Multi-label classification
Introduction to Deep Learning project

University of Helsinki

Faculty of Science

Group 8

Enrico Buratto - 015621911

Giacomo Grandi - 016336894

Rayyan Hassan - 015635635

Academic Year 2021-2022

Contents

1 Introduction

The goal of the project is to implement a Neural Network capable of recognizing several objects inside an image. Since a picture can have more than one label, though, this problem is different from a simple multi-classification task: it is, indeed, a multi-label multi-classification job.

As demonstrated several times over the years[1], Convolutional Neural Networks (CNNs) represent the gold standard when dealing with image classification tasks: their ability to recognize patterns and to take into account spatial coherence in the input, in fact, fits perfectly with image recognition; using a CNN is, therefore, the central hinge of our approach to the problem. However, as we will analyze thoroughly in this report, the possibility of having more than one class inside the same image brings with it inevitable challenges. As we will also explain later, a standard Convolutional Neural Network is, in fact, composed by convolutional and pooling layers followed by a simple Feed Forward Neural Network (FFNN), and the standard approach when using the latter is to consider *the brightest neuron* of the output layer (*i.e.* the node with higher weight) as the final prediction. However, for multi-label classification we need (possibly) more than one prediction for each picture; this is, therefore, a problem that must be, and is, addressed.

This document, which also reflects our approach to the problem, is structured as follows: first of all, we describe the data exploration we performed in order to btter understand the domain of the problem. Secondly, we discuss the different CNN models we have taken into consideration and we used. After that, we will explain all the regularization and optimization techniques we used and why; finally, we present the result we were able to achieve and we discuss them.

The entire project has been developed in Python, and both the training and the testing of the models have been performed on a local machine; the most important libraries we used are Numpy and Pandas for data handling, scikit-learn as an aid for one-hot encoding of the labels and, obviously, PyTorch for modeling and using the neural networks for training and testing tasks.

2 Data

Before starting with explaining the model we implemented and every other choice we made, a small introduction on the data should be done. This, among other things, reflects the approach we had with regards to the problem: before starting any other work, in fact, we performed some exploratory data analysis so that we could know better the problem itself.

In this section, therefore, we describe this first approach, followed by a brief description of the custom `Dataset` class we implemented in order to have a coherent collection of data that could be used for the training and the other tasks. In the end, we finally describe the general approach we had with regards to the *train-dev-test* split.

2.1 Exploratory Data Analysis

As we were saying in the introduction, in order to have a better understanding of the domain of the problem, we performed some exploratory data analysis.

First we tried to find out if the images had different formats or dimensions. We found out that there were some black and white pictures and some color ones, in particular there are 18758 color pictures and 1422 black and white ones.

Then we wanted to have a look at how the dataset is balanced, counting the number of occurrences for each of the 14 classes of the set. We used a bar chart to properly show this splitting of the dataset. As we clearly can see from the chart below, the dataset is strongly unbalanced: some classes have much more occurrences compared to others.

