Malaria Detection

Wesley Stanis

INTRODUCTION

Image recognition can be very powerful and useful for many different assignments. Some of the tasks may be facial recognition, media marketing, and healthcare assistance. This report is about a model used to detect malaria in images of blood cells. The model that is used for this analysis is a convolutional neural network.

Malaria is a disease that is caused by a parasite and can be fatal. It is commonly transmitted by mosquito bites, and effects approximately 2,000 people in the U.S. every year. Treatment for patients may vary, but treatment involves prescriptive drugs.

One issue that we may encounter in the medical field, is diagnosing a disease incorrectly. Another is possible time-constraints a medical professional may have when dealing with diagnosing patients.

This report is the results of a classification model built to read in thousands of images of blood cells, in order to predict if a cell has malaria.

DATASET

The dataset for this project was downloaded from www.kaggle.com, but is originally from the National Library of Medicines at https://ceb.nlm.nih.gov/repositories/malaria-datasets/. There over 27,000 images that are separated into two categories. These two categories contain infected and uninfected cells. Each image is a .png file and is an image of a blood cell.

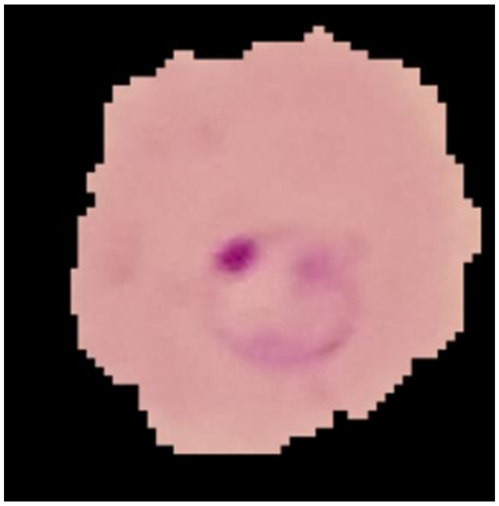
For more in-depth information of how the data was collected, please visit official NLM website listed above.

PRE-PROCESSING

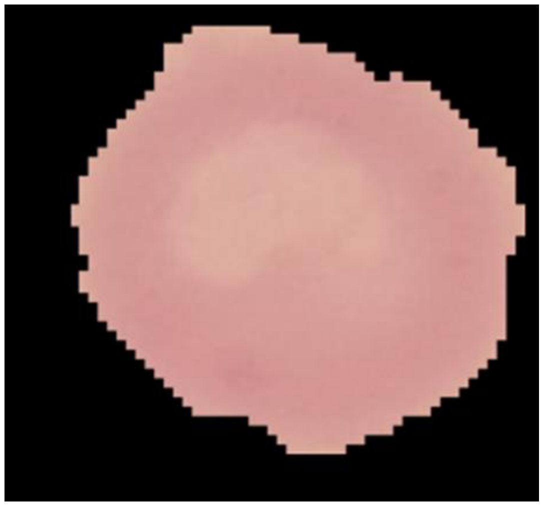
What was needed first was to enable a file path to our files on the computer. The data was separated into two files; one for training, and one for validation. Each folder then had two subfolders, which were split up into infected blood cells and uninfected blood cells.

After getting access to our image files, we wanted to explore a little on what these files actually look like.

Blood cell with malaria:



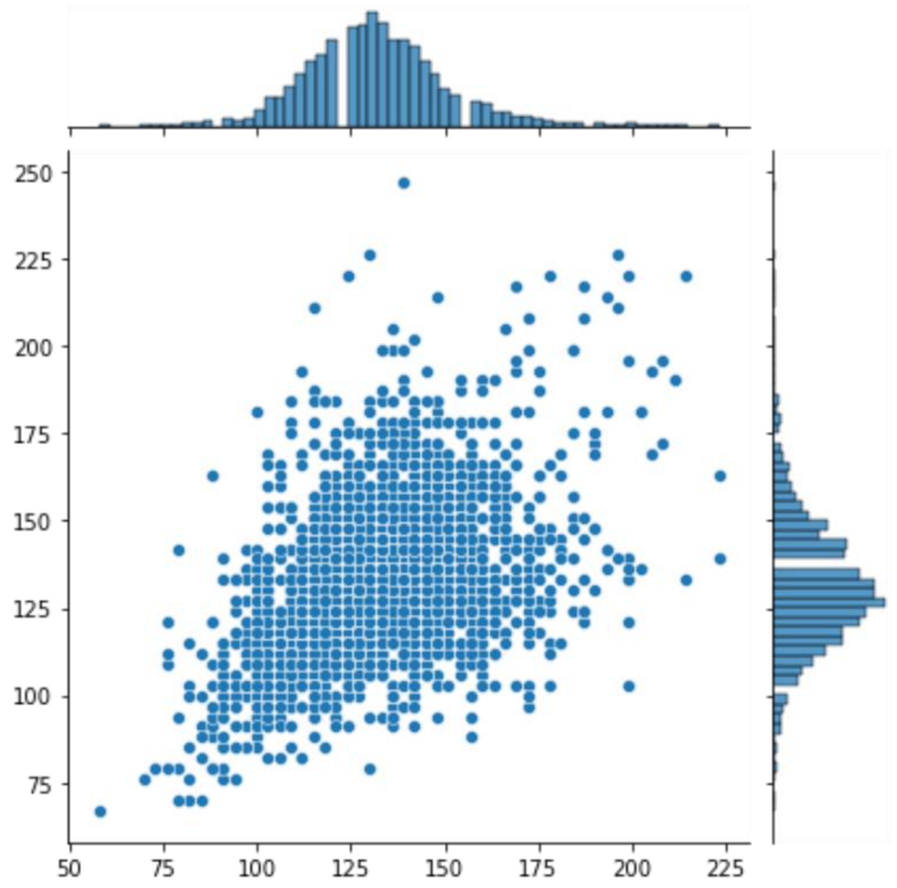
Blood cell without malaria:



These are random images. The human eye could easily differentiate these 2 cells.

These images range from many different sizes, but our model can only take in one image size for it to work. The next step needs to be figuring out what size our images should be for us to feed them into our model.

Here is a plot and distribution of the different image sizes in our data:



We can see these values range from small to larger. What we want to do is find the mean for both of these dimensions. By looking at this plot, we can see the means are somewhere around the 130 area, and in fact when we calculate the mean, we get values of 131 for each. This is used as our image size. Some images may be cropped, and some may be expanded, but we need to have one size for our model.

After selecting our image size, we then created an image generator. For a convolutional neural network to be effective, the more data we have would be better. It may be difficult to collect images of blood cells from people who have malaria, or another condition, so instead we randomly generate new images based on slights alterations of the existing ones.

Some of the modifications include slightly rotating the image, stretching it out, zooming in, and flipping the image. Due to the images we are dealing with, we can adjust many parameters when creating this image generator. These images look essentially like blobs, so we can rotate them, flip them, etc. If it were more detailed images, like faces or vehicles, we may need to be a bit more careful with how we generate new images.

