Capstone Three – Final Project Report

Cell Images Classification – Infected (Parasitized) or Uninfected

1. Introduction

Problem statement:

Predicting the parasite infected and uninfected cells from microscopic images

Context:

Infectious diseases such as malaria and dengue are caused by parasites like viruses. After infection, the human body cells start spreading these parasites all over the body. Once the body cells are infected with parasites, that is called ‘parasitized’ or ‘infected’ cells. Other hand, the cells that are not infected with parasites are called ‘uninfected’. So, it is crucial to detect these infected (parasitized) cells before they cause serious illness to patients. We can identify these infected cells using their microscopy images which are stained with chemical dyes.

Generally, two types of dyes are used to identify the ‘infected’ and ‘uninfected’ cells: 1. eosin (pink) and 2. hematoxylin (blue). Simply to understand, the images which contain the blue/pink stains are infected with parasites, whereas cell images not containing any stain are classified as uninfected cells. Here, we aim to classify the infected and uninfected images by training more than 10000 images that will help in predicting the right class upon testing.

Project goals:

- Labeling, feature extraction and training the two classes of images
- Building and testing a reliable model that can predict the infected images with high accuracy

2. Dataset

Data originally deposited by Baris Dincer in Kaggle datasets page. It contains more than 10000 images to train and test to identify the infected and uninfected class of images. Complete dataset can be downloaded from the below link.

Source: https://www.kaggle.com/brsdincer/cell-images-parasitized-or-not

The dataset contains separate train and test set images grouped into two classes:

1. Parasitized (infected) - (Train size: 12479, Test size: 1300)
2. Uninfected - (Train size: 12479, Test size: 1300)

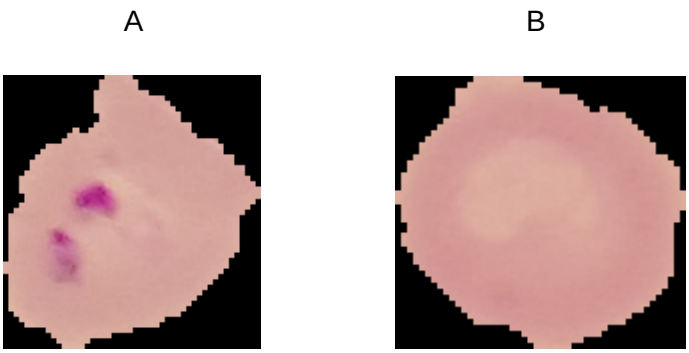


Figure 1: Sample images (A – Parasitized class, B - Uninfected class)

3. Data wrangling and EDA

The dataset had total 27558 cell images grouped equally between two classes. Initial wrangling involved cloning the original files from the GitHub repository into the Google Colaboratory environment. Then for image processing requirements we imported OpenCV, TensorFlow, Keras modules over the notebook. We first set the path for the images within the notebook using pathlib and glob modules. Followed by, we labelled the images into **"parasitized"** and **"uninfected"** classes.

Verifying the data

Next, we converted the image paths into series and then to DataFrame for easy data processing. After, we visualized few sample images from the DataFrame using matplotlib module to verify the class.

