AN2DL - First Homework Report
ANNalyzers

Edoardo Gennaretti, Francesco Lo Mastro, Riccardo Figini, Pierantonio Mauro

edogenna, francescolomi, riccardofigini, pierantonio

258290, 243817, 251187, 258737

November 24, 2024

1 Introduction

The goal of this challenge was to construct an **image classifier** capable of correctly classifying 8 types of **blood cells**. In this paper we are going to describe the key steps that led us to the realization of the networks that we used for the challenge. The key libraries that facilitated the creation of the network were Keras, Keras_CV and TensorFlow, used on Google Colab and Kaggle as development platforms.

2 Problem Analysis

The problem consists in correctly classifying a dataset of 96x96px RBG images. To do this we are required to **build a CNN** (for feature extraction) **and a classification head** able to assign the correct label to each provided image. Once the network is built, it will be **tested** on a test set never seen during training and evaluated on the accuracy reached in classifying those images.

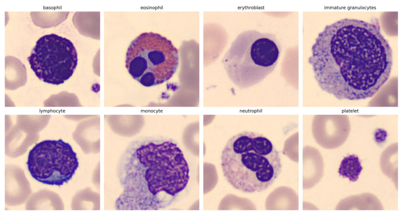

Figure 1: Blood cells images

3 Method

For enhancing the input data, we used data augmentation techniques provided by the Keras and KerasCV libraries, along with size manipulation.
To construct the CNN layers, both Keras primitives (e.g., Conv2D, GAP, BatchNormalization) and pre-trained models from Keras Applications (e.g., MobileNet, InceptionNet) were utilized.
To analyze our results, we employed a variety of evaluation metrics and visualization techniques, including confusion matrices, plots for validation loss, validation accuracy, and validation precision, as well as metrics like Precision, Recall, and ROC curves.
In our experiments, we measured progress using both **local** and **remote** (Codabench) metrics, aiming to align them as closely as possible to then optimize the actual performance. For brevity, we will use the following abbreviations:

Abbr.	Meaning
L_Acc	Training Accuracy on notebooks (local)
L_VAcc	Validation Accuracy on notebooks (local)
C_Acc	Test Accuracy on Codabench

Figure 2:

4 Experiments

Outliers_Remover.ipynb (1):
During a first inspection of public dataset, we identified the presence of repeated images, between these we can spot some **outliers**. The outliers were many copies of the 2 images shown in Figure 2.
Consequently, we developed a script to remove all images that were identical to those identified as outliers in either categories and return a clean dataset.

First_attempt.ipynb (2):
The first neural network developed for this study was intentionally simplistic in design, serving as a preliminary model to establish a baseline for the problem and to familiarize the team with the dataset, testing platform, and overall workflow.
This initial network comprised a minimal architecture with a small number of layers and parameters, ensuring that it could be implemented and tested rapidly. Basic configurations, such as two Conv2D layers followed by a Pooling layer and a dense output layer with Softmax activation, were employed.

L_Acc	0.99	L_VAcc	0.96	C_Acc	0.2

The strong difference from C_Acc and L_VAcc make us realize that Codabench test set surely has a different distribution w.r.t ours.

Second_attempt.ipynb (3):
We aimed to simulate Codabench performance on local development. To do this, we applied several **augmentations** from Keras_cv (e.g., randomPipeline, cutMix/mixUp) to both our training and validation sets
A slightly more complex, yet simple model was developed (Simple_CNN), using GlobalAveragePooling instead of a Flatten layer to preserve image patterns.

L_Acc	0.89	L_VAcc	0.67	C_Acc	0.39

We got a good reduction of local performance but still far from Codabench.
We then switched to a Normalized_Regularized_CNN model, which incorporated regularization and normalization into the training process, yielding marginal local improvements but almost no changes on Codabench.

L_Acc	0.82	L_VAcc	0.67	C_Acc	0.41

Third_attempt.ipynb (4)
Assuming that the poor results of the first solution were due to low network complexity, we tried adding Squeeze-and-Excitation block. We performed several tests with this network, changing structure and parameters.
The first step was to implement the network seen in class and make a first attempt, achieving on Codabench a score of 0.48. Two obvious signs of overfitting are present in the first test. The first is the stabilization of **val_accuracy** around 0.75, accompanied by significant fluctuations indicating poor generalization. The second is the **val_loss**, which does not show a progressively decreasing trend, suggesting that the model is not improving in fitting the validation data.
Another problem was obviously the limited accuracy achieved. To overcome these problems we have developed several networks, the techniques used are shown in the .ipynb. Here's a brief summary list: increase Dropout, increase layers, Regularization, optimization (Lion-SDG), weight class. A lot of attempts for improvement were made, trying different configurations, without producing noteworthy results on Codabench. We discover with next experiments that the main problem was not in the network structure, but in the augmentation of the images.

GoogleNet_parallel.ipynb (5):
Concurrently we implemented a model inspired by GoogleNet, useful for capturing different features at different scales thanks to the concept of **inception modules**, which allows for parallel feature extraction.
In each of these inception modules, the convolutional layers are arranged to simultaneously use several filters of various sizes to extract features from the input. This trial had worse performance compared to the others, so we decided to not use it.

