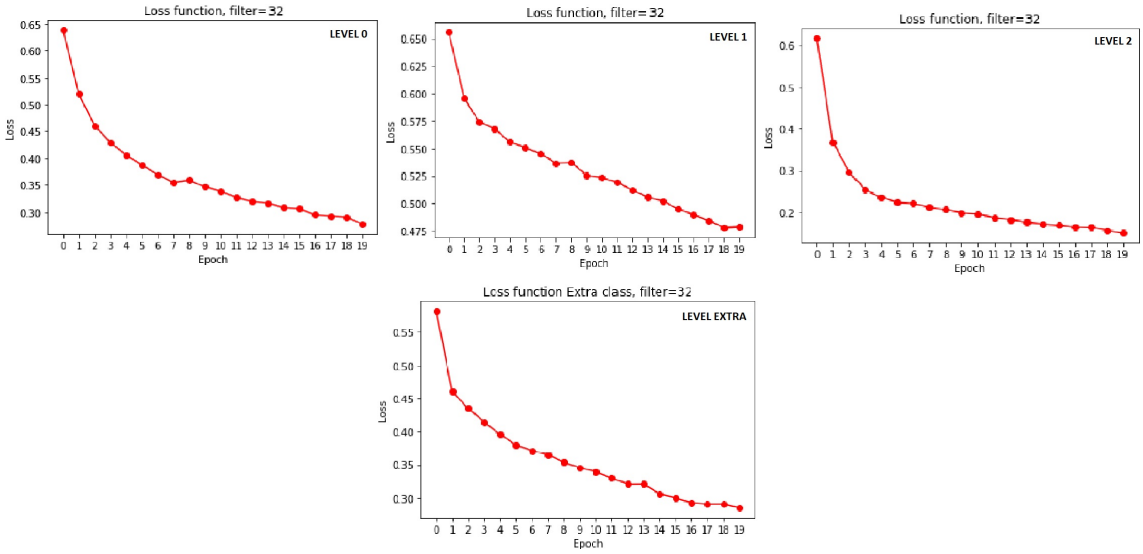


Figure 3: Loss with twenty epochs and filter size 32

In 3 the number of epochs are increased to twenty. The loss is quite smaller than the fist case. This is due to the fact that the network has more time to learn.

As second step the group changes the filter size to 256. In this way the image is divided in smaller details. So the CNN has more information. The net requires extra time than before to elaborate the image, but more details to distinguish them.

