Malaria Detection Using Deep Learning
Jin Xing

1. Executive Summary - *What are the key takeaways? What are the key next steps?*

Malaria is a severe and often fatal disease. Conventional malaria detections are performed by microscopists who analyze microscopic blood smear images in laboratory settings, which requires human expertise and large investments. These resources may be inadequate in developing counties, where malaria is more predominant. Deep learning models thus may play a role in facilitating malaria detection and reducing healthcare costs. In this study, I build a Convolutional Neural Network (CNN) based neural network algorithm to classify images of blood cells into being parasitized or uninfected. The algorithm achieves high overall accuracy, about 98%. Other CNN models with similar structures also perform very well with similar performance. Thus, the model performance is robust to CNNs with varying configurations. As such, deep learning techniques show potential in achieving high accuracy even being applied to the healthcare setting fully automatically. Further tuning of the model may obtain even better performance. As smartphones are widely used, apps based on deep learning could be developed so that malaria detection can be widely conducted even with smartphones, increasing the cost-efficiency and the number of tests. However, the algorithm shows limitations as certain uninfected cells are wrongly classified while it fails to detect certain parasitized cells. This provides a caveat in applying the algorithm to actual healthcare settings and points to a direction for future improvement.

2. Problem and Solution Summary

2.1 Research Question - *What problem was being solved?*

Malaria is a severe and often fatal disease caused by parasites such as Plasmodium falciparum. Conventional malaria detections are conducted by microscopists who analyze microscopic blood smear images in laboratories. Their accuracy and efficiency mainly depend on the level and availability of human expertise. For these reasons, deep learning models can aid microscopists in making decisions or even conducting the analysis automatically. They could significantly streamline the detection process and reduce health care costs if successful.

This project aims to build and evaluate Deep Learning algorithms based on the Convolutional Neural Network (CNN) to classify malaria-infected cells from non-infected ones. To this end, I use a dataset from the US National Institutes of Health (NIH), which contains 27,558 different malaria-infected or non-infected blood cell images from 150 malaria-infected patients and 50 healthy patients.

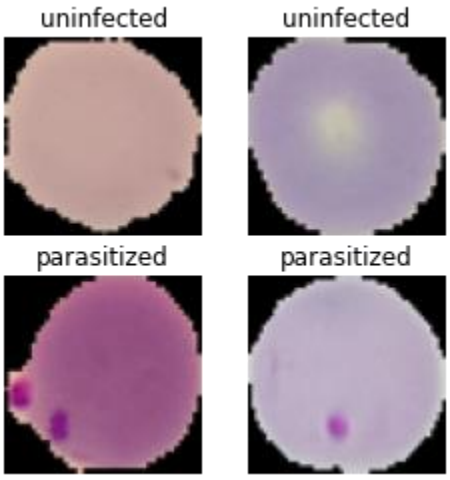
2.2 Methodology - *What are the key points that describe the final proposed solution design?*

2.2.1 Data Processing

Each image from the NIH has a label associated with it, either parasitized or uninfected, and contains three color layers, red, green, and blue (RGB). I resized each image to 64×64. There are 24,958 training images for learning the parameters of the neural network. Out of all the training images, 12,582 are parasitized, and 12,376 are uninfected. Also, there are 2,600 testing images to evaluate the neural network built. Out of all the testing images, 1,300 are parasitized, and 1,300 are uninfected. Thus, the numbers of parasitized or uninfected blood cell images are balanced for training and testing data.

Figure 1 shows sample images of parasitized and uninfected blood cells. We observe that parasitized blood cells are typically stained in dark purple or pink spots, while uninfected blood cells typically have no such spots and are uniform.

