Detecting malaria infections from pictures of cells: experiments with fractional max pooling

Malaria is a disease that is caused by parasites, which are transmitted into humans by the medium of certain kinds of mosquitoes. It is estimated that 216 million clinical cases of malaria occurred in 2016, which resulted in 445,000 deaths [1] According to the World Health Organisation, early diagnosis of malaria is an important part of fast and effective recovery. Misdiagnosis of malaria may lead to death, which is why high-quality diagnosis is important. The diagnosis of choice recommended by WHO is either microscopy or malaria rapid diagnostic test. Using deep learning, we can apply another layer of diagnosis into detecting malaria in cells, where the human eye might miss it. We propose a model, which utilises fractional max pooling (FMP) [2] to further the accuracy of deep learning models to detect malaria. The results show that replacing max pooling with FMP has substantial effects on the results. To further verify the results, we use ResNet and VGG as benchmarks. This notebook was written and run in Google Colab.

[1] https://www.cdc.gov/malaria/about/faqs.html (https://www.cdc.gov/malaria/about/faqs.html)

[2] https://arxiv.org/abs/1412.6071 (https://arxiv.org/abs/1412.6071)

In [0]:
```
### Let's import tensorflow to make sure that we have a GPU availab
le.
import tensorflow as tf
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
    raise SystemError('GPU device not found')
else:
    print('Found GPU at: {}'.format(device_name))
```

```
Found GPU at: /device:GPU:0
```

In [0]:
```
# Imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from glob import glob
from PIL import Image
from google.colab import files
from zipfile import ZipFile
import os
from tensorflow import keras
from sklearn.model_selection import train_test_split
import seaborn as sns
from sklearn.metrics import confusion_matrix
import gc
from tensorflow.keras import backend as K
from tensorflow.keras.applications.resnet50 import ResNet50
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense, BatchNormalizat
ion, Conv2D, Lambda, Dropout, MaxPooling2D
from tensorflow.keras.applications import VGG16

print(tf.__version__)
```

```
1.13.1
```

In [0]:
```
### Uploading the Kaggle credentials for obtaining the malaria data
set from kaggle.com
files.upload();
```

Choose Files no files selected

Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving kaggle.json to kaggle.json
```

In [0]:
```
### Moving the credentials (kaggle.json) to the correct folder and
downloading the malaria dataset
!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/
!chmod 600 /root/.kaggle/kaggle.json
!kaggle datasets download -d "iarunava/cell-images-for-detecting-ma
laria"
```

```
Downloading cell-images-for-detecting-malaria.zip to /content
 96% 322M/337M [00:06<00:00, 59.9MB/s]
100% 337M/337M [00:06<00:00, 52.4MB/s]
```

In [0]:
```
### Unzipping the downloaded zip file
zip_file = ZipFile('/content/cell-images-for-detecting-malaria.zip'
)
zip_file.extractall()
```

In [0]:
```
print('The two classes of images are in the folders', ' and '.join(
os.listdir("/content/cell_images")))
```

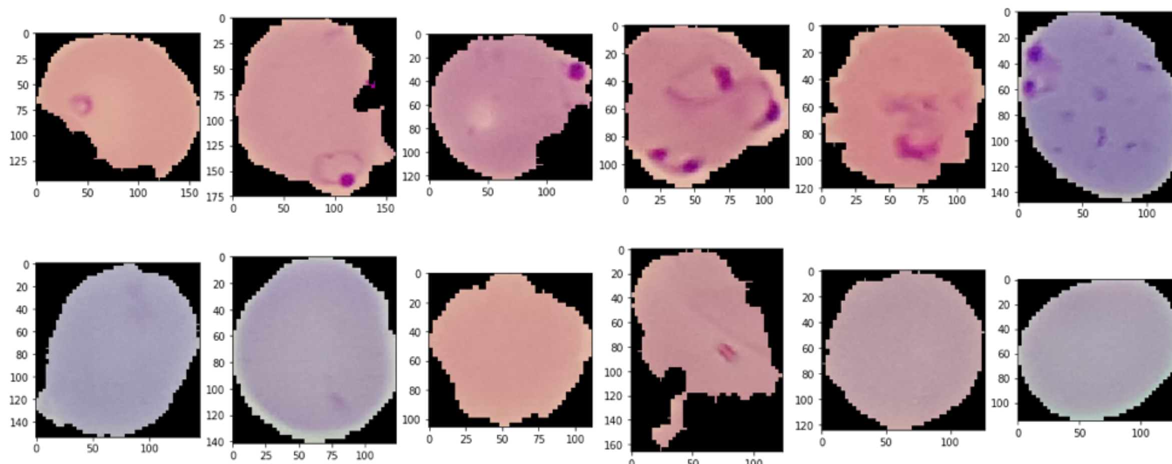
```
The two classes of images are in the folders Uninfected and Parasi
tized
```

In [0]:
```
### Collecting the image paths to the variables parasitized_imgs an
d uninfected_imgs
parasitized_imgs = np.array(sorted(glob("/content/cell_images/Paras
itized/*.png")))
uninfected_imgs = np.array(sorted(glob("/content/cell_images/Uninfe
cted/*.png")))
print('We have', len(parasitized_imgs), 'positive cases and', len(u
ninfected_imgs), 'negative cases of malaria in the dataset.')
print('Total size is', len(parasitized_imgs) + len(uninfected_imgs)
, 'images.')
```

```
We have 13779 positive cases and 13779 negative cases of malaria i
n the dataset.
Total size is 27558 images.
```

In [0]:
```
### Checking some random images
n_cols = 6
fig, ax = plt.subplots(2, n_cols, figsize=(20,8))
for i in range(n_cols):
    rand_image_path = parasitized_imgs[np.random.randint(len(parasi
tized_imgs))]
    img = Image.open(rand_image_path)
    ax[0,i].imshow(img)
for i in range(n_cols):
    rand_image_path = uninfected_imgs[np.random.randint(len(uninfec
ted_imgs))]
    img = Image.open(rand_image_path)
    ax[1,i].imshow(img)

print('Random images from the dataset')
print('Top row: Parasitized, bottom row: Uninfected')
plt.show()
```

