AN2DL - First Homework Report
Spanish Inquisition

Miguel Planas Díaz, Manuel del Carmen Fernández Fernández, Rayan Emara, Emanuele Paesano

miguelplanas, manuelferfer, rayanemara, emanuelepaesano

276442, 276383, 260145, 221974

November 24, 2024

1 Introduction

This project addresses the problem of classifying blood cell images into eight classes. The **goal** is to develop a deep learning model to classify blood cell types by means of a deep learning model.

2 Problem analysis

The first lesson we learned is "GIGO", therefore we inspected the data. We looked for any outliers using the t-SNE algorithim which reduced the image dimensions on 2 components confirming the presence of outliers.

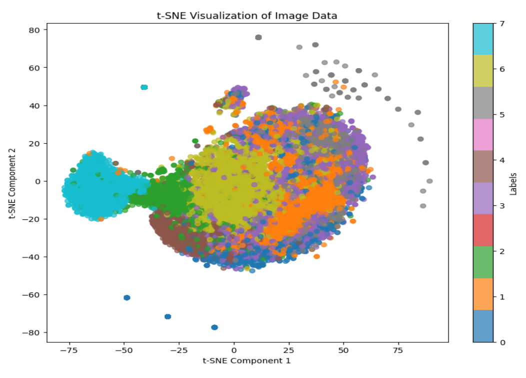


Figure 1: Outliers.

2.1 Cleaning Steps

To ensure the dataset is clean and free from duplicates and irrelevant images, the following steps were performed:

Hash Computation and Duplicate Removal: Perceptual hashes[4] were computed using the `imagehash` library to detect duplicates, which were then removed. Known bad images (e.g., **Shrek** and **Rick Astley**) were also excluded by retrieving and manually inspecting the images with greatest distance from the dataset mean.

Final Dataset: The cleaned dataset, reduced from 13,759 to 11,951 unique, noise-free images, including **class labels** and **original indices**, was saved as `training_data_cleaned.npz` for reproducibility.

3 Experiments

After different custom made models, we found performance to be extremely lacking, quickly realizing is better not reinventing the wheel. Keras provides a wealth of pre-trained models that are much more effective at extracting relevant features than we could ever achieve by hand. We therefore quickly pivoted to fine-tuning large pre-trained models.

We struggled to get traction at first, because we tried things too quick and not iteratively improving on a known working baseline. Our second lesson was "tensorflow often fails silently", we would flip dimensions without realizing it and the model would still perform well locally, while obviously failing on the private test set. We spent a couple of days implementing a good baseline that we knew worked, albeit with abysmal 0.35 accuracy which is very close to random chance on 8 classes. This meant that manually testing and inspecting recall and precision/confusion matrices on each iteration became a big part of our development process, focusing on **explainability** and keeping ourselves sane.

3.1 Fine-tuning

Our baseline consisted of a very simple fine-tuning script that followed the keras fine-tuning guide [7], our first findings were that freezing batch normalization layers underperformed, we believe this has to do with the fact they learn a moving average on the initial training data in order to "normalize" the feature extraction, effectively underfitting our local training data, which is very different from imageNet [5] We experimented with different class weights and found that not weighing the cross entropy yielded the best results on the **private test set** (0.59 vs 0.79 on equivalent EfficientNetV2M[6] implementations) , we believe this is because the classes in the training set aren't hugely imbalanced and we implemented heavy regularization.

On many models we also picked the best model based on the AUC metric, this helped as a sanity check in order to counteract the cross entropy loss which oftentimes led to overfitted models when the ReduceLrOnPlateau wasn't properly tuned, this is because a prediction $[0.1, 0.9]$ is, on account of the loss score, worse than $[0.0, 1.0]$ even though it means the model is learning the inherent noise in the dataset.

