Joseph Erickson

Nathan Mathews

Anthony Keaveney

December 10th 2019

CSC 461 Final Report

Blood Cell Type Classification Using DenseNet and Genetic Algorithms

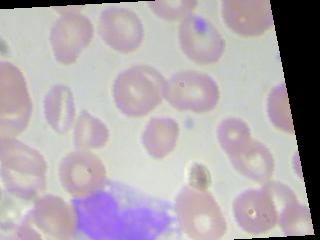
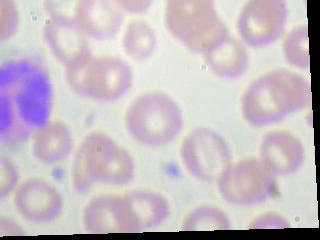
Medical science is a fast-growing field servicing an ever-growing population. Reducing the time needed for lab sample processing can be crucial to administering effective treatment for patients.  Automation in the lab can help reduce these times and increase efficiency of patient care. This all led to the question we wanted to try an answer for our final project, Can you train a neural network to distinguish between different types of blood cells?

 We used the Blood Cell Images Kaggle dataset[1] as the foundation for our implementation. The dataset contained 12,500 images of blood cells with cell type labels. The images were of four different evenly distributed cell types, Eosinophil, Lymphocyte, Monocyte, and Neutrophil. All of these images were also grouped into different folders according to cell type.

We then began with 9957 training images and 2487 testing images from the original data set. Below are images from the original dataset of the four blood types.

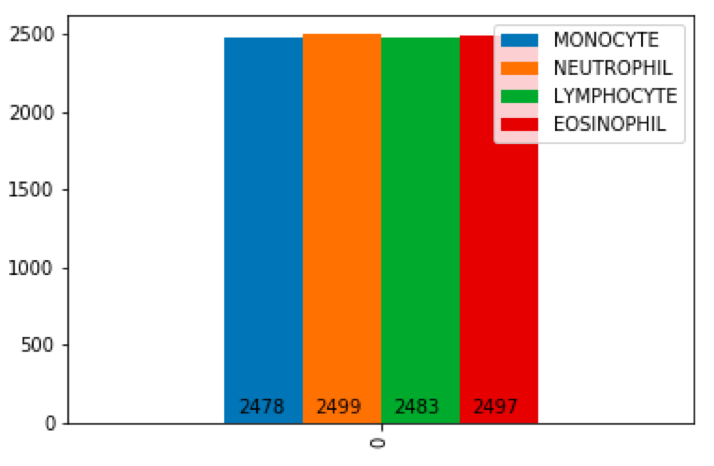
 

Eosinophil Lymphocyte

Monocyte Neutrophil

Our model then transformed the data by applying random rotations. While our model was already randomly rotating images, we decided to also rotate more images on our own. We did this to then add more images to our training set for further testing. We were then working with 99,570 training images, 9957 validation images, and 2487 testing images. The original size of the image was 320x240 which we decided to scale down to 40x30 to increase the speed of our model since we knew we were going to have a ton of testing to do. We then prepared our model to then normalize the images to get them on the proper scale of being in the range of zero to one.

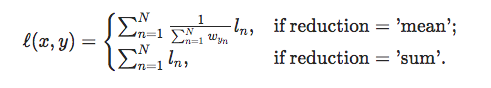


This graph here was used to check the even distribution across the original data set.

Once we finally had our data all set up and ready to use, we then decided to test out various approaches for initial results. Two approaches of ours were using regression and support vector machine. Since we were running tests on such large amounts of images, we found some of our testing to take multiple hours to conclude and it wasn’t efficient for us to get the best results. We then decided to work together using Google Collab running it on the GPU for quicker runtimes. Next, we would experiment and fine-tune hyperparameters for a neural network approach.

These hyperparameters consist of the number layers, layer widths, batch sizes, number of epochs, and validation percent. We then decided Implement a genetic algorithm to find hyperparameters that maximize testing accuracy.

Next was for us to start preparing our model. We used DenseNet161 [2] neural network with five linear hidden layers. DenseNet is used to connect each layer to every other layer in a feed-forward fashion. While trying to find out which loss function would be best for our model, we tried testing with Cross Entropy Loss, Multi Margin Loss, and The Negative Log Likelihood loss [3].Below is PyTorch’s implementation of the NLL Loss.

F:\NLL.png

After several test runs, we found our best accuracy to come with the negative log likelihood loss function with the Adam Optimizer.  We then found our best hyperparameters to be with 50 epochs and a batch size of 800, ~9% validation set. The linear layers in the network are used in conjunction with LogSoftMax. Our Genetic Algorithm [4] was then prepared with generations of 10, and each member a set of 5 layer widths. The top three performing members were then carried over to then populate over the next generation. We also set it up for one member of each generation to be generated randomly. The idea behind this was that we wanted to keep breeding the top-performing members until we reached optimal results. Below is a Genetic Algorithm flowchart which helped us with our implementation.

A close up of text on a white surface

Description generated with high confidence

A picture containing text, map

Description generated with very high confidence

Our results finally then told us that our best performing model with this technique achieved 95% training accuracy and 48% testing accuracy. Although, we did have another method which achieved approximately 60% testing accuracy, we decided to remain with this one. We saw low training and validation loss, along with high testing loss which proves that we are overtraining our data. The genetic algorithm then proved to do a little better than random weight values. We then plotted graphs to see how our results over the course of the epochs.

A screenshot of a cell phone

Description generated with very high confidence

After analyzing our results, we then began to discuss. Shrinking images drastically increased computed speed but lowered our testing accuracy by about 10%. In our case, we wanted to see results as quickly as we could without having to wait hours for our computers to run. In the future, using a more powerful machine could be much help in testing. Increasing the number of training and validation images improved training accuracy but not testing accuracy. The Genetic algorithm hyperparameters need to be more fine-tuned for this

specific problem. We also would have likely needed more time to run this algorithm to be effective.

We also were  unable to find optimal DenseNet hyperparameters for this problem, but from doing other research online, we saw achieving a testing accuracy of 90% is possible[5]. Considerations we could have for future research include Multi-class classification, Image detection[1], an SVM implementation, and Variations on hidden layers.

References

1. https://www.kaggle.com/paultimothymooney/blood-cells/kernels

2. https://pytorch.org/hub/pytorch\_vision\_densenet/

3.https://medium.com/udacity-pytorch-challengers/a-brief-overview-of-loss-functions-in-pytorch-c0ddb78068f7

4. https://towardsdatascience.com/genetic-algorithm-implementation-in-python-5ab67bb124a6 5. https://www.kaggle.com/drobchak1988/blood-cell-images-acc-92-val-acc-90

6. https://pytorch.org/docs/stable.htm