Figure 1: Sample images of parasitized and uninfected blood cells



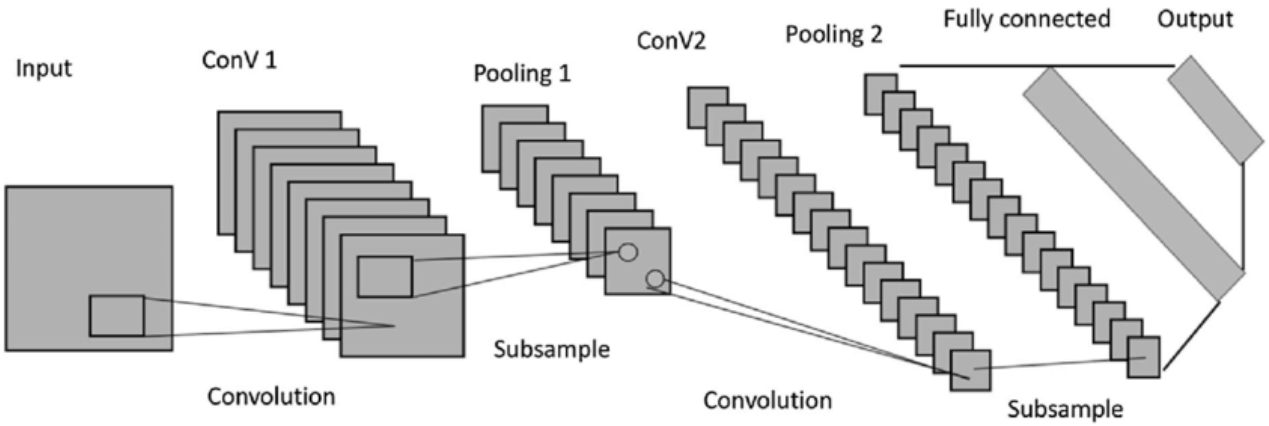
2.2.2 Building CNN-Based Neural Network

CNN classifies images by processing the image through several layers. Three types of layers contained in a typical CNN are listed and described below:

1. Convolutional layer. This first layer applies a kernel to scan the input image, creating an output layer.
2. Pooling layer. This intermediate layer applies operations (e.g., max pooling, dropout, and batch normalization) to alleviate the overfitting on the training data.
3. Fully connected layer. This last layer first flattens the processed image into a vector and then runs it through a fully connected neural network for classification.

Note that the convolutional layer and the pooling layer can be repeated before moving onto the final layer. Figure 2 illustrates a typical architecture for CNN.

Figure 2: Architecture for a typical CNN (Source: ResearchGate)



In this study, I built a CNN-based neural network with a kernel size of 2×2, a dropout rate of 40%, ReLU as the activation function, and softmax as the activation function for the final classification. Figure 3 shows the architecture of my neural network.

Figure 3: Architecture for the CNN used in this study

Layer	Output Shape	Parameter #
Conv2D	(None, 64, 64, 32)	416
Max Pooling	(None, 32, 32, 32)	0
Dropout	(None, 32, 32, 32)	0
Conv2D	(None, 32, 32, 32)	4128
Max Pooling	(None, 16, 16, 32)	0
Dropout	(None, 16, 16, 32)	0
Conv2D	(None, 16, 16, 32)	4128
Max Pooling	(None, 8, 8, 32)	0
Dropout	(None, 8, 8, 32)	0
Conv2D	(None, 8, 8, 32)	4128
Max Pooling	(None, 4, 4, 32)	0
Dropout	(None, 4, 4, 32)	0
Flatten	(None, 512)	0
Dense	(None, 512)	262656
Dropout	(None, 512)	0
Dense	(None, 512)	262656
Dropout	(None, 512)	0
Dense	(None, 2)	1026

Note: The number of total parameters is 539,138, the number of trainable parameters is 539,138, and the number of non-trainable parameters is 0.

2.3 Evaluation - *Why is this a 'valid' solution that is likely to solve the problem?*

Figure 4 shows the evaluation graph, in which the validation split is 0.2, i.e., 20 percent of the training data are kept aside to test the model performance during the model building phase. The model performance is measured by the overall accuracy. The training and validation accuracy become at the same level as the number of iterations (epochs) approaches 4.
The model achieves an overall accuracy of 98% for the testing data. Other models with similar structures, such as those with additional layers, those with batch normalization and LeakyReLU as the activation function, those with augmented images, or pre-trained models (e.g., VGG16), all achieve over 97% accuracy. Therefore, the model performance is robust with different similar structures of CNN.
Figure 5 shows the confusion matrix.
Figure 6 shows error samples, which are image samples that the CNN fails to classify correctly. Certain uninfected cells (labeled 0) have purple spots and therefore are wrongly classified as being parasitized by the CNN. On the other hand, certain parasitized cells (labeled 1) have purple spots, but the CNN fails to detect them.

