Chris Farris
Final Project

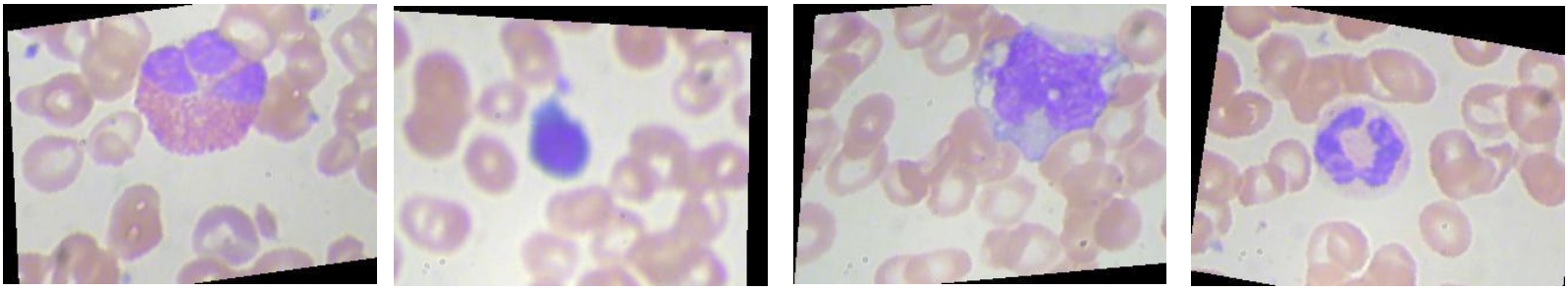
Classifying Blood Cell Images with Deep Learning

Introduction

Medical imaging is an extremely important field due to its ability to diagnose diseases, viruses, trauma, and other harmful conditions. X-rays, CT-scans, MRI, Ultrasound, and modern microscopes are several examples of the applications of imaging technology. While generating these images is performed mostly on the backs of machines, assessing the images and analyzing them is mostly performed by human experts. With the advancement in deep learning over the past decade, newer technologies are arriving to allow machines to also assist with this interpretation. Image recognition systems present an opportunity to aid human experts in diagnosis by quickly processing images and providing insights immediately.

In this project, I use images of white blood cells to train a neural network to recognize different types of such cells. I explore different deep learning techniques and demonstrate the capabilities of such systems in the medical domain. I examine the performance of optimizers, regularization, and architecture. The data is from the Blood Cell Count and Detection (BCCD) dataset under MIT license and is sourced from Github[1]. It contains nearly 10,000 training images of white blood cells and nearly 2,500 validation images. Labeling is stored in xml format. There are four classes of blood cell present: Neutrophil, eosinophil, monocyte, and lymphocyte.

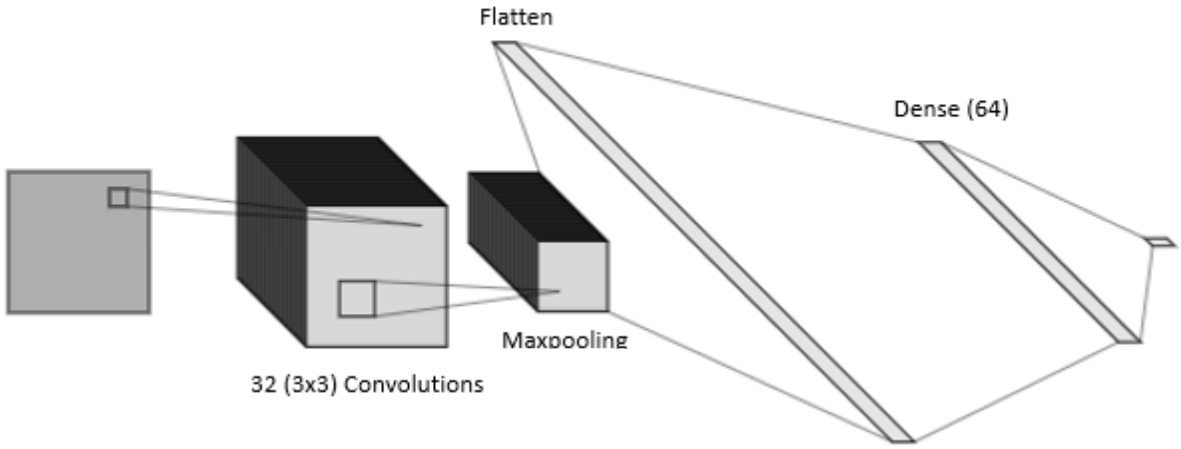
The deep learning in this project is conducted in Jupyter with Keras running tensorflow with GPU processing on a Nvidia Geforce 1080.



