Multilayer Perceptron EXAM 1 Report

Jason Witry

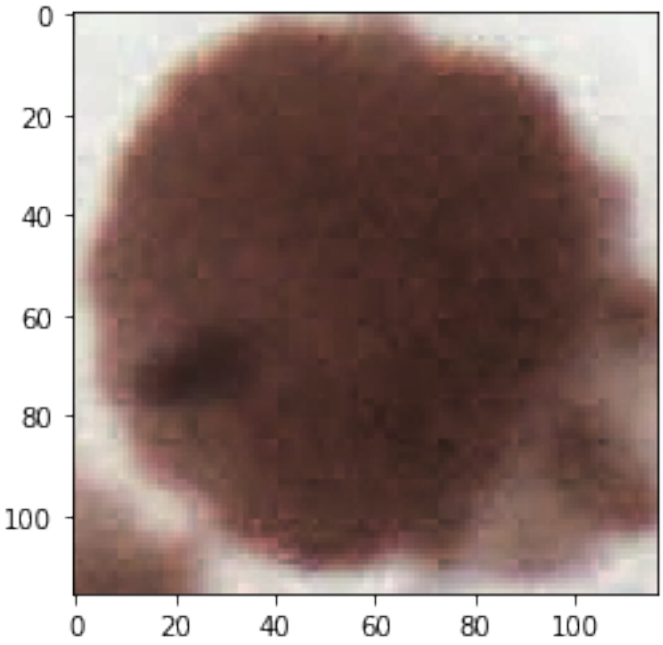
March 13, 2020

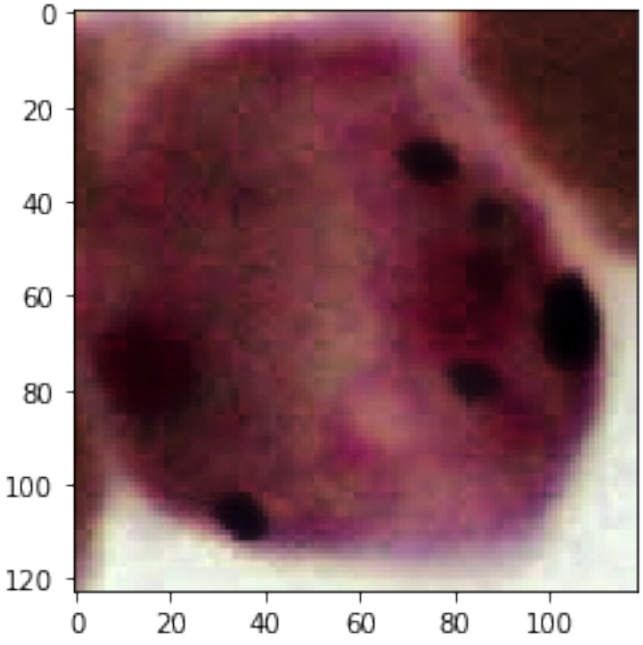
1 Data

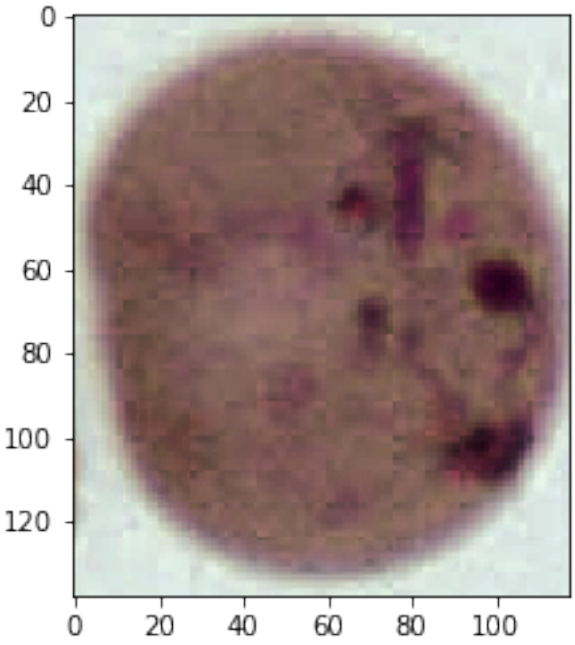
My objective here is to classify 4 types of cells based on their images: red blood cells, rings, schizonts and trophozoites. After downloading the data, and reading it in, I must first align the data so that each cell image is aligned with each respective target. The best way I found to do this is below:

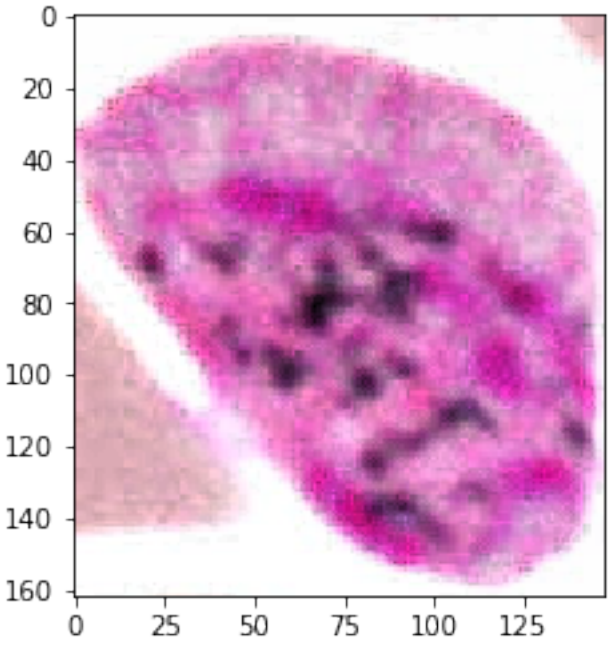
```
def align_data(im_list,y_list):
    ### Align data and target based on the number in the filename
    im_order = [int(im.split('_')[1].split('.')[0]) for im in im_list]
    y_order = [int(y.split('_')[1].split('.')[0]) for y in y_list]
    im_list_ord = []
    for im_ind,im in sorted(zip(im_order,im_list)):
        im_list_ord.append(im)
    im_list = im_list_ord

    y_list_ord = []
    for y_ind,y in sorted(zip(y_order,y_list)):
        y_list_ord.append(y)
    y_list = y_list_ord
    return im_list_ord,y_list_ord
```

After loading the images, I see that they have very different sizes. Therefore, I have to resize the images to a fixed (width,height) in order to feed to our model. I use the OpenCV module resize method to do this, choosing an initial value of 120x120. Below, I show some raw images.