Figure 4: Evaluation graph

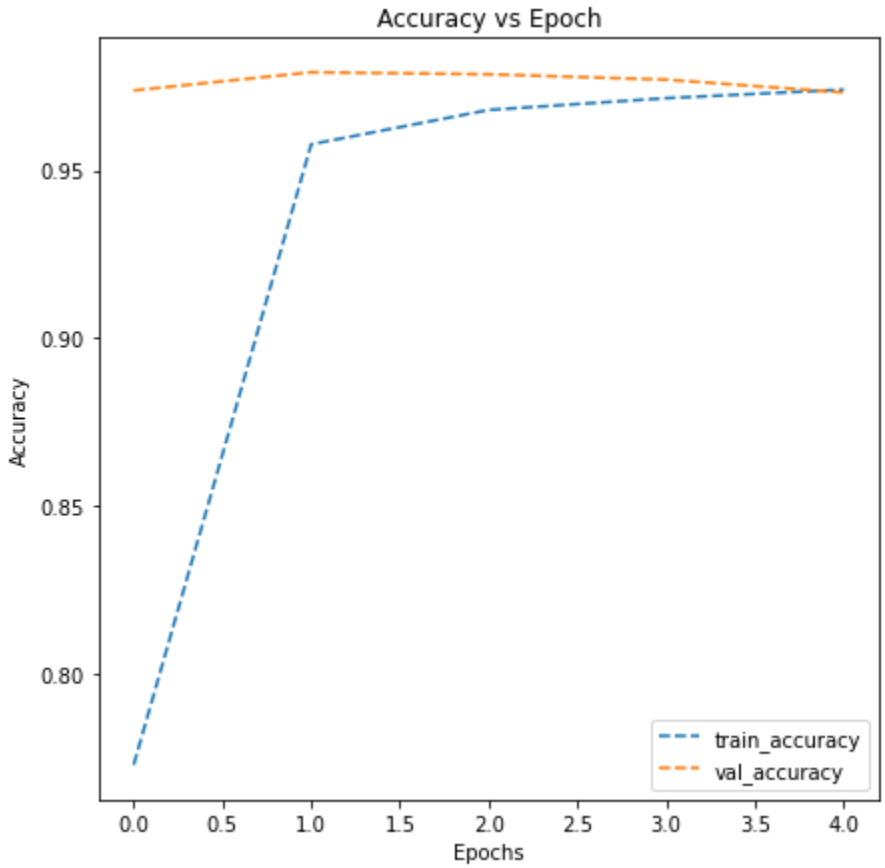


Figure 5: The confusion matrix

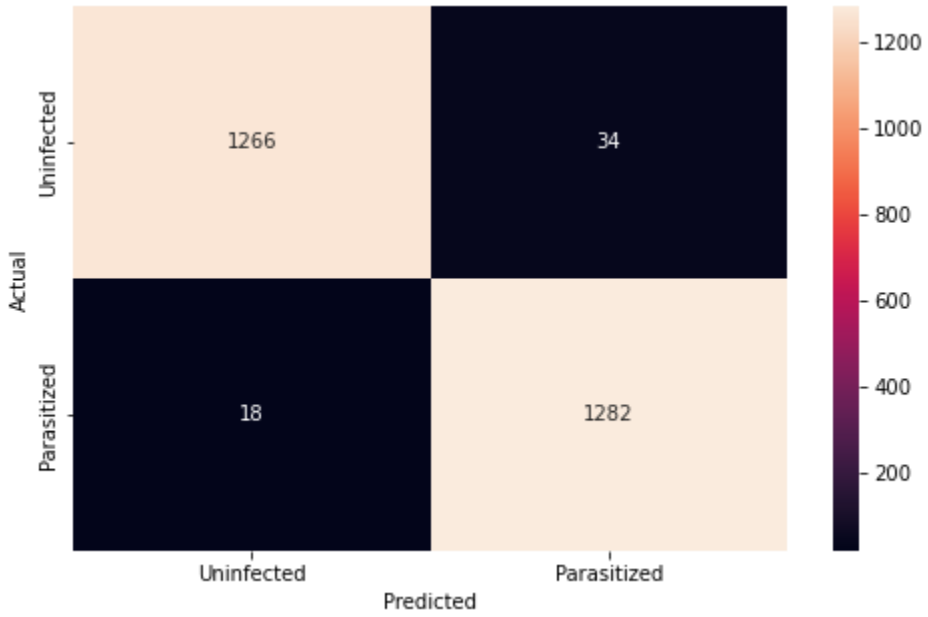
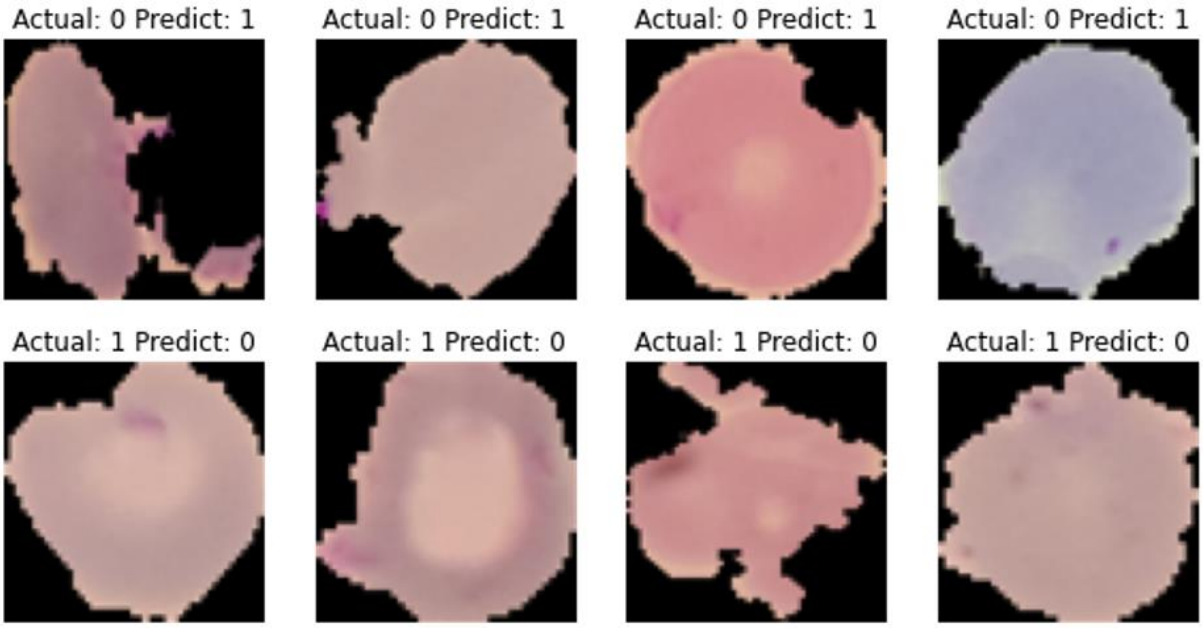


Figure 6: Error samples



Note: Uninfected cells are labeled 0. Parasitized cells are labeled 1.

3. Recommendations for Implementation - *What are some key recommendations to implement the solutions? What are the key actionables for stakeholders? What is the expected benefit and/or costs? What are the key risks and challenges? What further analysis needs to be done or what other associated problems need to be solved?*

Figure 6 shows the limitations of the algorithm built as certain uninfected cells are wrongly classified while it fails to detect certain parasitized cells. This points to a direction for future improvement. The model performance may be further improved by:

1. Adjusting the network structure (e.g., the number of layers, the number of nodes in each layer, activation functions, etc.)
2. Tuning hyperparameters (e.g., initial parameters, step size, regularization, stopping, the shape of a local detector, etc.)
3. Using alternative pre-trained models (e.g., VGG-19).
4. Adopting supervised learning (e.g., KNN, random forest, logistic regression) or unsupervised learning (e.g., PCA) for classification instead of a fully connected neural network.

Malaria Detection Using Deep Learning

Loading libraries

In [75]:
```
#Importing libraries required to load the data

import zipfile
import os
from PIL import Image
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, MaxPool2D, BatchNormalization, Dropout, Flatten, LeakyReLU, GlobalAvgPool2D
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import optimizers

#to ignore warnings
import warnings
warnings.filterwarnings('ignore')

# Remove the limit from the number of displayed columns and rows. It helps to see the entire dataframe while printing it
pd.set_option("display.max_columns", None)
pd.set_option("display.max_rows", 200)
```

Loading the data

In [76]:
```
#Storing the path of the data file
path = 'cell_images.zip'

#The data is provided as a zip file so we need to extract the files from the zip file
with zipfile.ZipFile(path, 'r') as zip_ref:
    zip_ref.extractall()
```

In [77]:
```
#Storing the path of the extracted "train" folder
train_dir = 'C:/Users/jxing/Desktop/MIT/cell_images/train'

#Size of image so that each image has the same size
SIZE = 64

#Empty list to store the training images after they are converted to NumPy arrays
train_images = []

#Empty list to store the training labels (0 - uninfected, 1 - parasitized)
train_labels = []
```

