Project: White Blood Cell Classifier

Author: Jonak, Paul

Abstract

White blood cells are the workers of your immune system. There are 5 main classes of white blood cells, each responding to different conditions such as bacterial infection or allergic reaction. An over- or under-representation of any class can provide vital information as to a patient's illness or condition. The goal of this work is to build a neural network able to identify the class of a white-blood-cell from an image. A small dataset of images found on kaggle.com was selected for training and testing the model. Augmentation was used to ensure a sufficiently-large training set. The final model is a convolutional neural network with L2 regularization to prevent over-fitting. The final model reached a validation accuracy of 93% though this would be higher still if longer training were allowed. Manual inspection of predictions made by the model confirmed the model is accurately labeling white blood cells.

Table Of Contents

1 Introduction

Blood count, or complete blood count, is the process by which a blood sample is inspected to determine its components and their prevalence[1]. These blood components include red blood cells, white blood cells, platelets, and more. Counts of these components allow a doctor to monitor one's health and to diagnosis conditions. For instance, low counts of red blood cells, hemoglobin and hematocrit indicate anemia. The physical symptoms of anemia include fatigue and weakness.

This project will focus on white blood cells. White blood cells are the workers of the immune system. They are called upon for such tasks as fighting bacterial infection or responding to an allergic reaction. A bacterial infection will result in a high white blood cell count[1], whereas a viral infection is usually associated with a low count[2].

A diagnosis is aided by identifying what types of white blood cells are present in the blood sample. The main classes are basophil, eosinophil, lymphocyte, monocyte, and neutrophil[3]. Basophils act as a warning system for allergic and antigen response. Fighting allergic reactions and parasitic infections is the job of eosinophils. Lymphocytes respond to cancer and viral infections, and neutrophils handle bacterial and fungal infections. The last class, monocytes, are often thought of as trash collectors because they will eat debris from cells breaking apart. However they also pass on important information to certain lymphocytes to help lymphocytes recognize and defeat infections more quickly. Given the roles undertaken by each class, a doctor would want to know the overall white blood cell count as well as the counts of each class.

A blood count consists of taking a sample of your blood, identifying the cells present in the sample, and then extrapolating. Identifying the cells present and measuring their prevalence may be done manually or automatically using image analysis, flow cytometers, or other methods. If images were taken, then the class of each white blood cell may be determined visually. Below is a sample image of each of the main classes (Fig 1). The images were taken from the source dataset found on Kaggle.com, as described within section **5 <u>Dataset</u>**.

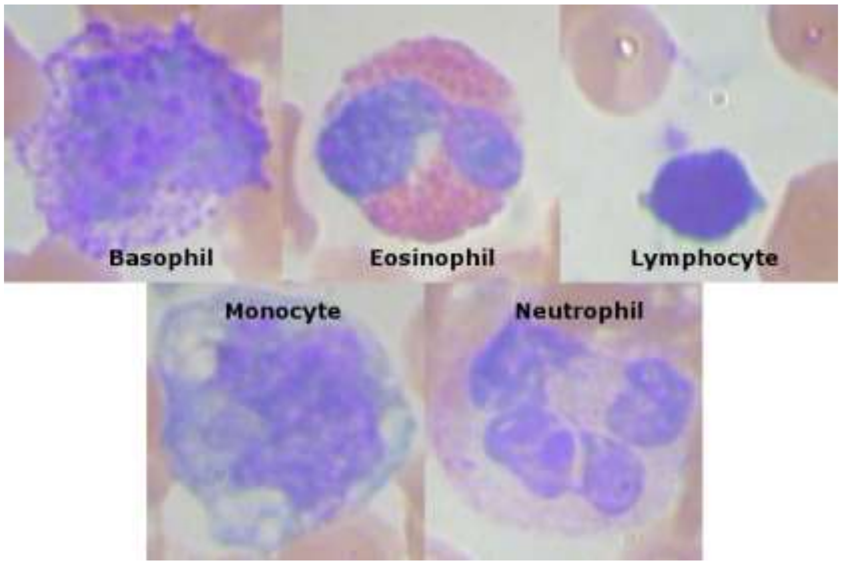


Figure 1: Sample images of each white blood cell class found in source dataset.

2 Problem Statement

Images were taken of a patient's blood sample to identify, characterize and count the various components present. Given an image containing a single white-blood-cell, classify what type of white-blood-cell is shown.

3 Description of Technology

The code was developed in Python 3,6 using Keras with Tensorflow as the backend. Both OpenCV and PIL were used for image handling though Keras' ImageDataGenerator class provided the image processing functions. Non-image data was handled and manipulated via Numpy and Pandas. Matplolib provided the visualization tools to plot training progress.

The machine running the code has two Nvidia GPUs. To take advantage of the CUDA cores present on each GPU, the GPU version of Tesnorflow was used in conjunction with Nvidia's CUDA Deep Neural Network library, cuDNN.

Installation and configuration instructions are available in **Appendix B: Code Setup**.

4 Description of Hardware

The machine used to build the model has the following configuration:

Component	Description
Operating System	Windows 10
CPU	AMD Ryzen 1950x (16-core, 3.4 GHz)
Memory	64GB DDR4
GPU	(2x) Nvidia GTX 1080 Ti (3854 CUDA cores, 11GB GDDR5X)

5 Dataset

5.1 Source Data

Source dataset comprises of approximately 400 images containing multiple red bloods cells and at least one white blood cell (see Figure 2). Ignoring images with more than one class, the frequency of each class of white blood cells within the dataset is given in Figure 3. The images are stored in the JPG format with datatype unit8 and dimensions 480 x 640 x 3.