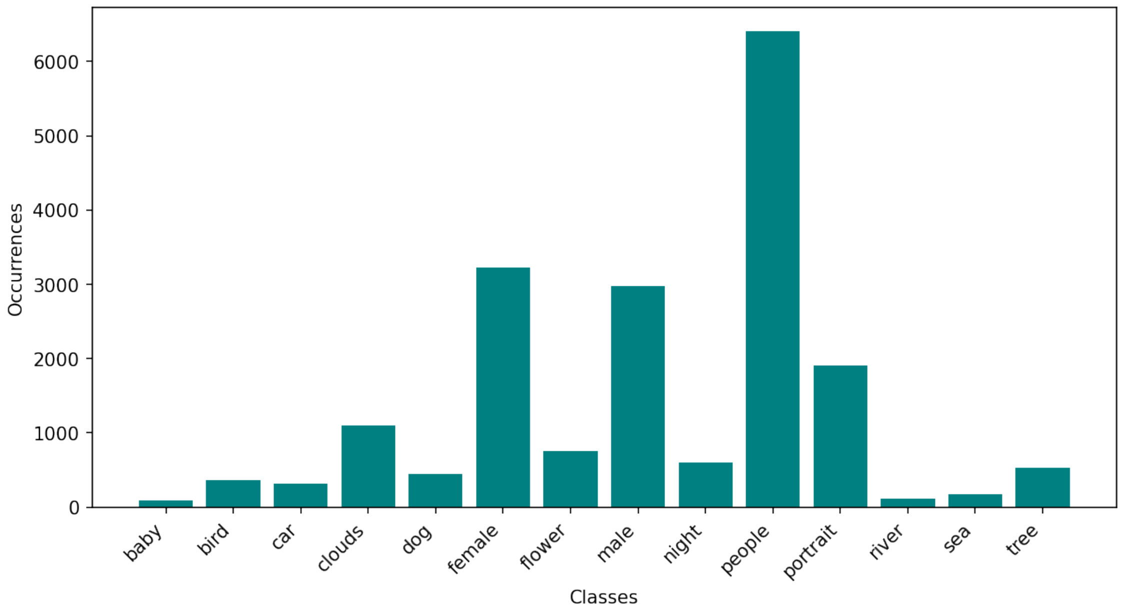

Figure 1: Classes' occurrences of the dataset.

Since the dataset is like this we understood that in order to get nice prediction results we had to perform some kind of regularization, which will be better explained later in §4.

Finally we counted the number of class labels for each image and grouped the pictures by the number of labels in them. We obtained what is shown in the table below.

Number of class labels	Number of images
0	9824
1	4161
2	2388
3	3230
4	388
5	9

Table 1: Number of class labels for each image.

Apparently there are a lot of images with no labels at all inside them and no image representing more than 5 classes.

After this data exploration we achieved a better understanding of our dataset that let us better organize our code and our network accordingly. The code for this data exploration can be found in `data_analysis/analysis.ipynb`.

2.2 Dataset class

As mentioned in the beginning of this section, we decided to implement a custom `Dataset` class, which is called `ImageDataset` and can be found in the `data_loader.py` file inside `utils` folder. We did this because of the non-trivial structure of the dataset itself, in which the labels are not reported grouped by image but by labels themselves.

As explained in the PyTorch documentation[4], a custom `Dataset` class can be created by inheritance from the standard PyTorch implementation. Three methods need then to be overwritten to make the class working: `__init__`, `__len__` and `__getitem__`; in addition to this, we implemented another method for convenience, `__get_labels`.

init This is the method that deals with the class fields initialization; it is then the first method that is called when the new dataset object is initialized. In our implementation, this method receives the directory path to the labels and to the images, the classes for the multi-label multi-classification task and, possibly, a `Transform` object for data augmentation. It then prepares the label for the training: in order to do that, we used the preprocessing feature from `sklearn`[5] to create an encoding for the labels. This encoding consists in the creation of one-hot vectors in which the position of an element (0 or 1) represents the class: for instance, if an image belongs to two classes, say the third and the fifth in the encoding, the one-hot vector will be `[0,0,1,0,1,0,0,0,0,0,0,0,0,0]`.
After this encoding, the images paths and the labels are loaded into a dataframe for future utilization, and the transform objects are initialized.

get_labels This is a convenience method that we used to find all the labels for each image. It receives an image name as input and returns the list of classes associated with that image.

len This is the method that returns the length of the dataset, *i.e.* its size; since our is a custom implementation with dataframes, the method has been overwritten using a suitable procedure to get the length.

getitem This is probably the core method of the class: its purpose is to read an image in the dataset given an index, apply the transformations and return both the image (with possibly its transformations stacked in the same image object) and the target, *i.e.* the labels. Our implementation is obviously adapted to the data we have, but nothing in particular has to be reported except for the mode with which we read the image. As mentioned in the previous section, in fact, some images were black and white and some had colors: therefore, we changed the mode to `ImageReadMode.RGB` in order to get images with a coherent size.

After creating the custom class, we used it to load all the images and labels into a single object. From this object we then divided the data into the different sets for training, validation and testing and we loaded them into different `DataLoader` objects. This operation is quite standard, as it is often used to iterate over data during the training and the testing phase; however, some considerations have to be done.
First of all, the data inside these iterators is not shuffled during the loading; even if this is a common procedure when dealing with multiple-class classification tasks, for this problems is not mandatory since the annotations files are randomly composed; moreover, we found that we had discording results using the exact same model and hyperparameters, and this was due to the more randomness of data during training and testing.
Secondly, the data is loaded into batches, and this is mandatory if we want to use batch training; however, being our dataset a custom object, we had to write also a collate function. This function is reported in the code with the name of `collate_fn`, and its purpose is to process the list of samples and zip them in order to form a batch which can be used by a standard implementation of the training procedure.