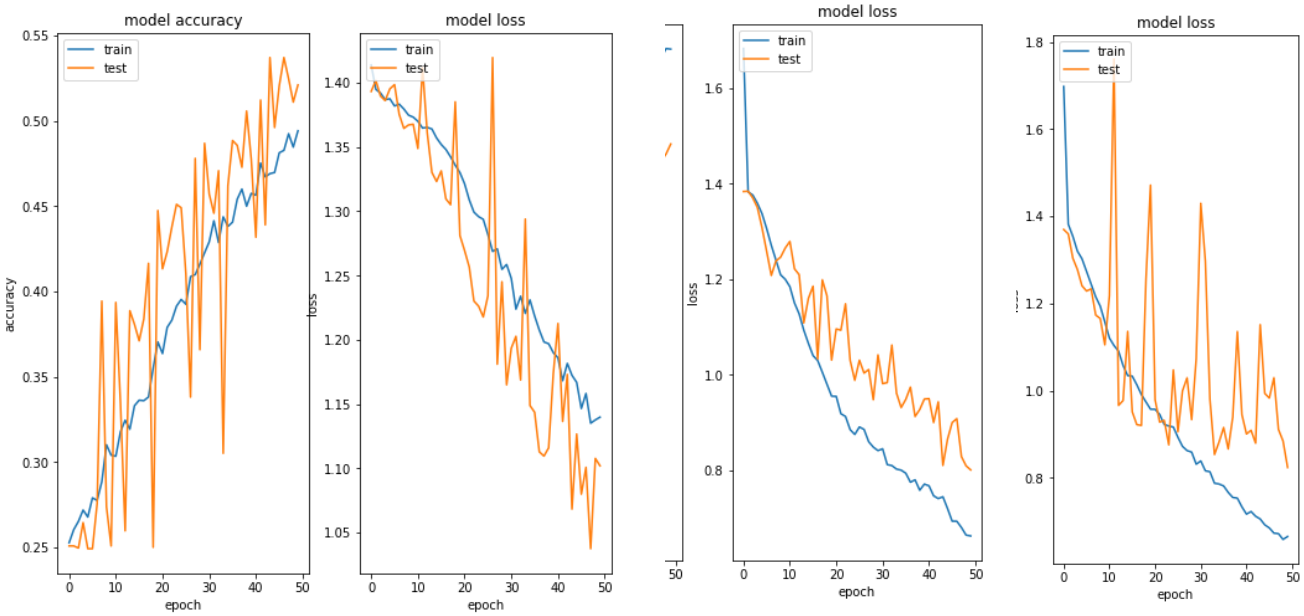
Four type of white blood cells, Neutrophil, eosinophil, monocyte, and lymphocyte dyed in blue and surrounded by red blood cells and platelets.