```
Random images from the dataset
Top row: Parasitized, bottom row: Uninfected
```



It is quite clear from the images above that darker spots in these cell images are a sign of malaria. The cells with no malaria (bottom row) seem to be quite uniform in color, whereas the cells with malaria (upper row) have darker areas of various size. However, the fourth cell in the bottom row has dark spot which is very similar to that of the first image of the top row. This suggests that the diagnosis is not so straightforward as it first seems.

We will use a standard 70%-30% split of data into training and test sets. We do not have a separate validation set for hyperparameter/model architecture tuning, since we are using existing architectures (VGG and ResNet) and keeping them as they are. In addition, we are more interested in the relative performance differences of fractional and max pooling instead of absolute performance of any model.

In [0]:
```
### Moving the PNG files to train and test folders

np.random.seed(2019) # fixing the random seed for reproducibility
split = 0.7 # we'll use a 70-30 split into training and test sets
train_index = list(np.random.choice(len(parasitized_imgs), replace=
False, size=int(split*len(parasitized_imgs))))

# making the empty folders
os.mkdir(r'/content/train')
os.mkdir(r'/content/test')
os.mkdir(r'/content/train/Parasitized')
os.mkdir(r'/content/test/Parasitized')
os.mkdir(r'/content/train/Uninfected')
os.mkdir(r'/content/test/Uninfected')

# moving the files
for i in range(len(parasitized_imgs)):
    img_path = parasitized_imgs[i]
    img_file = img_path.split('/')[-1]
    if i in train_index:
        os.rename(img_path, r'/content/train/Parasitized/' + img_fi
le)
    else:
        os.rename(img_path, r'/content/test/Parasitized/' + img_fil
e)

for i in range(len(uninfected_imgs)):
    img_path = uninfected_imgs[i]
    img_file = img_path.split('/')[-1]
    if i in train_index:
        os.rename(img_path, r'/content/train/Uninfected/' + img_fil
e)
    else:
        os.rename(img_path, r'/content/test/Uninfected/' + img_file
)
```

In [0]:
```
# Function for making confusion matrices later on
def print_confusion_matrix(confusion_matrix, class_names, figsize =
(10,7), fontsize=14):
    df_cm = pd.DataFrame(
        confusion_matrix, index=class_names, columns=class_names,
    )
    fig = plt.figure(figsize=figsize)
    heatmap = sns.heatmap(df_cm, annot=True, fmt="d", cmap='Blues')
    heatmap.yaxis.set_ticklabels(heatmap.yaxis.get_ticklabels(), ro
tation=0, ha='right', fontsize=fontsize)
    heatmap.xaxis.set_ticklabels(heatmap.xaxis.get_ticklabels(), ro
tation=0, ha='right', fontsize=fontsize)
    plt.title('Confusion Matrix', fontsize=fontsize*1.7)
    plt.ylabel('True label', fontsize=fontsize*1.3)
    plt.xlabel('Predicted label', fontsize=fontsize*1.3)
    return fig
```

Since the images have different sizes, we need to resize them to a constant resolution for the convolutional neural networks. We decided to use 50x50 as our resolution for a good compromise between speed and accuracy. Moreover, we are augmenting the training set with horizontally and vertically flipped images which allows us to have more training data (better and more robust results). This flipping is possible because cell images are quite symmetrical and the orientation of the cell should not affect the classification.

In [0]:
```
### Creating the train_generator and test_generator, both to increa
se variability in the training set and to not have all images in RA
M at the same time

# sizes of the train and test sets
N_train = 2 * len(train_index)
N_test = 2 * (len(parasitized_imgs) - len(train_index))

BATCH_SIZE = 128

train_aug = ImageDataGenerator(horizontal_flip=True, vertical_flip=
True)
test_aug = ImageDataGenerator()

print('Training generator...')
train_generator = train_aug.flow_from_directory(r'/content/train',
target_size=(50, 50), batch_size=BATCH_SIZE, class_mode='binary', s
huffle=True)
print('Test test_generator...')
test_generator = test_aug.flow_from_directory(r'/content/test', tar
get_size=(50, 50), batch_size=BATCH_SIZE, class_mode='binary', shuf
fle=False)

true_labels = test_generator.classes
print('Classes are coded into 0 and 1 as follows:')
print(test_generator.class_indices)
print('We have', N_train, 'images belogning to the training set,',
N_test, 'images to the test set, totalling', N_test+N_train, 'image
s.')
```

```
Training generator...
Found 19290 images belonging to 2 classes.
Test test_generator...
Found 8268 images belonging to 2 classes.
Classes are coded into 0 and 1 as follows:
{'Parasitized': 0, 'Uninfected': 1}
We have 19290 images belogning to the training set, 8268 images to
the test set, totalling 27558 images.
```

RESNET

ResNet is also a well-known network architecture for image classification problems. It is also a very deep network with a genuine solution to the experimental problem of having higher training errors for deep networks than shallower networks. The network utilizes “bottleneck” building blocks to that have skip connections, as formally introduces in the paper. Most onf the filter layers are 3 x 3 that are minimal to identify directions left-right and up-down. Originally, 50, 101 and 152 layered ResNets were constructed, and more layers led to better accuracies. It is noteworthy, that all these had lower complexity than VGG-16/19 despite of their depth. [1]

[1] He, K. et al. (2015) Deep Residual Learning for Image Recognition. [https://arxiv.org/pdf/1512.03385.pdf (https://arxiv.org/pdf/1512.03385.pdf)](https://arxiv.org/pdf/1512.03385.pdf)

In [0]:

```
# res_model1 uses the pretrained ResNet50, therefore we only train
the last layers
resnet = ResNet50(weights='imagenet', include_top=False, input_shap
e=(50, 50, 3), pooling='avg')
print('The ResNet50 has', len(resnet.layers), 'layers in Keras.')
res_model1 = Sequential()
res_model1.add(resnet)

# adding a new untrained classification head to the ResNet
res_model1.add(Flatten())
res_model1.add(BatchNormalization())
res_model1.add(Dense(1024, activation='relu'))
res_model1.add(Dropout(rate=0.5))
res_model1.add(BatchNormalization())
res_model1.add(Dense(512, activation='relu'))
res_model1.add(Dropout(rate=0.5))
res_model1.add(BatchNormalization())
res_model1.add(Dense(1,activation = 'sigmoid'))