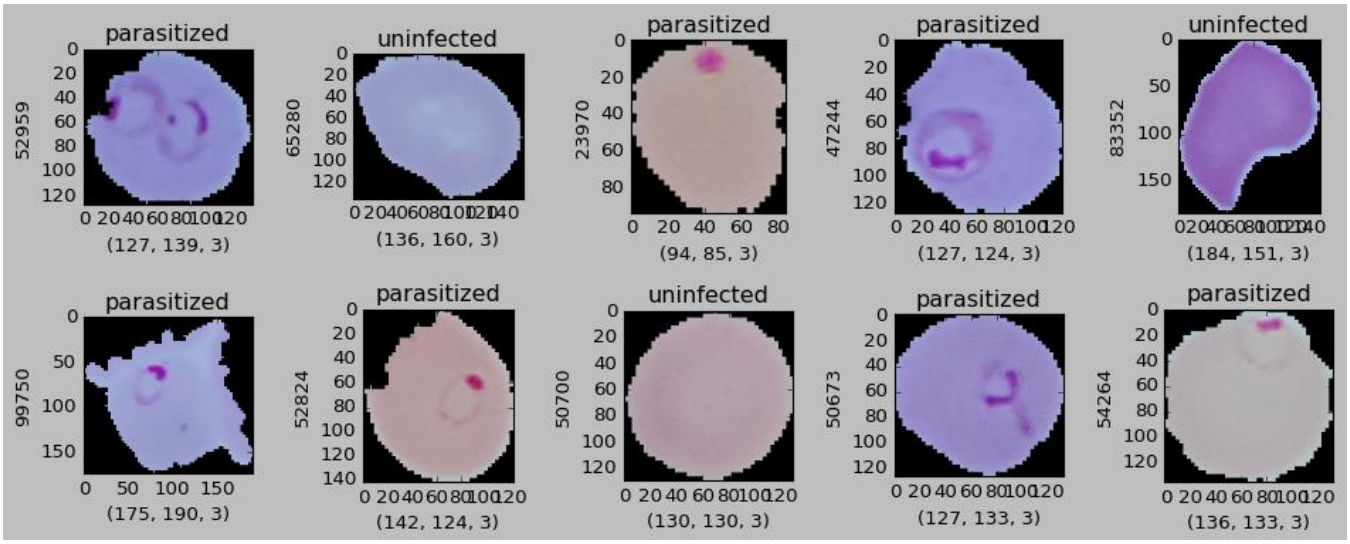


Figure 2: Sample images are visualized and verified for the labels

Once the image data and labels are verified, we proceeded to feature identification and extraction using the OpenCV module in python. Briefly, images were processed for edge detection, thresholding and finding contours.

Image edge detection

Random images were subjected to edge detection using OpenCV Canny() function. Upon trailing 3 to 4 images, the module function detected the edges correctly with minimal error. See below images,

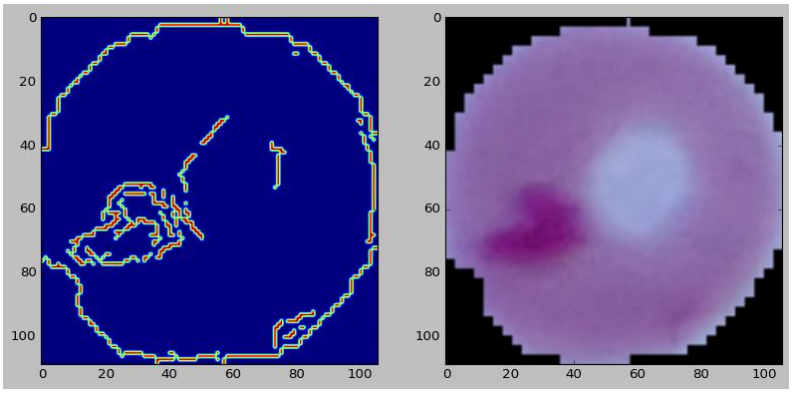


Figure 3: Image edge detection using OpenCV Canny function

Image thresholding

Similarly, random images were subjected to image thresholding test. OpenCV threshold module used along with THRESH_TOZERO_INV filter. See below images,

