Utilizing Convolutional Neural Networks to Classify White Blood Cells

Authors: Ryan Fischbach and Alex Weinstein

I. Background

The medical field has benefited significantly from machine learning and deep learning. Specifically, the prevalence of image data and other patient health data has facilitated neural networks automating the classification of image data to save healthcare workers time. While looking for datasets, we were inspired by the recent image classification algorithms that can classify a patient as having COVID-19 or not having COVID-19 from chest X-ray scans. These Convolutional Neural Networks (CNNs) can help save doctor's time while diagnosing scans, allowing them to focus on other patients or other more important work.

II. Methodology

For this problem, we closely followed the methodology outlined in chapter 4.5 of *Deep Learning with Python*, which offers a workflow to solve a machine-learning problem:

i. Defining the Problem and Assembling a Dataset

Our problem statement is: what is the best type of deep learning model to classify scans from patient images and what performance can we achieve from this model? We are curious to see the resulting performance metrics and if this automated approach to identifying new samples from patients can perform well enough to justify this method being used instead of a professional going through all images individually. Additionally, we are interested in what convolutional bases from other pre-trained models can be implemented in a healthcare setting via transfer learning to maximize our performance metrics.

We came across a dataset of white blood cell images from Kaggle and decided to pursue classification via CNNs.

ii. Choosing A Measure of Success

We are classifying a scan as 1 of 5 classes. Accuracy seems like a good universal metric of comparison that we can use to judge our classifier versus other models.

iii. Deciding on an Evaluation Protocol

This dataset provides plenty of data via data augmentation. Thus, we will use a hold-out validation set to evaluate our model. If needed, we will try other techniques such as cross-validation to evaluate our model's performance.

iv. Preparing the Data and Developing a Baseline

This Kaggle project provides two datasets: a dataset of 366 raw jpeg images with a class imbalance and a dataset of 12,054 augmented images with a nearly perfect class balance. Initially, we chose the smaller dataset because we wanted control over our data augmentation. We felt this control would facilitate higher performance. However, the

class imbalance resulted in our classifiers assigning the predominant class label to minimize the loss. To counteract this, we decided to undersample all classes to have the same number of samples as the minority class, but this limited our data to 84 samples. Even with data augmentation, our models did not have enough information to learn, resulting in the same issue of assigning one class label to all samples. The only solution was to parse the included XML annotations to create bounding boxes, however, we decided to switch to the augmented dataset because of the class balance and more data.

The augmented dataset is divided into 4 relatively evenly distributed classes: Neutrophil, Eosinophil, Monocyte, Lymphocyte, and other/multiclass (significantly smaller). Each of these has roughly 2,500 images, culminating in a total of 12,054 images. Our preprocessing pipeline was composed of importing the data from Kaggle into a Google Drive folder, mounting the Drive within Google Colab, and then utilizing code provided by the author of the dataset to scrape the images from the Drive directory. This provided code goes through each folder in the directory and correctly assigns labels to each image. The images were then resized from 320x240 to 60x80, normalized, and appended to a NumpyArray array. The images were downscaled to aid in training speed as well as to remove the useless information provided by the background. The white blood cell in the image is a small portion of the total image with red blood cells around it. Thus, we aimed to remove extra pixels from the background that do not aid in the classification task. The data was now ready to be classified.

v. Preparing the Data and Developing a Baseline

The baseline we are using is ~25% accuracy. Randomly classifying the images into one of these 4 classes yields a 1/4 chance of being correct,. A naïve classifier would assign the predominant class label, achieving roughly 25% accuracy. We aim to beat this baseline using a low capacity, baseline CNN.

We developed a simple CNN with a convolutional base, a max pool layer, and 2 dense layers. The model we created in order to beat the baseline utilized 30 epochs, culminating in a peak validation accuracy of 56.44%, which is ~31% higher than our baseline. Having achieved this validation accuracy on our baseline model, we moved on to developing a model that overfits in an effort to encode more information.

vi. Creating a Model that Overfits

In order to create a model that overfits, we increased the number of convolutional layers, max-pooling layers, and dense layers to create and store more information about the images. Our overfitting model had 3 convolutional layers, 3 max-pooling layers, and 6 dense layers, with a total of 2,889,925 trainable parameters.

