Object recognition in the wild using Convolutional Neural Networks

Practical Work 05

Francesco Monti

19.06.2022

Contents

Introduction

The goal of this project is to create an application that can recognize different age groups and units of a swiss Scout Brigade. The *Brigade des Flambeaux de l'Évangile* [1] is a cluster of 15 groups located in the french-speaking part of Switzerland. There are currently around 900 active members. The images that we will use are publicly available images to comply to the permissions given by the parents regarding the usage of those images. We will need to collect at least 700 images (100 for each possible outcome), but depending on how far we will take the classification we could need 1000 images at least. We will use an existing model, namely `MobileNetV2`, provided by the **Keras** package. We will probably need to do some data augmentation as we are not sure to be able to get 1000 usable images. To help ourselves with the dat preparation and the creation of the model, we will use the comprehensive guide on the **Keras** website [2].

The problem

This problem will be a *Multi-Label Classification* problem because we have the following classes:

Uniform color (age group)

- `[0]` **Blue**: *Petits-Flambeaux (M) / Petites-Flammes (F)* -> Participants
- `[1]` **Beige**: *Flambeaux (M) / Claires-Flammes (F)* -> Participants
- `[2]` **Red**: *Pionniers (M) / Cordées (F)* -> Participants
- `[3]` **Green**: *Responsables (M/F)* -> Leaders

Neckerchief color (gender)

- `[0]` **Blue and orange**: Male participants
- `[1]` **Green and yellow**: Female participants
- `[2]` **Full orange**: Male and female leaders

The total outcomes can be represented as such (first digit is age, second is gender) : `(0,0)`, `(0,1)`, `(1,0)`, `(1,1)`, `(2,0)`, `(2,1)`, `(3,2)` -> 7 possible outcomes.

Depending on how much images we find that have both the shirt and the neckerchief, we could backup to a *Single-Label Classification*. It's easier to find images with neckerchiefs rather than with the uniform (mainly because of the hot temperatures in the summer, when we take the most pictures).

[1] https://www.flambeaux.ch
[2] https://keras.io/guides/transfer_learning/

Data preparation

After collecting the needed images, and after filtering we end up with 122 total images. Which is really not a lot. We started from approximately 1000 original pictures. A lot were discarded because they didn't contain the wanted subjects (landscapes for instance), were too complex or just not tailored to our needs. We will make use of Data Augmentation to add more training data to the model.

Repartition

Here is the repartition of the different classes:

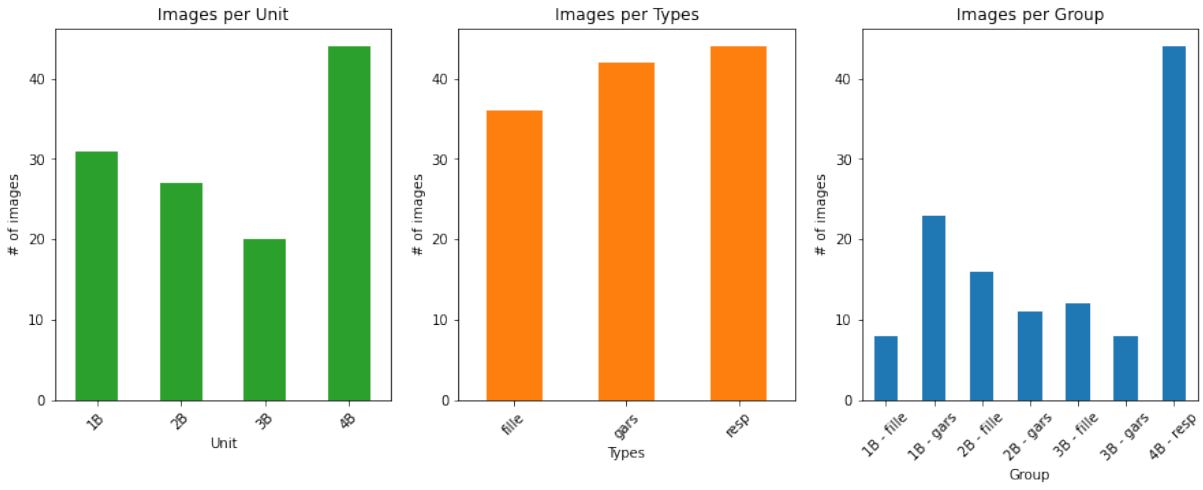


Figure 1: Number of images per class

We can see that if the take the subdivision in the 7 possible classes we have a large number of images (a third) that is only for the `[4B - resp]` class. This is due to the fact the the subclass `[4B]` is only associated to the `[resp]` subclass. The numerical values of the plots above are the following:

```
Unit:
1B    31
2B    27
3B    20
4B    44
```

```
Types:
fille    36
gars     42
resp     44
```

```
Group:
1B - fille    8
1B - gars    23
2B - fille   16
2B - gars    11
3B - fille   12
3B - gars     8
4B - resp    44
```

As the images repartition is not ideal, we need to find a little more images. After looking through 1400 more pictures, we found 55 more usable pictures. It is not a lot, but the pictures were older and the a lot did not contain the wanted subjects. Nonetheless, we have a pretty much balanced dataset of 167 pictures now. It should be enough data to train our model. Here is the summary of the new dataset:

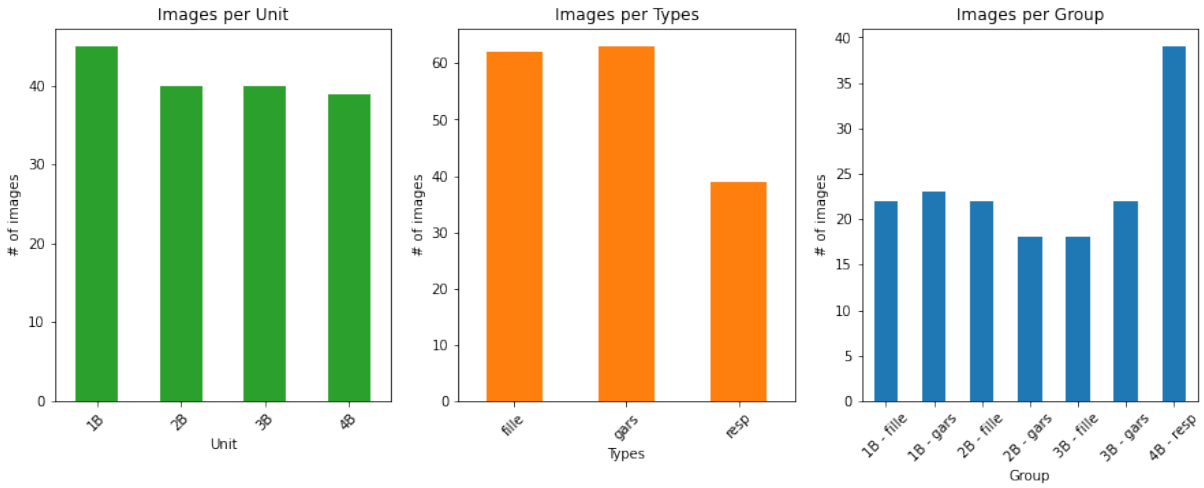


Figure 2: Number of images per class

The numerical values of the plots above are the following:

Unit:	
1B	49
2B	40
3B	40
4B	38

Types:	
fille	62
gars	67
resp	38

Group:	
1B - fille	22
1B - gars	27
2B - fille	22
2B - gars	18
3B - fille	18
3B - gars	22
4B - resp	38

Here are a few samples of the dataset :

Figure 3: Dataset samples

With this dataset we will now proceed to the data processing and data augmentation[3] steps.For the preprocessing part, it is a simple rescaling of $\frac{1}{255}$ to bring the pixel values from the range $[0, 255]$ to $[0, 1]$, the images are then rescaled to fit the 244×244 input of our base model. The images are not cropped to ration in fear of losing some features, we do not care a lot about the ratio of the image as

[3]Source: https://pyimagesearch.com/2019/07/08/keras-imagedatagenerator-and-data-augmentation/

it won't affect the model so much.

For the data augmentation we settled on a simple rotation of +/- 15%, a random horizontal flip and a random translation of +/- 10%. As we do not have a lot of images we will multiply the number of images by some factor to have more data to train on. Only the training data will be augmented, we do not want to validate our model on images that change all the time.

Further processing of the images

After some time testing a lot of different models, we came to the realization that our dataset was not good enough to train on. The images contained a lot of useless data, such as backgrounds or faces that would influence the model and prevent it from learning. The images have been reprocessed and cropped much more closely to the main "subjects" namely the shirt and neckerchief. After this stage the images should be more useful for the model and the results better.