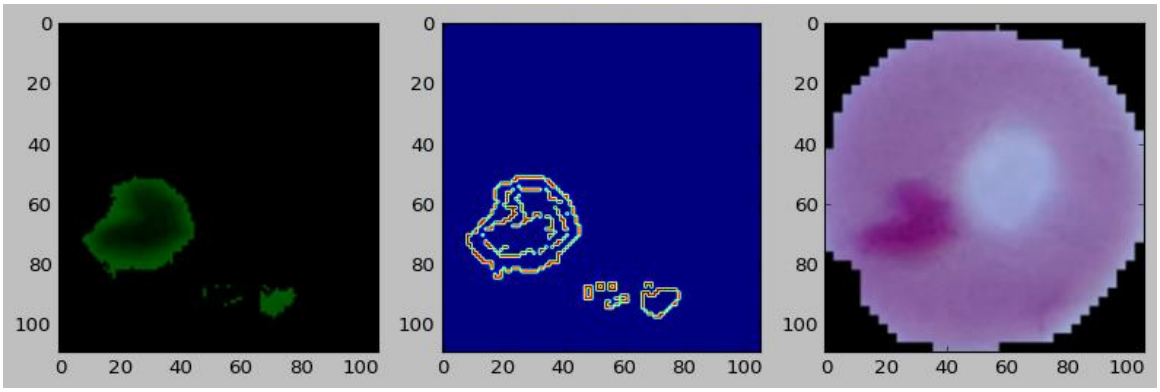
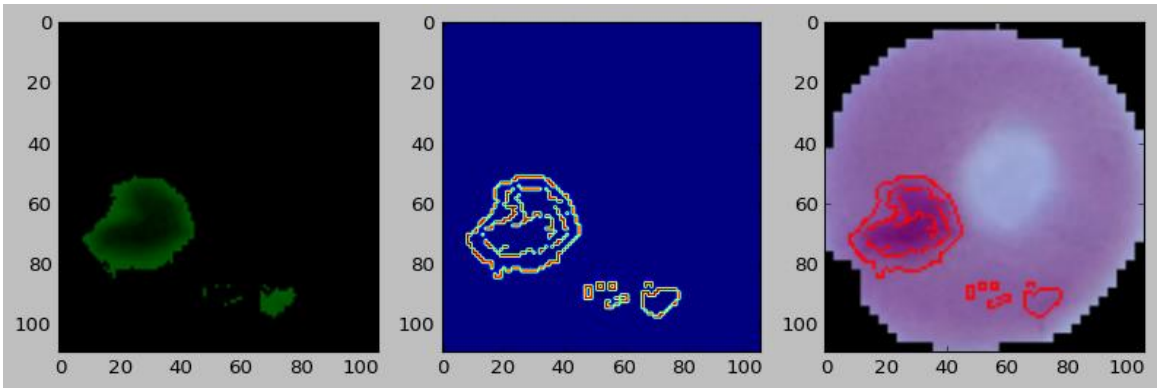


Figure 4: Comparison of threshold images along with edge detection. Left panel image shows the sample thresholding, whereas right panel image shows the original image. Middle image is the edge detected from previous step.

Finding contours

Next, we identified contoured section in the images using findCountour(), drawContour() function from OpenCV module. In findCountour() function we used RETR_EXTERNAL and CHAIN_APPROX_SIMPLE as parameters to retrieve the best contours. Along with, we used the drawContour() function to draw the contours around the image and line styled in red color. A bounding rectangle also was drawn around the contours using boundingRect() function. Example image below,

A



B

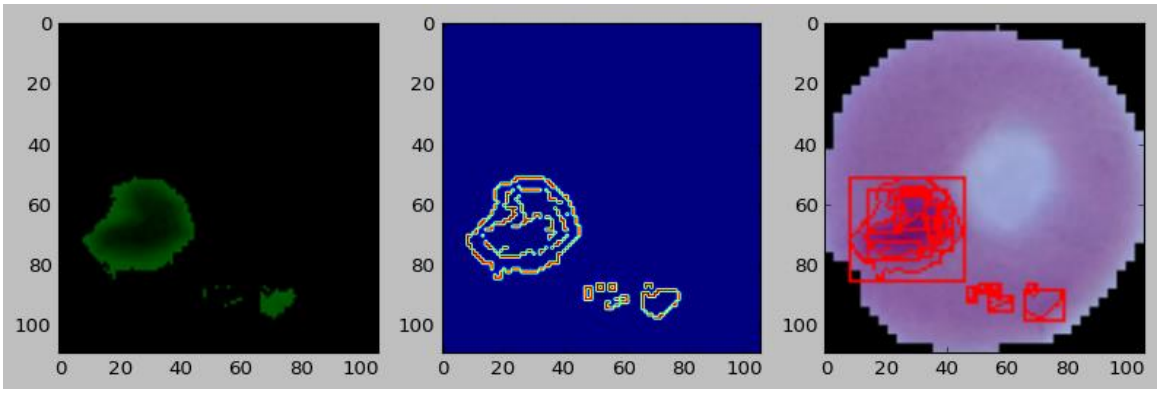


Figure 5: Contoured image comparison along with edge detection and thresholding. A) Right-side panel image shows the contoured areas within the image. B) Same right-side panel image with bound rectangle covering the contoured areas.