# fixing the weights of the pretrained ResNet
res_model1.layers[0].trainable = False
```

```
The ResNet50 has 176 layers in Keras.
```

Note: the number of layers can be calculated differently; this is still the 50-layer version of ResNet.

In [0]:
```
# train the network
res_model1.compile(optimizer='adam', loss='binary_crossentropy', me
trics=['accuracy'])

# First we will train only the classification head
H1 = res_model1.fit_generator(train_generator,
        validation_data=test_generator, steps_per_epoch=N_train //
BATCH_SIZE,
        epochs=5)

# Then we will also train some last layers of the pretrained ResNet
to boost accuracy
# training also the last resnet block...
for layer in res_model1.layers[0].layers[-11:]:
    layer.trainable = True

H2 = res_model1.fit_generator(train_generator,
        validation_data=test_generator, steps_per_epoch=N_train //
BATCH_SIZE,
        epochs=5)

# training also the second to last resnet block...
for layer in res_model1.layers[0].layers[-21:]:
    layer.trainable = True

H3 = res_model1.fit_generator(train_generator,
        validation_data=test_generator, steps_per_epoch=N_train //
BATCH_SIZE,
        epochs=5)

# training also the third to last resnet block...
for layer in res_model1.layers[0].layers[-33:]:
    layer.trainable = True

H4 = res_model1.fit_generator(train_generator,
        validation_data=test_generator, steps_per_epoch=N_train //
BATCH_SIZE,
        epochs=5)
```

```
Epoch 1/5
65/65 [==============================] - 11s 165ms/step - loss: 0.
3719 - acc: 0.8799
151/151 [==============================] - 37s 242ms/step - loss:
0.3129 - acc: 0.8727 - val_loss: 0.3719 - val_acc: 0.8799
Epoch 2/5
65/65 [==============================] - 10s 158ms/step - loss: 0.
2394 - acc: 0.9099
151/151 [==============================] - 34s 222ms/step - loss:
0.2342 - acc: 0.9060 - val_loss: 0.2394 - val_acc: 0.9099
Epoch 3/5
65/65 [==============================] - 10s 157ms/step - loss: 0.
2371 - acc: 0.9104
```