For the sake of comparison, after a week and around 100 different models, the best accuracy was 60% and the average was around 25%. The average loss was around 5. Here are some examples of the new dataset:

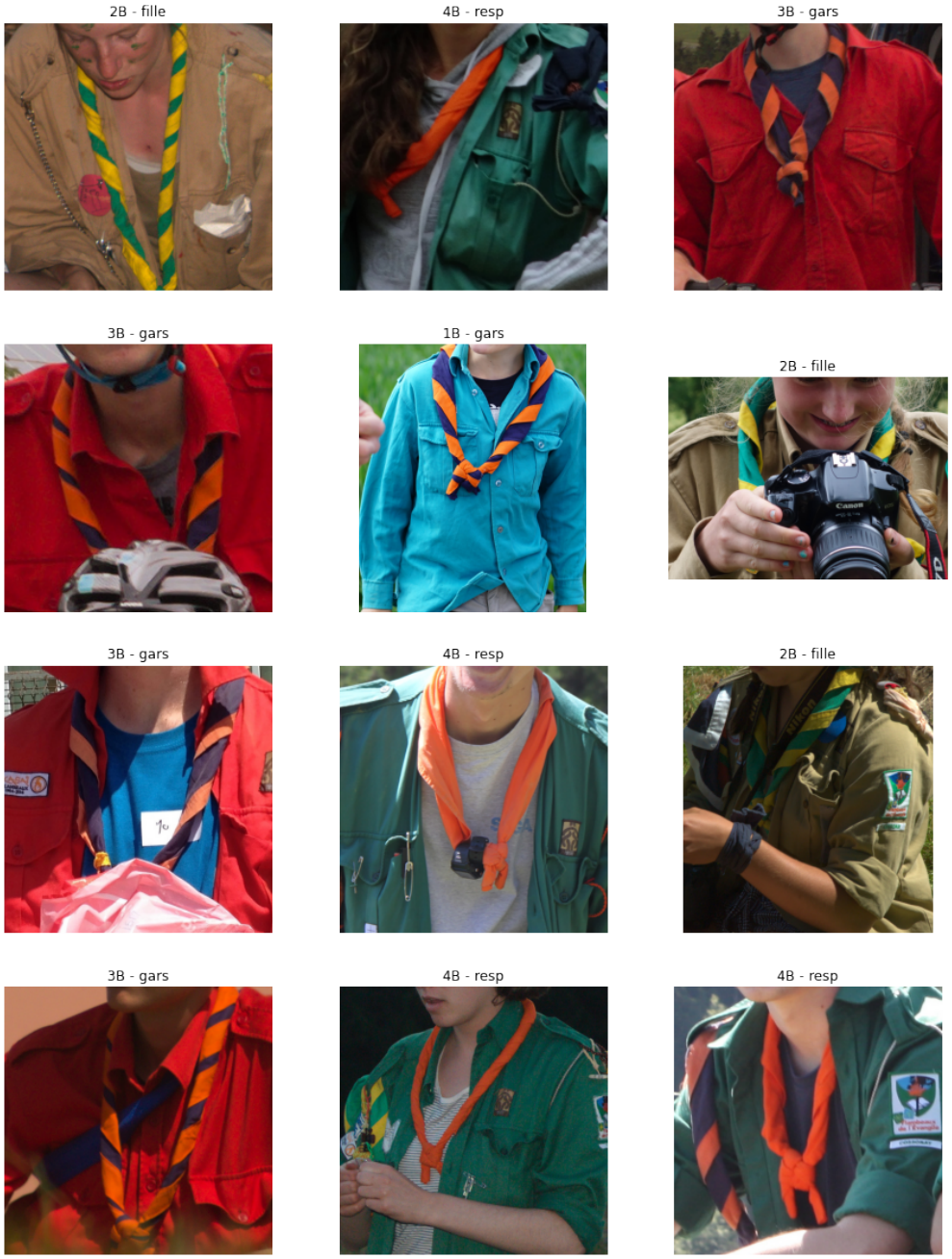
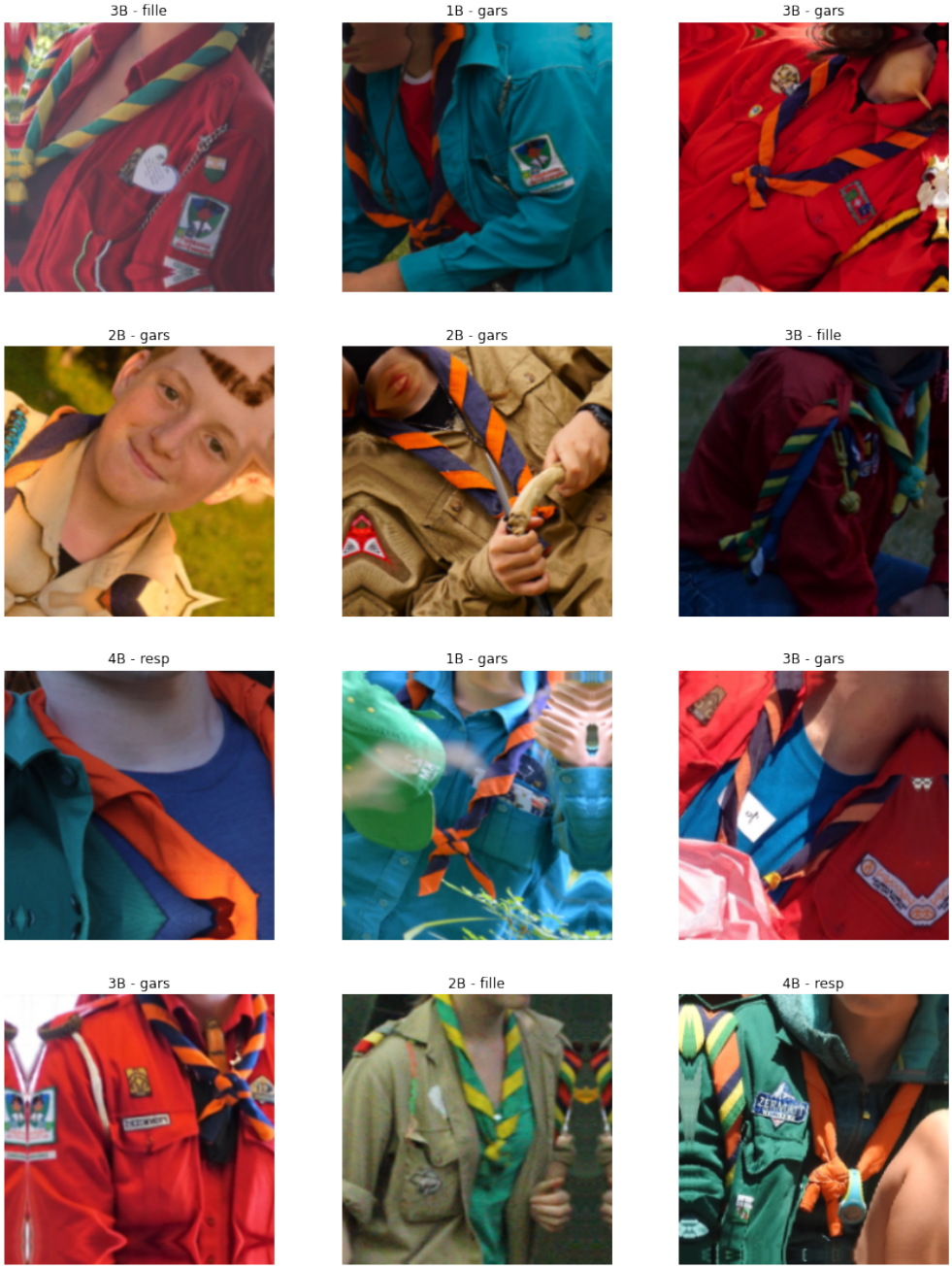


Figure 4: Cropped dataset samples

Here are some examples of the augmented images, ready to be fed to the model for training:



Datasets

The global dataset has been manually split between a train dataset of 140 images and a test dataset of 27 images. We will augment the training dataset up to 10x to have approximately 1400 training images. Pushing the augmentation rate further did not seem to improve the training.

The train dataset will be further split in a train and a validation datasets with a 0.25 ratio.

Model creation

As our problem is a multi-label classification problem, we had to change the given notebook. Based on an article[4] on the Internet, we tried to implement a single output multi-label classifier with Keras. We used a `MultiLabelBinarizer` to encode the labels with an "two-hot" encoding (one for each label) and a `BinaryCrossentropy` loss function. The activation function was changed to a *sigmoid* as we needed more than one output. The results were very disappointing as neither the loss nor the accuracy were good, with hyperparameters ranging from `Dense` layer with 16 neurons to 4 `Dense` layers with 4096 neurons each. The `Dropout` layers between the `Dense` layers had a rate of 0.1 up to 0.6.

After those failures we tried a different approach with a multiple output model. The reasoning behind the change of architecture was that having only one output for two different labels made the learning of the model very difficult, because if one label was right but the second wrong, the model would register that as a complete failure and not only a label failure.