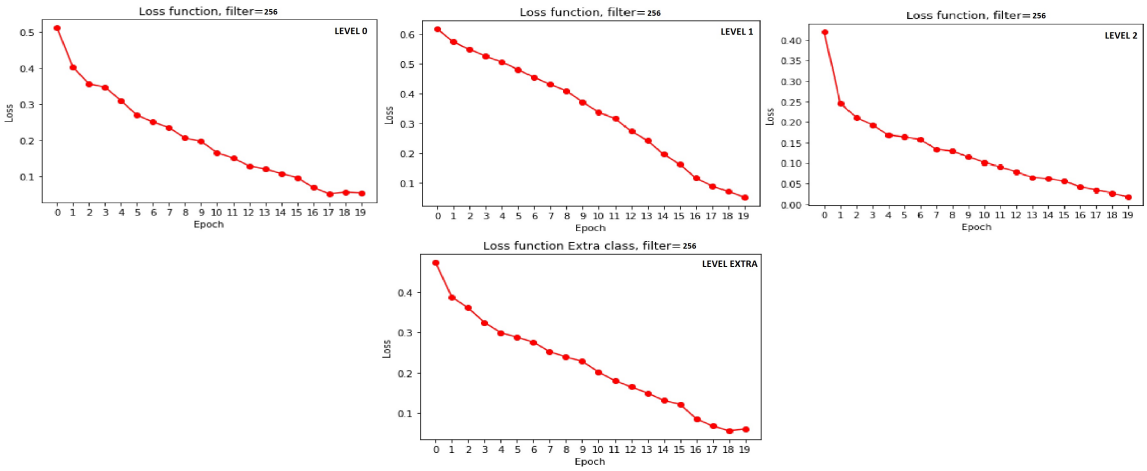


Figure 4: Loss with twenty epochs and filter size 256

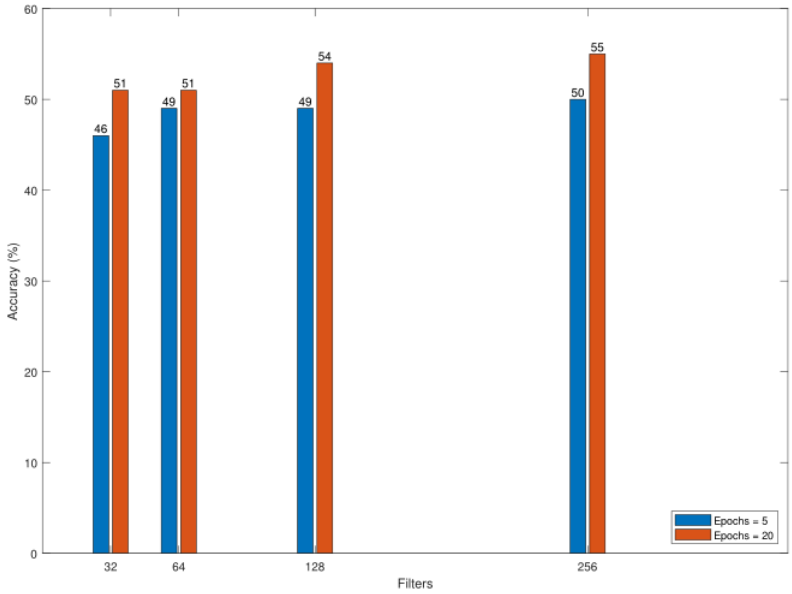


Figure 5: Accuracy

In 5 the accuracy is shown. As for the loss the accuracy is studied changing the filter dimensions and the epochs. On the y-axis there is the accuracy level in percentage, on the x-axis there are the filter dimensions. The red columns are the accuracy measured using twenty epochs, the blue ones are that one measured using only five epochs. As for the loss, the accuracy is improved increasing the filter dimension and the number of epochs. Instead the tables show the difference between the accuracy obtained by the network with only four classes, and the one obtained by adding the new network to the top of the tree. As it can be seen the two values are almost similar. This means that the Growing tree works correctly. In the first table the tree works with five epochs, in the second with twenty.

filter	Accuracy w/o extra class	Accuracy w extra class
32	46%	47%
64	49%	49%
128	49%	47%
256	50%	52%

filter	Accuracy w/o extra class	Accuracy w extra class
32	51%	51%
64	51%	54%
128	54%	57%
256	55%	55%

Now the results obtained by the medical dataset are analyzed. As for the other dataset, the group changes filter dimension and epochs. Also in this case the loss is improved increasing the two parameters, it can be seen in the following images.