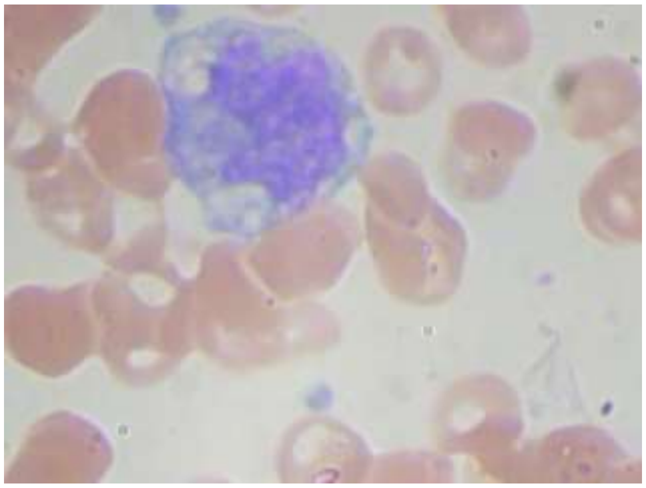


Figure 2: Sample image of a Monocyte and multiple red blood cells

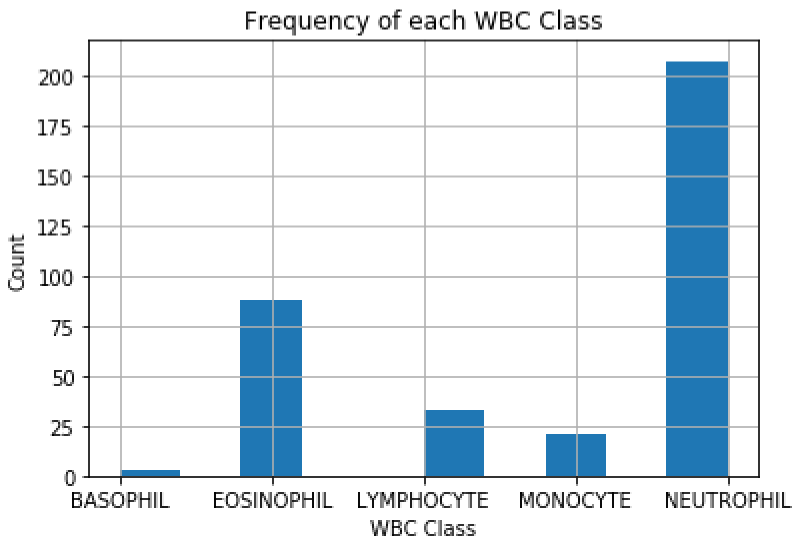


Figure 3: Histogram of the frequency of each class within the dataset. Images with multiple classes are ignored.

This collection of images and accompanying class labels were found on Kaggle.com under the dataset Blood Cell Images[4]. The direct download link to the data is:

https://www.kaggle.com/paultimothymooney/blood-cells/downloads/dataset-master.zip

The dataset was released into the public domain under **CC0: Public Domain** (https://creativecommons.org/publicdomain/zero/1.0/). The licensing of the data is given in https://github.com/Shenggan/BCCD_Dataset/blob/master/LICENSE and reproduced below:

5.2 Data Preparation

Data preparation includes

- Inspect data to see if anything should be removed
- Renaming each file according to the syntax CLASS.ID.jpg
- Augmenting the dataset
- Reorganizing the directory tree to have training and test folders, with each class having a separate folder
- Down-sampling the images
- Adjust pixel value range

To begin, a histogram of class frequency was built to inspect whether any classes should be removed. It was deemed the class Basophil has too few instances and therefore was removed. Next, the images were renamed such that they adhere to the syntax CLASS.ID.jpg. Augmentation was then used to create new images via ImageDataGenerator class from the Keras package. We set a target of 3,000 images per class. This figure is in line with the companion dataset included in the Kaggle listing, which has approximately 2,500 images per class. For the images generated for this project, the configuration for ImageDataGenerator is given in Table 1.

ImageDataGenerator Parameter	Value Used
rotation_range	180
width_shift_range	0.1
height_shift_range	0.1
shear_range	0.15
horizontal_flip	True
vertical_flip	True

Table 1: Augmentation parameters used with the Keras ImageDataGenerator class

After augmentation, we created the folders "train" and "test" to hold the training and test data, respectively. The training set will comprise 90% of the images from each class. Within each training and test set folder, there are subfolders for each class. All images associated with a particular class are then moved to their class folder. Having taken the above steps, the data is ready to be used with ImageDataGenerator flow() methods. The advantage to this approach is not needing to load all images into memory to train the model. Instead, the model is trained on batches of images.

With the 3,000 images per class, we built three datasets. The small set has 300 images per class and is used to investigate the structure of the model and the general size of each layer. The medium set has 900 images per class. This set is used for validating the model structure, fine tuning layers, changing parameters related to training, and making changes to address over-fitting. Lastly, the full set includes all 3,000 images per class and is used to build the final model. As a reminder, 90% of the images will be used for the training set.

When each image is fed into the model, we will want to lower the resolution and adjust the pixel value range. If we lower the resolution by a factor of 2, 4, and 5 then we will get the resolutions 240x320, 120x160, and 96x128, respectively. The resolution used in the Week 6 assignment was 150x150 therefore we opted for 120x160 as the nearest option. Lastly, we want all pixels to be within the range of 0 and 1. Given our images are of type uint8, then all pixel values will be divided by 255.