Having multiple outputs provided us with more flexibility in the fully-connected layers and helped the model learn better. We finally settled on a model that would branch right at the output of the *ImageNet* model and would have one or more `Dense` layers before the final layer, activated by the classic *softmax* function.

Hyperparameters

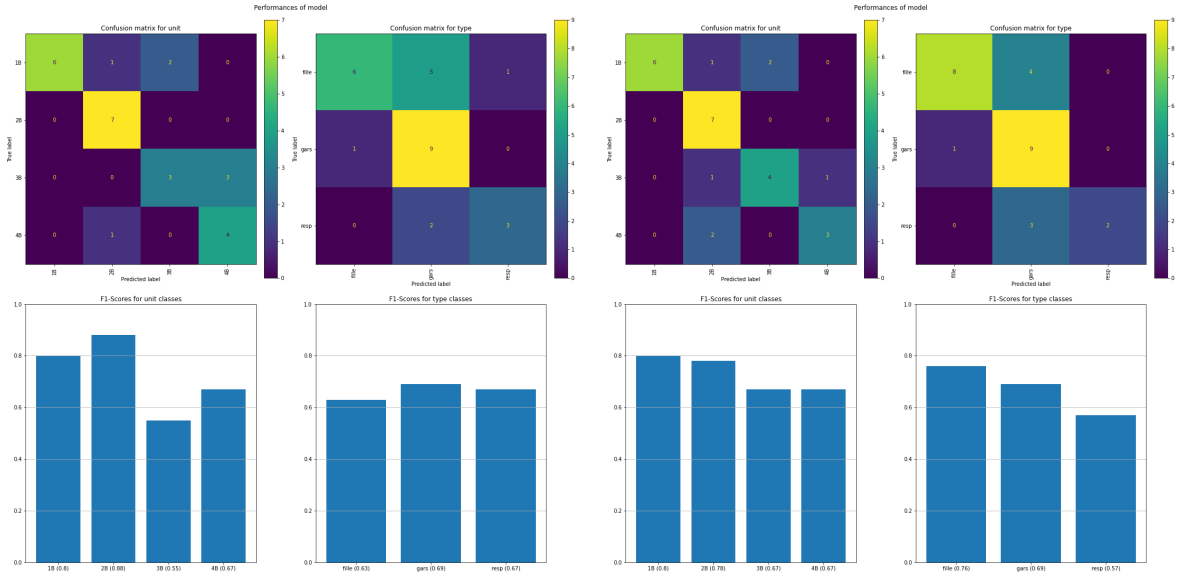
We explored a variety of "global" hyperparameters like the optimizer, the number of epochs, loss function and the augmentation rate. The values tested were as follow:

Optimizer

The choices we explored were *RMSprop* and *Adam*, we settled on *Adam* as we did not see a real improved of *RMSprop* over it. Here is an example of the many tests we made:

[4] https://pyimagesearch.com/2018/05/07/multi-label-classification-with-keras/

RMSprop (1 layer 96 neurons with 0.6 dropout) Adam (1 layer 96 neurons with 0.6 dropout)



Loss function

We started with a *BinaryCrossentropy* when we were using the single-output multi-label classifier but we switched to a *CategoricalCrossentropy* afterwards. We run into a little problem while trying to use a *SparseCategoricalCrossentropy* because we still had 2 labels. The workaround was to use the `MultiLabelBinarizer` for the model and a *CategoricalCrossentropy* for the loss function but to use two different `LabelEncoder` for the test dataset and the performance evaluation. It enabled us to plot two different confusion matrices for each label.

Epoch number

We tried several numbers of epochs while training our model, ranging from only 5 epochs to 50 epochs. After a bit we realized that after 10-15 epochs the models would not improve significantly and it was not worth the effort of computing 15 more. If a model would show signs of improvement after 15 epochs we would re-run it with more, but 15 epoch was enough for most models as they tended to overfit around 8-12 epochs anyway.

Augmentation rate

This hyperparameter was a little bit trickier to fine tune as some models performed better with fewer images than others. We tried a rate from 1 (no duplication of images) up to 20 (each image is augmented 20 times). The best results seemed to come from an augmentation rate of 10.

Model architecture

After a lot of hyperparameter exploration, we settled on a model with a `GlobalAveragePooling2D` on the output of the *ImageNet* base model, followed by two distinct branches composed of one `Dense` layer with 96 neurons and a *relu* activation function and a `Dropout` layer with a rate of 0.4. The unit branch ends with an output layer with 4 classes and a *softmax* activation function, the type branch is the same with only 3 classes instead of 4.

Here is a graphical visualization of the last layers of the model :

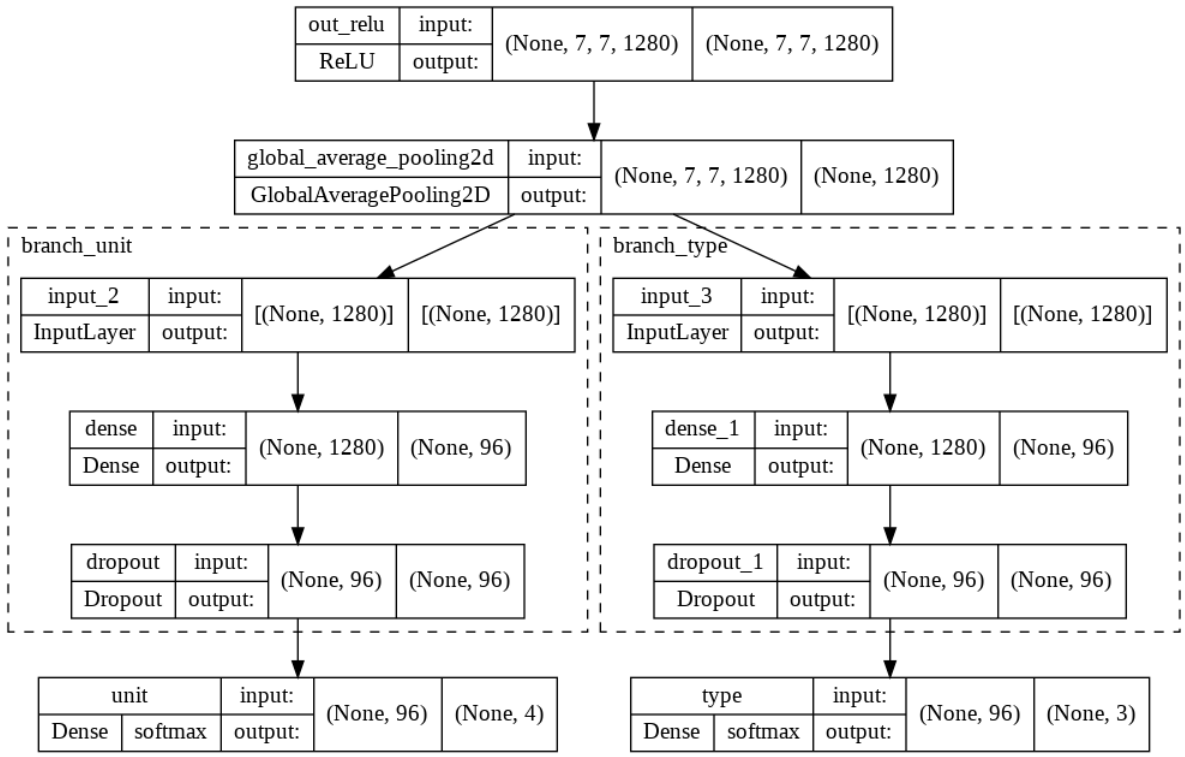


Figure 5: Final model

If we check the summary of the model we have this recap at the end :

```
Total params: 2,504,615
Trainable params: 246,631
Non-trainable params: 2,257,984
```

We can see that only the last layers are trainable and the total number of trainable parameters is `246'631`.

Transfer learning

We performed a transfer learning from the *ImageNet* model to help us classify our images. It is usually done for tasks where the dataset has too little data to train a full-scale model from scratch, as in our case. We could quickly train and validate a model with only a thousand images, where those other models had millions of images fed into them. We did not fine-tune our model as the time was running low, but it was the last part of the transfer learning process.

Results

Loss and accuracy

We let run our final model for 30 epochs to see were it would take us. We can see that our losses and accuracy are pretty decent,

As mentioned before, we can see a slight overfitting around 10 epochs, we will train our final model on 10 epochs only.