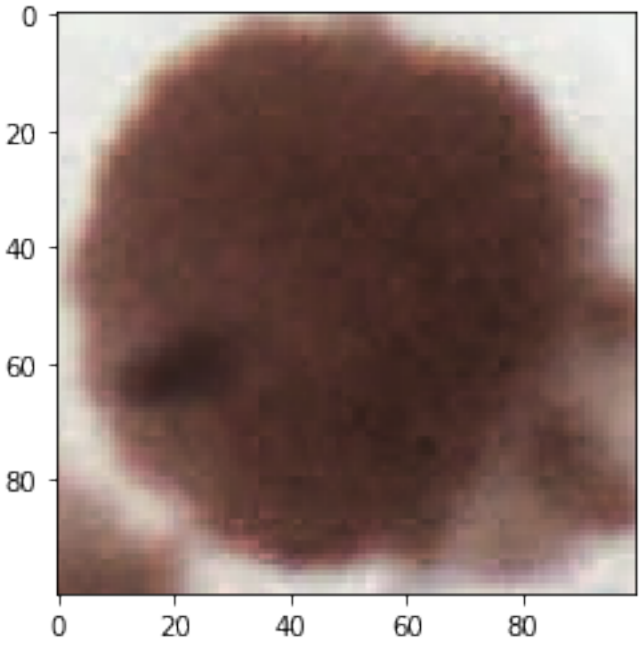

0
20
40
60
80
100
0
20
40
60
80
100

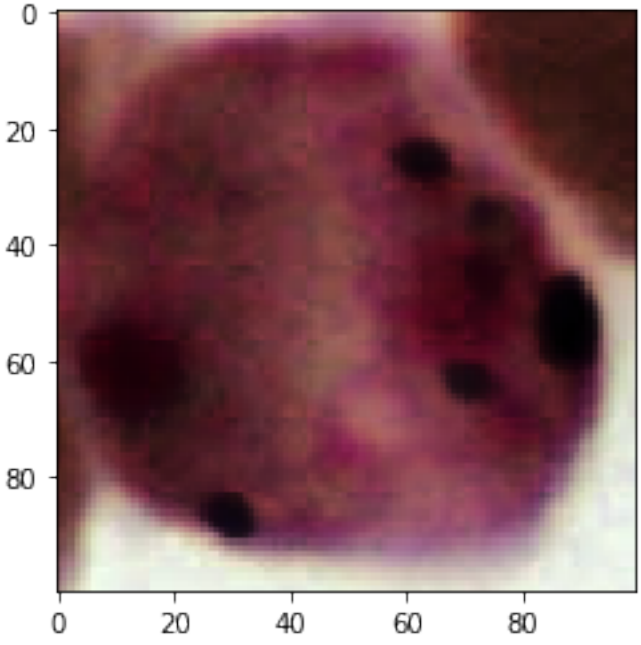

0
20
40
60
80
100
120
0
20
40
60
80
100

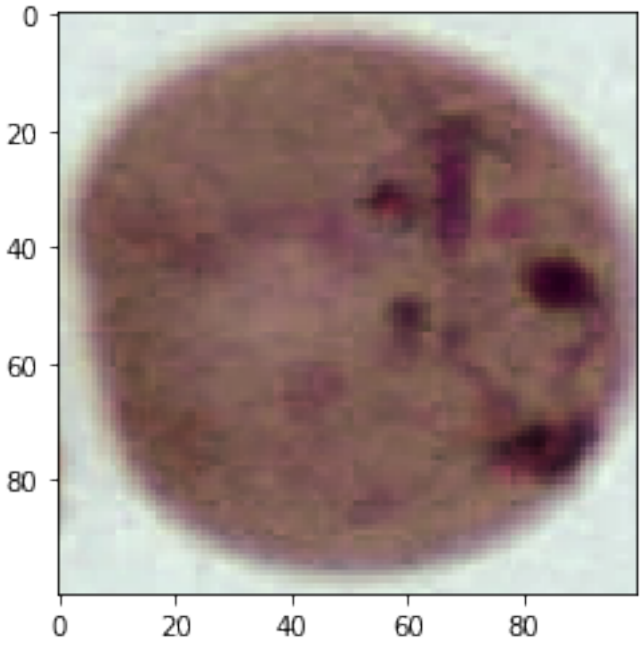


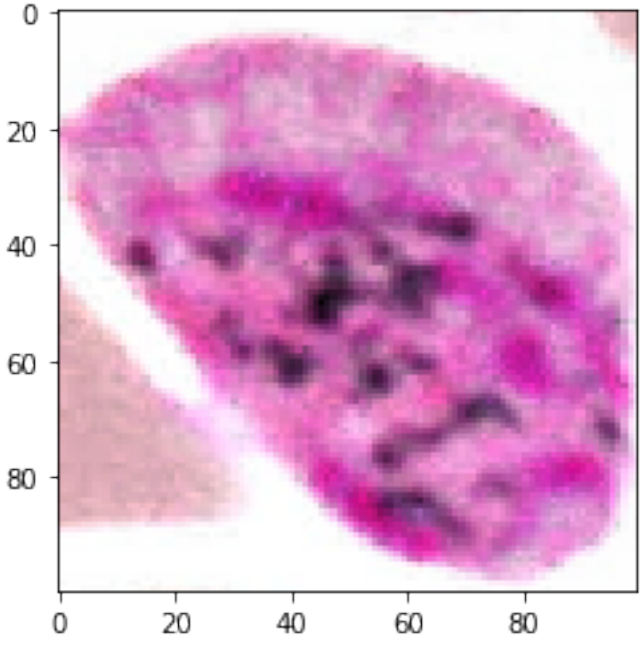



```
### Resize images to 120x120
X = list(map(resize_im,X))
```


0
20
40
60
80
0
20
40
60
80


0
20
40
60
80
0
20
40
60
80





I see that resizing does not alter the image very significantly and allows me to input the images into a multilayer perceptron. The resizing value (desired_size in resize_im) is now a hyperparameter. cv2 uses linear interpolation to fill or shrink the pixels in the image. I didn't get a chance to explore here much; however I did try one other value in the augmentation trial.