```
151/151 [==============================] - 35s 231ms/step - loss: 0.2122 - acc: 0.9132 - val_loss: 0.2371 - val_acc: 0.9104
Epoch 4/5
65/65 [==============================] - 10s 158ms/step - loss: 0.2213 - acc: 0.9153
151/151 [==============================] - 34s 226ms/step - loss: 0.2000 - acc: 0.9207 - val_loss: 0.2213 - val_acc: 0.9153
Epoch 5/5
65/65 [==============================] - 11s 162ms/step - loss: 0.2178 - acc: 0.9152
151/151 [==============================] - 34s 228ms/step - loss: 0.1986 - acc: 0.9191 - val_loss: 0.2178 - val_acc: 0.9152
Epoch 1/5
65/65 [==============================] - 10s 154ms/step - loss: 0.2141 - acc: 0.9203
151/151 [==============================] - 35s 228ms/step - loss: 0.1885 - acc: 0.9265 - val_loss: 0.2141 - val_acc: 0.9203
Epoch 2/5
65/65 [==============================] - 11s 171ms/step - loss: 0.2159 - acc: 0.9186
151/151 [==============================] - 36s 237ms/step - loss: 0.1855 - acc: 0.9270 - val_loss: 0.2159 - val_acc: 0.9186
Epoch 3/5
65/65 [==============================] - 10s 153ms/step - loss: 0.2142 - acc: 0.9185
151/151 [==============================] - 33s 221ms/step - loss: 0.1816 - acc: 0.9260 - val_loss: 0.2142 - val_acc: 0.9185
Epoch 4/5
65/65 [==============================] - 11s 167ms/step - loss: 0.2121 - acc: 0.9150
151/151 [==============================] - 34s 224ms/step - loss: 0.1768 - acc: 0.9298 - val_loss: 0.2121 - val_acc: 0.9150
Epoch 5/5
65/65 [==============================] - 10s 154ms/step - loss: 0.2133 - acc: 0.9162
151/151 [==============================] - 34s 228ms/step - loss: 0.1733 - acc: 0.9318 - val_loss: 0.2133 - val_acc: 0.9162
Epoch 1/5
65/65 [==============================] - 10s 155ms/step - loss: 0.2086 - acc: 0.9170
151/151 [==============================] - 35s 229ms/step - loss: 0.1671 - acc: 0.9328 - val_loss: 0.2086 - val_acc: 0.9170
Epoch 2/5
65/65 [==============================] - 11s 163ms/step - loss: 0.2153 - acc: 0.9169
151/151 [==============================] - 34s 228ms/step - loss: 0.1683 - acc: 0.9343 - val_loss: 0.2153 - val_acc: 0.9169
Epoch 3/5
65/65 [==============================] - 10s 154ms/step - loss: 0.2059 - acc: 0.9193
151/151 [==============================] - 33s 218ms/step - loss: 0.1614 - acc: 0.9362 - val_loss: 0.2059 - val_acc: 0.9193
Epoch 4/5
```

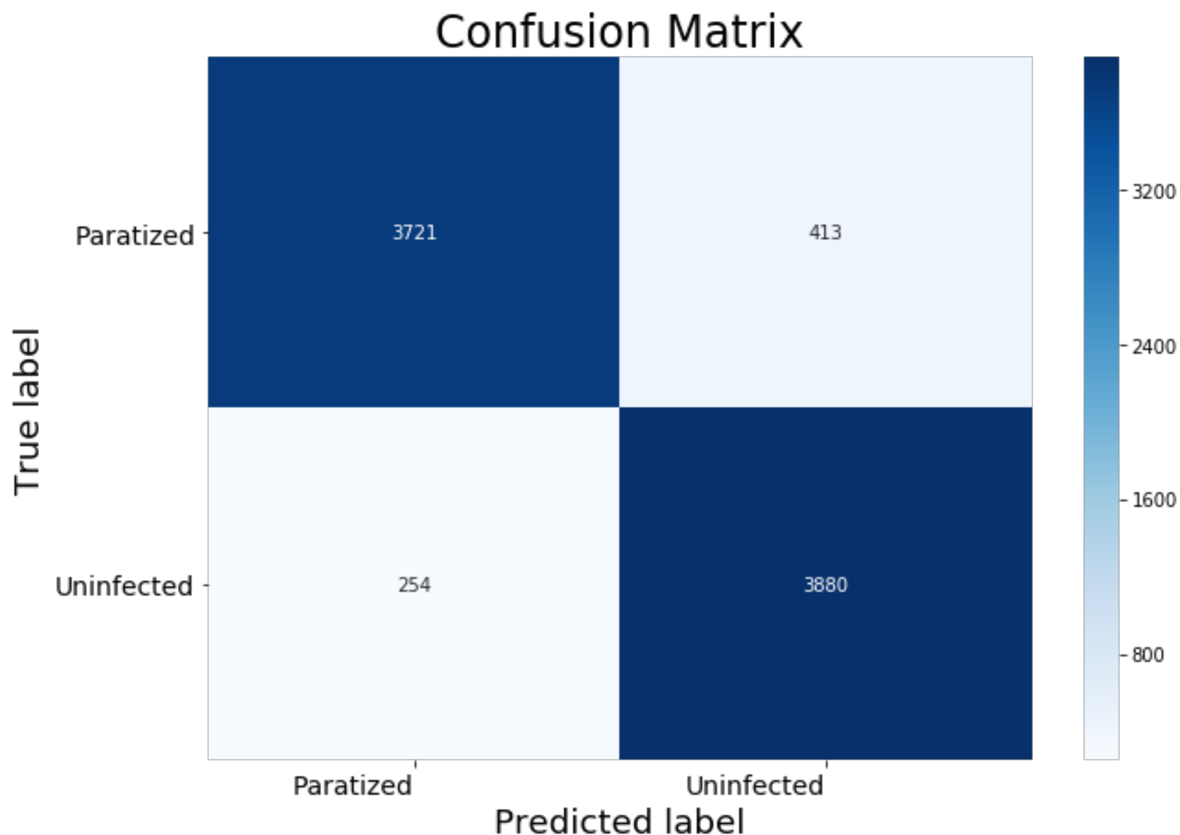
```
65/65 [==============================] - 10s 160ms/step - loss: 0.
2117 - acc: 0.9210
151/151 [==============================] - 33s 220ms/step - loss:
0.1641 - acc: 0.9342 - val_loss: 0.2117 - val_acc: 0.9210
Epoch 5/5
65/65 [==============================] - 10s 154ms/step - loss: 0.
2116 - acc: 0.9187
151/151 [==============================] - 34s 225ms/step - loss:
0.1571 - acc: 0.9367 - val_loss: 0.2116 - val_acc: 0.9187
Epoch 1/5
65/65 [==============================] - 10s 159ms/step - loss: 0.
2150 - acc: 0.9179
151/151 [==============================] - 36s 240ms/step - loss:
0.1521 - acc: 0.9407 - val_loss: 0.2150 - val_acc: 0.9179
Epoch 2/5
65/65 [==============================] - 10s 154ms/step - loss: 0.
2101 - acc: 0.9168
151/151 [==============================] - 34s 227ms/step - loss:
0.1502 - acc: 0.9406 - val_loss: 0.2101 - val_acc: 0.9168
Epoch 3/5
65/65 [==============================] - 11s 167ms/step - loss: 0.
2156 - acc: 0.9213
151/151 [==============================] - 34s 224ms/step - loss:
0.1477 - acc: 0.9428 - val_loss: 0.2156 - val_acc: 0.9213
Epoch 4/5
65/65 [==============================] - 11s 169ms/step - loss: 0.
2085 - acc: 0.9237
151/151 [==============================] - 35s 229ms/step - loss:
0.1446 - acc: 0.9440 - val_loss: 0.2085 - val_acc: 0.9237
Epoch 5/5
65/65 [==============================] - 10s 154ms/step - loss: 0.
2096 - acc: 0.9193
151/151 [==============================] - 33s 218ms/step - loss:
0.1394 - acc: 0.9459 - val_loss: 0.2096 - val_acc: 0.9193
```

Training of the pretrained layers did not yield any significant improvents in accuracy.

In [0]:
```
print('Accuracy of pretrained resnet:', H4.history['val_acc'][-1])

test_steps = N_test // BATCH_SIZE
res_predictions = res_model1.predict_generator(test_generator, test
_steps) # probabilities
res_predictions = np.array(res_predictions>0.5, dtype=np.int32) # c
lass labels
```

```
Accuracy of pretrained resnet: 0.91932756
```

In [0]:
```
cm = confusion_matrix(true_labels, res_predictions) #,['Paratized',
'Uninfected'])
print_confusion_matrix(cm, ['Paratized','Uninfected'], figsize = (1
0,7), fontsize=14);
```



In [0]:
```
### res_model2 uses random initial weights, therefore we train all
the layers ###
# result: takes ages and does not converge to good results...

res_model2 = ResNet50(weights=None, include_top=True, input_shape=(
50, 50, 3), classes=1)

# train the network
res_model2.compile(optimizer='adam', loss='binary_crossentropy', me
trics=['accuracy'])

H3 = res_model2.fit_generator(train_generator,
        validation_data=test_generator, steps_per_epoch=N_train //
BATCH_SIZE,
        epochs=10)

# val_acc: 0.5000 (no predictive power)
```

VGG

VGGNet is a famous image classification network introduced in 2014 ILSVRC, ImageNet Large Scale Visual Recognition Competition. It was originally developed by Oxford's Visual Geometry Group. The network is solely based on only 3x3 convolutional layers and 2x2 max pooling layers followed by fully connected layers. [1] We utilize the pretrained VGG16 network and train only the fully connected layers that summarize the output of the VGG16. Even though the network was originally introduced to take inputs of fixed-size 224 x 224, the architecture succeeds also with the input of size 50 x 50. The most significant differences of VGG to ResNet is that it has less layers and does not have skip connections attributable to ResNet.

[1] Simonyan, K. & Zisserman, A. (2015) Very Deep Convolutional Networks for Large-Scale Image Recognition, https://arxiv.org/pdf/1409.1556.pdf (https://arxiv.org/pdf/1409.1556.pdf)

In [0]:
```
# Clearing RAM for VGG
K.clear_session()