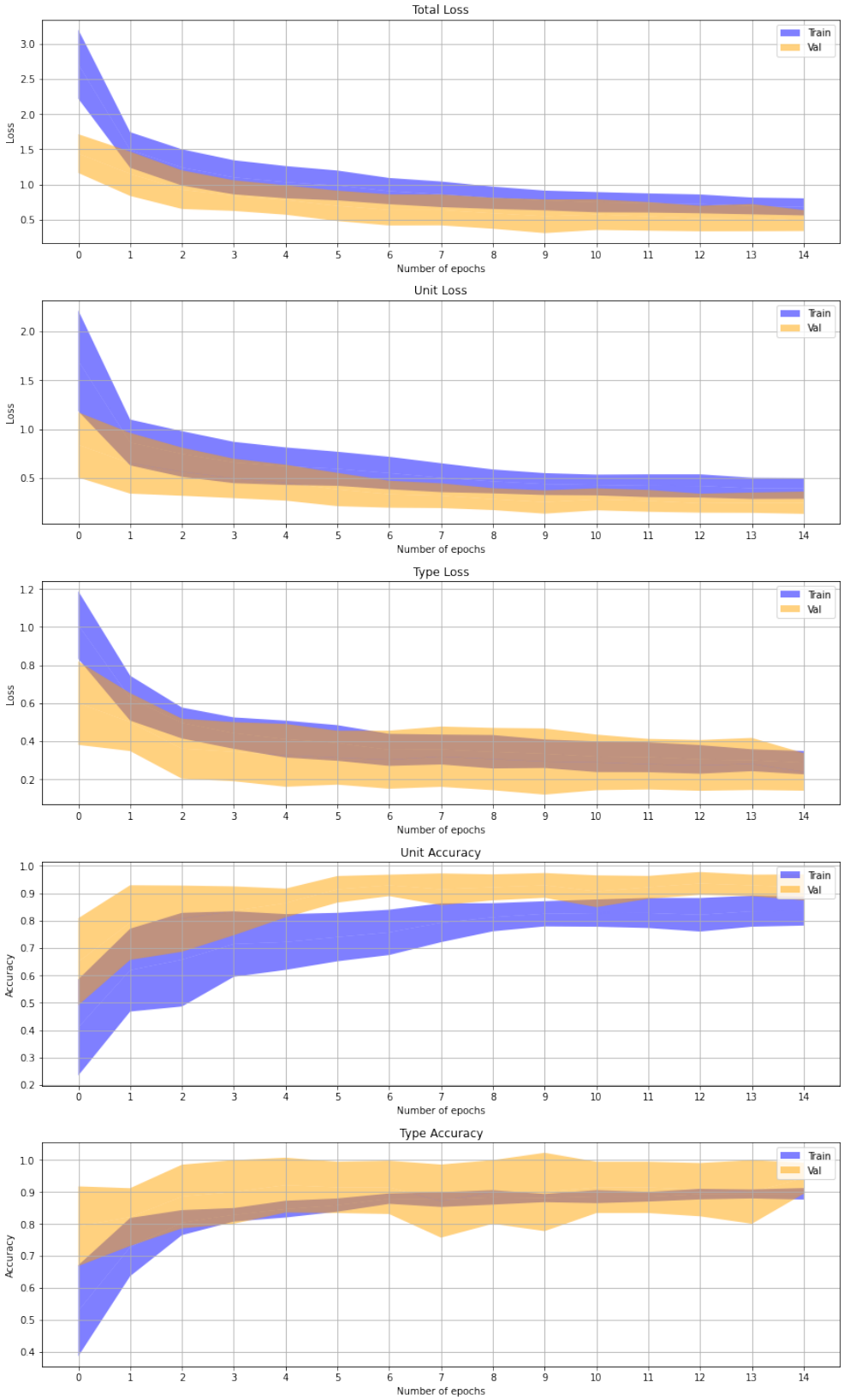


Figure 6: Final model losses and accuracies

Test set evaluation

Our test set is not very large (only 27 images) so the results can be a little bit exaggerated.

As we have two distinct labels we also have two sets of confusion matrices and f1-scores. We can see that our model performs pretty well, even if it's only on 27 images. We can observe that our model has a harder timing classifying the type rather than the units.

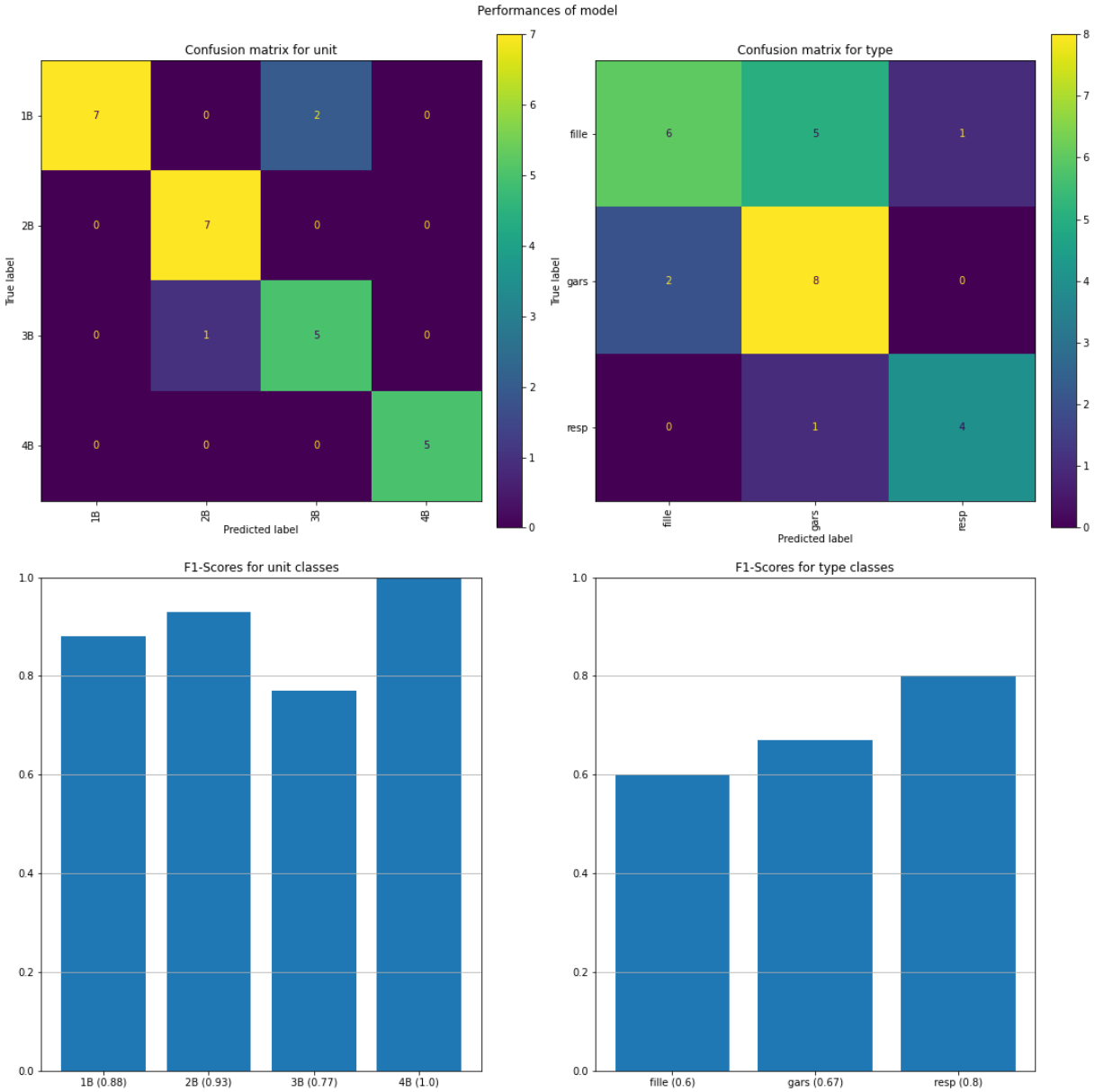


Figure 7: Final model performance

GradCAM++

We can observe that the model is not really looking at the shirts but seems to find other features in the image. Most of the time it looks at the neckerchief instead of the shirt.