Coming to the dataset partition, as already mentioned we adopted different dataset split techniques to deal with the training and the performance measurement of the program. Since we didn't have a proper test set during development, and the dataset wasn't too big, we first decided to not have a validation set, but to divide the data in 80% training set and 20% test set; in this implementation, we used the test set also as a validation set for the early stopping regularization, which we will discuss later. However, a boolean flag (`USE_VALIDATION`) in the code can be modified if one wants to have three different sets.

3 Model

In this section we explain the general approach we had in creating our model; after this, we describe the different models we tried and the motivations behind, giving also an insight on why one model could be better than the other.

3.1 General approach

The general approach we had has been to first starting with a simple Feed Forward Neural Network; we did so just to check if everything else, from the data loading to the training and testing phases, was working fine. We then built a simple network with an input layer of size $n = 128 \cdot 128 \cdot 3 = 49152$, a hidden layer with size $n = 128$ and an output layer with size $n = 14$, which is the number of classes of the dataset. We then ran the first tests and we ascertained that, as expected, the performance was very low: as already mentioned, images are composed of really sparse yet really localized data; therefore, a simple FFNN could not work fine.

After this first approach we created three different Convolutional Neural Networks with different number of layers and different layer sizes; we then used these three CNNs in our development, applying different regularization and optimization techniques and tweaking the hyperparameters. This, however, is described in the next sections.

3.2 CNN

As already mentioned, we decided to use Convolutional Neural Networks because they represent the gold standard when dealing with image recognition; this is due to several characteristics that differentiate them from standard FFNNs:

- CNNs permit locality recognition: in images, usually, nearby pixels are more strongly related than distant ones, and convolutional filters enable the recognition of them;
- CNNs hold the parameter sharing feature, *i.e.* they are able to share the weights by all neurons in a particular feature map; this, in particular, speeds up the computation;
- An image can be considered as an enormous set of features, where each feature corresponds to a pixel position. CNNs are capable to reduce this dimensionality using pooling layers; the input layer for the consequent FFNN is then smaller, and having less sparse data allows to learn faster and better.

As mentioned, we built three different models in order to compare them and see which model was the best for our problem; these three models are here described, while the discussion on the results is left for §6.

First CNN The first CNN we built is composed by three convolutional filters, two pooling layers, a possible dropout layer and a final FFNN with one hidden layer, which works as the classifier; the first two convolutional layers are followed by a pooling layer, and the dropout layer is placed between the hidden layer and the output layer of the FFNN. The convolutional and pooling layers are composed as follows:

- The first convolutional layer has 3 input channels, 6 output channels and a kernel size of 4;
- The first pooling layer has a kernel size of 3 and a stride equals to 2;
- The second convolutional layer has 6 input channels, 16 output channels and a kernel size of 4;
- The second pooling layer has a kernel size of 2 and a stride equals to 2;
- The third convolutional layer has 16 input channels, 32 output channels and a kernel size of 4.

The scheme of this architecture is here reported.

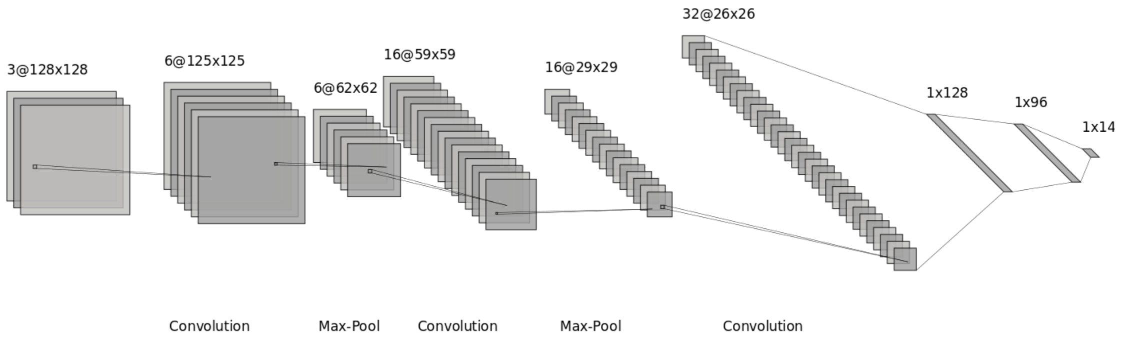

Figure 2: First CNN architecture.