# Import the pretrained VGG16 architecture
vgg = VGG16(include_top=False, weights='imagenet',input_shape=(50,5
0,3))

# Define the complete model
vgg_model1 = Sequential()
vgg_model1.add(vgg)
vgg_model1.add(Flatten())
vgg_model1.add(BatchNormalization())
vgg_model1.add(Dense(1024, activation='relu'))
vgg_model1.add(Dropout(rate=0.5))
vgg_model1.add(BatchNormalization())
vgg_model1.add(Dense(512, activation='relu'))
vgg_model1.add(Dropout(rate=0.5))
vgg_model1.add(BatchNormalization())
vgg_model1.add(Dense(1,activation = 'sigmoid'))


# Initailly prevent the imported VGG16 layers from learning
vgg_model1.layers[0].trainable = False

# Compile model
vgg_model1.compile(optimizer='adam', loss='binary_crossentropy', me
trics=['accuracy'])

# Train the model and gradually release last layers of the VGG16 fo
r training
V1 = vgg_model1.fit_generator(train_generator,
        validation_data=test_generator, steps_per_epoch=N_train //
BATCH_SIZE,
        epochs=4)

for layer in vgg_model1.layers[0].layers[-3:]:
    layer.trainable = True

V2 = vgg_model1.fit_generator(train_generator,
        validation_data=test_generator, steps_per_epoch=N_train //
BATCH_SIZE,
        epochs=5)

for layer in vgg_model1.layers[0].layers[-6:-3]:
    layer.trainable = True

V3 = vgg_model1.fit_generator(train_generator,
        validation_data=test_generator, steps_per_epoch=N_train //
BATCH_SIZE,
        epochs=5)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/ten
sorflow/python/keras/layers/core.py:143: calling dropout (from ten
```

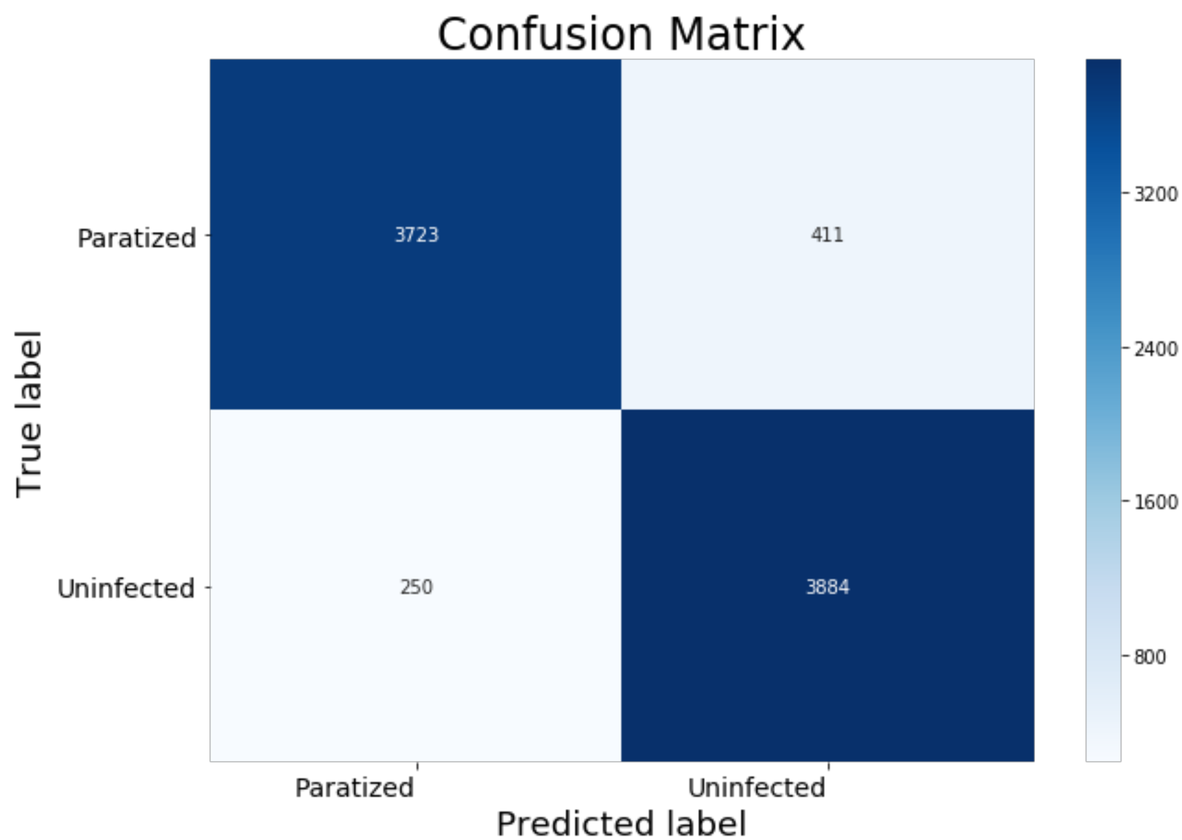
```
sorflow.python.ops.nn_ops) with keep_prob is deprecated and will b
e removed in a future version.
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `r
ate = 1 - keep_prob`.
Epoch 1/4
65/65 [==============================] - 9s 145ms/step - loss: 0.2
550 - acc: 0.9055
151/151 [==============================] - 32s 210ms/step - loss:
0.3113 - acc: 0.8803 - val_loss: 0.2550 - val_acc: 0.9055
Epoch 2/4
65/65 [==============================] - 9s 143ms/step - loss: 0.2
310 - acc: 0.9112
151/151 [==============================] - 32s 210ms/step - loss:
0.2540 - acc: 0.9007 - val_loss: 0.2310 - val_acc: 0.9112
Epoch 3/4
65/65 [==============================] - 9s 143ms/step - loss: 0.2
266 - acc: 0.9115
151/151 [==============================] - 30s 201ms/step - loss:
0.2327 - acc: 0.9092 - val_loss: 0.2266 - val_acc: 0.9115
Epoch 4/4
65/65 [==============================] - 10s 157ms/step - loss: 0.
2253 - acc: 0.9147
151/151 [==============================] - 32s 210ms/step - loss:
0.2216 - acc: 0.9137 - val_loss: 0.2253 - val_acc: 0.9147
Epoch 1/5
65/65 [==============================] - 11s 170ms/step - loss: 0.
2294 - acc: 0.9117
151/151 [==============================] - 34s 222ms/step - loss:
0.2210 - acc: 0.9116 - val_loss: 0.2294 - val_acc: 0.9117
Epoch 2/5
65/65 [==============================] - 9s 142ms/step - loss: 0.2
427 - acc: 0.9100
151/151 [==============================] - 30s 201ms/step - loss:
0.2150 - acc: 0.9157 - val_loss: 0.2427 - val_acc: 0.9100
Epoch 3/5
65/65 [==============================] - 9s 143ms/step - loss: 0.2
226 - acc: 0.9157
151/151 [==============================] - 32s 211ms/step - loss:
0.2114 - acc: 0.9145 - val_loss: 0.2226 - val_acc: 0.9157
Epoch 4/5
65/65 [==============================] - 9s 143ms/step - loss: 0.2
206 - acc: 0.9168
151/151 [==============================] - 30s 201ms/step - loss:
0.2053 - acc: 0.9194 - val_loss: 0.2206 - val_acc: 0.9168
Epoch 5/5
65/65 [==============================] - 11s 169ms/step - loss: 0.
2191 - acc: 0.9193
151/151 [==============================] - 32s 212ms/step - loss:
0.2042 - acc: 0.9188 - val_loss: 0.2191 - val_acc: 0.9193
Epoch 1/5
65/65 [==============================] - 9s 144ms/step - loss: 0.2
199 - acc: 0.9208
```