6 Model Building and Tuning

6.1 Initial Model

The initial model is adapted from course homework, the groundwork of which came from Francois Chollet's lab[5]. In Week6, we built a dog-cat classifier. The model structure is given below in Figure 4. For our initial model, we tested multiple variations of this structure that included or excluded layers conv2d_3, max_pooling2d_3, and conv2d_4. We also tested a matrix of values for the Conv2D size parameter of each convolutional layer across these variations. Additionally, we used various size parameters for the second-to-last dense layer, labeled here as dense_1. Lastly, the output from the last dense layer must be changed. Unlike the homework, we do not have a binary problem. Our last dense layer must have a size equal to the number of white blood cell classes. As we dropped the Basophil class due to lack of data, we have 4 classes. It follows that the last dense layer must be set to have an output of size 4, with 'softmax' as the activation function. Our model's loss function then becomes 'categorical_crossentropy' to account for these changes.

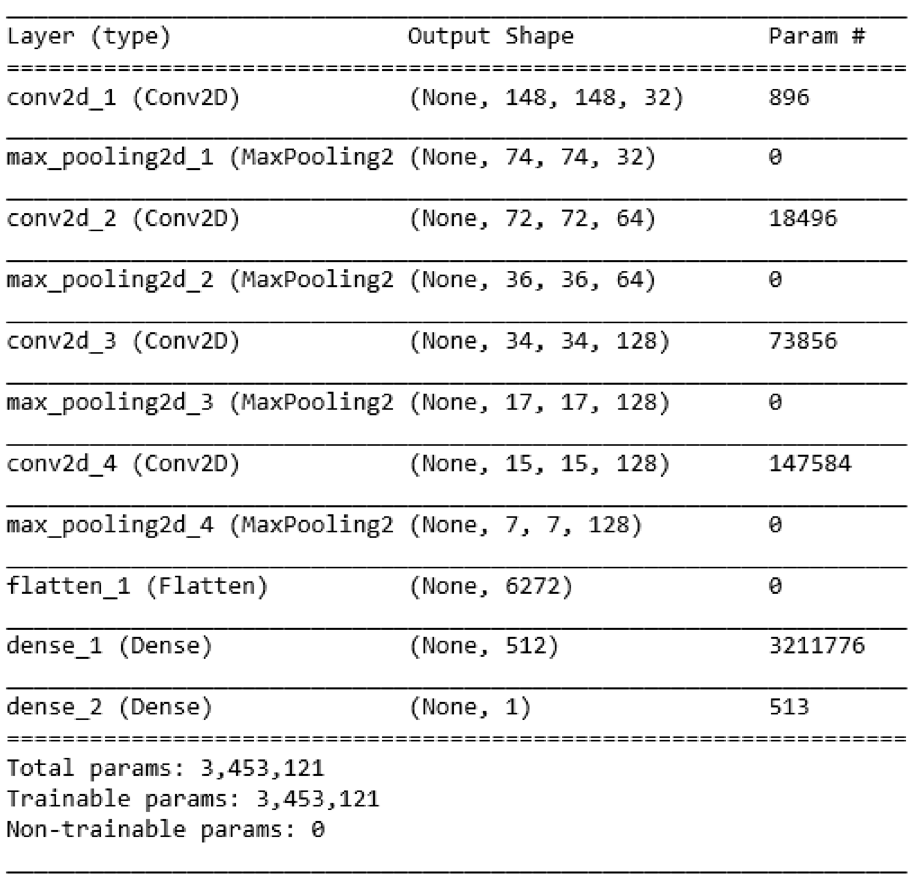
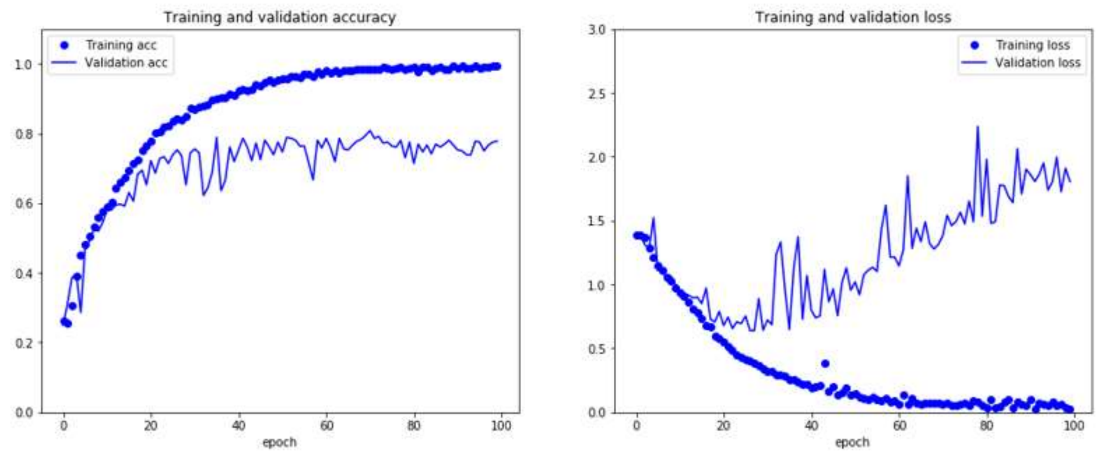