Second CNN The second CNN we built is composed by the same amount of layers, with the only exception of the channel sizes, which are different:

- The first convolutional layer has 3 input channels, 6 output channels and a kernel size of 4;
- The first pooling layer has a kernel size of 3 and a stride equals to 2;
- The second convolutional layer has 6 input channels, 6 output channels and a kernel size of 4;
- The second pooling layer has a kernel size of 2 and a stride equals to 2;
- The third convolutional layer has 6 input channels, 12 output channels and a kernel size of 4.

The scheme of this architecture is here reported.

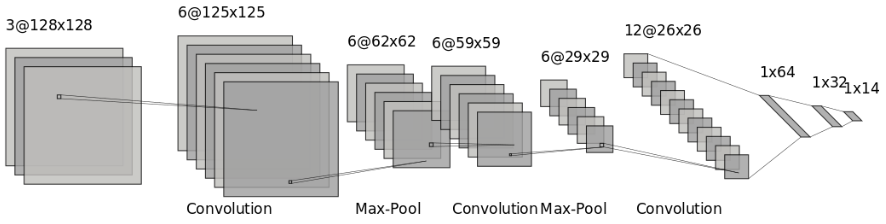

Figure 3: Second CNN architecture.

Third CNN The third CNN we built is composed by a higher number of convolutional and pooling layers. To be more precise, this model has 5 convolutional layers, 5 pooling layers (each one follows a convolutional layer) and the same amount of hidden layers for the CNN. The sizes of the layers are as follows:

- The first convolutional layer has 3 input channels, 6 output channels and a kernel size of 4;
- The first pooling layer has a kernel size of 3 and a stride equals to 2;
- The second convolutional layer has 6 input channels, 6 output channels and a kernel size of 4;
- The second pooling layer has a kernel size of 2 and a stride equals to 2;
- The third convolutional layer has 6 input channels, 12 output channels and a kernel size of 4;
- The third pooling layer has a kernel size of 2 and a stride equals to 2;
- The fourth convolutional layer has 12 input channels, 24 output channels and a kernel size of 4;
- The fourth pooling layer has a kernel size of 3 and a stride equals to 1;
- The fifth convolutional layer has 24 input channels, 48 output channels and a kernel size of 4;
- The fifth pooling layer has a kernel size of 2 and a stride equals to 1.

The scheme of this architecture is here reported.

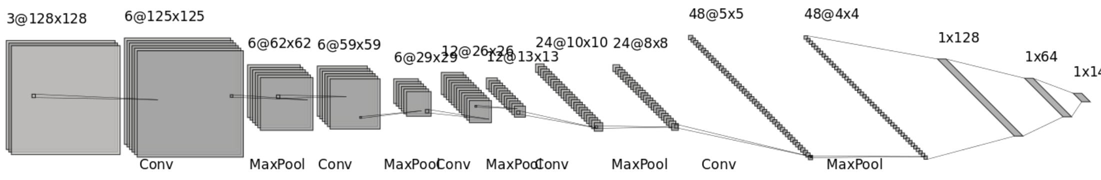

Figure 4: Third CNN architecture.

After having defined the network, some final explanations on how the training works must be given. As mentioned in the introduction the main idea of a FFNN, which is the final part of our network, is to output weights for each class in the set of classes: a higher weight corresponds to a higher probability of that class to be in the gold target. In a standard multi-class but single label classification, one gets the final results by just taking the highest output. In our problem, however, we need possibly more than one output for each tested sample in the dataset; this is why we decided to not take only the *brightest* neuron, but all the neurons with a weight greater or equal to a certain threshold. This threshold has been calculated experimentally, doing some *trial and error* to see which gave us the best results; the final value for this parameter is discussed in §5.

A final note should be done on the activation function for the output layer of the FFNN and the loss function we implemented. These are, respectively a sigmoid and the binary cross entropy loss. The former, in pytorch, is represented by the following function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

This function returns values between 0 and 1; in particular, the value is not null also before $x = 0$, unlike other functions such as the classical ReLU.