2 MLP Builing and Hyperparameter Training

To build the best model, first I needed to find a network architecture that performs relatively well on the data. I try the architectures detailed in the table below, and give the result.

Layers	Learning Rate	Epochs	Macro f1 Score	Cohen Kappa
(100,200,100)	1e-5	200	0.45	0.34
(100,200,100)	1e-5	400	0.35	0.14
(100,200,100)	1e-4	200	0.0081	0.00018
(100,200,100)	1e-4	400	0.0075	0.0
(100,200,100)	1e-3	200	0.35	0.11
(100,200,100)	1e-3	400	0.020	-0.00011
(100,200,200,100)	1e-5	200	0.30	0.10
(100,200,200,100)	1e-5	400	0.21	-0.01
(100,200,200,100)	1e-4	200	0.39	0.22
(100,200,200,100)	1e-4	400	0.0071	-0.00011
(100,200,200,100)	1e-3	200	0.54	0.54
(100,200,200,100)	1e-3	400	0.19	-0.015
(100,100,200,100,100)	1e-5	200	0.42	0.35
(100,100,200,100,100)	1e-5	400	0.0090	0.00045
(100,100,200,100,100)	1e-4	200	0.023	0.00045
(100,100,200,100,100)	1e-4	400	0.22	-0.00049
(100,100,200,100,100)	1e-3	200	0.31	0.095
(100,100,200,100,100)	1e-3	400	0.15	0.025

Looking at the models above, it appears that the best models have architechtures (100,200,200,100) and (100,200,100). Since most models for image processing that yield good results (i.e., (512,512) on the MNIST dataset) use more layers/parameters I choose the architecture with more layers/parameters. My network is larger than the MNIST dataset because the images I have are significantly larger, and I need more abstractness in my network because I have a more complex image. However, with more parameters I am increasing the risk of overfitting, so I need to decrease the learning rate.

Just to be sure, I compared the (100,200,200,100) architecture (arch1) with the (100,100,200,100,100) architecture (arch2), both having a learning rate of 5e-7 running for 600 epochs that saves every 200 epochs. The results on unseen validation data are as follows (both of these are with oversampling):

Layers	Macro f1 Score	Cohen Kappa	Chosen
(100,200,200,100)	0.70	0.74	Yes
(100,100,200,100,100)	0.66	0.71	No

```
print(len(red_blood), 'red blood cells')
print(len(troph),'trophozoites')
print(len(ring), 'rings')
print(len(schi), 'schizonts')
```

```
7000 red blood cells
```

```
1109 trophozoites
365 rings
133 schizonts
```

Above we can see the dataset is severely imbalanced. In order to avoid overfitting to red blood cells, we need to rebalance the data somehow. Naively, we can use random over sampling to balance the data by simply repeating the images. The results for the chosen architecture are shown in the table above.