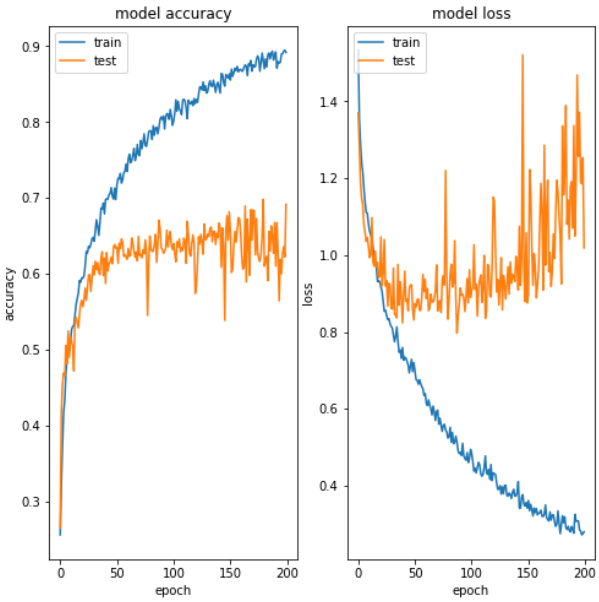
Optimization

I begin this project with a basic network architecture to assess capacity needs for the problem and to use for basic tests with optimization and regularization. This architecture consists of a single layer of convolutions, a maxpooling layer for dimension reduction, a dense layer, and the output layer.



I test three optimization algorithms beginning with standard Stochastic Gradient Descent with a learning rate of 0.01 followed by RMSProp and Adam with learning rates of 0.001 each. The training results can be seen below starting with SGD at the top left and working right and down in order mentioned.



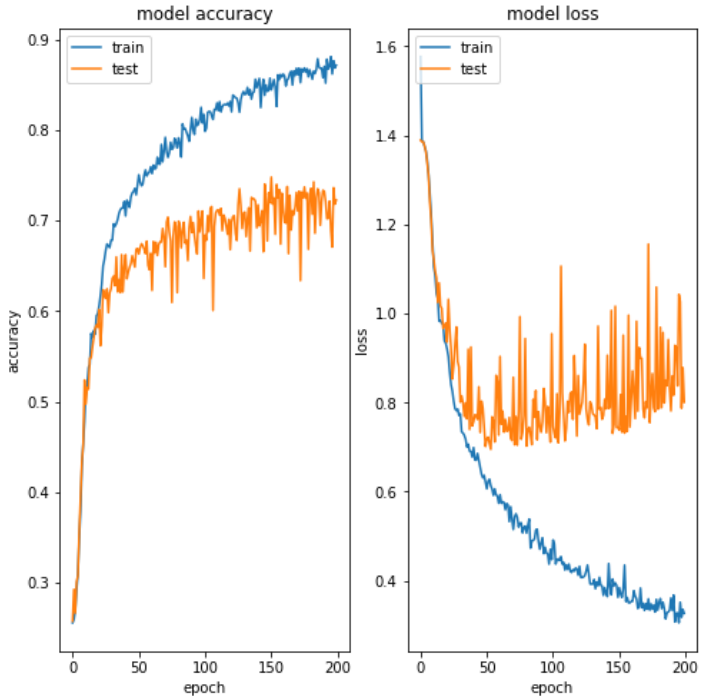


SGD provides consistent learning but is slow and after 50 epochs has unimpressive accuracy. However, both RMSProp and Adams have faster convergence, reaching 60% accuracy after 50 epochs. Another advantage of these is their stability though Adams appears to be the superior of the three. Thus, I will use Adams optimization for future experiments.

Regularization

A major issue in deep learning is overfitting the model. If we run the training with Adams for 200 epochs, we can see an example of overfitting:

In this case, our training accuracy continues to improve but out test accuracy stops improving and degrades in stability. The loss of our test data is even worse as it begins to diverge. This is a sign that the model is just memorizing answers and is not generalizable. We could either stop the learning at around 50 to 100 epochs to prevent this or we could add some regularization techniques to address this issue. The first is dropout[2] layers. Dropout randomly drops some outputs of a layer during training. This loss of information forces the model to try to learn broad features which are useful in generalization rather than memorizing data properties. One must be careful, however, to not set the rate of dropped neurons to be too high or the model won't be able to learn properly. This usually requires experimentation and some trainings to optimize. Playing with the dropout rate, I found that a 20% drop rate was roughly optimal in reducing overfitting but still allowing decent learning speed as seen below.