The latter is a loss function that compares each of the predicted probabilities from the FFNN to an actual class output that can be 0 or 1, unlike for instance the standard cross-entropy loss. In pytorch, the function then calculates the score that penalizes the probabilities based on the distance from the expected value.[3]

Even if controversial because apparently not supported by any proper research, this seemed to be a good combination when dealing with multi-label classification, as explained in [2]. This was, actually, experimentally verified by us trying also a softmax activation function combined with a normal cross-entropy loss function; the results using these two functions were overall worse than the first combination, yet this is not a scientific and proper proof.

Speaking of, the usage of this combination is also the motivation behind the fact that we did not use any class weighting during the training: pytorch implementation of binary cross-entropy loss, in fact, does not provide this functionality, yet the weighting is only on the batch; however, weighting the samples in a batch would not have improve anything, thus we relied on the regularization techniques described in the next section.

4 Regularization and optimization

4.1 Regularization

Regularization is the set of techniques used to prevent overfitting in the model. This is because the model is incapable of making good prediction on data it has not previously seen. The dataset for our project was unbalanced thus regularization was necessary to reduce the bias that would have been present due to the uneven data distribution. This technique functions by penalizing the model for every parameter that it adds thus preventing the formation of a complex model. This can be done in a multitude of ways such as the addition of dropout, weight, decay, data augmentation and early stopping. Each of these techniques assist the prevention of overfitting in their own unique way.

- **Dropout**:

  The dropout layer deactivates the neurons which are not overtly useful to the model thus they reduce the number of parameters being used and increase the regularization of the network. They are commonly used with CNN. In our project we applied them after the pooling layers. Their presence is essential in CNN as without them the probability of overfitting becomes quite high as the first batch of training data has an inordinately great effect on the overall learning of the model.

- **Weight Decay**:

  Weight decay adds a penalty term to the cost function of the algorithm so as to reduce the size of the weight during back propagation. This can also be referred to as the L-1 regularization technique as it takes advantage of the the same phenomenon. Thus the size of weights is reduced which in turn prevents the gradient vanishing problem as well.

- **Data Augmentation**:

  Data Augmentation is especially useful for image classification tasks as it greatly improves the accuracy of the algorithm. It does this by modifying the dataset. We have added the following transformations in our model : Color-jitter, Random adjust brightness, Random invert and Random rotation. These techniques are especially useful when the dataset is large as is the case in our project.

- **Early Stopping**:

  The performance of a model usually starts to deteriorate after a certain number of epochs therefore during training it is a common practice to add early stopping. Thus we choose an arbitrary value for the training epochs and the model automatically stops training when the loss starts increasing i.e the performance of the model starts to decrease

4.2 Optimization

The objective of all machine learning and deep learning algorithms is to create a model which has the lowest possible error when making predictions while also preventing overfitting. Thus, a proper balance between the two is required.

There are a variety of techniques which can be utilized to optimize the algorithm and these include hyper-parameter tuning, and optimization algorithms such as ADAM, ADAM-grad and Stochastic Gradient Descent (SGD). The initial step was to try the different optimizer algorithms and check which optimizer provides the best results. In our project, we obtained similar results with both the SGD and ADAM.

SGD is one of the most powerful optimizers as it combines the best qualities of gradient descent while fixing the problems associated with it. The biggest problem that came with the use of Gradient Descent (GD) was the high computational time that it took when working on large data but this was fixed through the utilization of SGD. This was accomplished by adding randomness in the GD algorithm as the algorithm now took random data points from the data set after each iteration.

ADAM was the optimizer which provided the best overall results on the performance metrics of precision, recall and F-1 score. The most significant difference between ADAM and SGD was that ADAM had a per-parameter learning rate whereas SGD utilized a constant learning rate for all the weight updates. Moreover, it reduces the noise on a large dataset as well which was quite advantageous as our dataset was relatively large. However, the greatest reason behind the utilization of ADAM was that it was computationally efficient while providing great results. After choosing the algorithm, the last step is to tune the different hyper-parameters such as learning rate, batch size, number of epochs, patience and weight decay. More information on these hyper-parameters and their tuning is mentioned in the next section

5 Results

After defining the model and explaining the choices we made, in this section we describe the results we achieved on the test dataset with the final model, regularization and optimization techniques. Before reporting the results, however, we firstly describe the performance metrics and the justification for them.