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 148, 148, 32)      896
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 74, 74, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 72, 72, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 36, 36, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 34, 34, 128)       73856
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 17, 17, 128)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 15, 15, 128)       147584
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 7, 7, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 6272)              0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               3211776
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 513
=================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
_________________________________________________________________
```

Figure 4: Model summary taken from Week 6 assignment

When using the small dataset of 300 images per class, we found the best performing models to have all four convolutional and three max pooling layers. Interestingly, the best results were found with much smaller size parameters for the convolutional and dense layers. In Figure 4, we see that the homework called for a dense layer with size 512. Our analysis revealed the optimum to be in the range 16 to 64, with 32 chosen after a second investigation using the medium dataset. Similarly, we had to lower the size parameter for the convolutional layers. The homework used 32, 64, 128 and 128 as the size parameter for the convolutional layers, in order from layer conv2d_1 to conv2d_4. We found optimal results with 8, 16, 16, 16 for the same ordering. The accuracy and loss plots are given below in Figure 5. These plots show the model has a validation accuracy near 80% which is not as high as we would like. The validation loss decreasing initially in line with training loss, as hoped for. After approximately 20 epochs, the validation loss rises steadily. This is clear evidence that our model is over-fitting the data. We will describe correcting this behavior in **6.3 <u>Over-fitting Correction</u>**.

```
dim_dense=32 | conv2d_1=8 | conv2d_2=16 | conv2d_3=16 | conv2d_4=16
```



```
acc=0.99 (val_acc=0.76) | loss=0.06 (val_loss=1.83) | lowest val_loss=0.80
```

Figure 5: Accuracy and loss plots for model (dense=32, conv2d=[8, 16, 16, 16]) with medium dataset

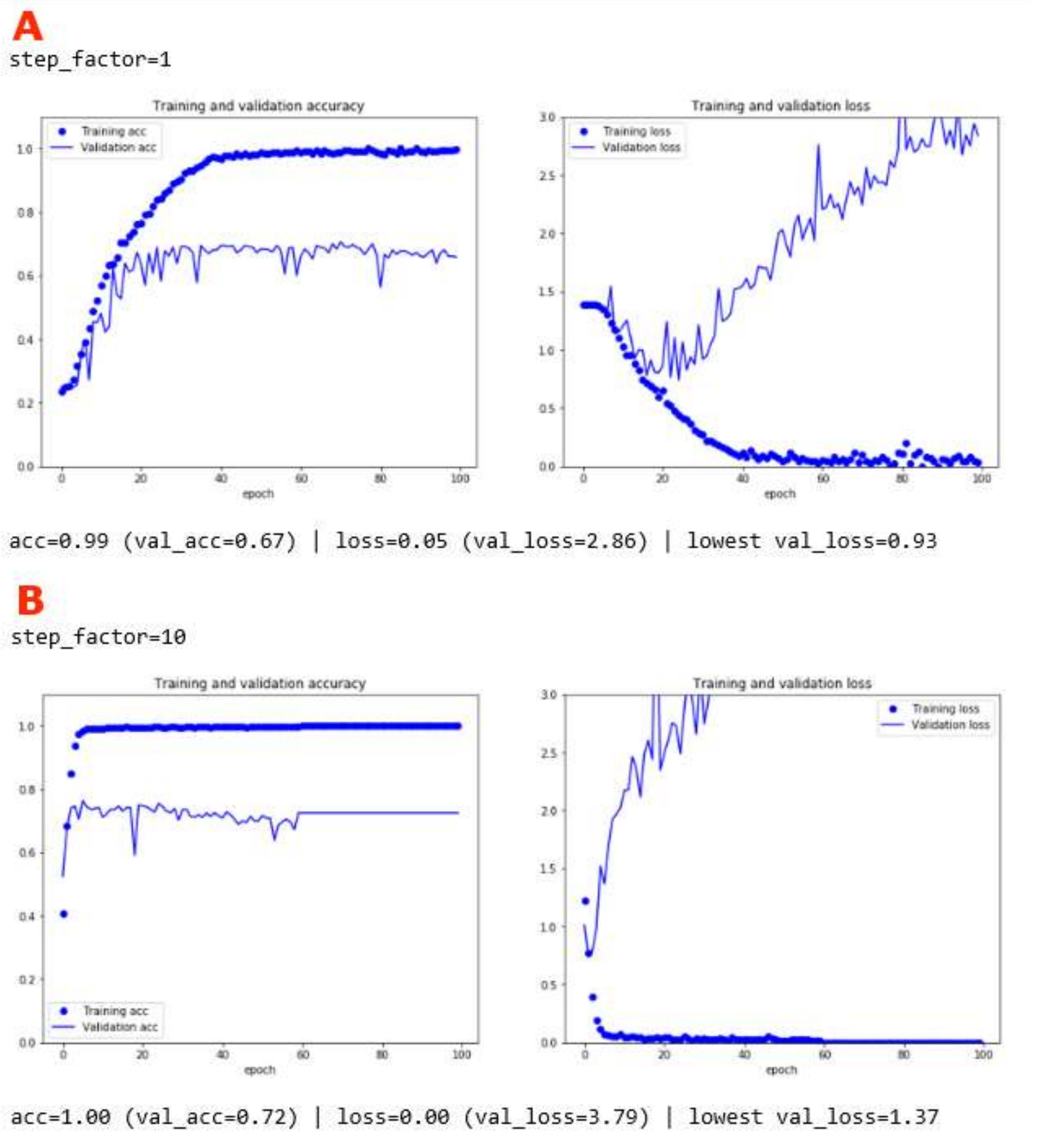
The changes to the size parameters result in a much simpler model in that there are far fewer trainable parameters as compared to the model from the homework. This is perhaps due to the inherent complexity of an animal's appearance compared to a cell. As an example, the stained cell stands out owing to its purple coloring whereas the dogs and cats have a variety of colors per image and across the images. There are also more complex shapes and textures to keep track of with animals.

6.2 <u>Adjust Model Fit Parameters</u>

Having established our model structure, we next investigated the optimal optimizer, number of steps per epoch, and the batch size of input images. For this project, we focused on the optimizers RMSProp, Adam and Nadam. These optimizers were selected as they appear to be widely used and because they automatically adjust the learning rate, therefore additional tuning is not required. We found the RMSProp optimizer to provide the best results.

When looking at the batch size of input images, we compared the sizes 16, 32, 64, 96 and 128. Above a batch size of 32, there is a steady drop in training accuracy and validation accuracy. It isn't clear why this would happen. Our best results appeared at a batch size of 16, which had the highest validation accuracy and lowest validation loss.

The third parameter tested was the number of steps per epoch. To test, we defined the number of steps as a multiple of the number of images for the dataset at hand. For instance, the small dataset has 300 images per class, with 270 per class devoted to training. With 4 classes, the training set holds 1080 images. The medium dataset, by comparison, has 3240 training images. If using the medium dataset, a step factor of 1 would give 3240 steps per epoch whereas a step factor of 2 gives 6480 steps. Above a step factor of 1, the model converges quickly at the cost of severe over-fitting, as shown in Figure 6. We opted to continue with a step factor of 1.