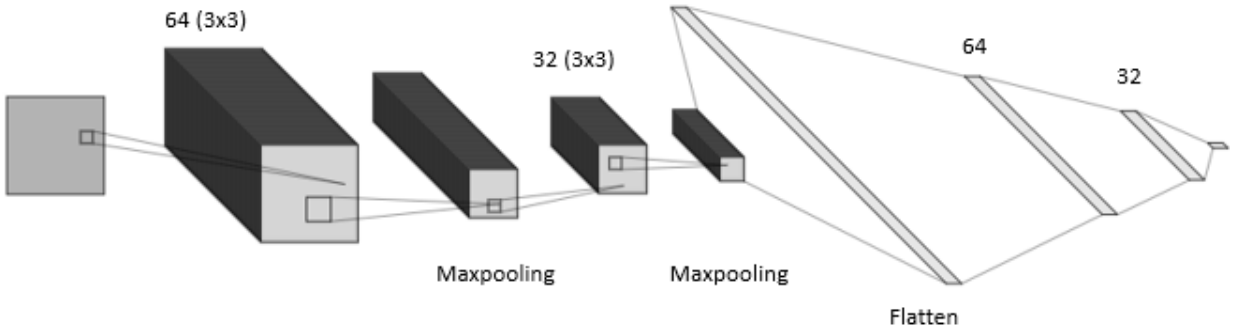
Compared to the previous training run, the addition of dropout has not only reduced the error of overfitting but has actually helped increase the test accuracy. Again, this is due to the generalized features the model is being forced to try to learn.

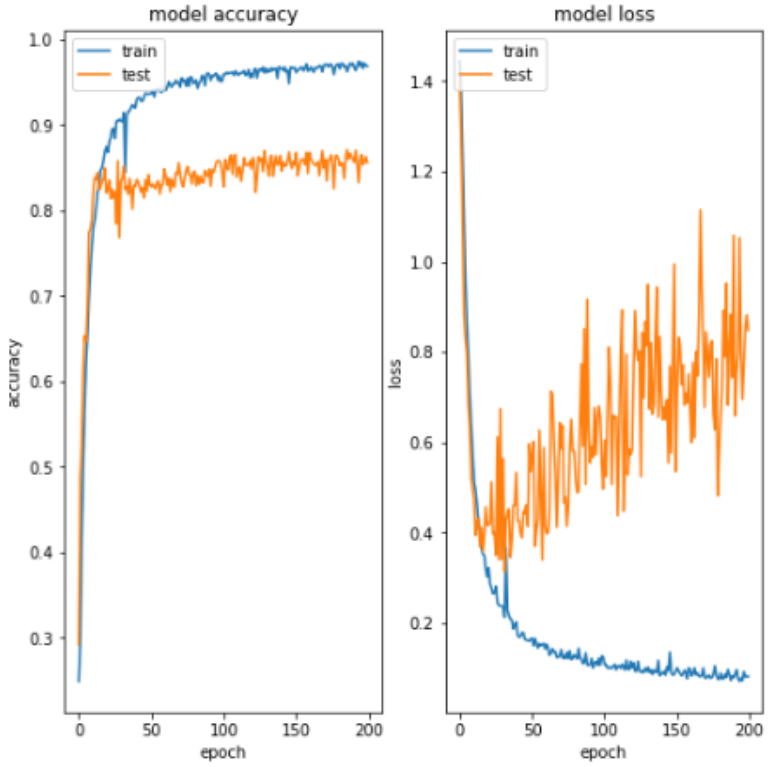
Another technique for combatting overfitting as well as improving performance is Batch Normalization[3]. Essentially, this technique attempts to normalize the output values of each layer such that they have zero mean and unit variance. Unfortunately, use of this technique introduced significant instability in the learning and was abandoned.

A final regularization method is feature-centering. This method is similar to batch normalization; however, it is only applied to the initial data before being ingested into the model. This method proved unreliable and heavily dependent on the initialization of the network. In some initializations, it helped accelerate convergence while in others it prevented it entirely.