DataFrame subset and image processing

We subset the original DataFrame into Parasitized and Uninfected sets. After subset, we processed all images in the individual set to verify the correct detection of image features.

Parasitized set

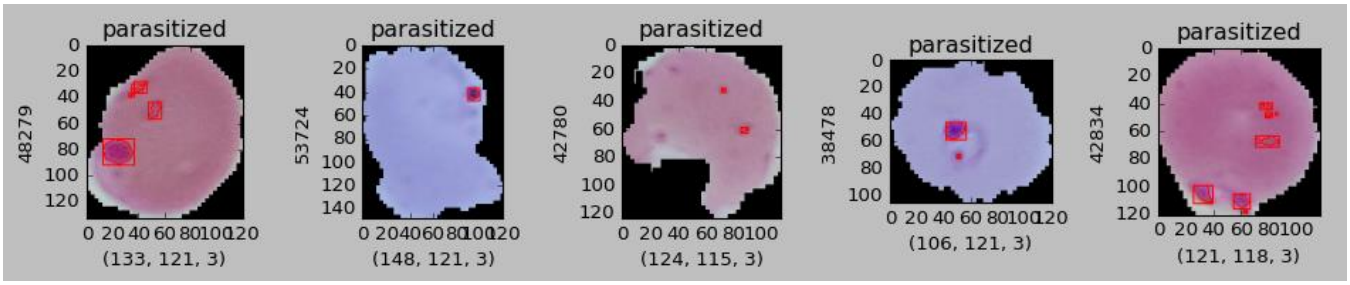


Figure 6: All the parasitized set images in the panel were correctly recognized for image features

Uninfected set

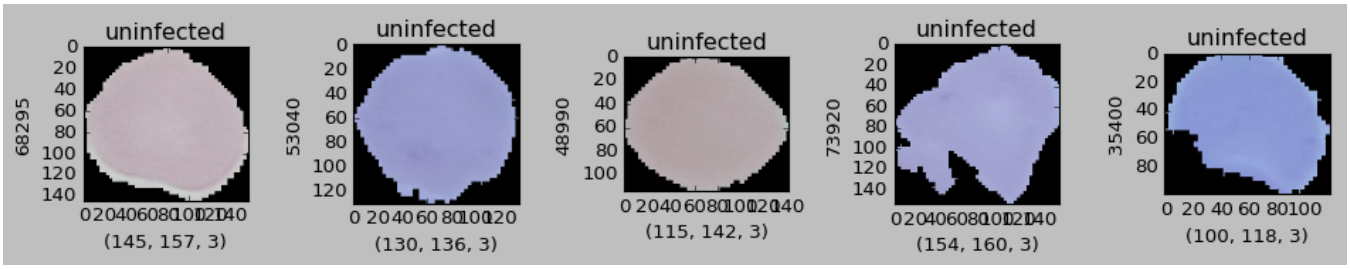


Figure 7: All the uninfected images were not recognized for any features as expected

Subsequent processing all images from parasitized and uninfected sets proved that extraction parameters correctly classified the image features in the parasitized set, whereas in the uninfected set it did not detect any features. Hence, it gave confidence on the set functional parameters to proceed further.

4. Preprocessing

Image augmentation

We used Keras Image Data Generator to generate new augmented images from exiting images of the original dataset. This will help to train our algorithm better in unseen images with variety of perceptions.