5.1 Performance metrics

Being this problem a multi-label and multi-class classification task, we could not measure the performance of the system with the usual accuracy score; the latter, in fact, is calculated on the number of correctly guessed targets divided by the number of total samples in a test dataset. Even if this could still work, it suffers from a big problem: it's hard to state how to count the semi-correct inferred samples; if we count only the totally correct samples, we would probably get a close-to-zero accuracy even with a pretty precise model, because only a wrong inferred label on a sample invalidates the full inference.
We then decided to use the three most typical performance measure for this kind of problem: precision, recall and F1-score.

Precision This metric is generically defined as number of true positives divided by the number of true positives and false positives:

$$precision = \frac{TP}{TP + FP}$$

In our domain, we express precision as the proportion of correct inferred labels on the total number of inferred labels; it is, therefore, an expression of *how precise* our model is, disregarding how many labels are inferred.

Recall This metric is generically defined as number of true positives divided by the number of true positives and false negatives:

$$recall = \frac{TP}{TP + FN}$$

In our problem, we define recall as the proportion of correct inferred labels among the total number of correct labels; it is, therefore, an expression of *how many* correct labels our model infers.

F1-score This metric has not a real counterpart in reality; it is, in fact, defined as the harmonic mean of precision and recall:

$$F1 = \frac{2 * precision * recall}{precision + recall}$$

It is however useful to compare the performance of two different systems, as explained below.

These three metrics enable us to fully understand how the performance of the system is going: low precision and high recall, for example, mean that the system is inferring more labels for every image in average but there is a high number of wrong guesses; vice versa, high precision and low recall mean that the system is probably inferring less labels, but most of them are correct.
In order to compare the overall performance of two or more versions of the system (*e.g.* different hyperparameters or a different network structure), we can finally use the F1-score: a lower score on system A means that system B is performing somewhat better, and vice versa.

In our implementation, all the three metrics are computed both during the training phase and during the testing phase: during the former we get these measurements after every epoch, in order to have an idea of how the training is going; in the latter, we obviously compute them in order to know the final performance of the system. The code for these metrics calculations can be found in `utils/performance_measure.py`.

5.2 Hyperparameter tuning

After having created the three models, we started tweaking the hyperparameters in order to find the combination which gives the best results. To be more precise, we combined these different values for the hyperparameters:

- **Batch size**: we tried a batch size of `10`, `100` and `1000` both for the training and the testing phase;

- **Learning Rate**: we tried with learning rates equal to `0.01`, `0.05`, `0.1` and `0.5`;
- **Number of epochs**: at first, we tried different number of epochs; however, due to the long time the models with higher epochs were taking, we decided to use only `40` epochs and tweak the patience parameter instead;
- **Patience**: we tried with a variable patience between `5` and `15`;
- **Activation threshold**: we tried with an activation threshold of `0.2` and `0.4`, sometimes with some adjustments such as `0.25` or `0.35`;
- **Weight decay**: we tried a weight decay equal to `0.01`, `0.1` and `0.5`.

With different hyperparameters we got, as expected, different results: this was especially true for the activation threshold and the weight decay.

For the former, as expected we found that the higher was the threshold, the higher was the precision metric but the lower was the recall metric; this is due to the fact that with a higher threshold the results were more accurate but, being the dataset unbalanced, more biased towards the most frequent classes, while with a lower threshold the model tended to predict more labels but with less precision.

For the latter we can make the same considerations, but in the other sense: the lower was the decay, the higher was the precision because the model was more biased towards the most common classes, while the higher was the decay the less overfitted the model was, and therefore the higher was the recall.

However, after this automated running we tried to change single hyperparameters in order to achieve better results. The best results we got on the test set (part of the set we had for training which has not been used during training itself) are reported below; the results for the proper training set are reported in file `final_results.txt` as one-hot-encoded values with the following formatting:

```
image_name class1 class2 ... class14
```

5.3 Results on the training set

Talking about models, we got overall best results with the first CNN described in §3. However, here are reported also some good results from the other networks. The legend to read the tables below is the following:

Parameter	Abbreviation
Batch size	**bs**
Learning rate	**lr**
Number of epochs	**e**
Patience	**p**
Activation threshold	**tr**
Weight decay	**wd**
Data augmentation	**da**
Dropout	**dr**

Table 2: Legend for the hyperparameters in the results section.

