AN2DL - First Homework Report
Team Rookie

Jiaxiang Yi, Yang Hao Mao, Simona Cai, Ying Zhou

rookieyi, leooo, cairookie, yingrookiee

251872, 248391, 252253, 276543

November 24, 2024

1 Introduction

The project's goal was to assign 96x96 RGB images of blood cells to one of eight categories, each indicating a distinct cell condition. This job is a standard *multi-class classification* issue that allows us to investigate both the theoretical and practical elements of **artificial neural networks and deep learning**. In the following report we are going to describe how we handled this problem in three stages:

- Analysis of the dataset and research of the best data augmentation
- Creating a CNN from scratch
- Applying Transfer Learning and Fine Tuning

2 Approaches

1. Dataset Analysis

   We discovered that there are some "outliers" in the dataset, such as meaningless or irrelevant images, so we did data cleaning, here is distribution of our dataset, which represent number of items per class.

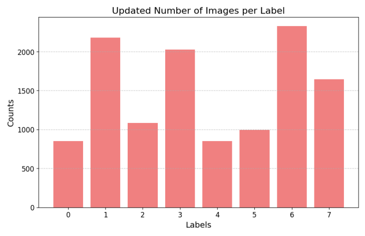


Figure 1: Label distribution of the filtered dataset

In the Figure 1 we noticed a significant discrepancy in the number of samples per class. To mitigate this imbalance, it might be necessary to apply various data augmentation techniques to generate new and unseen samples, and another strategy that could help is implementing **class weights**, which penalize over-represented classes to *balance the distribution.* We chose **accuracy** as our primary evaluation metric, paying particular attention to its trend on the validation set to keep the analysis straightforward. Regarding dataset splitting, we experimented with different proportions for the training and validation sets, initially using an 0.8-0.2 split. However, since the test set was already provided on Codabench, this offered a good indication of potential Condabench scores. Consequently, after reviewing the dataset's characteristics and need for a more robust training process, we decide to increase the training set size and reduced the validation set size to a 0.9-0.1 split. To address the increased *risk of over-fitting* with a larger training set, we adopted **early stopping** during the training process. Meanwhile, we preprocessed our training set dividing each value by 255 thus normalizing the images to stabilize training process.

2. Custom Model

   Initially, we create a simple CNN designed from scratch for image classification with few layers, 3 convolution and 3 max pooling followed by a flattening and a dense layer, and in the end trained the model obtaining a result of 25% on the test set. In order to improve the performance, we decide to add few more blocks composed by a Conv2D layer and MaxPooling2D layer, which extract features of varying complexity. After that we notice our model is over-fitted in validation set, so we inserted a Global Average Pooling layer and tried regularization with L1, L2 and L1L2 with different values of lambda, reaching our best result yet with L1L2 and with lambda equals to 0.001. We also tried to change the activation function ReLU and Leaky Relu, which the first one obtained our best new result. Finally we added dropout with rate euquals to 0.3 and HeUniform initialization to stabilzie training, map the features to the eight output classes using a softmax activation, to reach a 86% accuracy on the validation set.

3. Data Augmentation

   The biggest challenge we encountered was selecting the appropriate data augmentation strategies. It is crucial to determine which transformations may be used to achieve this aim. Initially, we applied only basic augmentations to our custom CNN model, which resulted in a leaderboard score of just 34%. To improve performance, we experimented with more advanced augmentation techniques, focusing primarily on **AugMix()**, **MixUp()**, and **RandomAug()**. However, combining **MixUp()** and **AugMix()** led to a worse score. Then, we chose **AugMix()** over **RandomAug()**, as it performed better individually. Additionally, we experimented with other augmentation techniques such as **GridMask()** and **CutMix()**, but they did not yield significant improvements. Finally, we introduced *noise* into the validation set to enhance robustness, which resulted in only a minimal improvement.

3 Model Selection

We found that applying a substantial amount of augmentation initially helped the model improve on the training set, but it led to over-fitting, as it didn't generalize well to the test set. Consequently, we decided to shift your approach and concentrate more on Transfer Learning to address the issue. The base models that have been used are as follows:

- InceptionV3
  The performance of the model at is not good, and the final validation accuracy is only 43.23% in the local test. Then, we enter a fine-tuning phase by unfreezing some of the end layers of InceptionV3, allowing some parameter updates. We tried a variety of fine-tuning parameters, which ultimately led to an increase in Final validation accuracy to 93.48% in the validation set and 49% in the test set.

- MobileNet
  Considering our limited computing resources, we started focus on lightweight models such as MobileNet and EfficientNet. However, with MobileNet, we observed extremely low accuracy during the training phase, reaching only around 30%, even with strong data augmentation.

- EfficientNetB0, EfficientNetB2
  The same issue occurred with EfficientNetB0 and EfficientNetB2. These lightweight models performed poorly during the local training phase, with the highest accuracy observed being around 45% and 42%. As we are not satisfied with these results, we have decided not to continue to use them.

- Resnet50V2
  With ResNet, we achieved our best local results, with a validation accuracy of approximately 94%. However, the score on the test set is only reached to 57%. We tried various fine-tuning strategies, such as freezing two or more convolutional layers, but these adjustments did not yield significant improvements in performance. We also tried DenseNet, but the result remains the same.