Generator Structure

```
Train_IMG_Generator = ImageDataGenerator(rescale=1./255,
                                         rotation_range=25,
                                         shear_range=0.2,
                                         zoom_range=0.1,
                                         brightness_range=[0.4, 0.8],
                                         horizontal_flip=True,
                                         vertical_flip=True,
                                         width_shift_range=0.2,
                                         height_shift_range=0.2,
                                         validation_split=0.1)
```

```
Test_IMG_Generator = ImageDataGenerator(rescale=1./255)
```

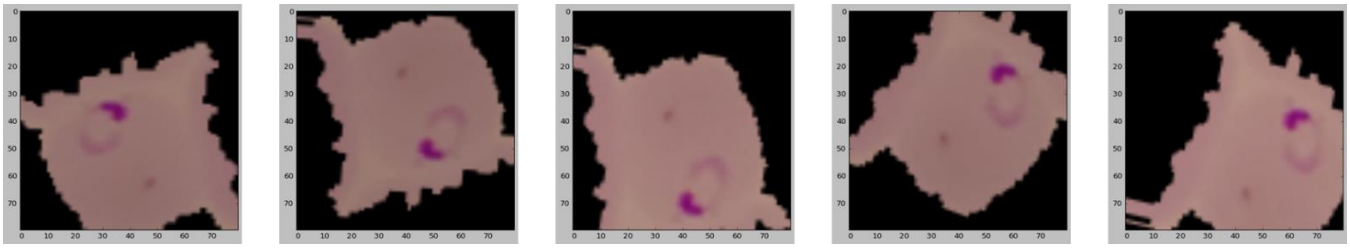


Figure 7: Image generator structure and augmented images generated from a single image.

Splitting the data (Train/Validation/Test)

Using the image data generator, we acquired new images and split the main train DataFrame into training image set and validation image set. Similarly, we generated test image set from main test DataFrame.

	Number of images	Number of classes	Binary class names
Training image set	22463	2	Parasitized: [0] Uninfected: [1]
Validation image set	2495	2	Parasitized: [0] Uninfected: [1]
Test image set	2600	2	Parasitized: [0] Uninfected: [1]

5. Modeling and Prediction

We used the convolutional neural network (CNN) model to process the cell images. Sequential model was set as input layer and SeparableConv2D() added as convolutional layer, followed batch normalization and Maxpooling2D() was used as pooling layer. On top of it dense layers were added with activation functions of RELU and SIGMOID.

Callback option was set using EarlyStopping() function with 'loss' as monitor parameter. Later, we compiled the model using ADAM optimizer with BINARY CROSS ENTROPY loss function and set the metrics as ACCURACY.

The compiled CNN model was fitted on the training image set with the prescribed callback and epochs set as five (epochs = 5). Approximately, each epoch taken around 11.6 minutes to complete the model training. Below are each epoch's accuracy and loss scores for train and validation sets.

Training the CNN model

Fitting

```
CNN_Model = Model.fit(Train_IMG_Set, validation_data=Validation_IMG_Set,
                      callbacks=Call_Back,
                      epochs=5)
```

```
Epoch 1/5
702/702 [==============================] - 728s 1s/step - loss: 0.5208 - accuracy: 0.7390 - val_loss: 0.2823 - val_accuracy: 0.8974
Epoch 2/5
702/702 [==============================] - 702s 999ms/step - loss: 0.2820 - accuracy: 0.8977 - val_loss: 0.2852 - val_accuracy: 0.9114
Epoch 3/5
702/702 [==============================] - 697s 992ms/step - loss: 0.2414 - accuracy: 0.9196 - val_loss: 0.2663 - val_accuracy: 0.9098
Epoch 4/5
702/702 [==============================] - 696s 991ms/step - loss: 0.2292 - accuracy: 0.9254 - val_loss: 0.2380 - val_accuracy: 0.9134
Epoch 5/5
702/702 [==============================] - 696s 990ms/step - loss: 0.2224 - accuracy: 0.9305 - val_loss: 0.2353 - val_accuracy: 0.9234
```

Figure 8: CNN model training and their performance metrics for each epoch

Evaluating the model

Model performance scores consisting accuracy and loss function were plotted in graphs and their convergence pattern was compared.