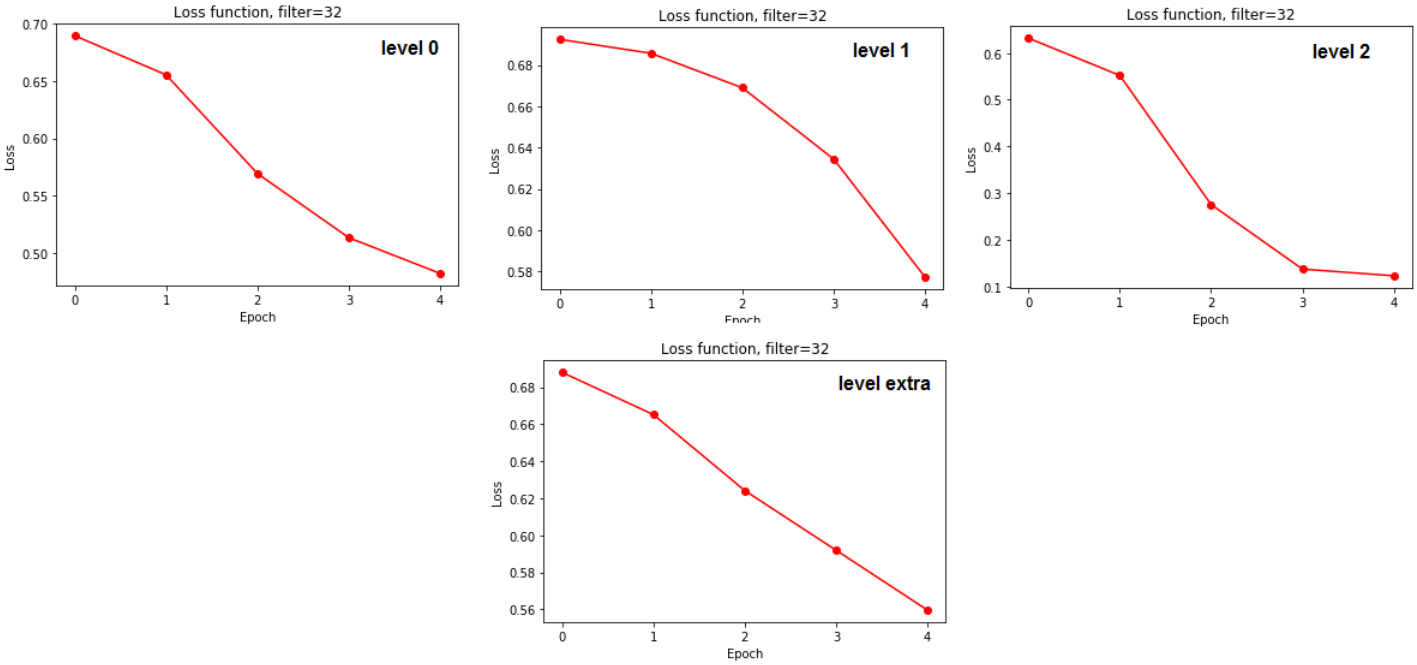


Figure 6: Loss filter=32 epochs=5

In the following the improvement of the loss due to the increase of the filters dimension.

As it can be seen in the table the accuracy is similar to the previous results, but this time pass from the old tree to the new one, decrease the accuracy.

filter	Accuracy w/o extra class	Accuracy w extra class
32	48%	49%
64	54%	48%
128	56%	42%
256	56%	51%