Then we implemented Glorot initialization to address unusually high initial losses observed during some training runs. Ideally, the initial loss for a multi class classification model should be close to $-\log(1/n_{\text{classes}})$, which in our case is approximately 2. However, the initial losses were sometimes significantly higher, leading to slower convergence. This initialization scheme ensures that the variance of the weights is balanced, preventing vanishing or exploding gradients at the start of training. This issue was particularly problematic given our limited compute resources [3] [2]

We found smaller batch sizes to effectively act as pseudo regularizers due to fewer data points, therefore introducing randomness in gradient estimation and helping escape local minima, there's also an inverse effect where really small batches can make it so that there's too much noise, we found a balance between performance and ***stochasticity*** at 64/128.

Our third lesson therefore is "understand what each part of your training script does, do not blindly follow online guides".

3.2 Data augmentation and best models

Our augmentation efforts started by adding the augmentation layers directly into the model layers, as a preprocessing step. We did this to benefit from the Test-Time augmentation paradigm, as a way to revert test images to a distribution closer to that of the training set [1]. This however didn't work out as planned since the keras_cv augmentations didn't play nice with the codabench runtime, where we struggled with deserialization errors. We therefore reverted to basic keras augmentation layers at test-time using RandomFlip, RandomRotation, Zoom, Contrast, Translation and Brightness. This resulted in a test accuracy of around 0.59 when fully fine-tuning an EfficientNetV2M model with all of the precuations from the previous paragraph. However, even though we implemented gradual unfreezing, due to the simple augmentation the model and dataset, it still showed overfitnes because of several training of the same layers.

Following Lomurno's advice, we doubled down on the keras_cv augmentations implementing a custom tf.data train-time only augmentation pipeline using almost all of the keras_cv augmentation layers (more on this in the next paragraph). This coupled with a different set of augmentations on the validation set allowed us to fine tune an EfficientNetV2M that generalized pretty well on unseen data, scoring a 0.79 accuracy on the private test set.

3.3 A note on aggressive augmentation

We invested a great deal of effort, late into the challenge, on making sure the augmentations weren't adding noise to the dataset, we regularly plotted augmented train set images and found out that overly aggressive augmentation led to certain samples becoming unrecognizable to the naked eye. On some samples, for example, separate smaller cells were a key feature in predicting that class, this informed our decision to reduce the maximum size on the GridMasking layer. While examining the training dataset we also found classes where the cells would be closer to the edges, we in turn reduced the translation layer's rates in order to make sure we didn't lose that information.

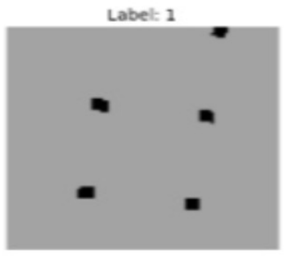


Figure 2: augmented 1.

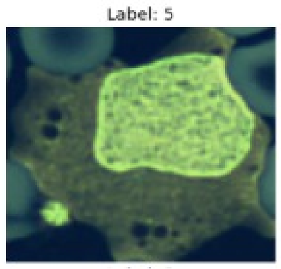
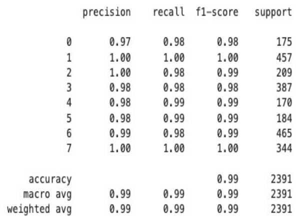


Figure 3: augmented 2.

4 Results

With the lessons we've learned we fully fine-tuned an EfficientNetV2M model, using separate augmentation pipelines on training and validation sets with which we managed to score 0.79 on the private test set, even though we know is not the best, we feel we've built a model we can actually explain, with results we can reproduce.

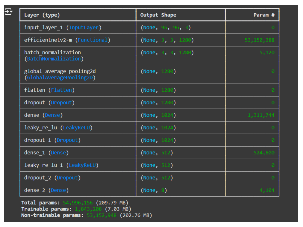


```
              precision    recall  f1-score   support

           0       0.97      0.98      0.98       175
           1       1.00      1.00      1.00       457
           2       1.00      0.98      0.99       209
           3       0.98      0.98      0.98       387
           4       0.98      0.99      0.99       170
           5       0.98      0.99      0.99       184
           6       0.99      0.98      0.99       465
           7       1.00      1.00      1.00       344

    accuracy                           0.99      2391
   macro avg       0.99      0.99      0.99      2391
weighted avg       0.99      0.99      0.99      2391
```

