AN2DL Homework 1 – Report

Filippo Desantis – 10677781 (HPC)

Liv Giacomin –10709077 (HPC)

Kamil Hanna – 10932895 (HPC)

18 November 2023

Abstract

The problem we were presented with was to train a network on how to classify pictures of leaves into healthy and unhealthy. We approached several different techniques, first cleaning the dataset then training with different architectures.

1. Data Loading, Cleaning and Balancing

We were given a dataset made from 5200 RGB images of size 96x96, with labels Healthy or Unhealthy. First, we imported the pictures and converted the labels into 0 and 1 format, with 0 as healthy. We then proceeded to scan through the pictures to see if there were any anomalies. Sure enough, we found that there were outliers, specifically two specific images (Fig. 1) that were each found 98 times in random positions. To eliminate these, we manually found one instance of each in the dataset and then we used the mean squared distance to find first the nearest 100 pictures, and when noticing there were just 98, we deleted from the dataset the 98 closest pictures to the manually found instances, making sure we zipped the labels correctly and deleted those too. So, we were left with a dataset of 5004 pictures: we noticed it was significantly unbalanced (3101 healthy and 1903 unhealthy) so we decided to balance it. We noticed that this increased the accuracy by a lot: not having enough unhealthy pictures caused us to overfit the model to healthy classes. We did this with the SMOTE function from the *imblearn* library, as can be seen in the notebook. Moreover, we applied some preprocessing to make sure all the pictures could be read normally, and we imported adequate preprocessing for the different models we were transferring, each time.

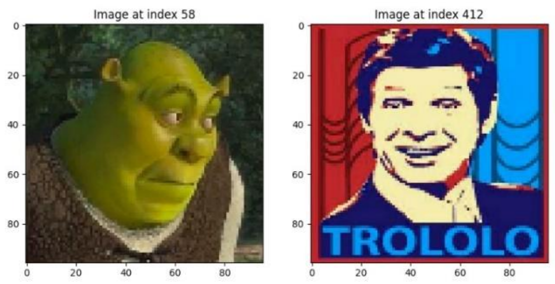


Figure 1

2. First attempts

The first attempts were with CNNs as seen in the labs, mostly from scratch. We implemented different combinations of feature extractions, max pooling, dense, augmentation and dropout layers. However, this didn't give us very good results: either it overfit sharply on our training set, or it didn't learn enough and didn't achieve any useful accuracy overall. Probably this was because the training set was too small, and either it learnt fast but did not extract relevant features or it tried to generalize but didn't have enough data to do so efficiently. The accuracy was also visibly noisy and oscillating, making us think that even the sporadic good results were just due to random guesses.

We tried to make this better by adjusting batch sizes, learning rate and stopping criteria, but unfortunately this didn't bring us a good result, especially on the hidden dataset on the competition evaluation. We quickly understood this approach wasn't going to take us anywhere. We also tried to treat the problem as an anomaly detection rather than a binary classification and tested out encoders/decoders, GAN's (BIGGAN), but we were unable to reach an accuracy higher than 0.76 on the leaderboard.

3. Transfer learning attempts

We therefore moved onto transfer learning. This quickly brought us to better results, achieving a 0.81 on CodaLab with just transfer learning using Efficient Net V2 B0. We chose this net because it performed very well with running times that weren't too long, which allowed us to experiment. This specific score was achieved without the SMOTE and with an exponential learning rate decay, so overall the model was very simple. At this point we were also experiencing difficulties, as our local GPUs weren't working very well with different TensorFlow versions (one member tried ROCm tensorflow to make good use of his AMD GPU; but it didn't work...), or different nets. We tried loading weights from different nets, such as MobileNet, ResNet and Xception, but EfficientNet gave us better results, specifically the version stated above. We then went on to fine tuning, trying to freeze only the first layers to use them as feature extractors, but in this case it made the results on CodaLab worse. So, we lost hope for a while, not really understanding what approach to take since nothing was doing better than our completely frozen transfer learning. Which brought us to focus on different aspects, which came together to form our final model.