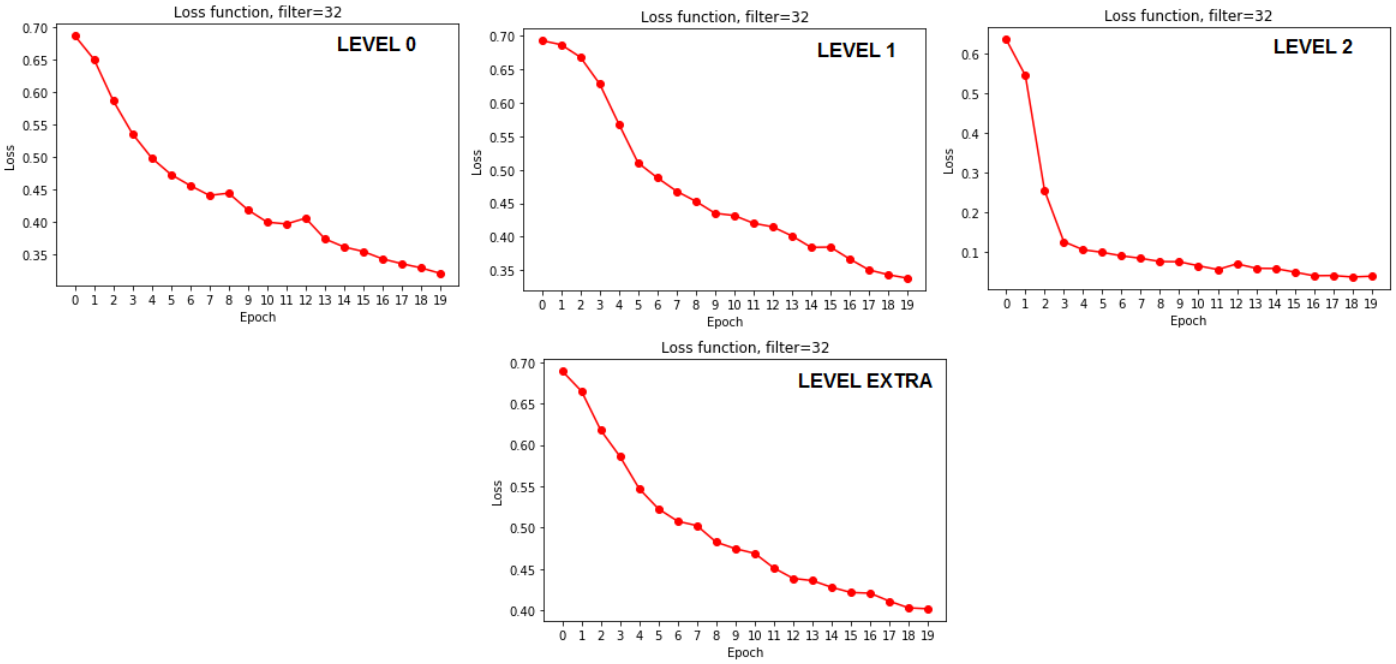


Figure 7: Loss filter=32 epochs=20

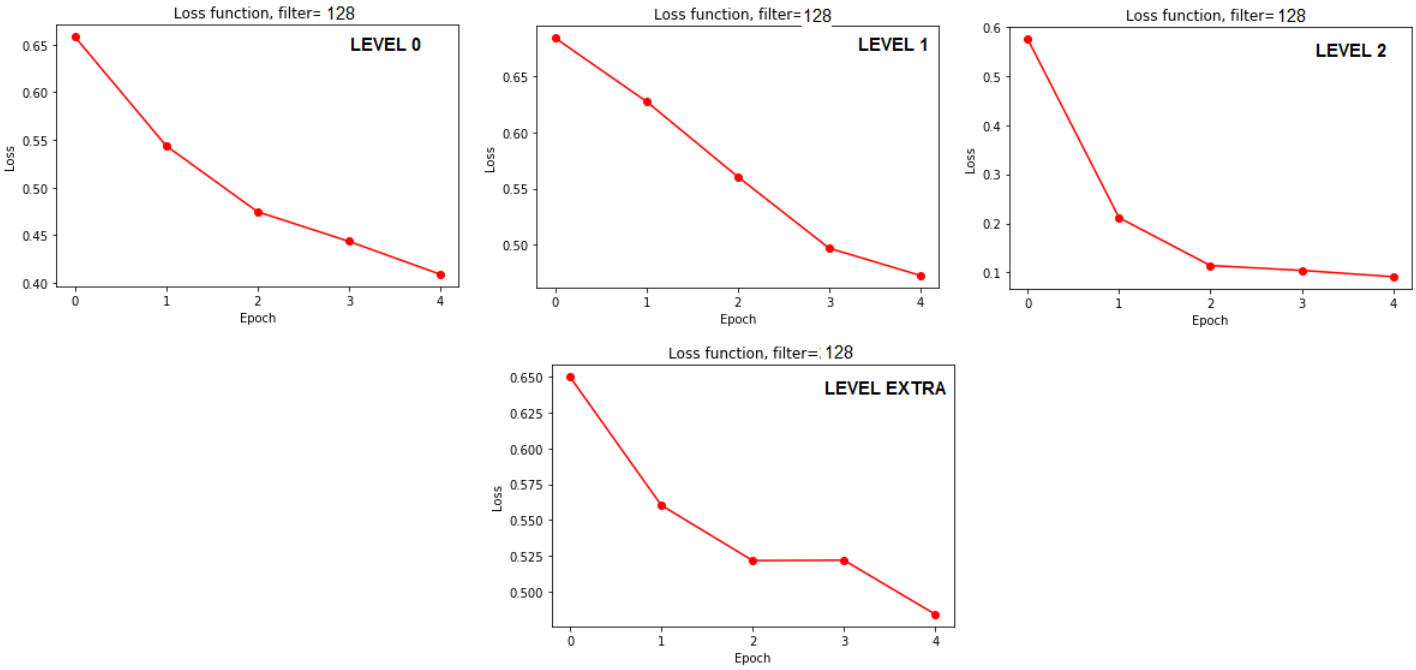


Figure 8: Loss filter=256 epochs=5

filter	Accuracy w/o extra class	Accuracy w extra class
32	60%	56%
64	66%	56%
128	61%	53%
256	71%	62%