Figure 4: confusion matrix.



```
Layer (type)                          Output Shape               Param #
input_layer_1 (InputLayer)            (None, 96, 96, 3)                0
efficientnetv2-m (Functional)         (None, 3, 3, 1280)      53,150,388
batch_normalization                   (None, 3, 3, 1280)           5,120
(BatchNormalization)
global_average_pooling2d              (None, 1280)                     0
(GlobalAveragePooling2D)
flatten (Flatten)                     (None, 1280)                     0
dropout (Dropout)                     (None, 1280)                     0
dense (Dense)                         (None, 1024)             1,311,744
leaky_re_lu (LeakyReLU)               (None, 1024)                     0
dropout_1 (Dropout)                   (None, 1024)                     0
dense_1 (Dense)                       (None, 512)                524,800
leaky_re_lu_1 (LeakyReLU)             (None, 512)                      0
dropout_2 (Dropout)                   (None, 512)                      0
dense_2 (Dense)                       (None, 8)                    4,104
Total params: 54,996,156 (209.79 MB)
Trainable params: 1,843,208 (7.03 MB)
Non-trainable params: 53,152,948 (202.76 MB)
```

Figure 5: model summary.

4.1 Architecture Summary

The architecture of the model is based on a combination of a pre-trained convolutional neural network (EfficientNetV2) followed by some of the following layers implemented by us:

Use of EfficientNetV2: used to extract powerful features from the images. EfficientNetV2 is known for its efficiency and ability to learn robust representations of images.

Batch Normalization Layers: Improves training stability and accelerates convergence by reducing the risk of activations deviating too much, which can hinder the model's learning.

Dropout: Prevents overfitting by making the model less dependent on specific neurons and promotes better generalization.

Dense Layers: The dense layers transform the features extracted by the convolutional and **GlobalAveragePooling2D** layers into useful representations for the final task. These representations are processed through multiple progressively smaller dense layers (1024, 512, 8 units), allowing the model to focus on the most relevant features for classification.

LeakyReLU: The `LeakyReLU` used to avoid the "dying neurons" problem by allowing a small gradient even for negative inputs. This improves the model's ability to learn and generalize better.

5 Conclusion

We think the main weakness in our training strategy is the augmentation, which could've been stronger. We couldn't employ test time augmentation which would've greatly improved generalization. In hindsight, spending an entire week trying to work on a custom model was a mistake, we wasted a lot of time without any guarantee of a return on our investment. Instead, we would've benefited from trying many different architectures once we wrote a good training script. We also focused too much on keeping our models tiny, sometimes less is not more and we clearly lacked the expertise necessary to expertly fine tune, even while following some great online guides. On the other hand we learned **a lot**,we especially gained an intuition for why the things that worked did so. We'd like to keep working on the problem, our roadmap includes implementing a hyper-parameter tuning script and more exotic optimizers and learning rate schedulers.

6 Work

Rayan worked on the fine-tuning models, final report and research. Emanuele Paesano worked on the custom model architectures and fine-tuning models. Manuel worked on the report, fine-tuning, research on optimizers and class imbalance. Miguel worked on custom model architectures and fine-tuning models.

References

[1] The importance of efficient models in machine learning, 2024. Accessed: 2024-11-24.

[2] S. Dutta. Understanding glorot and he initialization: A guide for college students, 2020. Accessed: 2024-11-24.

[3] A. Karpathy. The "recipe" for training neural networks, 2019. Accessed: 2024-11-24.

[4] C. Prathima and N. B. Muppalaneni. A novel framework for handling duplicate images using hashing techniques. In *2023 3rd International Conference on Innovative Mechanisms for Industry Applications (ICIMIA)*, pages 984–991. IEEE, 2023.

[5] Stack Overflow. How to freeze batch norm layers during transfer learning, 2020. Accessed: 2024-11-24.

[6] M. Tan and Q. V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks, 2020.

[7] TensorFlow. Transfer learning with keras, 2024. Accessed: 2024-11-24.