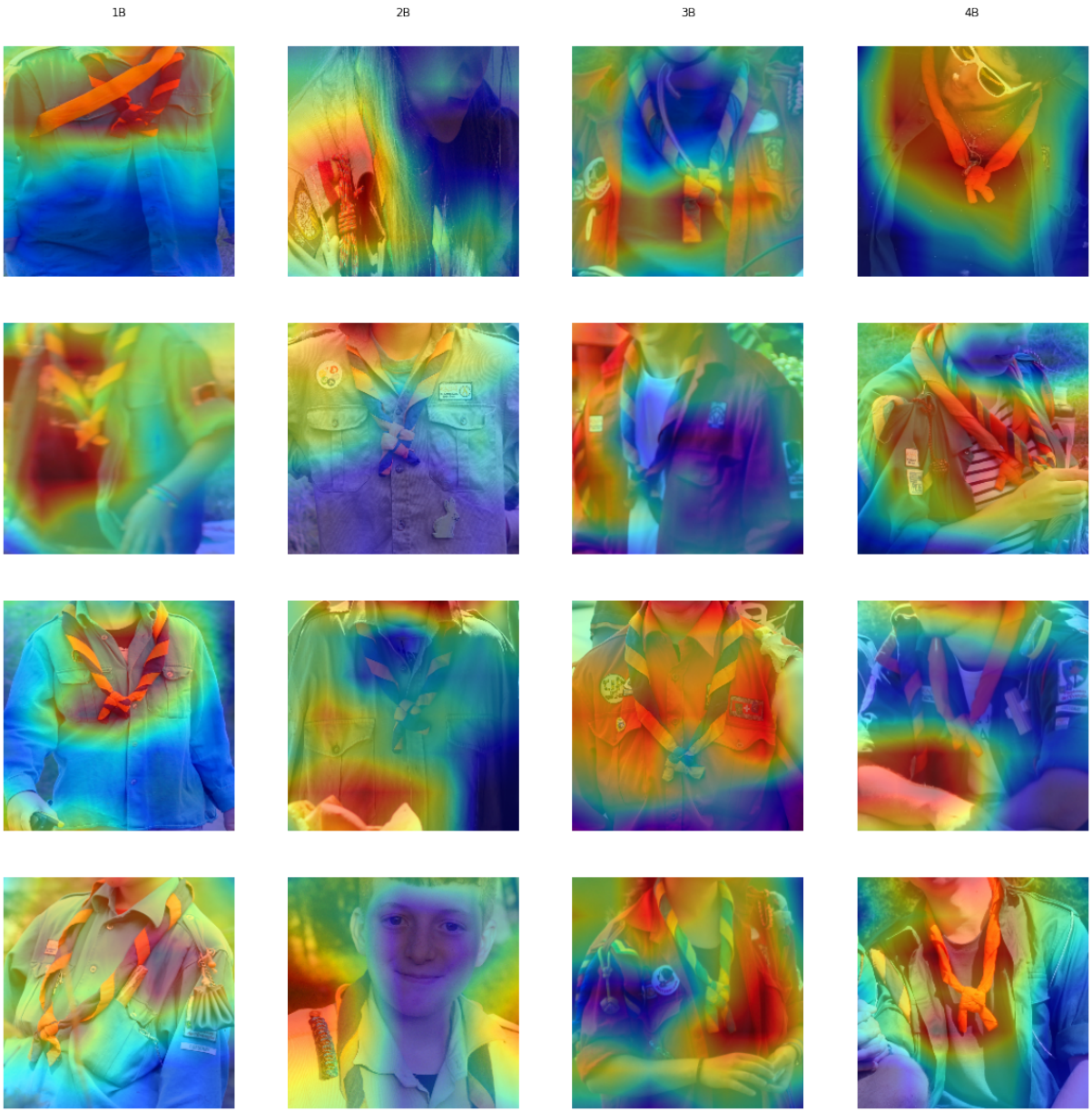


Figure 8: GradCAM++ on unit label

For the neckerchief the model seems to look at the right feature on the image, even if it is a little bit too focused on a point or looks a the wrong neckerchief.

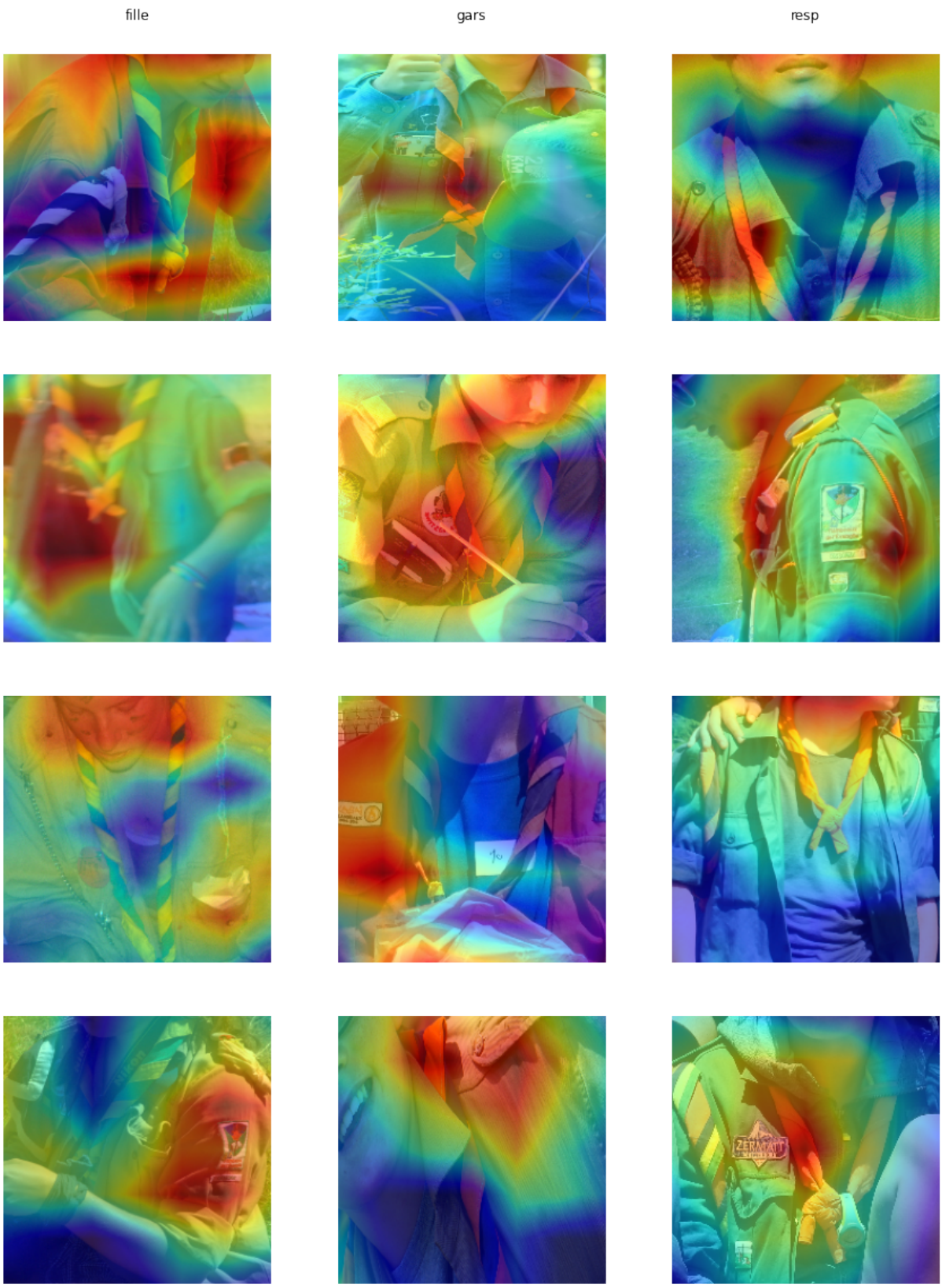


Figure 9: GradCAM++ on type label

Misclassified images

We can see that in the case of these misclassified images, the model did not look at the primary features but was either “distracted” by other features or was not able to recognize the right class. This can be explained by the similarities between the classes, where almost only the color can separate the different classes. We recon that it is a quite difficult task to rightfully recognize features that close.