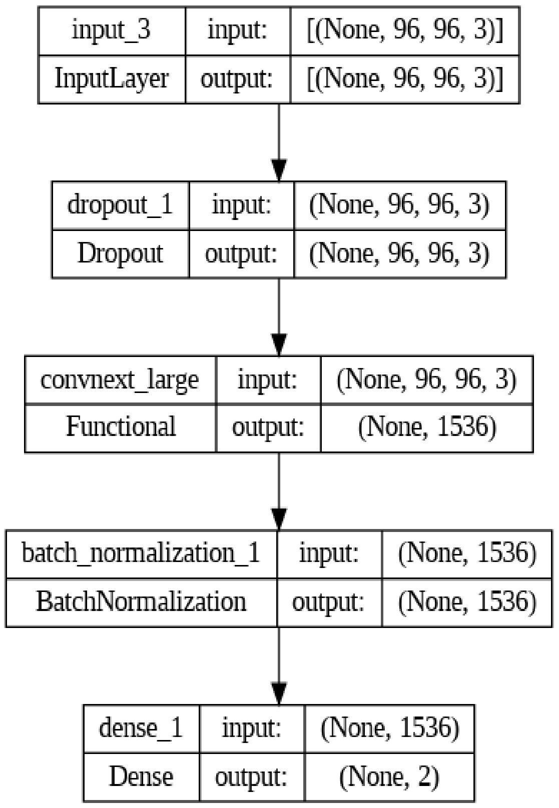
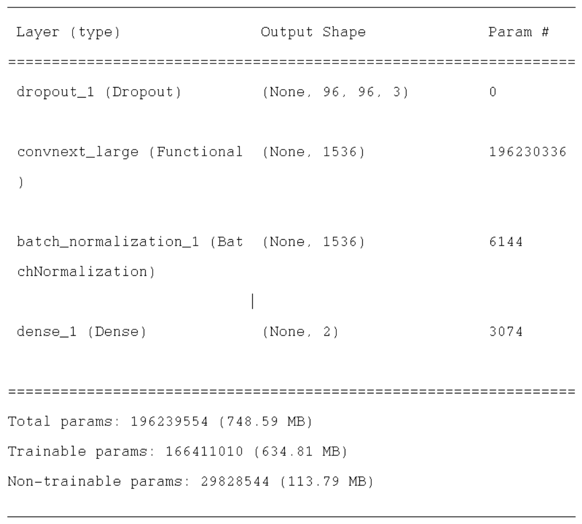

input_3 | input: | [(None, 96, 96, 3)]
InputLayer | output: | [(None, 96, 96, 3)]
dropout_1 | input: | (None, 96, 96, 3)
Dropout | output: | (None, 96, 96, 3)
convnext_large | input: | (None, 96, 96, 3)
Functional | output: | (None, 1536)
batch_normalization_1 | input: | (None, 1536)
BatchNormalization | output: | (None, 1536)
dense_1 | input: | (None, 1536)
Dense | output: | (None, 2)

Figure 2

4. Final model

1. ConvNeXt

In our final model, we managed to transfer the weights from ConvNeXt large, loading the 296 layers. We did this on Colab since due to compatibility issues we didn't manage to run it locally on GPU. We observed different outcomes freezing different numbers of layers and found that a good result was obtained when freezing the first 90 layers. By doing this we made the most of the feature extraction from the large net, and then trained it specifically to our kind of data.



```
Layer (type)                Output Shape              Param #
=================================================================
dropout_1 (Dropout)         (None, 96, 96, 3)         0

convnext_large (Functional  (None, 1536)              196230336
)

batch_normalization_1 (Bat  (None, 1536)              6144
chNormalization)
                           |
dense_1 (Dense)             (None, 2)                 3074

=================================================================
Total params: 196239554 (748.59 MB)
Trainable params: 166411010 (634.81 MB)
Non-trainable params: 29828544 (113.79 MB)
```

Figure 3

2. Cyclical learning rate

We tried with exponential decay, but it wasn't giving us good results, so after some research we found code[1] for a class that implemented a Cyclical Learning Rate, changing it each epoch and therefore being able to "pick up" the learning when it was slowing down too much. We tinkered a bit with the parameters and saw some improvements, so we left it in the model.