We utilized 30 epochs again, culminating in a peak training accuracy of 97.85% and a peak validation accuracy of 91.42%. This differential, along with the plateau of validation accuracy around epoch ~17 suggests that the model is overfitting. While it isn't overfitting by very much and the performance of this classifier is already very good, we want to maximize the generalizability of our model. If this model is used in a healthcare setting, we want to ensure it can be used on a variety of scans, even from different machines. Thus, learning the intricacies of the training data will hurt the

performance of the model when used on other data. Thus, we will now attempt to mitigate overfitting using various strategies.

vii. Regularizing the Model and Tuning Hyperparameters

To determine the impact of different hyperparameters on overfitting, we took the existing overfitting architecture and applied the different strategies one by one. This gave us a fair performance evaluation of each and indicated which strategies should be included in our final model for the data.

a. Adding Dropout

To create a model utilizing dropout, we first took the existing architecture of the overfitting model. In between the dense layers, we tested dropout. In other words, the layers will lose a percentage of their information while passing it to the next layer. This can help prevent overfitting by losing some of the information specific to the training set.

We trained this model using dropout = 0.2 for 30 epochs, yielding a peak 94.71% training accuracy and peak validation accuracy of 89.37%. The validation accuracy still plateaued but progressed through more epochs before doing so. This suggests that dropout between dense layers = 0.2 helped prevent overfitting, however, more work can be done. Thus, we will try another dropout model with a higher dropout rate to determine if it will mitigate overfitting.

We trained another model using dropout = 0.4 for 30 epochs, yielding a peak training accuracy of 88.32% and peak validation accuracy of 88.97%. The validation accuracy barely plateaued at the end, with the training and validation accuracies barely nearly identical. Thus, dropout = 0.4 will be used in a combined model to mitigate dropout.

b. Reducing the Capacity of the Network

To reduce the capacity of the network, we took the existing architecture of the overfitting model and removed the top two dense layers with the most nodes. This took the trainable parameters of the model from 2,889,925 to 759,237. This reduced the parameters by roughly a factor of 4.

This new architecture yielded a peak training accuracy of 97.74% and a peak validation accuracy of 93.46%. The validation accuracy was very high, but it did plateau around epoch 15 with high volatility thereafter. This discrepancy with the plateau suggests overfitting but did yield our highest validation accuracy yet.

c. Adding Weight Regularization

Using the existing overfitting architecture, we added weight regularization to each of the dense layers. Large network weights can suggest that small changes in the input produce large changes in the output, suggesting that the model is overfitting. In other words, the model increases its weights while learning the noise in the training data, which can't be generalized. To combat this, weight regularization punishes the network for having large weight values by considering the size of the weights in the loss function. Now, to minimize the loss, the network must also keep the weights small.

We began using an L2 activity regularization of 0.005 on the dense layers, yielding a peak training accuracy of 98.33% with a peak validation accuracy of 90.37%. This is a large gap and the training accuracy plateaus around epoch 12, suggesting overfitting. Next, we implemented kernel regularization for both l1 and l2 at 0.0001. We expanded the number of epochs to 50 to ensure that the model had enough time to learn the training data. This process kept the validation and training accuracy nearly identical during the training.

This model produced a peak training accuracy of 96.24% and a peak validation accuracy of 91.87%. The validation accuracy did plateau, however, it did so at higher epochs. This differential was smaller than other methods to mitigate overfitting, suggesting that weight regularization can help combat overfitting and thus should be used in our final model.

d. Utilizing Transfer Learning

To utilize transfer learning, we took several convolutional bases from two well known pre-trained CNNs and froze their convolutional layers. We then added the dense layers from our overfitting model and judged performance. The pre-trained CNNs tested were VGG16 and ResNet50. These are well regarded as the top pre-trained CNNs for image classification. Each was trained on different image datasets, with convolutional bases being modified for their specific tasks. We attempted to determine the best convolutional base for white blood cell scans, and more generally, medical scans.