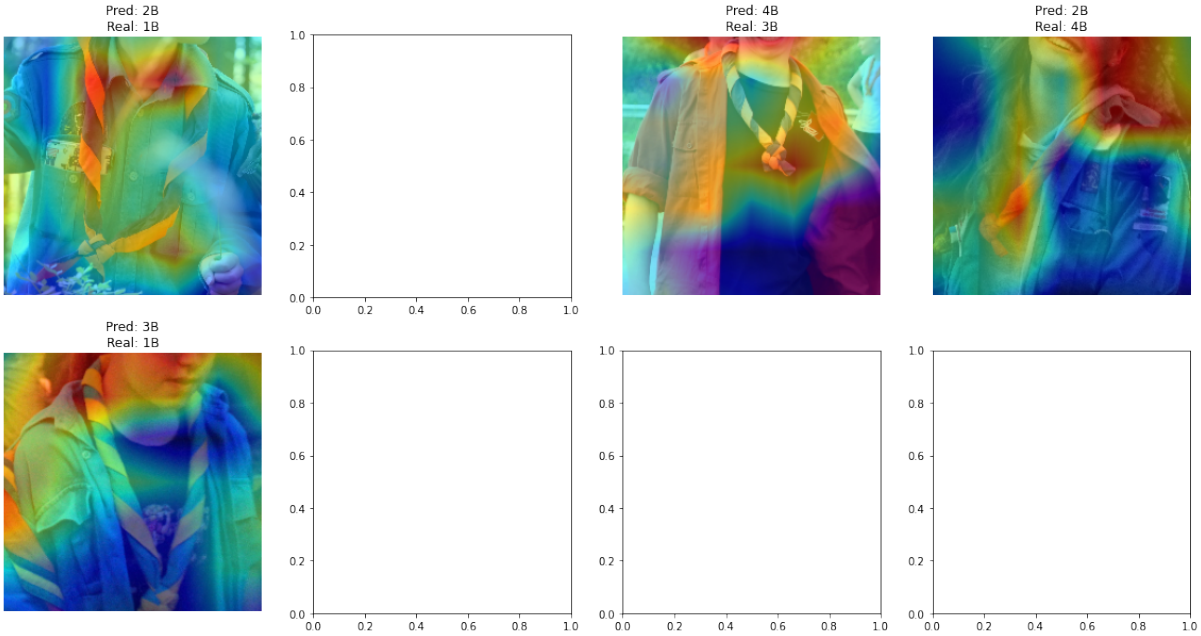


Figure 10: GradCAM++ on missclassified unit label

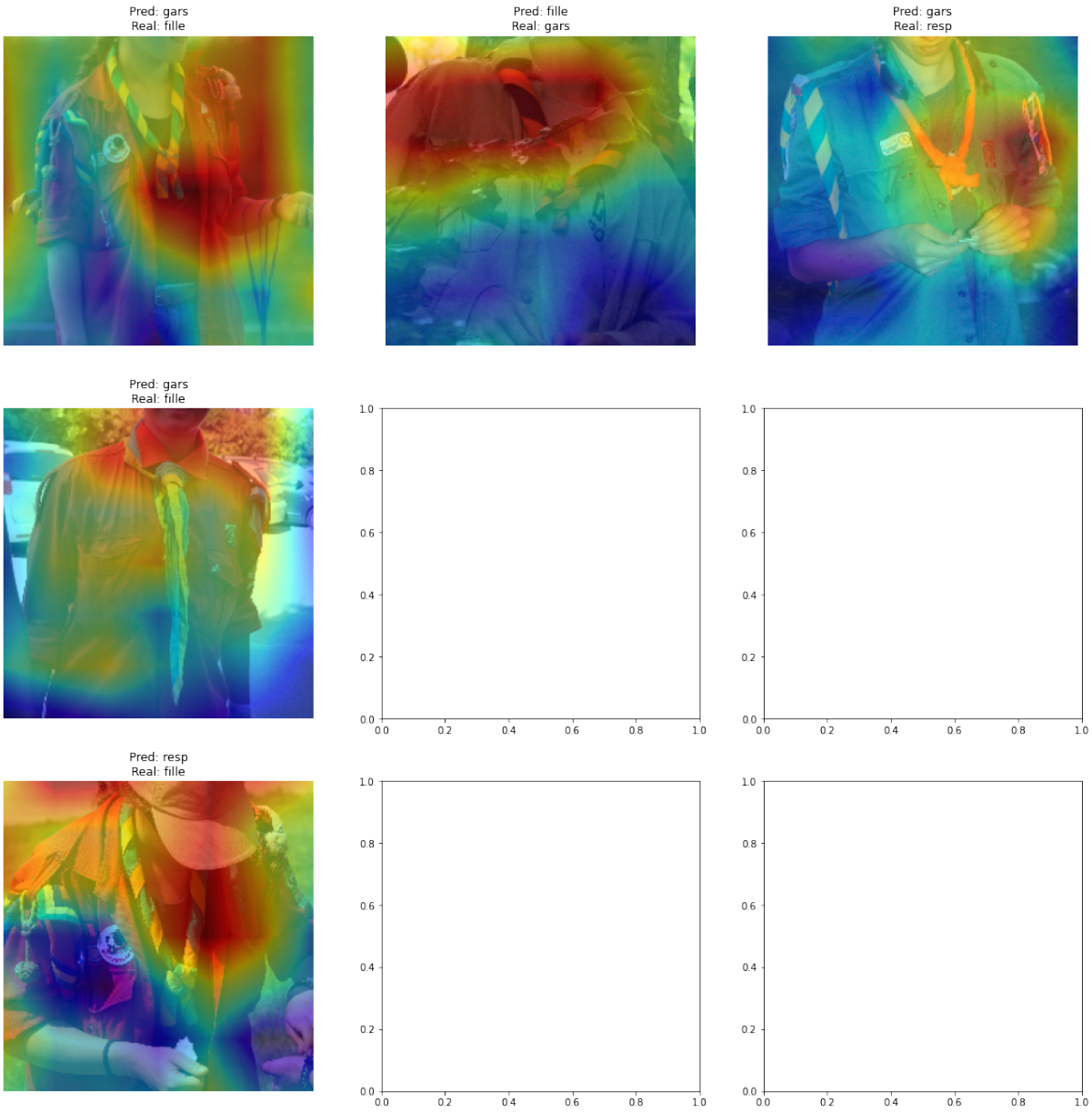


Figure 11: GradCAM++ on missclassified type label

Dataset improvement

The first problem we encountered was the too broad images, we needed to refocus our subject and zoom more on the features we wanted to classify. We resolved this problem in the dataset by cropping all the images to focus on the shirt and neckerchief. Another problem is the lack of images. Those images aren't easy to find on the Internet, we needed to pick them from specific databases, every time making sure we had the rights to use those images. If we had more time we could have sieved through more pictures to augment our base dataset.

Another option could come from the mobile app. We could ask the users to rate the prediction, giving the right answer in case of error from the model. We could the save the model prediction and the true labels and upload them to a server were we could continuously fine-tune the model and releasing new model weights every so often. In this case the more we would use our app, the more our model would learn. But it could be more complicated than that…

Confused classes

As said previously the classes are really close and it is not a surprise that the model confuses them. The most confused classes are the two *participants* type, namely *fille* and *gars*. The reason could be that the two neckerchiefs have a striped pattern that could easily confuse the model if it does not look at the colors.

Conclusion

All in all, the model was not so bad. One thing that we lack is the real life test. Unfortunately the provided mobile app does not support multiple outputs and the code snippet to export the model does not work either. It could have been fun to try on real occasion or other subjects to see how our model behaves.

The training of the model has been a difficult task, especially because we choose a multi-label classification and the given notebook was not crafted for this purpose. A lot of the time was “lost” on troubleshooting problems related to this challenge. Looking back we would not have chosen such a hard problem, but the motivation to complete such a personal project was high.

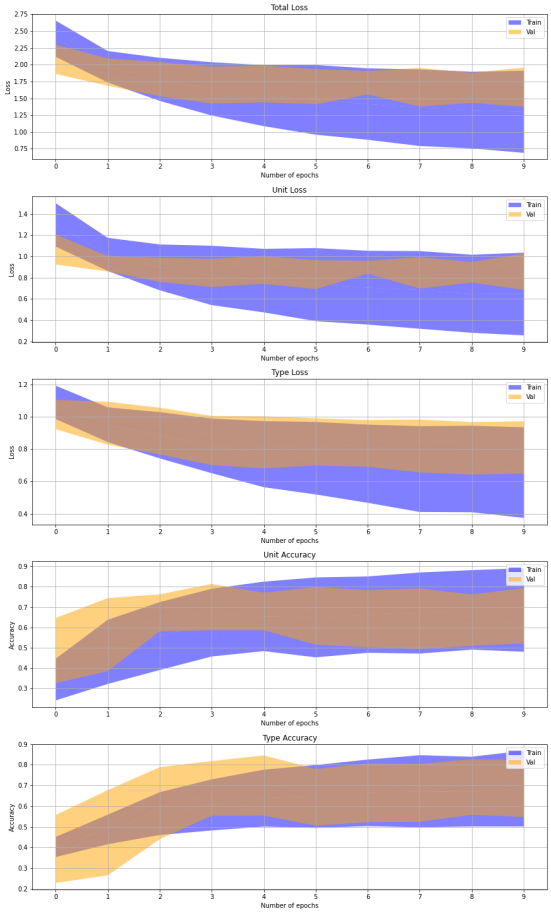
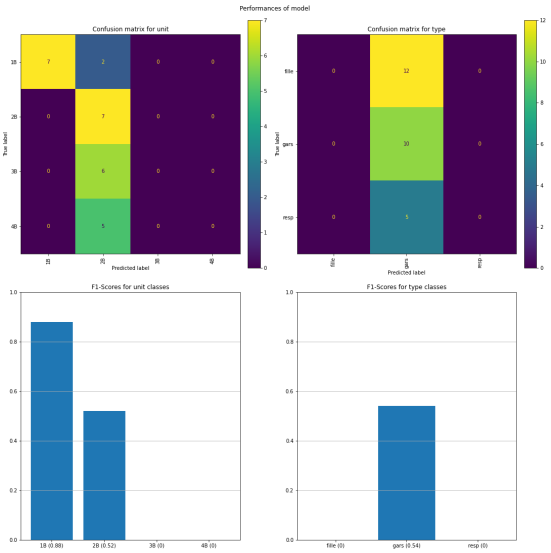
In the future, we would like to continue to work on this project and try to implement a mobile app to properly test the model in real-life situations. One further goal would be to implement a little server application to retrieve the predictions of the mobile app and continuously fine-tune the model to improve its performances and generality.