3 Augmentation

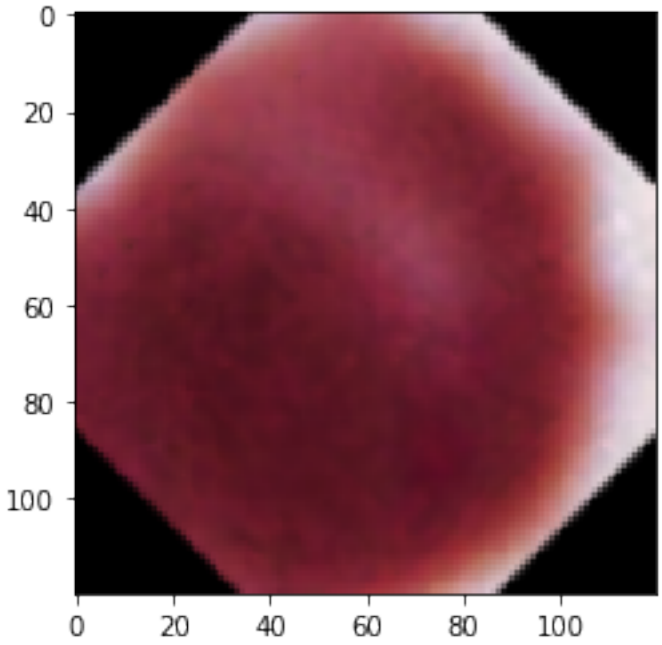
This is good, but a really simple way to deal with the imbalance. As the model is not really getting any new data points, it can still overfit to the specific images. A better way to resample this dataset would be to use data augmentation, i.e. applying image transformations to artificially generate new data. The transformations applied were brightness adjustments and rotations, applied randomly as shown below:

```
def augmenter(self,X_train,y_train,class_repeats):
    print("Augmenting...")
    rot = {0:0,1:cv2.ROTATE_90_CLOCKWISE,2:cv2.ROTATE_180,3:cv2.ROTATE_90_COUNTERCLOCKWISE}
    bright = np.arange(0.9,1.15,0.05)
    for i in range(len(X_train)):
        for j in range(class_repeats[y_train[i]]-2):
            X = self.apply_random_transform(X_train[i],rot,bright)
            X_train.append(X)
            y_train.append(y_train[i])
    return X_train,y_train

def apply_random_transform(self,X,rot,bright):
    pick_rot = np.random.randint(0,4)
    pick_bright = np.random.randint(0,len(bright))
    if rot[pick_rot] == 0:
        im = Image.fromarray(X)
        im = ImageEnhance.Brightness(im)
        im = im.enhance(bright[pick_bright])
        return np.array(im)
    else:
        X = cv2.rotate(X,rot[pick_rot])
        im = Image.fromarray(X)
        ### All PIL enhancements below
        im = ImageEnhance.Brightness(im)
        im = im.enhance(bright[pick_bright])
        return np.array(im)
```

So the augmenter will decide between rotating 90,180, and 270 degrees and altering the brightness by a factor of 0.5 to 1.0, both according to a uniform distribution. Here 'class repeats' is the number of red blood cells (majority class) divided by the number of the cell of interest, i.e. a factor

that when subtracted by 1 tells me how many times I would have to repeat one image to get the same number of that cell as red blood cells. The code here subtracts 2 because I would get memory errors from trying to subtract 1. Note that because cells have no preferred direction, we could rotate any degrees we wanted. However, rotating by an angle in between these values can lead to the image getting cut off at the edges. We see this below, rotating by 135 degrees:

```
plt.imshow(cv2.resize(imutils.rotate(X[index_red[100]],135),(120,120)))
        plt.show()
```

