AN2DL - First Homework Report
Natural Stupidity

Camilla Bocciolone, Davide Fileccia, Luca Forte, Matilde Simonetti

camibocciolone, goofymushroom0, calu02, matix

280315, 273638, 280428, 279810

November 24, 2024

1 Introduction

The aim of this project is to classify images of blood cells by developing a **Convolutional Neural Network** (CNN) based on the concepts learned in the Artificial Neural Network and Deep Learning course. The dataset provided to train our network contains 13,759 images of **blood cells** (96x96 pixels, RGB), categorized into 8 classes: Basophil, Eosinophil, Erythroblast, Immature Granulocytes, Lymphocyte, Monocyte, Neutrophil and Platelet.

2 Problem Analysis

1. **Dataset characteristics**
We started by inspecting the dataset, printing casual batches of images for manual review, which helped us identify several anomalies. The last 200 images were identical, labeled as class 5, but had manually altered backgrounds. A similar issue was observed from image 11,959 to 13,559, where identical images with modified background appeared under different labels. Our final code processes the data by filtering out duplicates and saving the cleaned dataset in a new file named `data_processed.npz`.

2. **Main challenges**
After cleaning the dataset by removing duplicates, we reduced it to 11,943 images. Upon further examination, including printing batches of images from each class, we observed significant class imbalances, with some classes containing nearly one-third the number of images compared to others. Another challenge was the strong similarity between certain classes where some cells appeared almost identical making differentiation more difficult.

3. **Initial assumptions**
We divided the dataset into training and validation sets using a 90/10 split. All the models described in the next section were trained with a batch size of 128, a learning rate of 0.001, and a maximum of 1000 epochs, incorporating Early Stopping to mitigate overfitting. Additionally, we opted to defer addressing class imbalance until later stages of the project. The initial models were trained on the original, cleaned dataset.

3 Experiments

We started by building two custom CNN models; The first had a basic architecture with 2 convolutional layers (32 and 64 filters) followed by ReLU activations, 2x2 max pooling, and a dense layer. The second, more complex model inspired by VGG featured 5 convolutional layers (64, 128, 256, 512, 512 3x3 filters). To reduce the number of parameters in the deeper model, we replaced the Flattening layer with a Global Average Pooling layer. However, from the training and initial testing, the tendency to overfit was evident (for the more complex model the training accuracy was 1.00). As a matter of fact, the accuracy scores in the training and validation parts didn't match the ones obtained in the test one on Codabench.

Based on the performance of the custom models, we decided to adopt a different approach.
After studying the principles of **Transfer Learning** and **Fine-Tuning** during the course, we integrated these techniques into our workflow. Specifically, we employed the pre-trained MobileNetV3 model from Keras [2], leveraging its ImageNet-trained weights to benefit from the model's generalized feature extraction capabilities. Initially, all layers of the pre-trained model were frozen to retain the previously learned features, and we appended a fully connected dense layer followed by a dropout layer (with a dropout rate of 0.3) to mitigate overfitting. This method resulted in a notable improvement on test accuracy on Codabench.
Next, we experimented with the **EfficientNetB0** model, which resulted in further accuracy gains. Given the promising results, we decided to apply Fine-Tuning to better adapt the model to our specific task. We froze the first 30 layers of the model and unfroze the remaining convolutional layers to allow fine-tuning. Additionally, we added a **Dropout Layer** (with a dropout rate of 0.3), we also used **L2 Regularization** (equation 2) to reduce the risk of overfitting. This allowed the network to learn more task-specific features while retaining the general knowledge captured by the pre-trained model.

We decided to implement **Data Augmentation** directly on the dataset to enhance the model's ability to generalize by exposing it to a wider variety of transformations. Given the strong visual similarity between certain classes, we hypothesized that augmentation would help the model capture subtle differences in cell morphology. Additionally, augmentation would act as a form of regularization, reducing the risk of overfitting by preventing the network from relying solely on specific features of the training data. Initially, we applied mild data augmentation, focusing on slight geometric transformations such as translation, rotation, and horizontal and vertical flips. However, this approach did not result in any significant improvement. Subsequently, we experimented with more aggressive augmentation techniques to enhance model performance.
We used **RandAugment** from kerascv on our train set with the followings parameters: augmentations per image=1, magnitude=0.5, magnitude stddev=0.15, rate=0.90, geometric=True; to optimize the training process and reduce computational cost, we chose to perform data augmentation outside of the network. This approach allows the network to train more efficiently without the overhead of augmentation during runtime.
This augmentation is what led to the improvement on Codabench testing.

From this point on we focused on **Hyperparameters Tuning**. We conducted several experiments to improve model performance. First, we adjusted the batch size to 32 and 64, and modified the data augmentation parameters by increasing the magnitude to 0.6, the magnitude standard deviation to 0.2, the number of augmentations per image to 2 and 3, and reducing the augmentation rate to 0.7. However, these changes resulted in similar validation accuracy as before and a decrease in Codabench accuracy. Next, we lowered the learning rate to 0.0001, as fine-tuned models often benefit from smaller learning rates [1]. This adjustment yielded a slight improvement. Additionally, we experimented with replacing the Adam optimizer with Lion, testing learning rates of 0.001, 0.0001, and 0.00005. These trials did not produce any significant improvement. A subsequent attempt with a learning rate of 0.0001/3 resulted in better validation accuracy but worse Codabench accuracy. Based on these results, we decided to retain Adam as the optimizer with a learning rate of 0.0001. To make the net more precise we also tried adding a dense layer with 128 neurons and a batch normalization layer, but unfortunately this didn't lead to actual improvement on both validation and Codabench accuracy. To ensure that EfficientNetB0 was the optimal choice, we compared its performance against other pre-trained networks, including ResNet50, DenseNet121, and EfficientNetB4. All of them achieved validation accuracies comparable to EfficientNetB0; however, they required a substantially higher computational cost due to the higher number of parameters, making EfficientNetB0 the more efficient option.