1 https://github.com/bckenstler/CLR/blob/master/clr_callback.py , more info in the notebook code

3. Dropout and batch normalization

We added a dropout layer to our model so that we avoided too much overfit and found that 0.1 and 0.2 were the best parameters to use since otherwise it wouldn't learn enough, seeing as the dataset was still quite small. We also added batch normalization, and the overall model can be seen here in Fig. 2 and 3, with the number of parameters and the different sizes.

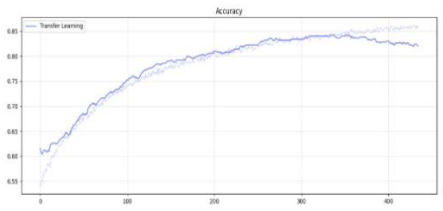


Figure 4– training accuracy in dotted line, validation accuracy in dark blue

4. Stopping criteria, batches, epochs

As a stopping criterion we monitored the validation accuracy, with a patience of 5. In the end our best model only ran for about 15 epochs, since we had a hard time with GPU and RAM usage on Colab (this is discussed in the conclusions). Also, we saw that the best results when it came to training speed and accuracy were achieved with a batch size of 512.

5. Conclusions

We would be lying if we said this challenge didn't bring us many difficulties. It was hard for us to train the models locally and we also found limitations online, so we had a much lower computation power than expected (we hope to fix these problems by the next homework). Something we found very discouraging was that it took us a long time to improve on the score that we first got with a very simple transfer learning (it seemed to learn better but didn't reach high accuracy, as seen in Figure 4), and this made us feel like all our efforts weren't useful. One field where we had to give up was image augmentation: we tried to implement it many times with different transformations, but it never gave us good results, and most times it took too long to even see results. We tried to implement it at test time by modifying the model.py file, but again this didn't improve the performance significantly. We suppose that maybe we just didn't find the right transformations, or we didn't have enough computational power. We mainly tried to use random contrast, rotations and flips. Another thing we tried was resizing the pictures and making them bigger with interpolation, but also this didn't help us much. We even tried running very slow models for 10+ hours but even that didn't improve even on the transfer learning from EfficientNet. The only way to improve was to use ConvNeXtLarge, and this made us think that in the end we just needed to rely more on external nets than on our own wits and skills. In the end, the best model was the one with 0.1 dropout (Figure 5) instead of the one with 0.2 dropout (Figure 6) because even if the validation accuracy seemed to fluctuate more it gave slightly worse results. Another difficulty we had was making the preprocessing match between the model.py file and our model training, and that took us a long time to figure out since it was preventing a realistic output from the CodaLab website.

However we are proud of building something that worked even if our final accuracy online wasn't very high, and we feel like we have learned much about CNNs and adjacent fields.

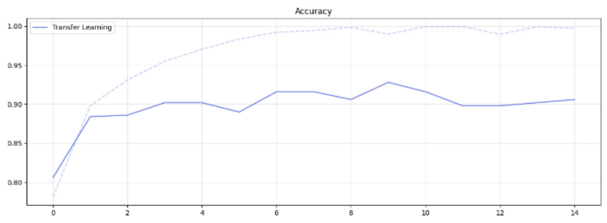


Figure 6 - training accuracy in dotted line, validation accuracy in dark blue

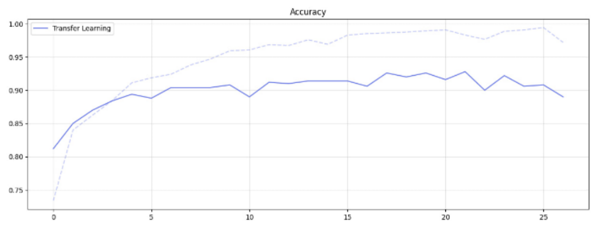


Figure 5 – training accuracy in dotted line, validation accuracy in dark blue

Contributions

We all worked in parallel during the homework, in the first days Filippo focused on outlier removal, while Kamil was trying different convolution networks/GAN and Liv was testing different nets for transfer learning. We then landed on the best model and every one of us tried different parameters, testing in parallel, also with the aim of exploiting the hardware resources (Colab's and local GPUs) as well as we could. The report was written together, with Liv mainly writing the text and Filippo and Kamil obtaining the pictures and plots. Overall it was a group effort and it is hard to say who specifically worked on what. Maybe working so much in parallel was a professional deformation since we are all studying to be High Performance Computing Engineers.