```
151/151 [==============================] - 32s 213ms/step - loss:
0.2033 - acc: 0.9204 - val_loss: 0.2199 - val_acc: 0.9208
Epoch 2/5
65/65 [==============================] - 9s 144ms/step - loss: 0.2
213 - acc: 0.9173
151/151 [==============================] - 30s 202ms/step - loss:
0.1964 - acc: 0.9221 - val_loss: 0.2213 - val_acc: 0.9173
Epoch 3/5
65/65 [==============================] - 9s 144ms/step - loss: 0.2
170 - acc: 0.9165
151/151 [==============================] - 32s 212ms/step - loss:
0.2018 - acc: 0.9192 - val_loss: 0.2170 - val_acc: 0.9165
Epoch 4/5
65/65 [==============================] - 9s 143ms/step - loss: 0.2
141 - acc: 0.9186
151/151 [==============================] - 31s 202ms/step - loss:
0.1929 - acc: 0.9259 - val_loss: 0.2141 - val_acc: 0.9186
Epoch 5/5
65/65 [==============================] - 10s 158ms/step - loss: 0.
2165 - acc: 0.9201
151/151 [==============================] - 31s 208ms/step - loss:
0.1906 - acc: 0.9251 - val_loss: 0.2165 - val_acc: 0.9201
```

In [0]:

```
print('Accuracy of pretrained VGG:', V3.history['val_acc'][-1])

test_steps = N_test // BATCH_SIZE
vgg_predictions = vgg_model1.predict_generator(test_generator, test
_steps) # probabilities
vgg_predictions = np.array(vgg_predictions>0.5, dtype=np.int32) # c
lass labels
cm = confusion_matrix(true_labels, vgg_predictions) #,['Paratized',
'Uninfected'])
print_confusion_matrix(cm, ['Paratized','Uninfected'], figsize = (1
0,7), fontsize=14);
```

```
Accuracy of pretrained VGG: 0.92005324
```



2x2 Max Pooling - For Comparison

CNN utilizing 2x2 Max Pooling to use for comparsions.

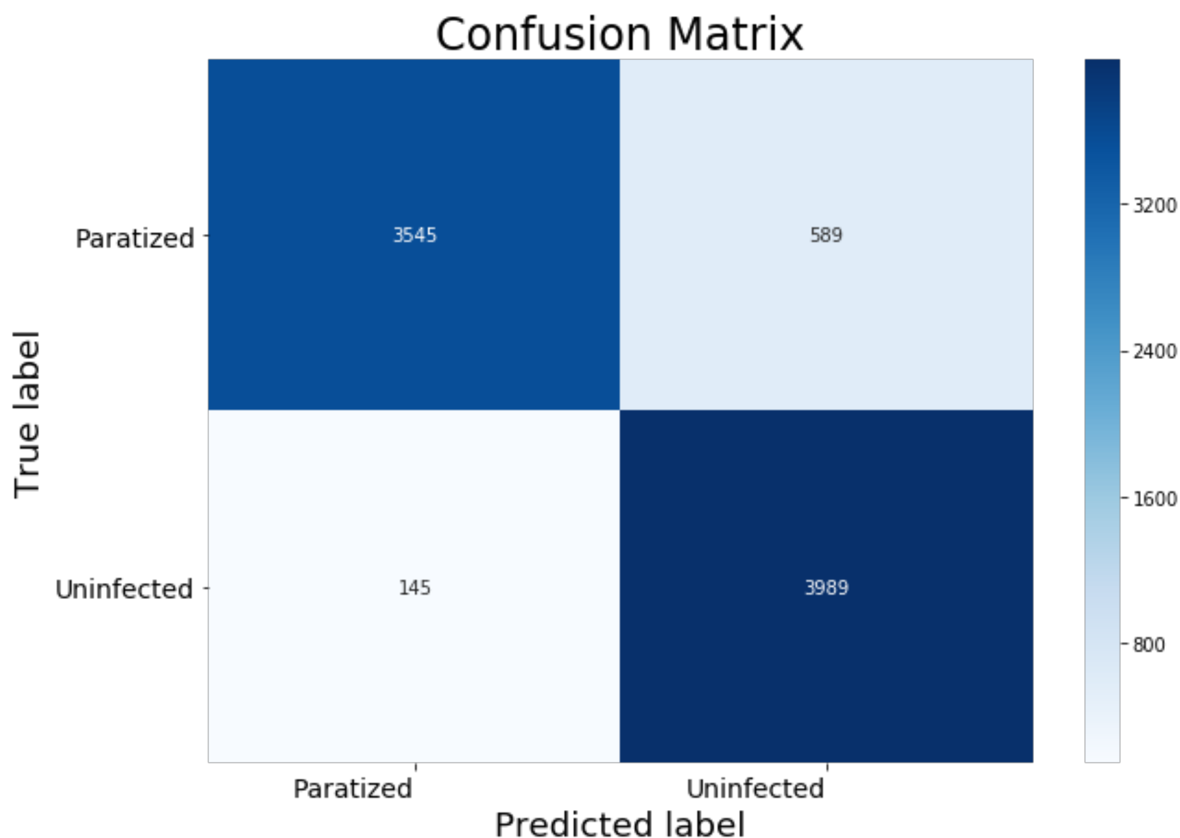
In [0]:
```
#Create the model
comp_model = Sequential()