5.3.1 First CNN

bs	lr	e	p	tr	wd	da	dr	Prec	Rec	F1
1000	0.1	40	5	0.2	0.5	Yes	No	0.3542	0.5357	0.4263
1000	0.1	40	5	0.3	0.1	Yes	No	0.3398	0.6593	0.4483
100	0.1	40	5	0.4	0.1	No	No	0.3528	0.5180	0.4182
100	0.5	40	4	0.2	0.1	Yes	No	0.3365	0.5620	0.4192
100	0.5	40	5	0.2	0.1	Yes	Yes	0.3298	0.6661	0.4399

Table 3: Best results with the first CNN.

5.4 Second CNN

bs	lr	e	p	tr	wd	da	dr	Prec	Rec	F1
1000	0.1	40	6	0.2	0.1	No	No	0.3356	0.6565	0.4439
1000	0.1	40	5	0.3	0.5	No	No	0.3573	0.4893	0.4130

Table 4: Best results with the second CNN.

5.5 Third CNN

bs	lr	e	p	tr	wd	da	dr	Prec	Rec	F1
1000	0.1	40	5	0.3	0.1	Yes	Yes	0.3561	0.5233	0.4235

Table 5: Best results with the third CNN.

6 Conclusion and error analysis

In conclusion, from the data we have collected we can state the the first CNN works better than the others, especially than the third one. Therefore, from this we can infer that for this data and this type of problem, a simpler CNN with less convolutional layers and with a limited number of neurons in the FFNN is more appropriate than a more complex one. However, a too small network as the second is, is not appropriate either.

The explanation we gave ourselves, and we are therefore proposing, relies on how Convolutional Neural Networks work. Each layer has, in fact, the task of extracting a certain type of information from the image: the first layer extracts low-level information, while the last extracts higher-level features. Adding more layers, sometimes, enables the recognition of more interesting and useful features, but this is true only as long as these features are present. Evidently, with the data of this problem this is not entirely true: the consequence of this is, then, that more layers actually learn only non-relevant features, bringing the model to be more overfitted and excessively focused on these non-relevant features.
Regarding the fact that the second CNN gives worse results than the first, the explanation could be similar: while a too big network would likely overfit, a too small one would probably underfit, and therefore the results are less precise.

From the results, moreover, we can see that the two parameters don't change usually:

- Patience is basically always equal to 5, except for two cases where it was 4 or 6. This means, as we also verified manually, that the loss on the validation set is not increasing too often after having reached the minimum;
- Dropout seems to decrease the performance, except for the biggest model we had. This probably means that the first and the second models are not too complex, thus dropout is not needed.

We can also see that some parameters especially are crucial to the results, since different combinations of them influences the performance of the model; this is particularly true for learning rate, activation threshold and data augmentation:

- The learning rate is highly dependent on the batch size. This is as expected, since a smaller batch size means that the optimizer is approaching slower to an optimal solution; therefore, the learning rate can be higher because the two balance each other;
- The activation threshold is the main responsible for the precision/recall trade-off: as already explained in this document, a higher threshold means more correct yet less results, and vice versa. As already mentioned, we very much played with this parameter, since we had to find the *perfect spot* which gives reasonable precision and recall scores;
- Data augmentation is also crucial as this is, as already explained, a useful technique to avoid overfitting.

Finally, we can say that the results we got are probably not the best that could be achieved, yet are quite good: even if a precision equals to ~ 0.35 may seems quite low, we also have to consider that we are dealing with multiple labels, and the same holds for recall, therefore we can be satisfied.

References

[1] Neha Sharma, Vibhor Jain, Anju Mishra, An Analysis Of Convolutional Neural Networks For Image Classification, Procedia Computer Science, Volume 132, 2018, Pages 377-384, https://doi.org/10.1016/j.procs.2018.05.198.

[2] Sigmoid Activation and Binary Cross entropy — A Less Than Perfect Match?, retrieved from https://towardsdatascience.com/sigmoid-activation-and-binary-crossentropy-a-less-than-perfect-match-b801e130e31.

[3] Binary Cross Entropy/Log Loss for Binary Classification, retrieved from https://www.analyticsvidhya.com/blog/2021/03/binary-cross-entropy-log-loss-for-binary-classification/.

[4] Writing Custom Datasets, DataLoaders and Transforms, retrieved from https://pytorch.org/tutorials/beginner/data_loading_tutorial.html.

[5] Preprocessing data, retrieved from https://scikit-learn.org/stable/modules/preprocessing.html.