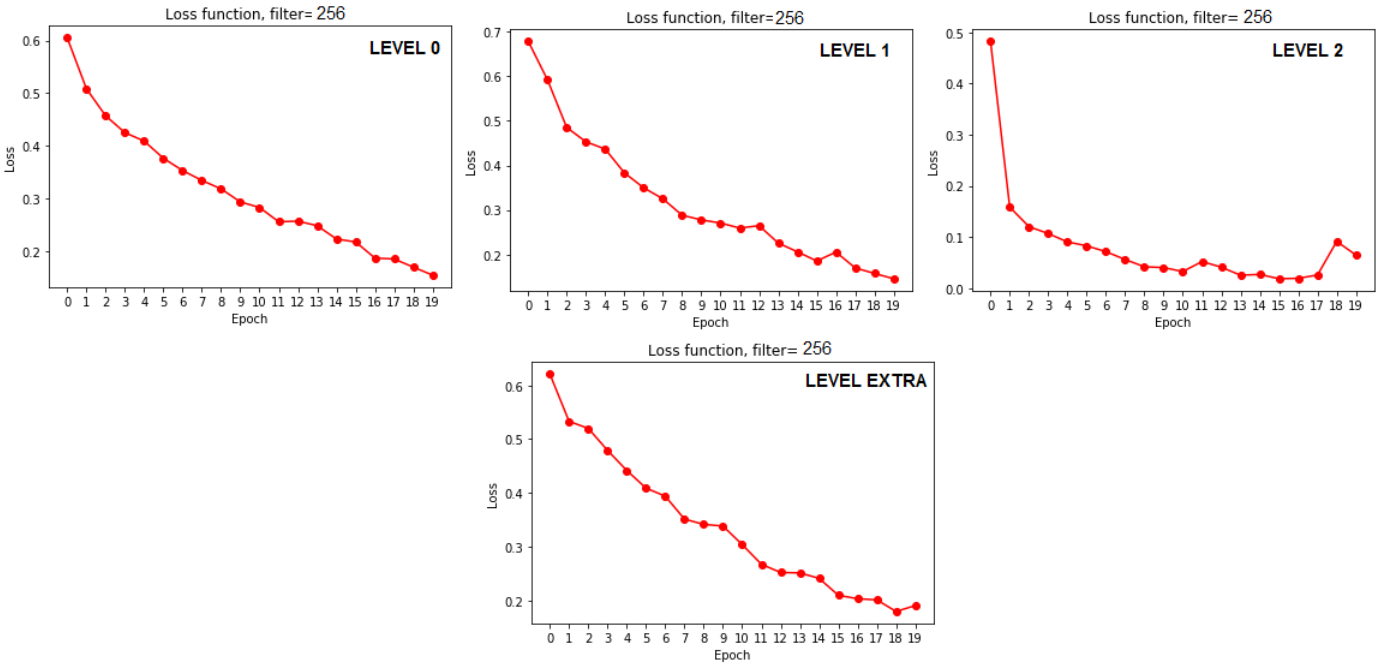


Figure 9: Loss filter=256 epochs=20

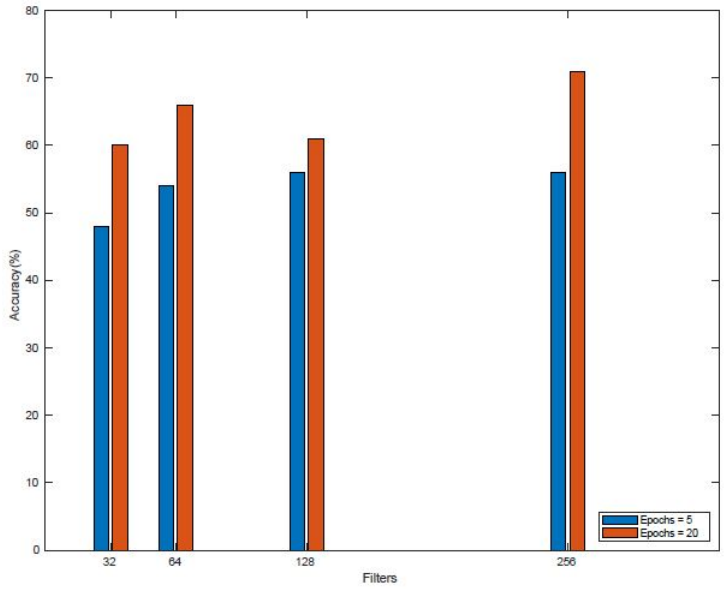


Figure 10: Accuracy

7 Conclusion

In conclusion, increasing the epochs the network work better, but this especially in the second dataset increase a lot the time. Adding an extra class the network is able to recognize it without training all the nets, but in this way the accuracy is a little reduced. Some subsequent improvements can be added to CNN to improve it. For example, the ratio rating can be increased in line with the size of

the filters. This operation can be dangerous because, the time is increased but above all it risks the overfitting. Other blocks that improve accuracy can also be added to the network.