In [78]:
```
#We will run the same code for "parasitized" as well as "uninfected" folders within the "train" folder
for folder_name in ['/parasitized/', '/uninfected/']:

    #Path of the folder
    images_path = os.listdir(train_dir + folder_name)

    for i, image_name in enumerate(images_path):
        try:
            #Opening each image using the path of that image
            image = Image.open(train_dir + folder_name + image_name)

            #Resizing each image to (64,64)
            image = image.resize((SIZE, SIZE))

            #Converting images to arrays and appending that array to the empty list defined above
            train_images.append(np.array(image))

            #Creating labels for parasitized and uninfected images
            if folder_name=='/parasitized/':
                train_labels.append(1)
            else:
                train_labels.append(0)
        except Exception:
            pass

#Converting lists to arrays
train_images = np.array(train_images)
train_labels = np.array(train_labels)
```

In [79]:
```
#Storing the path of the extracted "test" folder
test_dir = 'C:/Users/jxing/Desktop/MIT/cell_images//test'

#Size of image so that each image has the same size (it must be same as the train image size)
SIZE = 64

#Empty list to store the testing images after they are converted to NumPy arrays
test_images = []

#Empty list to store the testing labels (0 - uninfected, 1 - parasitized)
test_labels = []
```

In [80]:
```
#We will run the same code for "parasitized" as well as "uninfected" folders within the "test" folder
for folder_name in ['/parasitized/', '/uninfected/']:

    #Path of the folder
    images_path = os.listdir(test_dir + folder_name)

    for i, image_name in enumerate(images_path):
        try:
            #Opening each image using the path of that image
            image = Image.open(test_dir + folder_name + image_name)

            #Resizing each image to (64,64)
            image = image.resize((SIZE, SIZE))

            #Converting images to arrays and appending that array to the empty list defined above
            test_images.append(np.array(image))

            #Creating labels for parasitized and uninfected images
            if folder_name=='/parasitized/':
                test_labels.append(1)
            else:
                test_labels.append(0)
        except Exception:
            pass

#Converting lists to arrays
test_images = np.array(test_images)
test_labels = np.array(test_labels)
```

In [81]:
```
# shape of images
print(train_images.shape)
print(test_images.shape)
```

```
(24958, 64, 64, 3)
(2600, 64, 64, 3)
```

In [82]:
```
# shape of labels
print(train_labels.shape)
print(test_labels.shape)
```

```
(24958,)
(2600,)
```

Data processing

In [83]:
```
# try to use min and max function from numpy
print(np.min(train_images), np.max(train_images))
print(np.min(test_images), np.max(test_images))
```

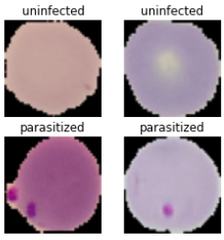
```
0 255
0 255
```

In [84]:
```
# try to use value_counts to count the values
print(pd.DataFrame(train_labels).value_counts())
print(pd.DataFrame(test_labels).value_counts())
```

```
1    12582
0    12376
dtype: int64
0    1300
1    1300
dtype: int64
```

In [85]:
```
# try to normalize the train and test images by dividing it by 255 and convert them to float32 using astype function
train_images = (train_images/255).astype('float32')
test_images = (test_images/255).astype('float32')
```

In [86]:
```
# This code will help you in visualizing both the parasitized and uninfected images
np.random.seed(42)
plt.figure(1 , figsize = (4 , 4))

for n in range(1, 5):
    plt.subplot(2, 2, n)
    index = int(np.random.randint(0, train_images.shape[0], 1))
    if train_labels[index] == 1:
        plt.title('parasitized')
    else:
        plt.title('uninfected')
    plt.imshow(train_images[index])
    plt.axis('off')
```



Building CNN-Based Neural Network

In [87]:
```
#Clearing backend
from tensorflow.keras import backend
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D,MaxPooling2D,Dense,Flatten,Dropout
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
from random import shuffle

backend.clear_session()
#Fixing the seed for random number generators so that we can ensure we receive the same output everytime
np.random.seed(42)
import random
random.seed(42)
tf.random.set_seed(42)
```

In [88]:
```
# Encoding Train Labels
train_labels=to_categorical(train_labels,2)
# Similarly let us try to encode test labels
test_labels=to_categorical(test_labels,2)
```