Annexes

Hyperparameters exploration

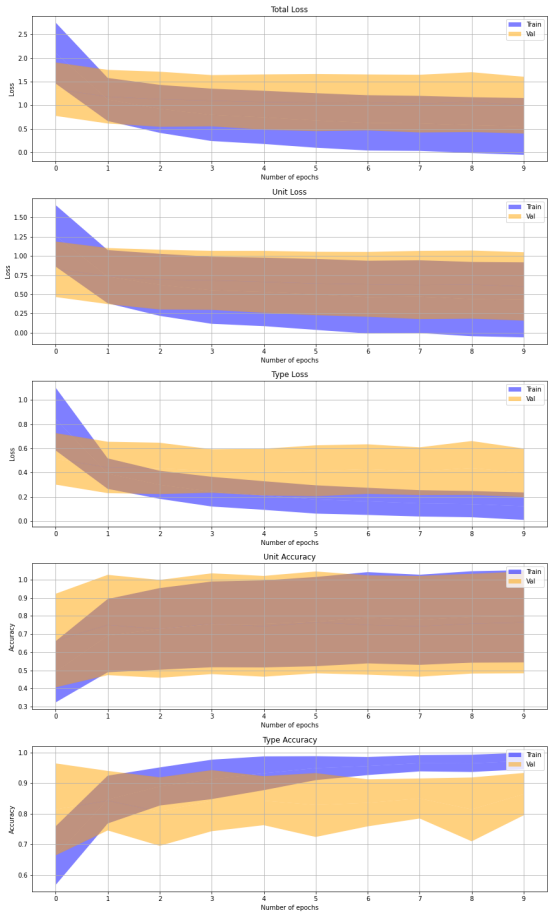
Here are some examples of hyperparameters exploration that we looked at during our training:

- **2 branches with 2 layers of 16 neurons and a dropout of 0.2**

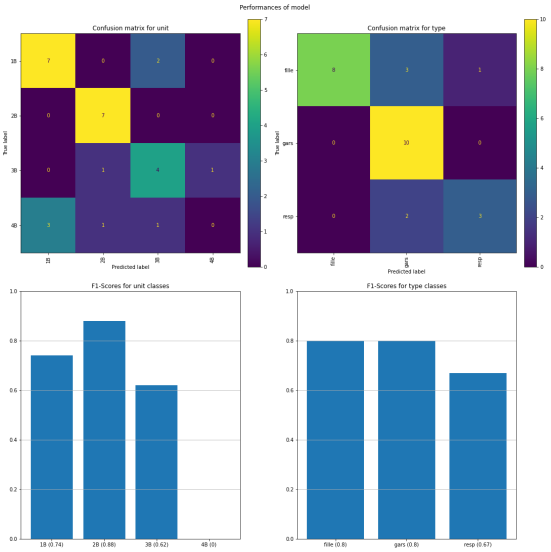
Losses and accuracy	Confusion matrices and f1-scores
	

- **2 branches with a layer of 32 neurons and a dropout of 0.25**

Losses and accuracy

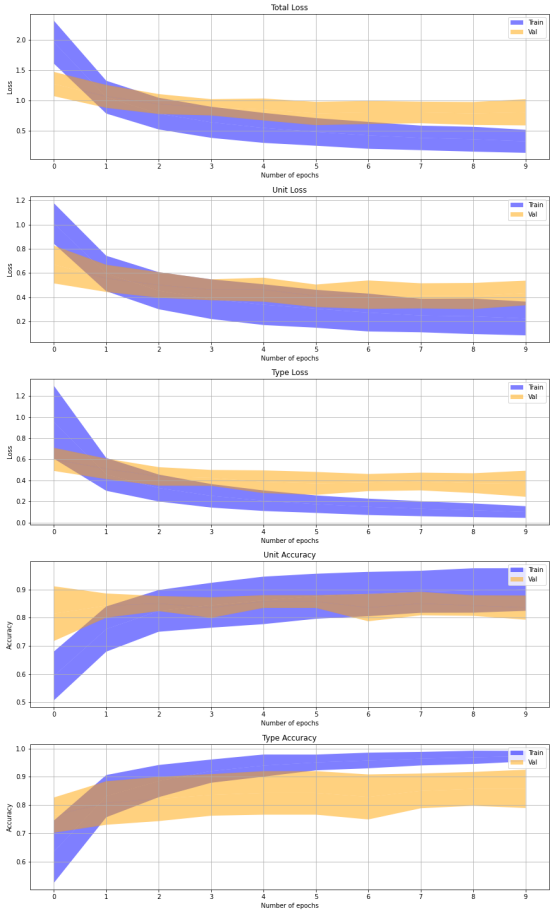


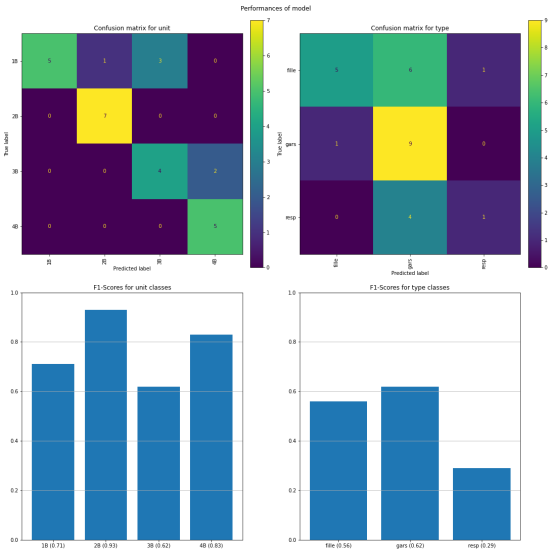
Confusion matrices and f1-scores



- **2 branches with a layer of 32 neurons and a dropout of 0.3 for the unit branch and 0.1 for the type branch**

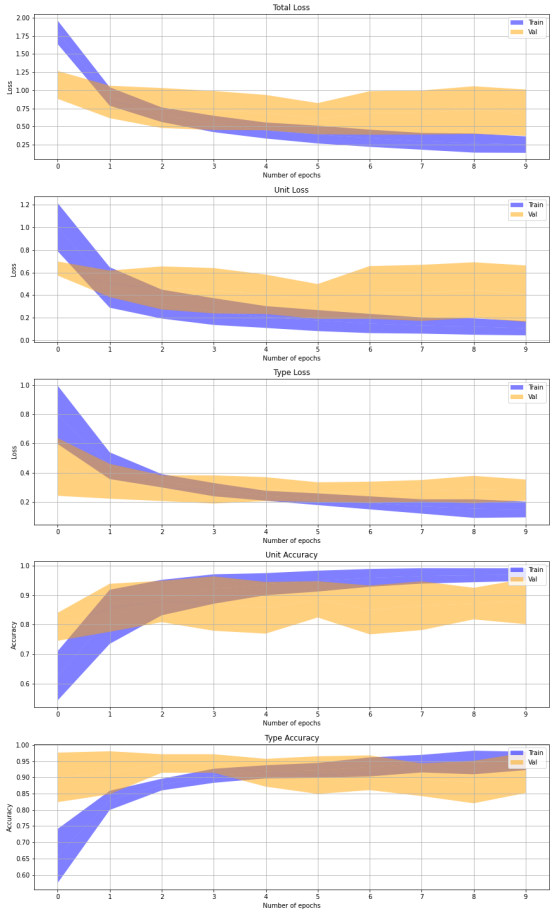
Losses and accuracy	Confusion matrices and f1-scores



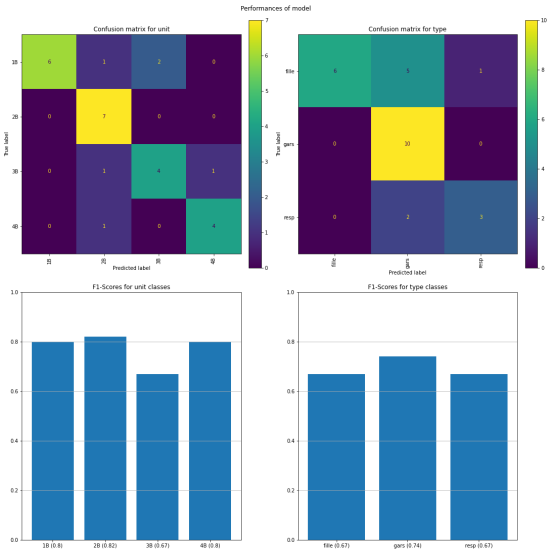


- **Unit branch with a layer of 64 neurons and a dropout of 0.3 and type branch with a layer of 32 neurons and a dropout of 0.3**