MODEL

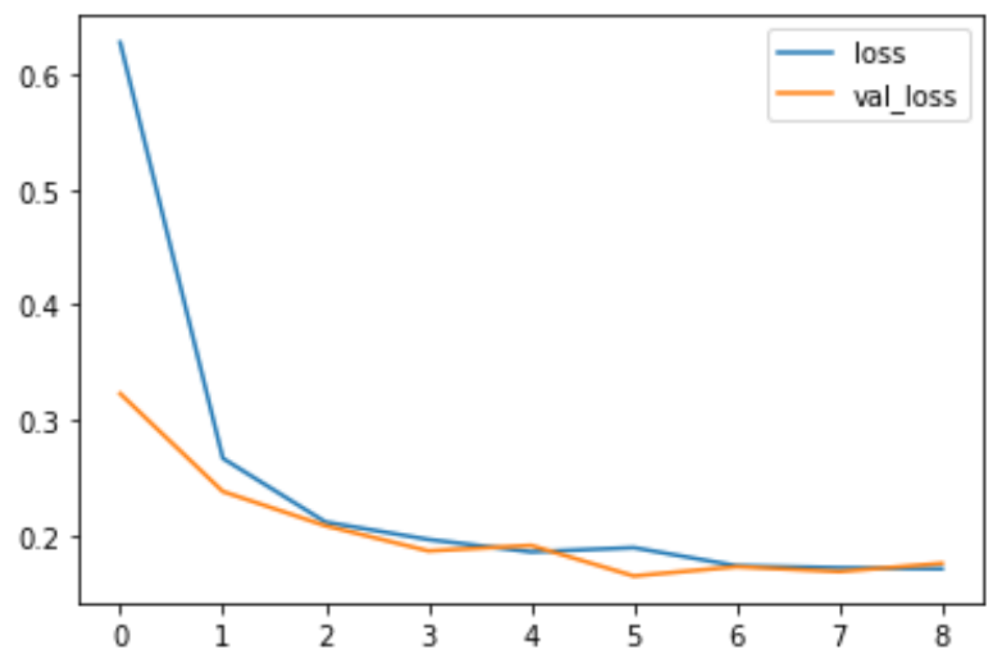
We then proceeded to build our convolutional neural network (CNN) model. A CNN model has convolutional layers, pooling layers, and dense layers. There is no particular format of how these can stack up, but the convolutional and pooling layers need to be in place before the dense layers.

Our model used 3 convolutional layers, 3 pooling layers, 1 dense layer with a .2 dropout rate, 1 dense layer with a sigmoid function, and compiled with a binary cross-entropy loss function.

Before the data can reach the dense layers it needs to get flattened into a one-dimensional array, which we did. The .2 dropout rate mentioned, means that 20% of the neurons will be turned off at random, which is to prevent overfitting. We used the sigmoid function because it is binary data, as well as using the binary cross-entropy loss function.

We trained the data for 10 epochs, and added an early stopping patience of 2, which means once the validation loss begins to rise, it will wait 2 more epochs before it stops to accommodate possible noise in the data.

After training our model, we wanted to visualize the loss:



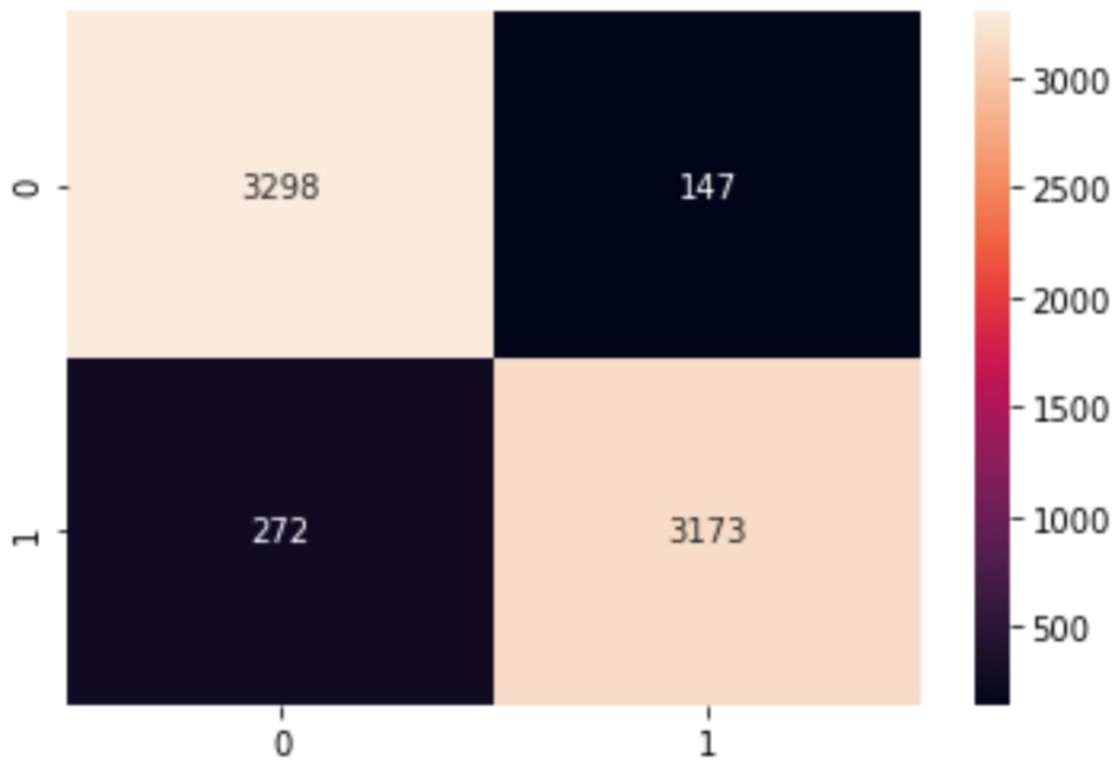
We can see that when the model begins the loss is high, which is expected. Then after a few epochs the loss greatly falls. The early stopping looks to have been triggered around 6 epochs, but it looks like we could have trained on a few more epochs as the validation loss didn't appear to be rising much.

This chart looks pretty good, as it doesn't appear to show any signs of overfitting.

To make our predictions, we set a threshold for our model to have at least a 0.9 probability before it classifies a cell to be uninfected. Due to the nature of this disease and the risk at hand, we want to be assured of our predictions, so we refrain from a .5 cutoff.

After we visualized our training and validation loss, we wanted to visualize the performance metrics of our model. The visualization used for this is a confusion matrix, which pairs up the predicted values against the true values. The numbers we want to reduce is our false positive and false negative errors

For binary data the chart is simple:



The correctly categorized predictions are where the chart matches up at (0,0) or (1,1). In this case 0 is an infected cell and 1 is an uninfected cell.

The 147 represents our false negative errors, and these are the errors that we want to reduce the most, due to our data. This is when we predict the cell to be uninfected, but it is in fact infected with the disease. Although a false positive is not ideal, it is much safer to predict someone to have the disease, then after further exploration by a professional, to find out that the individual does not. The last thing someone in the medical field would want to do is to send someone home who actually had a disease or certain health condition, because they were thought to have been healthy.

Our number of false negatives were a little lower than our false positives, which is what is desired.

Our metrics for the model show:

	precision	recall	f1-score	support
0	0.92	0.96	0.94	3445
1	0.96	0.92	0.94	3445

The f1 score is .94, which is pretty good. The main metric to pay attention to here is the recall for the 0 (infected) class. This is the metric that is our main focus. Recall is the ratio of our correct predictions compared to all of the data. The recall for 0 is .96, which is good.

DEPLOYMENT

The next step would be to deploy this model to incoming images of blood cells, and it should be able to generate its predictions. We would need to preprocess the data slightly, such as resizing the images, but we then can feed the data into our model.

This model is not the answer to diagnose malaria in patients, but can defiantly assist. It can save those medical professionals valuable time by filtering out some images, or bringing certain images to focus on. It may be used to help prioritize a medical professional's time when diagnosing malaria by supplying images of blood cells. We would still need the medical professional to do the diagnosing, but this can be used as a tool to assist them.

While this deployment may be used, another one can be trained. We can continue to attempt to create a better model as we acquire more data. As our data supply increases, our performance in the model should as well, while we continue our efforts for a near perfect model.