In [94]:
```
#creating sequential model
model1=Sequential()

model1.add(Conv2D(filters=32,kernel_size=2,padding="same",activation="relu",input_shape=(64,64,3)))
model1.add(MaxPooling2D(pool_size=2))
model1.add(Dropout(0.4))
model1.add(Conv2D(filters=32,kernel_size=2,padding="same",activation="relu"))
model1.add(MaxPooling2D(pool_size=2))
model1.add(Dropout(0.4))
model1.add(Conv2D(filters=32,kernel_size=2,padding="same",activation="relu"))
model1.add(MaxPooling2D(pool_size=2))
model1.add(Dropout(0.4))
model1.add(Conv2D(filters=32,kernel_size=2,padding="same",activation="relu"))
model1.add(MaxPooling2D(pool_size=2))
model1.add(Dropout(0.4))

model1.add(Flatten())

model1.add(Dense(512,activation="relu"))
model1.add(Dropout(0.4))
model1.add(Dense(512,activation="relu"))
model1.add(Dropout(0.4))
model1.add(Dense(2,activation="softmax"))
model1.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_4 (Conv2D)           (None, 64, 64, 32)        416

 max_pooling2d_4 (MaxPooling  (None, 32, 32, 32)       0
 2D)

 dropout_6 (Dropout)         (None, 32, 32, 32)        0

 conv2d_5 (Conv2D)           (None, 32, 32, 32)        4128

 max_pooling2d_5 (MaxPooling  (None, 16, 16, 32)       0
 2D)

 dropout_7 (Dropout)         (None, 16, 16, 32)        0

 conv2d_6 (Conv2D)           (None, 16, 16, 32)        4128

 max_pooling2d_6 (MaxPooling  (None, 8, 8, 32)         0
 2D)

 dropout_8 (Dropout)         (None, 8, 8, 32)          0
```

```
 conv2d_7 (Conv2D)           (None, 8, 8, 32)          4128

 max_pooling2d_7 (MaxPooling  (None, 4, 4, 32)         0
 2D)

 dropout_9 (Dropout)         (None, 4, 4, 32)          0

 flatten_1 (Flatten)         (None, 512)               0

 dense_3 (Dense)             (None, 512)               262656

 dropout_10 (Dropout)        (None, 512)               0

 dense_4 (Dense)             (None, 512)               262656

 dropout_11 (Dropout)        (None, 512)               0

 dense_5 (Dense)             (None, 2)                 1026

=================================================================
Total params: 539,138
Trainable params: 539,138
Non-trainable params: 0
_________________________________________________________________
```

In [95]:

```
model1.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

In [96]:

```
callbacks = [EarlyStopping(monitor='val_loss', patience=2),
             ModelCheckpoint('.mdl_wts.hdf5', monitor='val_loss', save_best_only=True)]
```

In [97]:

```
history1=model1.fit(train_images,train_labels,batch_size=32,callbacks=callbacks,validation_split=0.2,epochs=20,verbose=1)
```

```
Epoch 1/20
624/624 [==============================] - 73s 116ms/step - loss: 0.4579 - accuracy: 0.7730 - val_loss: 0.2110 - val_accuracy: 0.9738
Epoch 2/20
624/624 [==============================] - 76s 122ms/step - loss: 0.1232 - accuracy: 0.9577 - val_loss: 0.1016 - val_accuracy: 0.9792
Epoch 3/20
624/624 [==============================] - 77s 124ms/step - loss: 0.0964 - accuracy: 0.9679 - val_loss: 0.0837 - val_accuracy: 0.9786
Epoch 4/20
624/624 [==============================] - 77s 123ms/step - loss: 0.0915 - accuracy: 0.9715 - val_loss: 0.0875 - val_accuracy: 0.9770
Epoch 5/20
624/624 [==============================] - 76s 122ms/step - loss: 0.0821 - accuracy: 0.9740 - val_loss: 0.0908 - val_accuracy: 0.9732
```

In [98]:

```
accuracy1 = model1.evaluate(test_images, test_labels, verbose=1)
print('\n', 'Test_Accuracy:-', accuracy1[1])
```

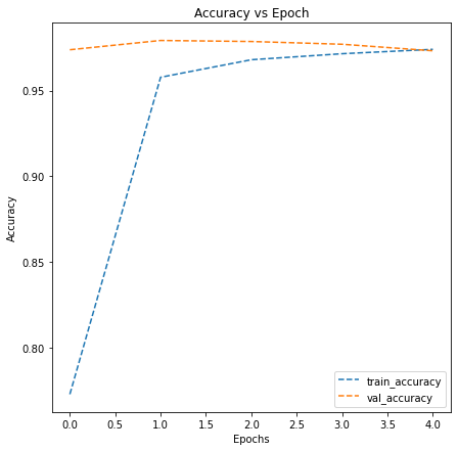
```
82/82 [==============================] - 2s 19ms/step - loss: 0.0815 - accuracy: 0.9800

 Test_Accuracy:- 0.9800000190734863
```

In [103…

```
# function to plot train and validation accuracy
def plot_accuracy(history):
    N = len(history.history["accuracy"])
    plt.figure(figsize=(7,7))
    plt.plot(np.arange(0, N), history.history["accuracy"], label="train_accuracy", ls='--')
    plt.plot(np.arange(0, N), history.history["val_accuracy"], label="val_accuracy", ls='--')
    plt.title("Accuracy vs Epoch")
    plt.xlabel("Epochs")
    plt.ylabel("Accuracy")
    plt.legend(loc="lower right")
```

In [104…

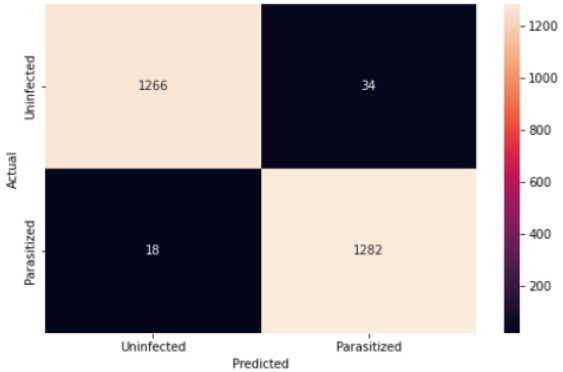
```
# Plotting the Train and validation curves
plot_accuracy(history1)
```



In [106…

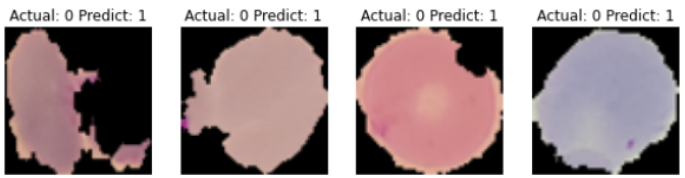
```
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix

pred = model1.predict(test_images)
pred = np.argmax(pred,axis = 1)
y_true = np.argmax(test_labels,axis = 1)

#Printing the classification report
print(classification_report(y_true,pred))

#Plotting the heatmap using confusion matrix
cm = confusion_matrix(y_true,pred)
plt.figure(figsize=(8,5))
sns.heatmap(cm, annot=True,  fmt='.0f', xticklabels=['Uninfected', 'Parasitized'], yticklabels=['Uninfected', 'Parasitized'])
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```

```
82/82 [==============================] - 2s 19ms/step
              precision    recall  f1-score   support

           0       0.99      0.97      0.98      1300
           1       0.97      0.99      0.98      1300

    accuracy                           0.98      2600
   macro avg       0.98      0.98      0.98      2600
weighted avg       0.98      0.98      0.98      2600
```



In [152…

```
# Error samples
index=0
index_errors= []
for label, predict in zip(y_true, pred):
    if label != predict & predict==1:
        index_errors.append(index)
    index +=1

plt.figure(figsize=(10,8))
for i,img_index in zip(range(1,5),random.sample(index_errors,k=16)):
    plt.subplot(1,4,i)
    plt.imshow(test_images[img_index])
    plt.title('Actual: '+str(y_true[img_index])+' Predict: '+str(pred[img_index]))
    plt.axis('off')
plt.show()
```



In [153…

```
# Error samples
index=0
index_errors= []
for label, predict in zip(y_true, pred):
    if label != predict & predict==0:
        index_errors.append(index)
    index +=1

plt.figure(figsize=(10,8))
for i,img_index in zip(range(1,5),random.sample(index_errors,k=16)):
    plt.subplot(1,4,i)
    plt.imshow(test_images[img_index])
    plt.title('Actual: '+str(y_true[img_index])+' Predict: '+str(pred[img_index]))
    plt.axis('off')
plt.show()
```