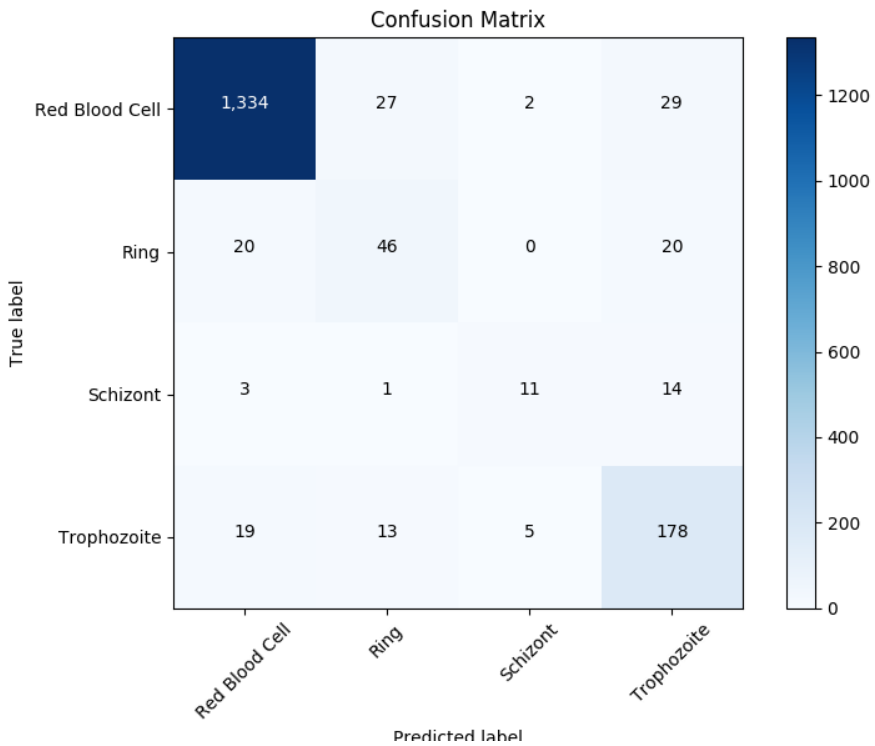


We can see that the corners have been cutoff here. There is a way to fix this by using imutils.rotate_bound, but I have yet to try and pursue that option, so I stuck with 90 degree increments. The results from when the validation loss stopped decreasing for the augmented data are below:

Macro f1 Score	Cohen Kappa
0.65	0.69

So I get slightly worse results than in random over sampling. Obviously, data augmentation *should* be better than naive oversampling; my thought is I need to reduce the range of the brightness factor and add other image enhancements, as well as augment the data in a systematic way rather than random. I also was required to reduce the size of each image to 100,100 because of memory issues. Because oversampling is better, I chose to continue with it although I should revisit augmentation, potentially by combining with oversampling, i.e. systematic augmentation where small imbalances are removed with a random over sampler.

For the oversampling model,from epoch 400 to epoch 600 the loss changes from 0.34 to 0.37. So more learning does not seem to be the answer here for improving this model. Looking at 1,



Confusion matrix for Random Oversampling. Ref:https://www.kaggle.com/grfiv4/plot-a-confusion-matrix

we see that the model is only doing well on red blood cells and trophozoites. The schizont's and rings are difficult, most likely due to the fact these have the least amount of data. So, I decided to train 4 more models using OvA training (one vs all) to model each cell individually. For example, I set up the y_train variable in the code to be 1 for red blood cells, and 0 for the rest for one model, rings as 1 and the rest as 0 for the rest in another model, etc.

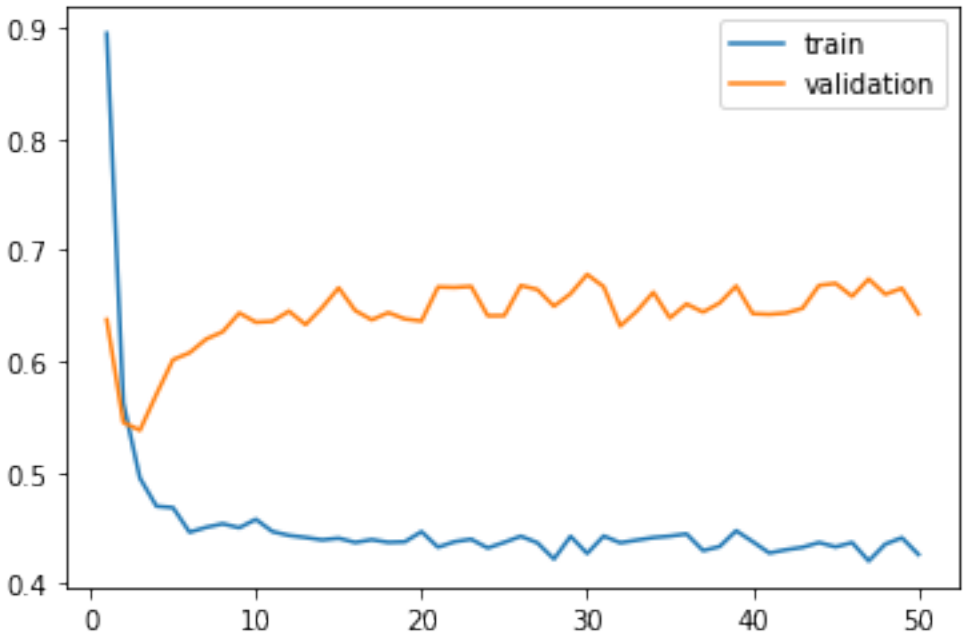
The results *could* be done using a voting scheme; however in case there are any complexities I will input the outputs of these network into a final neural network to make the final decision based on the probabilities of the 5 models (one for each cell plus the large for predicting all). The idea is this final network can discover and correct the inefficiencies of the 5 models and give better predictions.