- **VGG16(Final model)**
  During the fine-tuning phase, we tested with unfreezing different percentages of layers to find the best configuration for performance. Specifically, We obtained an accuracy of 96.32% on our local test set by freezing the first 3 layers of the VGG16 architecture. In

Figure 3 we can see the confusion matrixes. To improve the output even more, we added a normalization layer following the Global Average Pooling layer, which we identified as missing from the original architecture. To further improve performance, we introduced new features such as a dynamic learning rate that decreases when the validation loss increases, along with additional fine-tuning strategies. We noticed that our model might have an overfitting issue, see in Figure 2. However, after adding dropout layers with rate 35% and 20% respectively after each Dense Layer the results reach to 67%.

Table 1: **Local Results** During local testing, the models obtained the following results.

Model	Validation Accuracy(%)	Validation Loss(%)
Resnet50V2	96.15	15.26
EfficientNetB0	45.40	191.81
EfficientNetB2	42.31	179.78
VGG16	**96.32**	**12.64**
InceptionV3	93.48	21.67

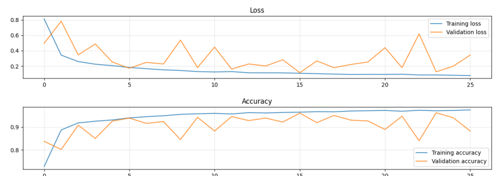
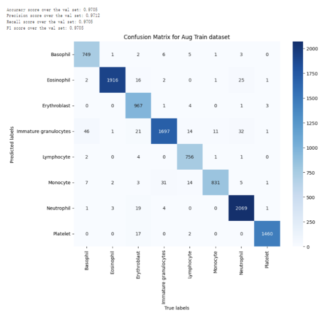


Figure 2: VGG16's Loss and Accuracy

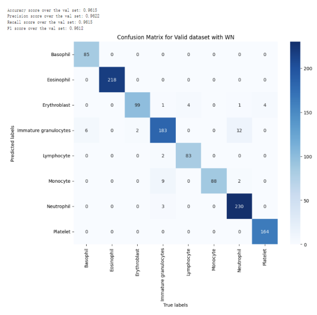


```
Accuracy score over the val set: 0.9705
Precision score over the val set: 0.9712
Recall score over the val set: 0.9705
F1 score over the val set: 0.9705
```

Confusion Matrix for Aug Train dataset

Predicted labels \ True labels	Basophil	Eosinophil	Erythroblast	Immature granulocytes	Lymphocyte	Monocyte	Neutrophil	Platelet
Basophil	749	1	2	6	5	1	3	0
Eosinophil	2	1916	16	2	0	1	25	1
Erythroblast	0	0	967	1	4	0	1	3
Immature granulocytes	46	1	21	1697	14	11	32	1
Lymphocyte	2	0	4	0	756	1	1	0
Monocyte	7	2	3	31	14	831	5	1
Neutrophil	1	3	19	4	0	0	2069	1
Platelet	0	0	17	0	2	0	0	1460

Confusion Matrix For Train dataset



```
Accuracy score over the val set: 0.9615
Precision score over the val set: 0.9622
Recall score over the val set: 0.9615
F1 score over the val set: 0.9612
```

Confusion Matrix for Valid dataset with WN

Predicted labels \ True labels	Basophil	Eosinophil	Erythroblast	Immature granulocytes	Lymphocyte	Monocyte	Neutrophil	Platelet
Basophil	85	0	0	0	0	0	0	0
Eosinophil	0	218	0	0	0	0	0	0
Erythroblast	0	0	99	1	4	0	1	4
Immature granulocytes	6	0	2	183	0	0	12	0
Lymphocyte	0	0	0	2	83	0	0	0
Monocyte	0	0	0	9	0	88	2	0
Neutrophil	0	0	0	3	0	0	230	0
Platelet	0	0	0	0	0	0	0	164

Confusion Matrix For Valid dataset

Figure 3: Confusion Matrix for VGG16

4 Discussion

- Strengths and weaknesses: We believe our model performs quite well in learning image features, even after applying strong augmentation, and the accuracy appears to be good. However, we suspect we might be experiencing an overfitting illusion. Despite adding more dropout layers and L1L2 regularization, we haven't been able to find the best solution to resolve this issue, as the problem persists.

- Limitations and assumptions: Using proper augmentation has proven to be an effective strategy for addressing the small dataset size and has helped improve our test set performance. However, we believe there is room for improvement and suspect we either need to add more augmentations or find a better combination of augmentation layers.

5 Conclusions

- Restate main contributions

  Every teammate gave their best effort, and everyone worked hard to achieve the best possible results. Each teammate contributed in different ways during various phases of the project.

- Suggest improvements

  We haven't tested many augmentation layers and different optimizer due to issues with Codabench, and we also haven't extensively tested hyperparameters. Therefore, we've decided to experiment with different models and focus on fine-tuning. We believe that our model can be improved with more hyperparameter adjustments.

- Propose future work

  We tried many models, but only found VGG16 to be the most robust and adaptable model, if not the best local performer, at the end. If we have more time we will further tune and try VGG16 accordingly. What's more, we plan to test different optimizers (like **Lion Optimizer** and **Ranger Optimizer**) to evaluate their impact on our model's performance, and different hyperparameters.