4 Method

The final model we presented, incorporating the pre-trained EfficientNetB0 network, utilizes the following layer types: CONV2D, DeepCONV2D, BatchNormalization and Average Pooling. All the hidden layers in EfficientNetB0 employ the Swish activation function and for the output layer, we employed softmax function, which is well-suited for classification tasks.

The Swish activation function:

$$f(x) = \frac{x}{1 - e^{-x}} \tag{1}$$

The L2 regularized loss function used during training is given by:

$$L_{\text{regularized}} = L_{\text{original}} + \lambda \sum_i w_i^2 \tag{2}$$

Where:
L_{original}: Original loss function. λ: Regularization strength. w_i: Model weights. η: Learning rate.

5 Results

Table 1 provides a summary of the performance metrics for the key models developed during this project.
Table 2 compares different pre-trained networks after fine-tuning.

6 Discussion

After finalizing our model through hyperparameter tuning, we analyzed its performance using the confusion matrix 1. Notably, we observed a consistent pattern of misclassification for *Immature Granulocytes*, which were frequently misclassified. We hypothesized that this issue stemmed from class imbalance within the dataset. To address this, we considered two potential solutions:

- balancing the classes by applying offline data augmentation using the ImageDataGenerator with minimal transformations.
- employing class weights, a technique that assigns higher costs to minority class samples during training, thereby influencing the loss function to improve representation of underrepresented classes.

However, neither of these techniques led to any significant improvement, so we decided to continue using our imbalanced dataset.

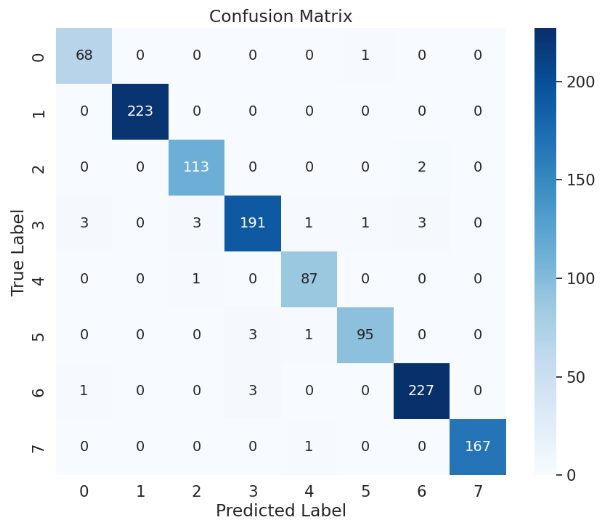

Figure 1: Final Model Confusion Matrix

7 Conclusions

After conducting a sufficient number of tests on custom networks and various pre-trained networks, we concluded that the best approach was to use a pre-trained network, incorporating **Transfer learning** and **Fine-tuning** techniques. This allowed the network to adapt to the specific classification problem at hand, combined with randomized data augmentation to maximize the network's generalization capabilities. Finally, the code implements the adopted network, which is structured as follows:

- **EfficientNetB0**
- **Dropout Layer**
- **Output Layer**

Despite our efforts, the network we presented requires further improvements:

Table 1: The scores of some of our models

Model	MobileNetV3	EfficientNetB0_1	EfficientNetB0_2
Validation Accuracy	91.72	98.58	98.74
Codabench Accuracy	42.00	**77.00**	76.00

Notes: EfficientNetB0_1 Adam optimizer learning rate=0.0001, while EfficientNetB0_2 with Lion optimizer and learning rate=0.0001/3.

Table 2: The scores of some of our models

Model	ResNet50	DenseNet121	EfficientNetB4
Validation Accuracy	97.41	97.49	98.58

- **More accurate and problem-specific fine-tuning and hyperparameter tuning.**
- **Customized augmentation tailored to the specific problem** (we discussed about removing some photographic augmentation and geometric augmentation).
- **Identifying the correct way to balance the classes**, as all the hypotheses we proposed seemed to worsen the network's performance.

References

[1] Addison-Wesley Professional. Howard, J. and Gugger, S. (2020). *Deep Learning for Coders with fastai and PyTorch: AI Applications Without a PhD.*

[2] *Keras Apllications*: Link

[3] Keras Augmentation Layers: Link

8 Members' contribution

Regarding the contribution of each member: Camilla implemented the first CNN model and handled data augmentation to balance the classes. Luca developed the MobileNetV3 model. Matilde designed the second custom model inspired by VGG and collaborated with Davide to create the EfficientNetB0 model. Additionally, Davide integrated RandAugment into the workflow. For all other tasks, such as hyperparameter tuning, we contributed collectively.