4 Final Model Difficulties

After running the final model tying a neural network to the softmax outputs of the 5 models (one that classifies all 4; 4 others that classify each individual cell. The results are disappointing; either the model does not train, or it overfits to the training data. After trying multiple architectures, I found that nothing would train well.

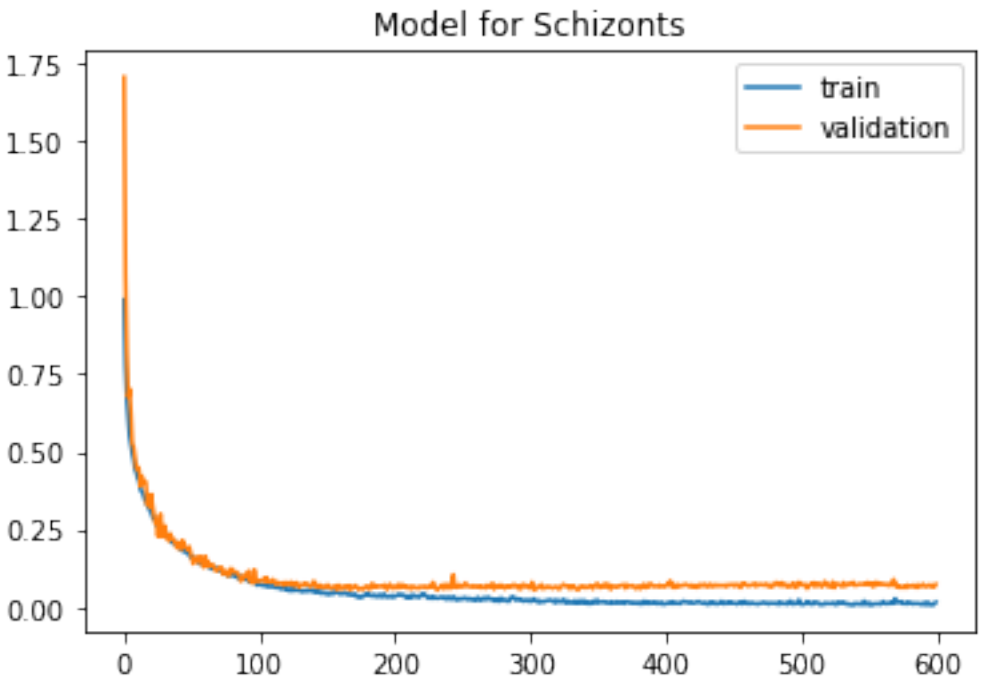
```
epochs = np.arange(1,51)
plt.plot(epochs,train_loss,label='train')
```

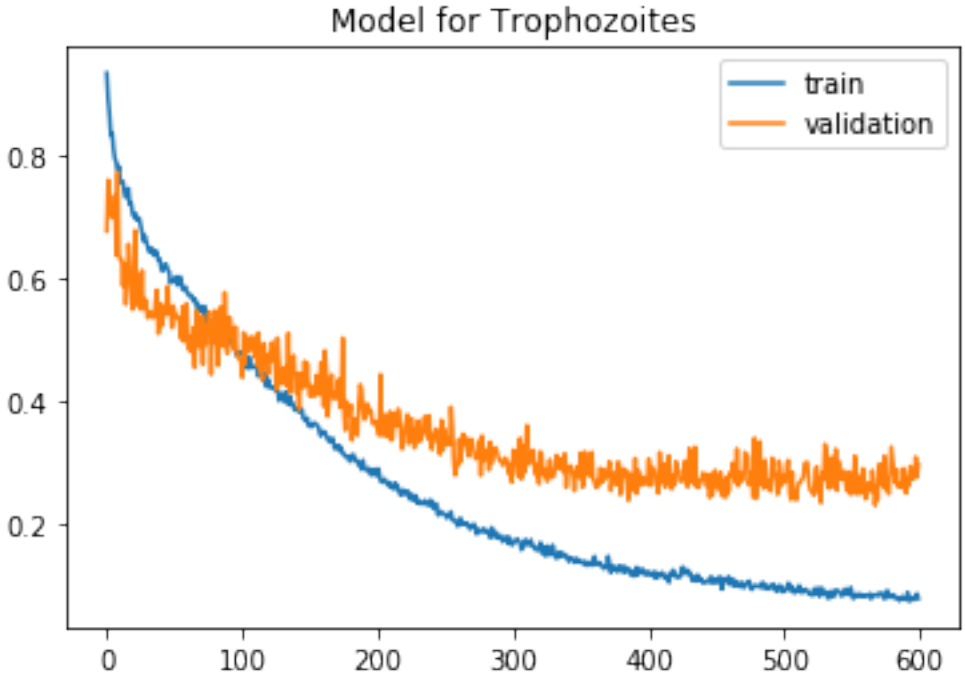
```
plt.plot(epochs,test_loss,label='validation')
plt.legend()
plt.show()
```