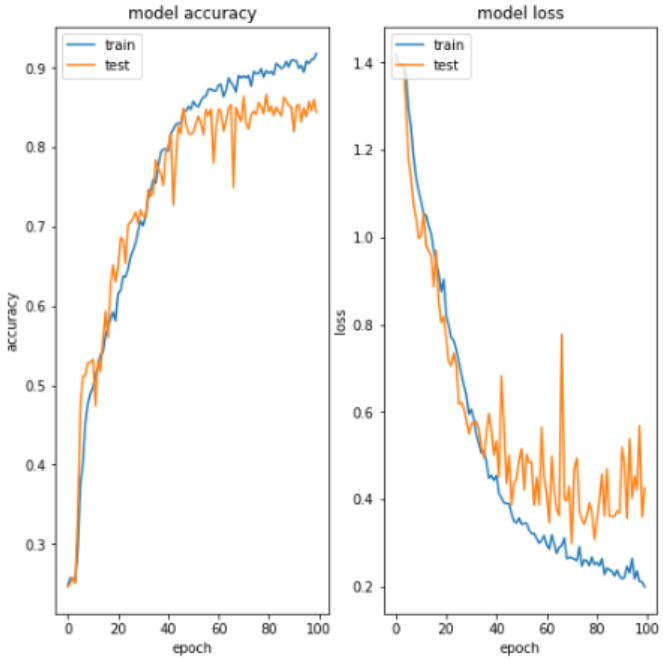
Architecture

In experimenting with architecture, I tested adding more layers, different numbers of kernels, different kernel sizes, and different dropout levels. There were some different convergence behaviors, but many models performed similarly. One interesting change is with feature centering. As explained previously, feature centering was inconsistent in the test model. I attempted to use it again in the new architecture below.



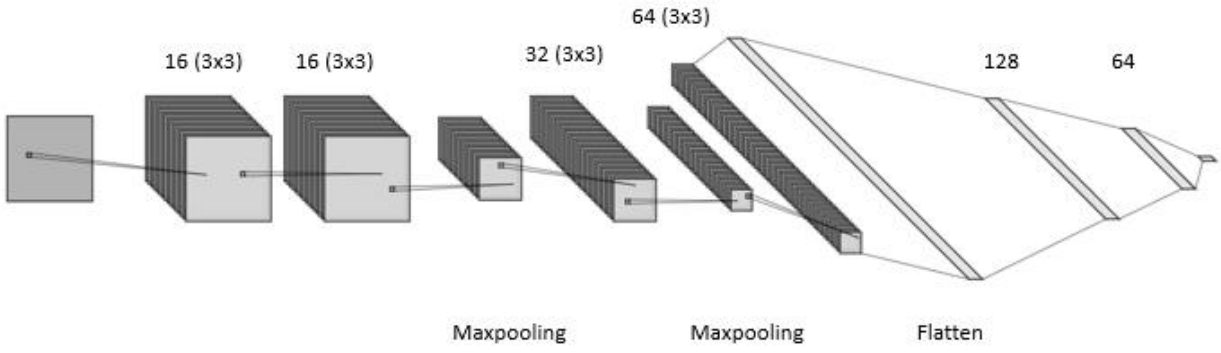


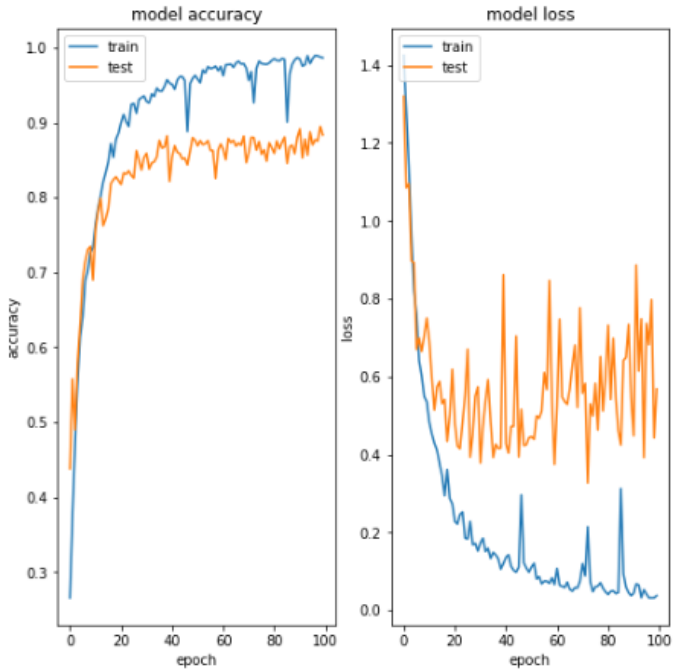
The stark improvement by the additional layers is apparent. However, looking at the divergent loss, it appears there is some overfitting occurring. Interestingly, when I repeat the training but don't use feature centering, the loss is more stable, and I achieve roughly the same accuracy:



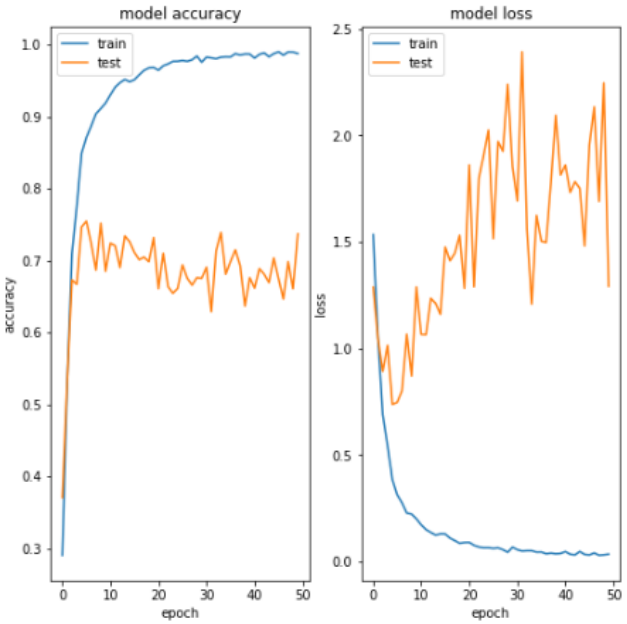
Feature centering sped up the training but at the cost of overfitting later. After witnessing inconsistent behavior in the simple model and overfitting in the expanded one, I abandon using feature centering for future experiments.

Some improvements can be made by increasing the number of layers again:





This architecture proved to be the most accurate. Additionally, convergence was faster, though the loss was slightly less stable. Many additional designs were tested including hyperparameter adjustments, but none improved upon this accuracy. In addition to testing my own architectures, I tested an existent architecture, VGG[4], with transfer learning. VGG is well known and simple network structure composed of many layers of 3x3 convolutions to build to a robust system. Keras includes published pre-trained network weights from the ImageNet dataset. In this test, I preserve the base layers of the network which have already learned standard image shapes like lines. I free the upper convolutions for learning and retrain on the blood cell data. The hope is that the pretrained lower weights will heavily accelerate learning. This model also has far more capacity than my own networks, so I also expected better accuracy. Below are the results on training VGG:



Unfortunately, while VGG did accelerate and increase learning on the training set, the test set suffered from massive overfitting. The capacity of VGG was overkill for this problem and hampered learning.

Conclusion

Deep learning image recognition tasks are typically interesting problems, and this was no exception. Medical imaging and diagnosis are important capabilities and creating automated or machine-assisted processes would dramatically improve healthcare. Blood cell classification is not a trivial task. Though white blood cells have defining traits, the imaging quality and orientation of the frame can obscure these. Additionally, the image contains multiple objects other than the white blood cell subject in the form of red blood cells and platelets. This noise imagery is close in appearance to the subject trait imagery can is a source of possible confusion in the system. Combining convolutional neural networks, maxpooling, and dropout, I achieved roughly 85% classification accuracy in a stable model which can be trained efficiently. Though standard models like VGG enabled overfitting, future work could be possible with further experimenting with layer composition and regularization techniques.

References

1. Shenggan, Chen. https://github.com/Shenggan/BCCD_Dataset

2. Srivastava, Hinton, Krizhevsky, Sutskever, Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting (Journal of Machine Learning Research 15, 2014). https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

3. Ioffe, Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift (Cornell University,2015) https://arxiv.org/abs/1502.03167

4. Simonyan, Zisserman. VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION (University of Oxford, 2015) https://arxiv.org/pdf/1409.1556.pdf