AN2DL - First/Second Homework Report vacis-vaccarino-palladino

Nicolò Vacis, Giovanni Vaccarino, Vittorio Palladino

vacisnicolo, giovanniv, vittpall

274316, 274402, 278541

June 1, 2025

1 Introduction

The AN2DL challenge involves developing a **supervised classification model** to categorize eight classes of blood cells. The primary objective is to maximize classification accuracy.

2 Problem Analysis

Dataset: The dataset contains around 14,000 blood cell images across eight classes, requiring thorough preprocessing.

Architecture: Our goal was to identify the best model architecture through systematic experimentation and performance analysis.

Main challenges: Achieving high remote test set accuracy while dealing with potential inconsistencies in validation scores was a primary challenge. Designing a model that balances performance and spacial efficiency was crucial.

Initial assumptions: We initially assumed that dataset balancing would aim to avoid biases during training, augmentation would introduce variety and prevent overfitting by simulating diverse scenarios, and fine-tuning would enhance training performance by carefully adjusting parameters to avoid overfitting while maximizing model accuracy.

3 Methods applied

Dataset preprocessing: Initially, we analyzed the dataset using a *cosine similarity matrix* [5] [10] to identify similar images and targeting outliers. Features were extracted using a *pre-trained VGG Net model* [3]. This analysis revealed duplicate and erroneous images, which were excluded. To facilitate this process, all NPZ files were converted into images for a thorough manual review.
In order to firstly balance the dataset, **data augmentation** was applied progressively, starting with basic transformations like zoom and translation to avoid overcomplicating the dataset. Gradually, we fully augmented the dataset while introducing more advanced techniques like *CutMix and MixUp* [4] and random color degradation to enhance dataset diversity. Offline augmented datasets were precomputed to reduce training iteration times, and additional lightweight online augmentations were applied during training. To find a good representative dataset: it was initially **split into static proportions** (70% training, 20% validation, 10% testing). Later, *K-fold cross-validation*[1][9] was employed to evaluate multiple dataset combinations and optimize performance. To maximize train/val data, we **excluded a test split**, relying on the remote test for evaluation. For the final experiments with ConvNeXt, we utilized the entire dataset for training to fully leverage its potential and maximize model accuracy.

Simple Architectures: Drawing inspiration from architectures discussed during lessons and popular models like MobileNet, we experimented with several custom designs better explained in the "Experiments" section.

Transfer Learning and Fine Tuning: We explored transfer learning with several pre-trained models to strengthen and enhance our architectures:

- **MobileNet**: We began with a lightweight model, testing layers added after MobileNet. Dense sequences outperformed conv sequences, and adjustments to depth, size, dropout, and **L2 regularization** slightly improved performance. Fine-tuning achieved 47% accuracy on the remote test but overfitting persisted.
- **VGG19, EfficientNet and ResNet**: We applied a simple architecture, consisting of one or two dense layers with added dropout, to three pre-trained networks: ResNet, EfficientNet, and VGG19. With the same training conditions and fine-tuning, VGG19 outperformed the others, achieving a 62%.
- **ConvNeXt**: [8] This architecture delivered the best results. Additionally, a *dynamic learning rate* and refinment structure on the dense layers optimized with **Keras Tuner** [6], further enhanced performance.

4 Experiments

Going deeeper in the previous points, those are some expreriments made.

Network architectures: As outlined in "Methods Applied," we began experimenting with architectures, starting with a custom model [11]. The most effective featured two blocks of convolutional layers with MaxPooling2D, followed by two dense layers for feature unification. Dropout layers were added to reduce overfitting, but performance remained limited. We then tested various pre-trained architectures (Table 1), finding that simple additional layers worked best.
Finally, for ConvNext, we used Keras Tuner [6] to optimize parameters, as shown in Table 2.

Fine Tuning: To improve fine-tuning, we implemented a dynamic learning rate, enhancing training efficiency by adjusting it during the process. We used **class weights** to balance the dataset and address misclassified classes. Analyzing the confusion matrix and prediction confidence, we manually adjusted weights for frequently misclassified classes, slightly improving accuracy. *Early stopping* was also employed to prevent overfitting and ensure optimal performance. During fine-tuning, we gradually unfroze the final 10 layers of the imported net and eventually the entire model to test performance improvements. We also tested triplet loss to enhance separability of similar images with different labels, but it produced negative results. Our final experiment applied basic TTA [2] (only zoom, rotation and flip) to augment image sequences and average predictions, but it actually showed a negative improvement on the remote test set.

Table 2: Hyperparameters Configuration with Keras Tuner.

Parameter	Value
Dropout Layer 1	0.39
Units Dense Layer 1	384
Activation Dense Layer 1	tanh
Units Dense Layer 2	224
Activation Dense Layer 2	relu
Dropout Rate Layer 2	0.33
Optimizer	adam

5 Results

Dataset Preprocessing: Data quality and balancing were key to performance, while challenging augmentations hurt smaller networks but benefited larger ones. K-fold cross-validation slightly improved accuracy, used only to determine the best initial split, but was omitted for the final network due to limited impact. Using the full dataset boosted accuracy but made training monitoring harder without a validation set.