```
acc=0.99 (val_acc=0.67) | loss=0.05 (val_loss=2.86) | lowest val_loss=0.93
```

```
acc=1.00 (val_acc=0.72) | loss=0.00 (val_loss=3.79) | lowest val_loss=1.37
```

Figure 6: Training accuracy and loss when tuning steps per epoch. (A) Steps Per Epoch is set to the number of images in the training set. (B) Steps Per Epoch is set to 10 times the number of images in the training set. We see faster convergence with (B) but also evidence of severe over-fitting.

6.3 Over-fitting Correction

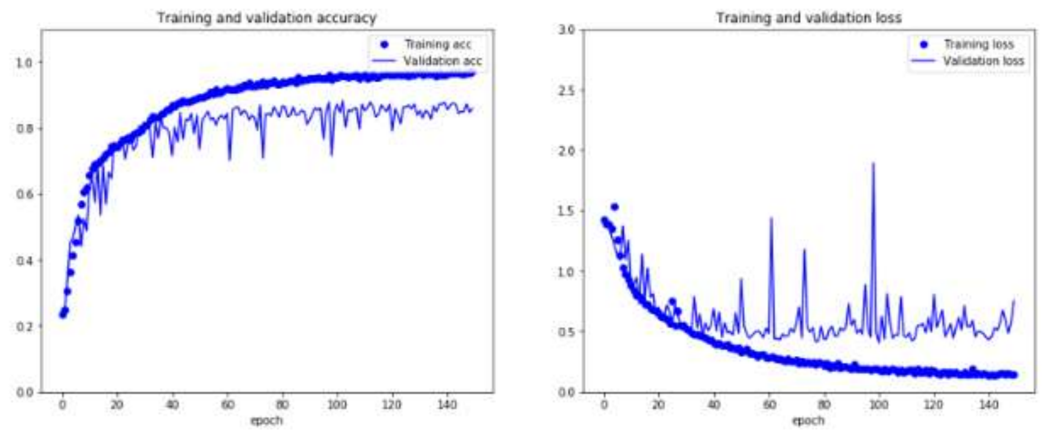
Evidence of over-fitting lies in the rising validation loss as seen in Figure 6. In our course, we employed both L2 regularization and dropout independently of one another to correct for over-fitting. These are the two approaches that were focused on for this project. We built one L2 regularization configuration, four different dropout configurations, and also two configurations that combined L2 regularization with dropout. Combining L2 regularization with dropout was found to lower accuracy for a given number of epochs. This approach may work well if the number of epochs is increased substantially, however the training time may become prohibitively long. This was not investigated further as a result.

When focusing on L2 regularization and dropout independently of one another, we find they both perform well after tuning. Figure 7 A highlights the best results seen with L2 regularization. Over-fitting no longer appears to be an issue. The optimal results with dropout are shown in Figure 7 B. We find dropout leads to faster convergence of the validation accuracy and lower validation loss than L2 regularization. The particular configuration shown here is "2_v2" which has two dropout layers. These dropout layers are placed immediately before and immediately after the first dense layer. It is not immediately clear why this dropout configuration performed better than the others.

Having found the optimal configurations for L2 regularization and dropout, the models were trained again on the full dataset. The plots of accuracy and loss are presented in Figure 8. With the full dataset, L2 regularization performs far better than dropout, whereas with the medium dataset the two options were nearly equally effective. With these results in hand, we have our final model.

A

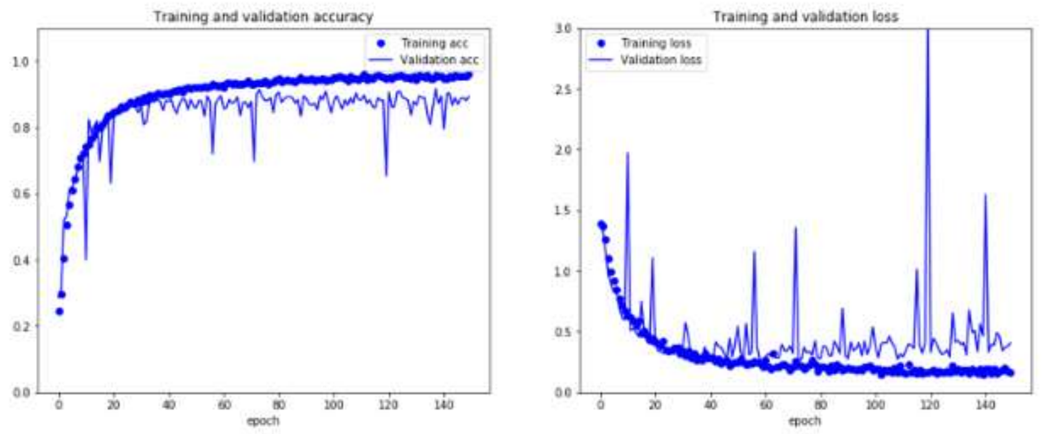
```
lambda_l2reg=0.01
```