#Build the convolutional network
inputShape= (50,50,3)
comp_model=Sequential()

comp_model.add(Conv2D(32, (3,3), activation = 'relu', input_shape =
inputShape))
comp_model.add(BatchNormalization(axis = -1))
comp_model.add(MaxPooling2D(2,2))

comp_model.add(Conv2D(32, (1,1), activation = 'relu'))
comp_model.add(BatchNormalization(axis = -1))
comp_model.add(MaxPooling2D(2,2))

comp_model.add(Conv2D(32, (3,3), activation = 'relu'))
comp_model.add(BatchNormalization(axis = -1))
comp_model.add(MaxPooling2D(2,2))

comp_model.add(Conv2D(32, (3,3), activation = 'relu'))
comp_model.add(BatchNormalization(axis = -1))
comp_model.add(MaxPooling2D(2,2))

comp_model.add(Flatten())

comp_model.add(Dense(512, activation = 'relu'))
comp_model.add(BatchNormalization(axis = -1))
comp_model.add(Dropout(0.5))
comp_model.add(Dense(1, activation = 'sigmoid'))
```

In [0]:
```
comp_model.compile(optimizer='adam', loss='binary_crossentropy', me
trics=['accuracy'])
```

In [0]:
```
comp_model.fit_generator(train_generator,
        validation_data=test_generator, steps_per_epoch=N_train //
BATCH_SIZE,
        epochs=10)

comp_model.save_weights("comp_CNN.h5")
```

```
Epoch 1/10
65/65 [==============================] - 9s 144ms/step - loss: 0.7
882 - acc: 0.5011
151/151 [==============================] - 31s 206ms/step - loss:
0.5542 - acc: 0.7540 - val_loss: 0.7882 - val_acc: 0.5011
Epoch 2/10
65/65 [==============================] - 9s 133ms/step - loss: 0.2
919 - acc: 0.8824
151/151 [==============================] - 29s 189ms/step - loss:
0.3088 - acc: 0.8759 - val_loss: 0.2919 - val_acc: 0.8824
Epoch 3/10
65/65 [==============================] - 9s 133ms/step - loss: 0.2
932 - acc: 0.8824
151/151 [==============================] - 29s 190ms/step - loss:
0.2489 - acc: 0.9048 - val_loss: 0.2932 - val_acc: 0.8824
Epoch 4/10
65/65 [==============================] - 9s 132ms/step - loss: 0.3
152 - acc: 0.8693
151/151 [==============================] - 30s 198ms/step - loss:
0.2217 - acc: 0.9142 - val_loss: 0.3152 - val_acc: 0.8693
Epoch 5/10
65/65 [==============================] - 9s 132ms/step - loss: 0.2
397 - acc: 0.9069
151/151 [==============================] - 28s 189ms/step - loss:
0.2087 - acc: 0.9214 - val_loss: 0.2397 - val_acc: 0.9069
Epoch 6/10
65/65 [==============================] - 9s 133ms/step - loss: 0.2
478 - acc: 0.9099
151/151 [==============================] - 29s 189ms/step - loss:
0.2032 - acc: 0.9250 - val_loss: 0.2478 - val_acc: 0.9099
Epoch 7/10
65/65 [==============================] - 9s 133ms/step - loss: 0.2
088 - acc: 0.9224
151/151 [==============================] - 30s 198ms/step - loss:
0.1977 - acc: 0.9235 - val_loss: 0.2088 - val_acc: 0.9224
Epoch 8/10
65/65 [==============================] - 10s 156ms/step - loss: 0.
2150 - acc: 0.9214
151/151 [==============================] - 30s 199ms/step - loss:
0.1917 - acc: 0.9284 - val_loss: 0.2150 - val_acc: 0.9214
Epoch 9/10
65/65 [==============================] - 10s 152ms/step - loss: 0.
2051 - acc: 0.9291
151/151 [==============================] - 30s 199ms/step - loss:
0.1900 - acc: 0.9276 - val_loss: 0.2051 - val_acc: 0.9291
Epoch 10/10
65/65 [==============================] - 9s 141ms/step - loss: 0.2
267 - acc: 0.9112
151/151 [==============================] - 29s 193ms/step - loss:
0.1833 - acc: 0.9306 - val_loss: 0.2267 - val_acc: 0.9112
```

In [0]:
```
test_steps = N_test // BATCH_SIZE
predictions = comp_model.predict_generator(test_generator, test_steps) # probabilities
predictions = np.array(predictions>0.5, dtype=np.int32) # class labels

cm = confusion_matrix(true_labels, predictions) #,['Paratized','Uninfected'])