Architectures: The simple custom architecture faced significant limitations due to its small size, causing underfitting during predictions. This was

Model	Validation Accuracy	Test Accuracy
Custom Model	88.82%	22.00%
ResNet18	89.61%	/
MobileNetV3Large	90.16%	50.00%
VGG19	97.34%	62.00%
EfficientNetB7	86.29%	/
ConvNext	**97.21%**	**81.00%**

Table 1: Validation: internal dataset. Test: remote platform.

challenging given the limited dataset size, as the model struggled to capture enough complexity.
After testing MobileNet, we moved on to VGG19, ResNet, and EfficientNet. Among these, VGG19 demonstrated the most potential. Dropout outperformed L2 regularization in this setup, further improving performance.
ConvNeXt outperformed other architectures, with optimal performance achieved using dense layers (512, 128 neurons) preceded and followed by 0.3 dropout.

HyperParameters Tuning: Initial tests with MobileNet highlighted the differences between L2 regularization and Dropout. L2 reduced overfitting while preserving all neurons, but Dropout was more effective, particularly with larger dense layers.
Keras Tuner helped optimize parameters for ConvNeXt during initial training providing values for the number of neurons and rate for the custom dense layers. Using the largest possible batch size (32) proved beneficial as it ensured more consistent gradient updates during training. The **Adam optimizer** required less memory and improved performance. A **dynamic learning rate** provided the best results, adding adaptability during training and improving the overall process. Categorical cross-entropy consistently outperformed other loss functions, including triplet loss, which, despite its potential to better differentiate classes, performed worse, possibly due to suboptimal anchor pairs. Gradually **unfreezing** layers—first 50%, then the other 50%—helped initialize the network and improve convergence, enhancing performance.

6 Discussion

Careful dataset balancing and advanced augmentations significantly improved performance without adding biases. Architecturally, the simplest dense and Dropout layers added after transfer learning models provided the best results, demonstrating that simplicity can work effectively in conjunction with transfer learning. Among hyperparameter tuning strategies, unfreezing several layers and employing a dynamic learning rate stood out.
However, a key limitation was the validation-test correlation. While augmentations initially helped, progress stopped once validation accuracy surpassed 80%, highlighting a gap in alignment with the test set. Additionally, the architectures used, despite their simplicity in post-transfer learning layers, were computationally heavy. Lighter networks often underperformed due to their less complex structures, resulting in lower accuracy. This highlights a trade-off between computational efficiency and model performance.

7 Conclusions

We developed a robust ConvNeXt-based system [7], dynamic learning rates, fine-tuning, and emphasizing preprocessing and data balancing. Future work could explore GANs for dataset augmentation to improve fine-tuning of smaller networks. Additionally, efforts could focus on reducing model computational demands while maintaining accuracy.

Contributors

vacisnicolo: Preprocessing data, transfer learning, and triplet loss experiments.

giovanniv: Focused trainings and fine-tuning optimization models.

vittpall: Tuned hyperparameters using Keras Tuner and refined the model architectures.

References

[1] christianversloo. how-to-use-k-fold-cross-validation-with-keras. https://github.com/christianversloot/machine-learning-articles/blob/main/how-to-use-k-fold-cross-validation-with-keras.m.

[2] P. Dufour. How to correctly use test-time data augmentation to improve predictions. https://stepup.ai/test_time_data_augmentation/.

[3] fareid. Image similarity using cnn feature embedding. https://medium.com/@f.a.reid/image-similarity-using-feature-embeddings-357dc01514f.

[4] lukewood. Cutmix, mixup, and randaugment image augmentation with kerasc. https://keras.io/guides/keras_cv/cut_mix_mix_up_and_rand_augment.

[5] O. Okonj. Cosine similarity - measuring similarity between multiple image. https://onyekaokonji.medium.com/cosine-similarity-measuring-similarity-between-multiple-images-f289aaf40c2b, 2023. Accessed: 2024-11-16.

[6] T. O'Malley, E. Bursztein, J. Long, F. Chollet, H. Jin, L. Invernizzi, et al. Kerastuner. https://github.com/keras-team/keras-tuner, 2019.

[7] VacisVaccarino-Palladino. Convnext-train-script. code.zip/train/Final_ConvNeXt_Train.ipynb.

[8] VacisVaccarino-Palladino. Convnext_train_script. code.zip/train/SimpleNet_Train.ipynb.

[9] VacisVaccarino-Palladino. Kfold_script. code.zip/train/Kfold_Train.ipynb.

[10] VacisVaccarino-Palladino. Similarity_matrix_script. code.zip/preprocessing/K-fold_Train.ipynb.

[11] VacisVaccarino-Palladino. Simplenet_train_script. code.zip/train/SimpleNet_Train.ipynb.