A B

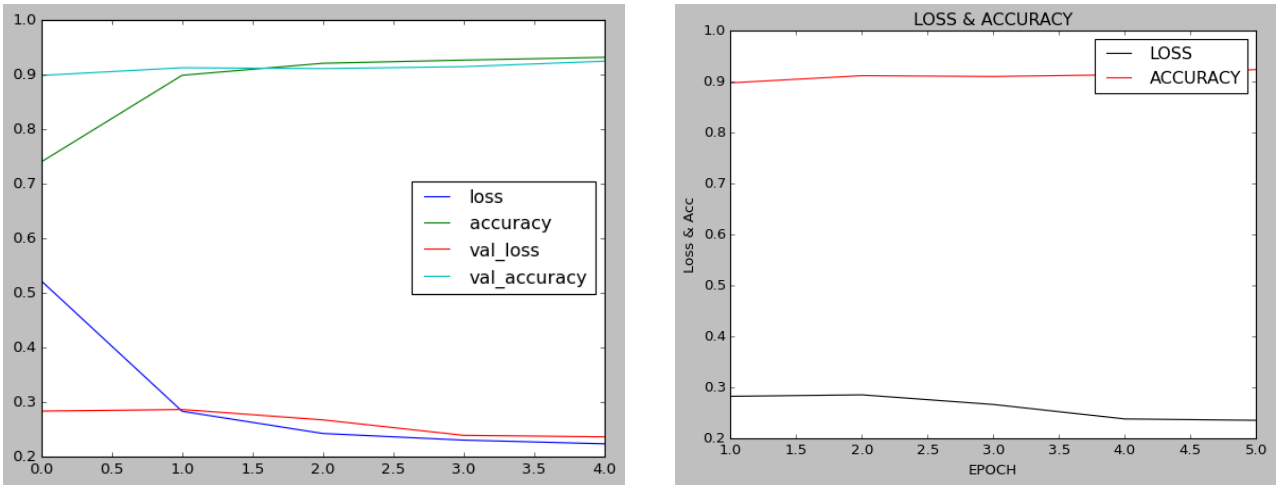


Figure 9: Accuracy and loss comparison graph for training and validation image sets. A) Graph shows the convergence of accuracy and loss scores over the time with maximum accuracy at the top and with minimum loss at the bottom. B) Graph shows the comparison between loss and accuracy function for five epochs. Accuracy score was stable along the epochs, but loss score was nicely converged to the lower bottom at the end of five epoch.

Prediction

The final model was evaluated in the test image set and their performance metrics were obtained. Approximately, the model took 8.2 minutes to complete the set with accuracy score of 0.92 and loss score of 0.24. These scores compare well with our training and validation sets. For full comparison see below,

	Accuracy	**Loss**
Training image set	0.93	0.22
Validation image set	0.92	0.23
Test image set	0.92	0.24

The prediction was executed on the test image set and respective predicted classes were obtained.

Predicted classes:

- 0 ≫ -- Parasitized
- 1 ≫ -- Uninfected

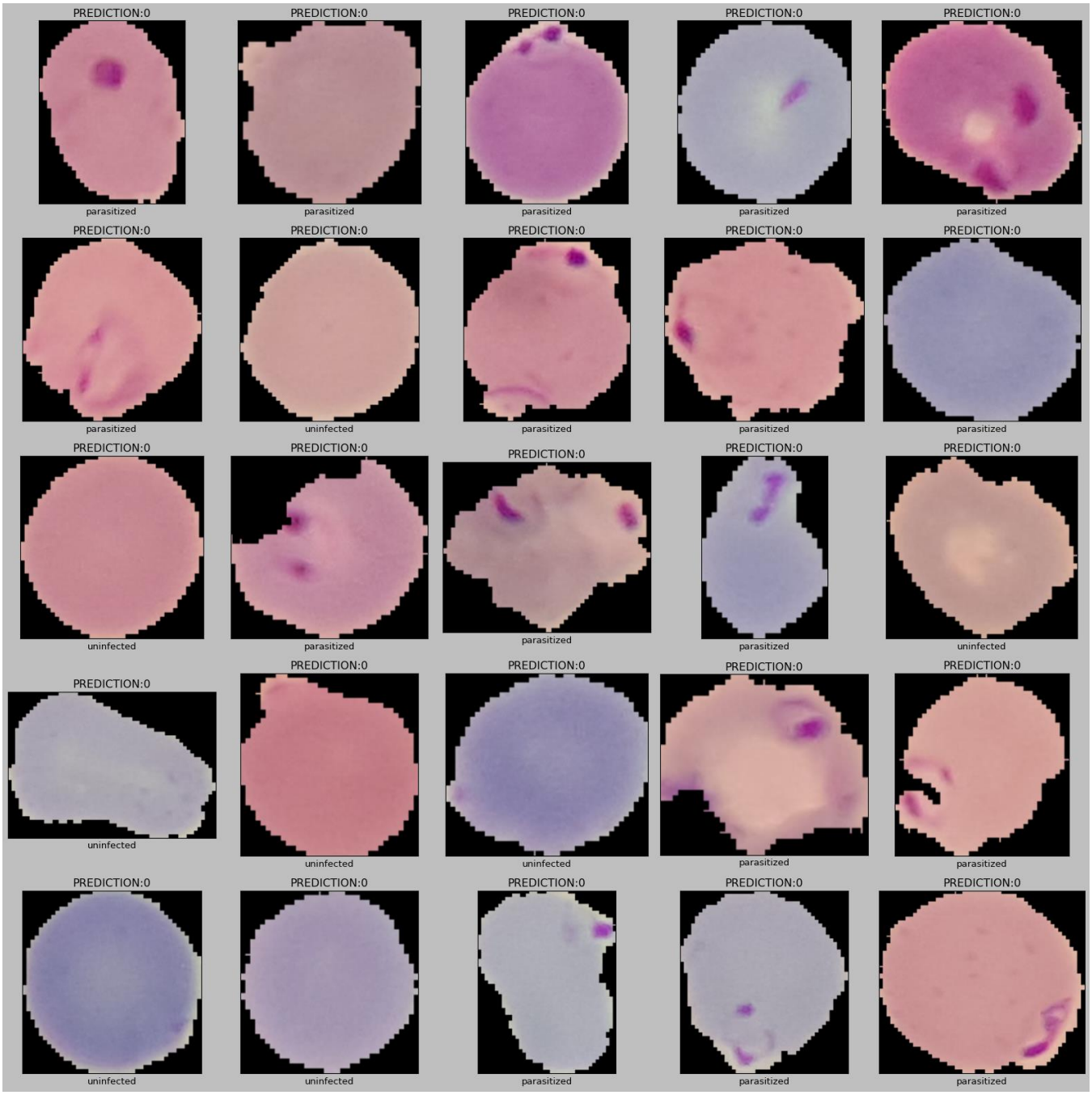


Figure 10: Test images with predicted class labels

6. Summary and Next Steps

- Using the CNN model, we successfully classified images from this dataset with high accuracy and minimal loss.
- There were very few images misclassified into other classes, which can be rectified by training the model with increased sample size.
- Apart form CNN model, other models can be also tested in future for model comparison study.

Supplementary data:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 separable_conv2d (Separable  (None, 254, 254, 12)     75
 Conv2D)

 batch_normalization (BatchN  (None, 254, 254, 12)     48
 ormalization)

 max_pooling2d (MaxPooling2D  (None, 127, 127, 12)     0
 )

 separable_conv2d_1 (Separab  (None, 127, 127, 24)     420
 leConv2D)

 dropout (Dropout)           (None, 127, 127, 24)      0

 max_pooling2d_1 (MaxPooling  (None, 63, 63, 24)       0
 2D)

 time_distributed (TimeDistr  (None, 63, 1512)         0
 ibuted)

 bidirectional (Bidirectiona  (None, 63, 64)           395520
 l)

 bidirectional_1 (Bidirectio  (None, 63, 64)           18816
 nal)

 flatten_1 (Flatten)         (None, 4032)              0

 dense (Dense)               (None, 256)               1032448

 dropout_1 (Dropout)         (None, 256)               0

 dense_1 (Dense)             (None, 1)                 257

=================================================================
Total params: 1,447,584
Trainable params: 1,447,560
Non-trainable params: 24
```

Supplementary Figure 1: Displaying the architecture of the trained CNN model

References:

1. https://www.tensorflow.org/tutorials/images/cnn
2. https://towardsdatascience.com/introduction-to-convolutional-neural-network-cnn-de73f69c5b83