This is *terrible*. The validation loss has actually increased, meaning our predictions are getting worse! This is clearly a sign of overfitting; one method I should have tried was adding more dropout to the model (currently set at 0.2).

```
epochs = np.arange(600)
plt.title('Model for Schizonts')
plt.plot(epochs,train_losses[0],label='train')
plt.plot(epochs,test_losses[0],label='validation')
plt.legend()
plt.show()

plt.title('Model for Trophozoites')
plt.plot(epochs,train_losses[1],label='train')
plt.plot(epochs,test_losses[1],label='validation')
plt.legend()
plt.show()
```





Another issue I am having is overfitting in the trophozoites. The data was balanced by a procedure that randomly oversampled the all the data (X and y train) for each cell individually,

then combined the cell types we are not training on into one batch, and randomly undersamples down to the target. I believe this process encourages overfitting, because naive oversampling is not changing the data, I am still fitting the same data the model just sees it more often. I believe this is another problem in the mixed model; many of the individual models are overfit, so when the results go into the stacked neural network the results cannot possibly perform much better on the data. A better option would have been to use data augmentation to augment the data, then potentially train the same model comparing red blood cells to others on *each other cell type individually*. For example, I would train on red blood cells against trophozoites (making the data balanced with augmentation), red blood cells against schizonts, etc. If there was still over fitting in the model, I would try increasing the dropout and potentially adding some other forms of regularization.

The final, best model I ended up with was a combination of the model that trained on all cells (output softmax of 4) and the individual models training using OvA. Once the predicted probabilities came out of the individual models, I concatenated them into a (nsamples,4) matrix using hstack:

```
models = ['mlp_witryjw0.hdf5','mlp_witryjw1.hdf5','mlp_witryjw2.hdf5','mlp_witryjw3.hdf5','mlp_w
y_pred0 = model0.predict(x)
for i in range(2, len(models)):
    model_cur = load_model(models[i])
    ### Get predictions of current model
    pred = model_cur.predict(x)
    # y_pred[:,i-1] *= pred[:,1]
    ### stack each prediction of positive
    y_pred = np.hstack((y_pred, pred[:, 1].reshape(-1, 1)))


### multiply (hadamard product) probabilities predicted by softmax
y_pred *= y_pred0
```

Multiplying the probabilities reflects the idea that if the model built on all 4 cells is not sure about a prediction (i.e. a red blood cell prediction about 0.45, trophozoite about 0.55), but the individual model is very sure about the cell being a red blood cell (0.9) and the trophozite model is fairly sure it is a trophozoite (0.7), after multliplying the predictions will be 0.405 for red blood cell and 0.385 for trophozoite, eliminating the ambiguity and making the est choice from all the models. This only modestly improved performance however:

Macro f1 Score	Cohen Kappa
0.70	0.75

5 Conclusions and How to Improve

In sum, I should have paid more attention to validation plots telling me that I was overfitting many of my models, and increased the dropout/added augmentation (in a more intelligent way). I think a uniform distribution of brightness over 0.5-1.5 was too large of a range; and rather than randomly transform the images I think it may be better to systematically transform them (i.e. for

all rotations do different brightness variations). I may include different rotations besides 90,180 and 270 in the future as well using the rotate_bound solution. All in all, more image processing would have helped. However, while this model is not great, it could be much worse. It is interesting that multiplying the probabilities helped the model somewhat in the end; I believe if my models for each individual cell were not as overfit as they are I could have a much better model.