```
acc=0.96 (val_acc=0.86) | loss=0.15 (val_loss=0.54) | lowest val_loss=0.61
```

B

```
dropout_2_v2=0.3
```

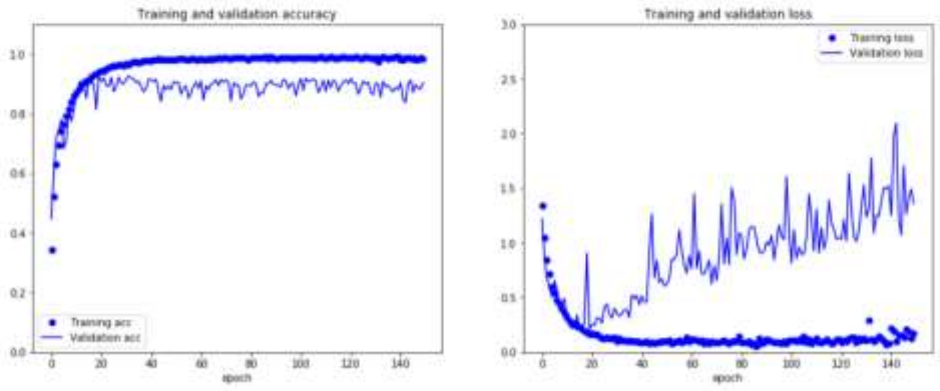


```
acc=0.95 (val_acc=0.87) | loss=0.18 (val_loss=0.48) | lowest val_loss=0.38
```

Figure 7: Accuracy and loss during training with the medium dataset. (A) Optimal results with L2 regularizaiton occurred for lambda value of 0,01. (B) The optimal dropout configuration was model "2_v2" with rate of 0,3.

A

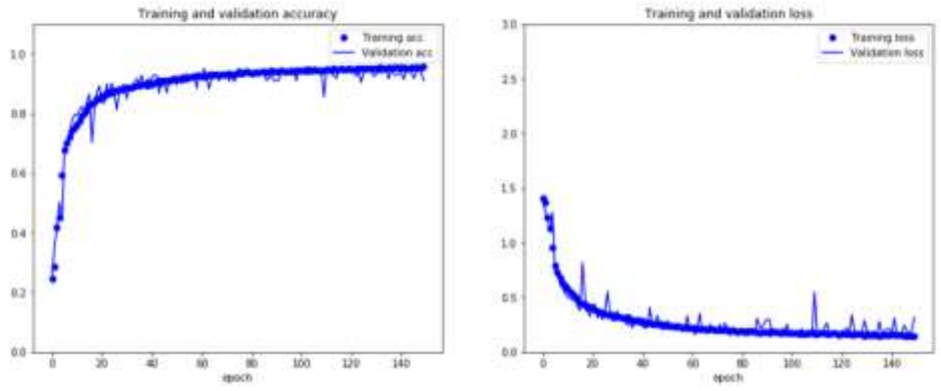
```
Standard model
```



```
acc=0.99 (val_acc=0.89) | loss=0.14 (val_loss=1.42) | lowest val_loss=0.34
```

B

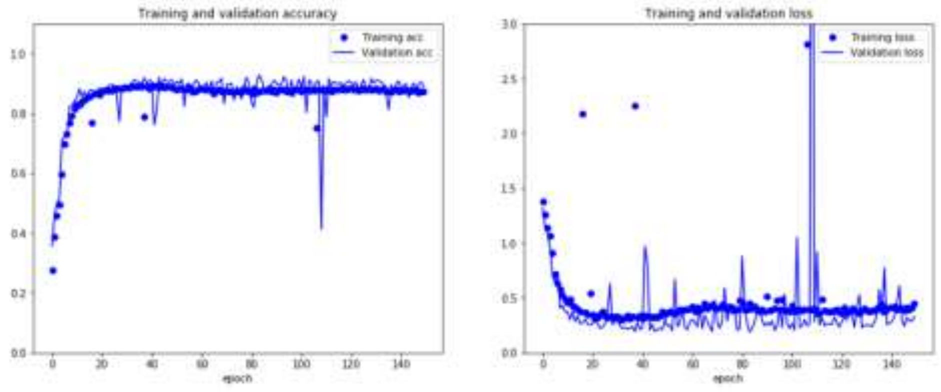
```
lambda_l2reg=0.01
```