Losses and accuracy

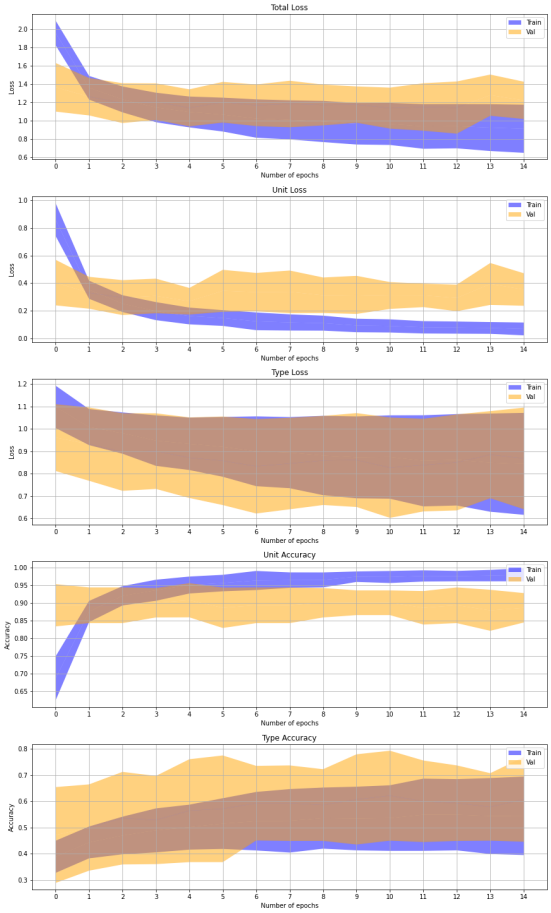


Confusion matrices and f1-scores

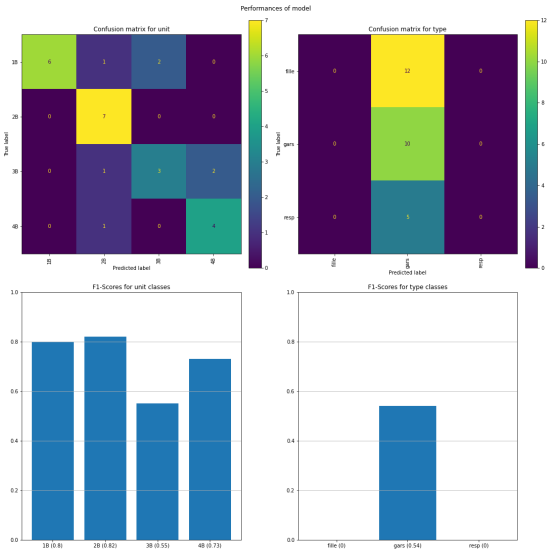


- **Unit branch with a layer of 128 neurons and a dropout of 0.6 and type branch with 2 layers of 32 neurons and a dropout of 0.6 and 0.4**

Losses and accuracy

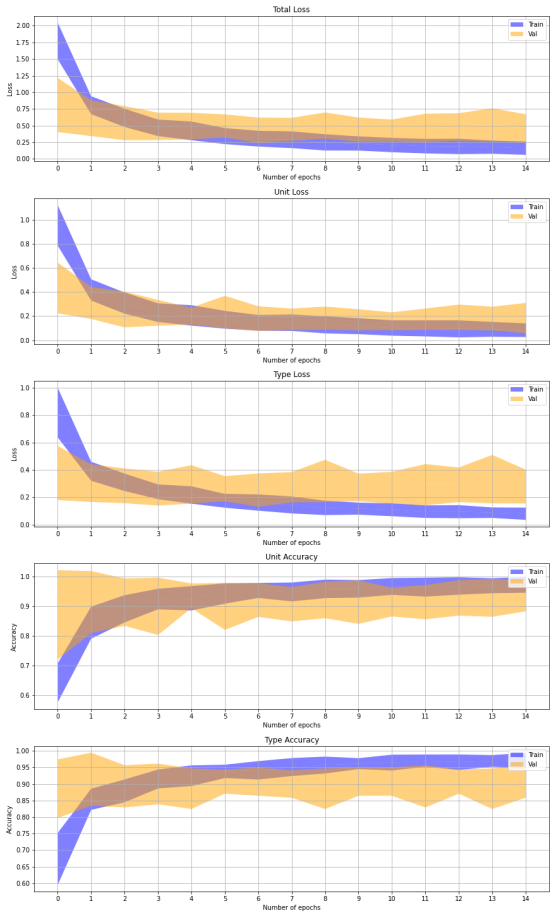


Confusion matrices and f1-scores

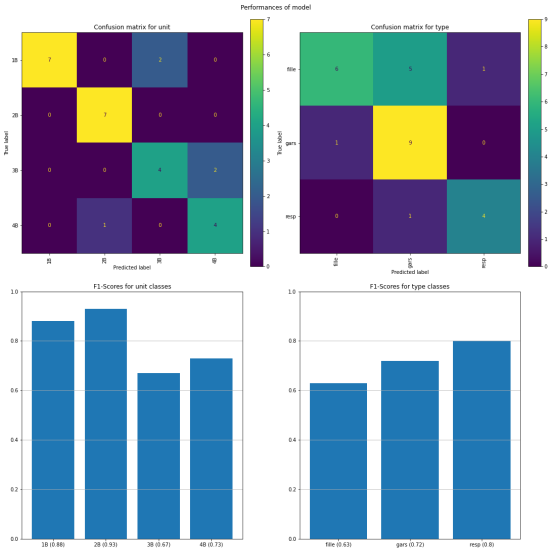


- **2 branches with a layer of 128 neurons and a dropout of 0.6**

Losses and accuracy

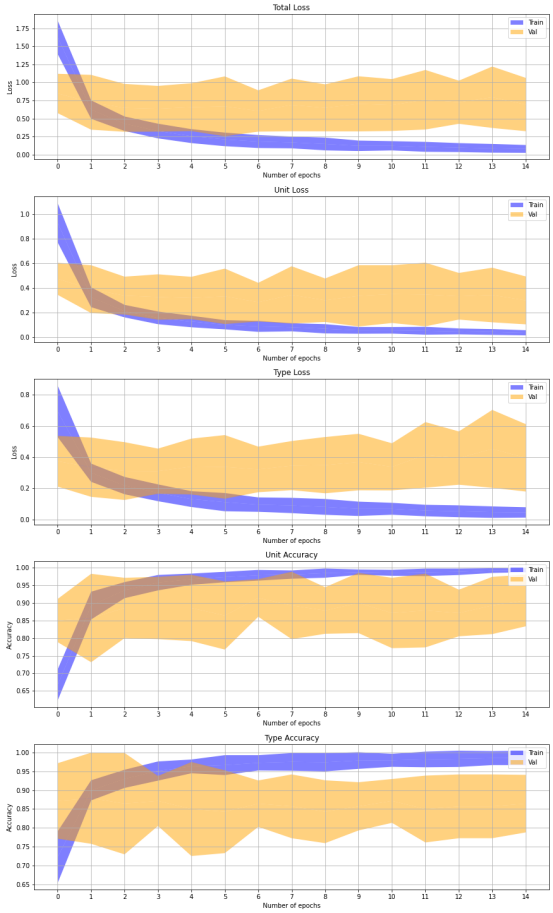


Confusion matrices and f1-scores

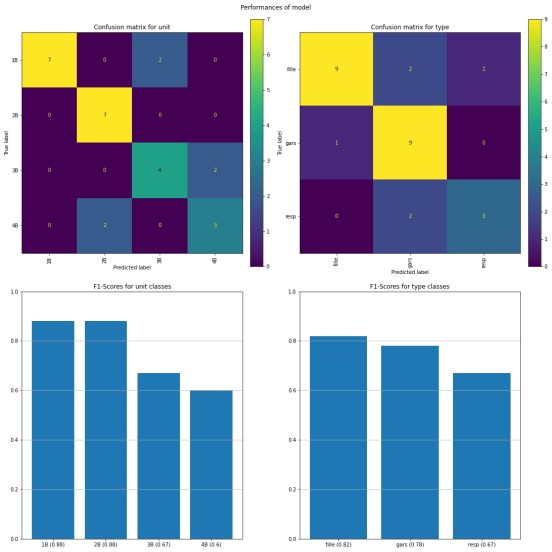


- **2 branches with a layer of 96 neurons and a dropout of 0.5**

Losses and accuracy



Confusion matrices and f1-scores



- **Unit branch with a layer of 96 neurons and a dropout of 0.4 and type branch with 2 layers of 32 neurons and a dropout of 0.1**

Losses and accuracy	Confusion matrices and f1-scores