VGG16 yielded the best performance, although it overfits drastically. This was expected as it had many more convolutional layers than our first overfitting model. VGG16 could be leveraged with the other techniques to potentially create our final model.

III. Results

Our final model was composed of 3 convolutional layers and 3 max-pooling layers, then flattened, with 2 dense layers with dropout in between. This totaled 423, 237 trainable parameters, which is significantly less than what was used in our overfitting model.

Layer (type)	Output Shape	Param #
conv2d_51 (Conv2D)	(None, 58, 78, 32)	896
max_pooling2d_50 (MaxPooling	(None, 29, 39, 32)	0
conv2d_52 (Conv2D)	(None, 27, 37, 64)	18496
max_pooling2d_51 (MaxPooling	(None, 13, 18, 64)	0
conv2d_53 (Conv2D)	(None, 11, 16, 128)	73856
max_pooling2d_52 (MaxPooling	(None, 5, 8, 128)	0
flatten_27 (Flatten)	(None, 5120)	0
dense_126 (Dense)	(None, 64)	327744
dropout_12 (Dropout)	(None, 64)	0
dense_127 (Dense)	(None, 32)	2080
dropout_13 (Dropout)	(None, 32)	0
dense_128 (Dense)	(None, 5)	165

Total params: 423,237
Trainable params: 423,237
Non-trainable params: 0

Figure 1: The summary of the final mode's architecture.

The final model achieved a peak training accuracy of 95.84% and a peak validation accuracy of 93.96%. There is very little discrepancy between the two, and we theorize this is due to the weight regularization being applied to the model, with other mitigation techniques helping as well.

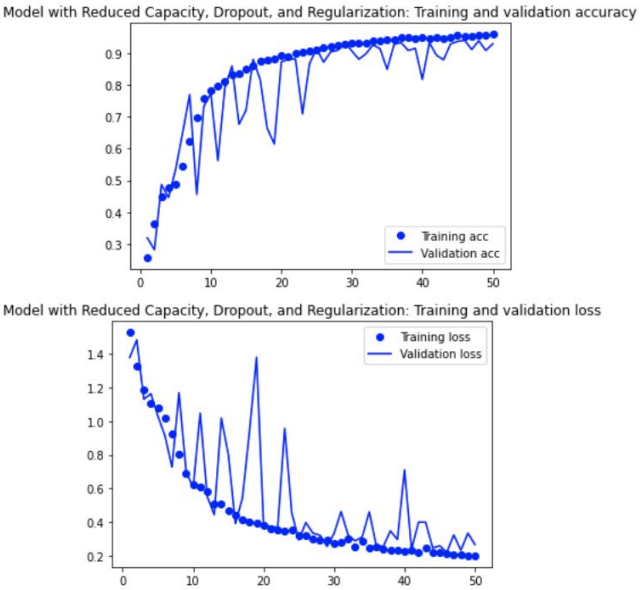

Figure 2: Graphs of the final model's training and validation accuracy as well as training and validation loss. The training and validation accuracy both rapidly improved the first 20 epochs, before slowly leveling out. The validation accuracy steadily increased until epoch 45, where it plateaued. Both losses slowly decreased, but training loss decreased smoothly while the validation loss was very volatile. This suggests that convergence did not take place until around epoch 45, where the number of validation accuracy swings decreased.

We trained our model for 50 epochs. While this is higher than the traditional 30 suggested by *Deep Learning with Python* and other resources, we determined that in order to have the best convergence, this model should be trained for more epochs to reduce the amount of volatility in the validation loss as well as narrow the gap between the training and validation accuracy.

	precision	recall	f1-score	support
NEUTROPHIL	0.65	0.83	0.73	624
EOSINOPHIL	0.84	0.76	0.80	623
MONOCYTE	0.95	0.78	0.86	620
LYMPHOCYTE	1.00	1.00	1.00	620
accuracy			0.84	2487
macro avg	0.86	0.84	0.85	2487
weighted avg	0.86	0.84	0.84	2487

```
[517,  85,  21,   1]
[145, 475,   1,   2]
[133,   6, 481,   0]
[  0,   0,   1, 619]
```

Figure 3: Classification Report and Confusion Matrix from the model's performance on the hold-out validation set.