```
acc=0.95 (val_acc=0.93) | loss=0.15 (val_loss=0.20) | lowest val_loss=0.21
```

C

```
dropout_2_v2=0.3
```

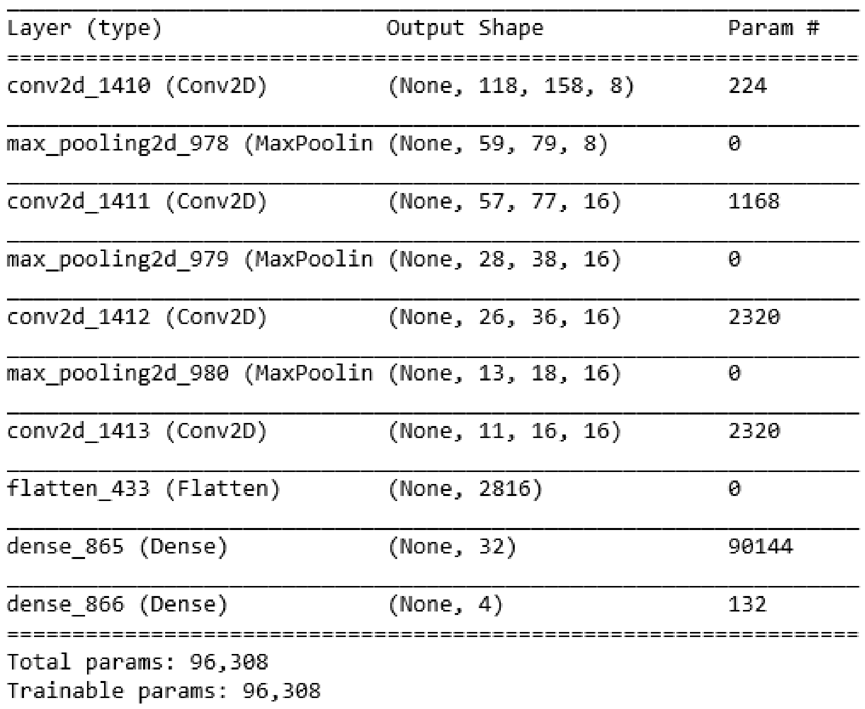


```
acc=0.88 (val_acc=0.89) | loss=0.40 (val_loss=0.35) | lowest val_loss=0.33
```

Figure 8: Accuracy and loss during training with the full dataset. (A) Standard model does not include L2 Regularization nor Dropout, as evident by increasing validation loss. (B) Model with L2 Regularization uses the optimal lambda of 0,01. (C) Model with Dropout uses config "2_v2" at dropout rate of 0,3.

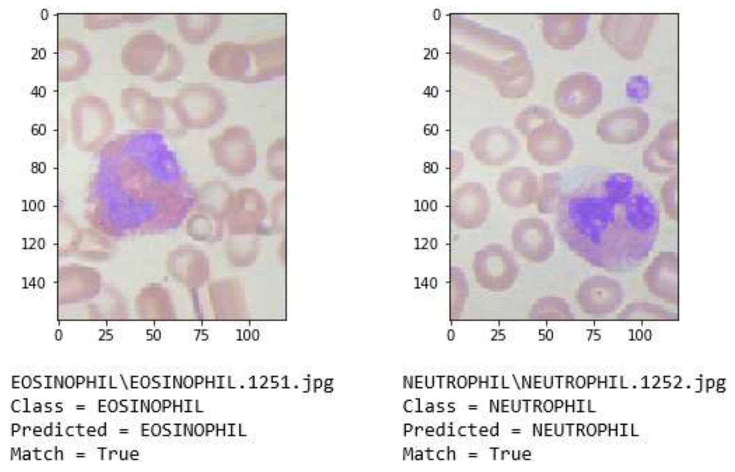
6.4 Final Model

Our final model is composed of four convolutional and three max pooling layers, followed by a flattening layer and two dense layers. The model summary is given below in Figure 9. To correct for over-fitting, we found L2 regularization to be most effective. Subsequently, the first dense layer has L2 regularization applied with a lambda of 0,01. Our final validation accuracy was found to be 93% with a validation loss of 23%. As seen in Figure 8 B above, the training accuracy has not fully converged after 150 epochs meaning we may be able to achieve even higher validation accuracies with larger number of epochs. To inspect the results, images from the dataset were randomly selected and fed into the model. For all samples tried, the model was able to predict the correct class of white blood cell (see Figure 10).



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1410 (Conv2D)         (None, 118, 158, 8)       224
_________________________________________________________________
max_pooling2d_978 (MaxPoolin (None, 59, 79, 8)         0
_________________________________________________________________
conv2d_1411 (Conv2D)         (None, 57, 77, 16)        1168
_________________________________________________________________
max_pooling2d_979 (MaxPoolin (None, 28, 38, 16)        0
_________________________________________________________________
conv2d_1412 (Conv2D)         (None, 26, 36, 16)        2320
_________________________________________________________________
max_pooling2d_980 (MaxPoolin (None, 13, 18, 16)        0
_________________________________________________________________
conv2d_1413 (Conv2D)         (None, 11, 16, 16)        2320
_________________________________________________________________
flatten_433 (Flatten)        (None, 2816)              0
_________________________________________________________________
dense_865 (Dense)            (None, 32)                90144
_________________________________________________________________
dense_866 (Dense)            (None, 4)                 132
=================================================================
Total params: 96,308
Trainable params: 96,308
```

Figure 9: Model summary for final model with L2 Regularization



```
EOSINOPHIL\EOSINOPHIL.1251.jpg        NEUTROPHIL\NEUTROPHIL.1252.jpg
Class = EOSINOPHIL                    Class = NEUTROPHIL
Predicted = EOSINOPHIL                Predicted = NEUTROPHIL
Match = True                          Match = True
```

Figure 10: Example of two sample images fed into the final model and the correct classes returned.