L_Acc	0.63	L_VAcc	0.5	C_Acc	0.45

Fourth_attempt.ipynb (6):
We hypothesized that the existing augmentations were insufficient, as validation accuracy was still far from Codabench results.
To address this, we introduced heavier augmentations such as Solarization, Brightness, and GridMask, also increasing the number of augmentations per image from 1 to 3. This approach forced the model to learn from greater variations in color, shape, and subject positioning.

Now the Normalized_Regularized_CNN improved pattern learning and has closer local perfomance:

L_Acc	0.83	L_VAcc	0.52	C_Acc	0.55

From now on the local validation can give us a **reliable estimate of Codabench performance**.

Fifth_attempt.ipynb (7):
Even though the previous attempt showed improvements on the generalization capability of our model, we were worried that the augmentation could start to even **destroy significative features** of the images. For that reason we tried new augmentation approaches that use just the Keras_cv.RandAug function. This strategy is purposely designed by Keras to avoid augmentation pipelines that ruin the image features.

L_Acc	0.99	L_VAcc	0.91	C_Acc	0.62

As the table shows, a new significative improvement in Codabench occurred, from now on, this will be our base to improve.
A downside of the new augmentation approach is that it caused our local validation estimates to diverge further from the Codabench results, again.

Further_tests.ipynb (8):
The new augmentation technique shows great promise, but it is evident that achieving [L_Acc = 0.99] may indicate **overfitting**.
To address this, we incorporated L1 and L2 regularization layers into the Dense layers, experimenting with different lambda values as well as various Dropout rates.
However, the final results were either excessively high ([L_Acc = 0.98], still suggesting overfitting) or significantly low ([L_Acc = 0.5] or worse) with no improvement on validation.
We also attempted to **strengthen weak classes** by applying **"class weights"** to the model. But still, there weren't any noticeable improvements (Codabench faced serious availability issues during this period, that's why we do not have C_Acc estimates for this experiments).

tl-ft-mobilenet/convnextbase.ipynb (9):
Our models struggled to **learn the correct patterns**, prompting us to both trying to provide more samples for weak classes (0,2,4,5) and using **Transfer Learning with Fine Tuning**. In addition, since Codabench did not allow testing this approaches, we devised a local strategy to assess performance:
We applied RandAug for the **Training set** and the augmentation from Fourth_attempts.ipynb for the **Validation set** as it appeared to better approximate Codabench performance.
So our focus was trying to improve results on this Validation set, aiming to surpass [L_Vacc = 0.52] from the Fourth_attempt by a lot more. In this way we could be enough sure that even Codabench performance would have been improved too.
We first experimented with **MobileNetV3**, a model with a moderate number of parameters, achieving:

L_Acc	0.99	L_VAcc	0.74

Next, we used a model with very high amount of parameters like **ConvNeXtBase**, limiting the Early Stopping patience to prevent overfitting and really long training time. This approach led to even better performance:

L_Acc	0.99	L_VAcc	0.83

tl-ft-InceptionNet.ipynb (10):
Further attempts at transfer-tuning learning were carried out with InceptionNet50. Of particular note is the attempt to lower the number of misclassifications between Monocyte and Immature Granulocytes, which were the major cause of error. In order to solve this problem, we tried duplicating specific classes and/or dwelling on other metrics during training. On codabench: 0.81

5 Results & Conclusions

In the Final Phase our models achieved almost the same results as the Developement Phase:

Best custom model	Best tuning model
Fifth_attempt: 0.63	tl-tf-convnextbase: 0.86

In conclusion, the development of the blood cell classifier highlighted the critical role of augmentations in improving generalization. The custom networks we built were good but limited in their ability to extract features effectively due to architectural constraints.
As predictable, fine-tuning pre-trained models, such as ConvNext, provided the best results by leveraging their feature extraction capabilities. Final results were coherent with local performance confirming that our strategies on emulating them have been effective.
Contributions: **Francesco** and **Pierantonio** worked on notebooks (2,3,6,7,8) while **Riccardo** and **Edoardo** worked on notebooks (1,4,5,9,10).

6 Bibliography

[1] Keras.io. Keras. [Online] https://keras.io/api/applications/.

[2] Image augmentation. Keras. [Online] https://keras.io/api/layers/preprocessing_layers/image_augmentation/

[3] *Symbolic Discovery of Optimization Algorithms.* Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Yao Liu, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, Quoc V. Le. https://arxiv.org/abs/2302.06675

[4] *A Comprehensive Survey on Transfer Learning.* Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, Qing He. https://arxiv.org/abs/1911.02685

[5] *Parameter-Efficient Fine-Tuning for Large Models: A Comprehensive Survey.* Zeyu Han, Chao Gao, Jinyang Liu, Jeff Zhang, Sai Qian Zhang. https://arxiv.org/abs/2403.14608

[6] *A ConvNet for the 2020s.* Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, Saining Xie. https://arxiv.org/abs/2201.03545

[7] ConvNeXt newtowrk. Keras. [Online] https://keras.io/api/applications/convnext/