The model's performance was analyzed by classifying the hold-out validation set, data that was never seen before by the model. The model averaged 84% accuracy, roughly 9% less than the peak validation accuracy. The model had the highest success at predicting the LYMPHOCYTE class, having precision and recall of 1. The model predicted 619/620 LYMPHOCYTE images correctly in the test set. The model had the second-highest performance with the MONOCYTE class but predicted 133/620 MONOCYTES as NEUTROPHIL. The EOSINOPHIL class had the second to worst performance, with 145/621 EOSINOPHIL scans being incorrectly classified as NEUTROPHIL. Lastly, NEUTROPHIL had the worst performance via precision and recall, with 85 and 21 NEUTROPHIL scans being incorrectly classified as EOSINOPHIL and MONOCYTE respectively.

IV. Analysis and Discussion

The first thing we noticed while developing a model was that very small networks were required to yield good performance with this classification task. This was likely due to the small image size, and the even smaller subset of each image with our target. Our targets are very similar in appearance, making the classification task even harder. Thus, a few pixels here and there are the important ones to correctly classify. For this reason, many dense would quickly overfit because there wasn't much information to learn. Additionally, many convolutional layers did not provide much information either because no additional layers would be able to provide information useful to the model. Thus, hyperparameter tuning became very important to yield a model with good accuracy.

On this note, transfer learning also did not provide benefit. Transfer learning models are largely trained on huge datasets, with large images, and are used to classify multiple items. Thus, their convolutional bases are very deep and have many trainable parameters. For a task like this, those additional frozen convolutional layers on top of the ones we already had did not add much information, making transfer learning nearly useless for this application.

Once we had achieved the rough estimate of our final model's architecture, things like regularization and dropout really helped overfitting in this context. We theorize that dropout helps remove some of the useless information about training carried by the model through layers, though other reasons could explain its performance benefit as well. Weight regularization penalized the model for having weights that were too large, also assisting performance.

Our final model struggled with differentiating the NEUTROPHIL, EOSINOPHIL, and MONOCYTE classes, but classified LYMPHOCYTES nearly perfectly. We hypothesize that the model struggled with those three classes because of the similarities in the scans of the white blood cell types. These three all have a purple shell with lower opacity with a more dense purple core in the middle. The LYMPHOCYTE class seems to only have this denser purple core, potentially helping in classification. For further development of this model, we would suggest providing a different image format or type of scan that could further help differentiate these images. Additionally, the Keras Functional API could be

used to provide multiple inputs on top of these images, potentially assisting in classification as well.

To answer the question of: “Can this model be used to automate white blood cell detection to remove the chore for medical professionals?”, our answer depends on the context. 84% test accuracy suggests that if the application is very dependent on getting each scan correct, this would not be a good model. If the classifier would be fed thousands of these scans in one go and the distribution of white blood cells need to be correct, this solution holds more promise. Ultimately, a medical professional will have a much higher accuracy classifying these white blood cells, so it is for them to say the importance of classifying every scan correctly.

V. References

Chollet, F. (2018). *Deep learning with Python*. Shelter Island, NY: Manning Publications.

Huilgol, P. (2020, August 18). Top 4 Pre-Trained Models For Image Classification with Python Code. Retrieved December 03, 2020, from https://www.analyticsvidhya.com/blog/2020/08/top-4-pre-trained-models-for-image-classification-with-python-code/

Mooney, P. (2018, April 21). Blood Cell Images. Retrieved December 05, 2020, from https://www.kaggle.com/paultimothymooney/blood-cells

Mooney, P. (2018, April 12). Identify Blood Cell Subtypes From Images. Retrieved December 05, 2020, from https://www.kaggle.com/paultimothymooney/identify-blood-cell-subtypes-from-images

ResNet pre-trained model. https://keras.io/api/applications/resnet/

VGG16 pre-trained model. https://keras.io/api/applications/vgg/