7 Next Steps

To further this project's goal, there are a number of areas to investigate, including additional preprocessing of the images, building new models, and changes to model training. Model training may potentially be improved with the use of callbacks. This was seen in the course during Week 11 when building a translation system. The callback served to compare the model at the current epoch with a previous epoch, and to proceed with the best version. It was not explored in the course whether this approach improves the final model, though it should allow for faster and more consistent training. Another option is to test different model structures. In our course, we found autoencoders to be quite powerful. This was not investigated here owing to time constraints.

At a more fundamental level, it's possible that modifying the preprocessing approach could lead to improved results. A straightforward modification would be to convert the input RGB images to greyscale. This was not tried here in the belief the distinct coloring of the white blood cells would aid the model in isolating class-specific features. Perhaps more promising would be to use image processing to isolate the white blood cells entirely first, and then feed them into the model. The downside to this approach is the need to develop a robust image processing routine which may prove difficult. Additionally, this may add bias into the model as the user is handling a portion of the work that the model would have to do otherwise. Lastly, the impact of image resolution should be investigated. The selected resolution was based on resolutions used in the course. Using resolutions that are larger or smaller may have a substantial impact. For this project the original resolution was also tested but the GPUs did not have sufficient memory to train the model. It is possible to re-test without the GPUs active.

8 Lessons Learned

The approach for this project was to find a practical example from the course, apply the example to the problem at hand, and then refine the model. The course example was from the Week 6 homework where we built a model to classify images of dogs and cats. The convolutional neural network used in the homework proved to be a good foundation for this project. The lesson learned is that one is able to build a reasonably accurate model by finding and adapting pre-existing models that are relevant to the case at hand.

It was also learned that planning and significant time is required for model tuning. We looked at changing the number of layers, the parameters for each convolutional layer and dense layer, and more. These parameters were mostly investigated independently of one another. With more resources, one matrix of parameter values would have been built to test for interaction across parameters. This was partially done at the beginning when changing the size of the convolutional and dense layers. We were able to speed up this particular test initially by using the smallest subset of the dataset. We then moved to the medium dataset to ensure test results would be more in-line with what we would see using the full dataset.

For some tests, it wasn't immediately clear which configuration was best. We decided to focus on the final validation accuracy and the lowest point on the validation loss plot. The lowest validation point is of interest because we were seeing over-fitting. Subsequently, if we would be able to correct over-fitting, then the validation loss to be expected would be near or lower than the lowest point on the current validation loss plot. In other words, we are using the lowest validation loss as a proxy for our expected validation loss after correcting for over-fitting.

We turned to the course homework for examples of how to correct over-fitting. In our course, L2 regularization and dropout were used. It was found that both could be effective but tuning is required. For instance, four different model configurations were tested with dropout. Surprisingly, the configuration taken from the homework performed the worst. We conclude that multiple dropout layer configurations with varying dropout rates must be tested to identify what works best for each problem. Alongside this sentiment, L2 regularization should also be included in the testing though this adds another set of tuning trials.

The last lesson learned is that we can build a white blood cell classifier with reasonably high accuracy. We were able to reach 93% accuracy with the validation set. It may be possible to reach a higher accuracy by following some of the suggestions in **7 Next Steps** but for the present work we are quite happy with these results.

9 References

1) "Complete Blood Count." Mayo Clinc, https://www.mayoclinic.org/tests-procedures/complete-blood-count/about/pac-20384919
2) "Low White Blood Cell Count." Mayo Clinc, https://www.mayoclinic.org/symptoms/low-white-blood-cell-count/basics/causes/sym-20050615
3) "What Are White Blood Cells?" University of Rochester Medical Center, https://www.urmc.rochester.edu/encyclopedia/content.aspx?ContentTypeID=160&ContentID=35
4) "Blood Cell Images." Kaggle, https://www.kaggle.com/paultimothymooney/blood-cells
5) "5.2 – Using Convnets With Small Datasets." Francois Chollet

Appendix A: Youtube URLs

YouTube URLs

Short: https://youtu.be/Ma6As_9qeGM
Long: https://youtu.be/kaxgFZIQDNk

Appendix B: Code Setup

The code developed for this project is publicly available at:

https://github.com/pjonak/Classify_WhiteBloodCells

To run the code, you with need python 3,6 and following packages:

- cv2
- Keras
- Matplotlib
- Numpy
- Pandas
- PIL
- Tensorflow

It is recommended to begin by installing Anaconda. Anaconda is freely available at https://conda.io/docs/user-guide/install/download.html. Once Anaconda is installed, packages may be installed by using the following command within Anaconda Prompt

```
conda install PACKAGE_NAME
```

However, before beginning, please first follow the instructions for installing Tensorflow. The instructions are available at https://www.tensorflow.org/install/. These instructions will ask you to create a new virtual environment, activate the environment, and then install Tensorflow. If you are using GPU processing, then you may also have to install Nvidia's cuDNN libraries. Once these steps are done and Tensorflow has been installed, you may then proceed to install the remaining packages in your new environment. As a last note, the 'cv2' package should be installed via 'conda install –c conda-forge opencv'.