print_confusion_matrix(cm, ['Paratized','Uninfected'], figsize = (10,7), fontsize=14);
```



Fractional Max Pooling

Nearly all convolutional networks make use of some sort of spatial pooling. The choice of spatial pooling is most often the alpha times alpha max pooling, with the chosen alpha being most often 2 (MP2). Max pooling reduces the size of the layers, by discarding 75% of your data while still keeping the most relevant data. The side effect from this, however, is that convolutional layers cannot simply be alternated with pooling layers, since the spatial size is reduced rapidly. Thus for this problem, we try to achieve better results by replacing MP2 layers with fractional max pooling (FMP) which uses the alpha sqrt(2), which lowers the spatial reduction between layers and allows us to put more convolutional layers after one another.

In [0]:
```
pooling_ratio = [1.0, 1.44, 1.44, 1.0]
def fractional_max_pool(x):
    return tf.nn.fractional_max_pool(x, pooling_ratio)[0]
```

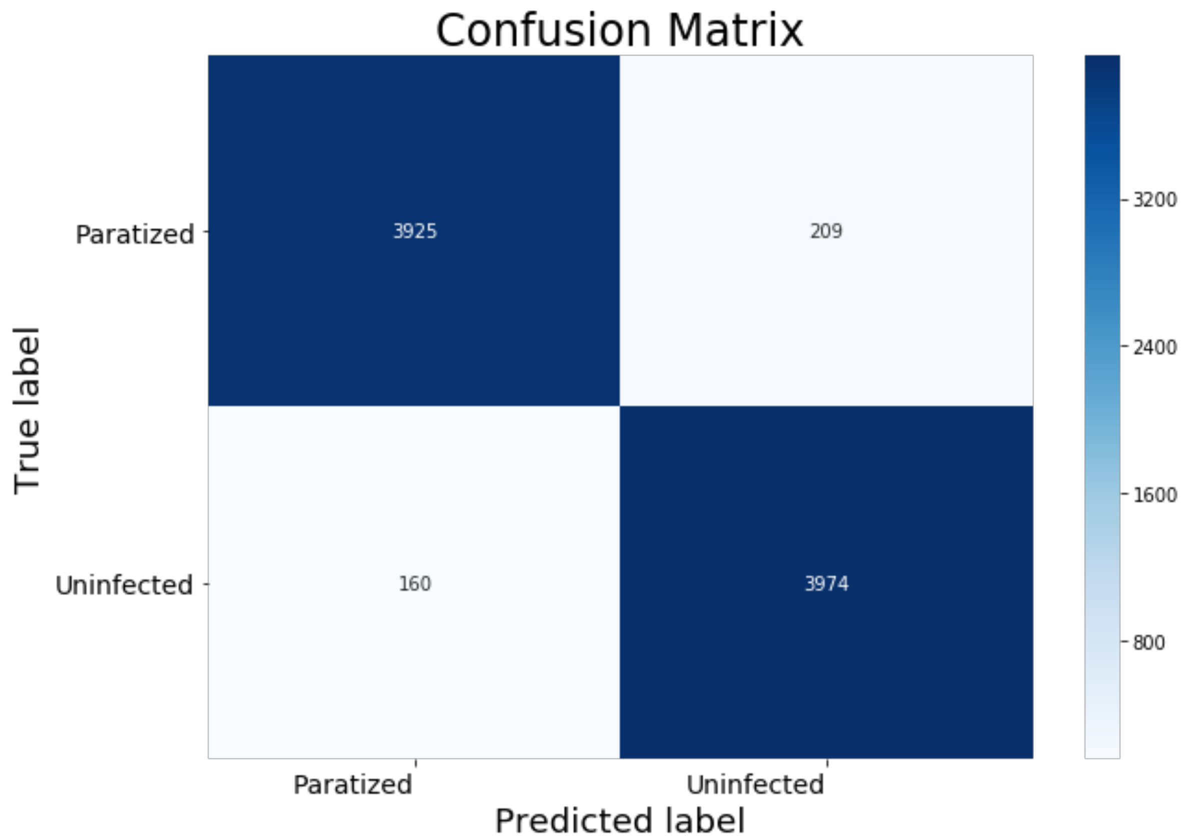
In [0]:
```
#Basic Convolutional Deep Neural Network, which utlizes Fractional
Max Pooling instead of the common 2x2 Max Pooling.

#Create the model
model = Sequential()

#Build the convolutional network
inputShape= (50,50,3)
model=Sequential()

model.add(Conv2D(32, (3,3), activation = 'relu', input_shape = inpu
tShape))
model.add(BatchNormalization(axis = -1))
model.add(Lambda(fractional_max_pool))

model.add(Conv2D(32, (1,1), activation = 'relu'))
model.add(BatchNormalization(axis = -1))
model.add(Lambda(fractional_max_pool))

model.add(Conv2D(32, (3,3), activation = 'relu'))
model.add(BatchNormalization(axis = -1))
model.add(Lambda(fractional_max_pool))

model.add(Conv2D(32, (3,3), activation = 'relu'))
model.add(BatchNormalization(axis = -1))
model.add(Lambda(fractional_max_pool))

model.add(Conv2D(32, (3,3), activation = 'relu'))
model.add(BatchNormalization(axis = -1))
model.add(Lambda(fractional_max_pool))


model.add(Flatten())

model.add(Dense(512, activation = 'relu'))
model.add(BatchNormalization(axis = -1))
model.add(Dropout(0.5))
model.add(Dense(1, activation = 'sigmoid'))
```

In [0]:
```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics
=['accuracy'])
```

In [0]:
```
model.fit_generator(train_generator,
        validation_data=test_generator, steps_per_epoch=N_train //
BATCH_SIZE,
        epochs=10)

model.save_weights("frac_CNN.h5")
```

```
Epoch 1/10
65/65 [==============================] - 11s 166ms/step - loss: 0.
4245 - acc: 0.8310
151/151 [==============================] - 42s 279ms/step - loss:
0.4406 - acc: 0.8164 - val_loss: 0.4245 - val_acc: 0.8310
Epoch 2/10
65/65 [==============================] - 9s 144ms/step - loss: 0.2
390 - acc: 0.9168
151/151 [==============================] - 40s 263ms/step - loss:
0.1921 - acc: 0.9337 - val_loss: 0.2390 - val_acc: 0.9168
Epoch 3/10
65/65 [==============================] - 11s 162ms/step - loss: 0.
1876 - acc: 0.9396
151/151 [==============================] - 41s 272ms/step - loss:
0.1606 - acc: 0.9448 - val_loss: 0.1876 - val_acc: 0.9396
Epoch 4/10
65/65 [==============================] - 9s 145ms/step - loss: 0.2
180 - acc: 0.9354
151/151 [==============================] - 40s 263ms/step - loss:
0.1480 - acc: 0.9511 - val_loss: 0.2180 - val_acc: 0.9354
Epoch 5/10
65/65 [==============================] - 10s 161ms/step - loss: 0.
1673 - acc: 0.9446
151/151 [==============================] - 41s 271ms/step - loss:
0.1415 - acc: 0.9526 - val_loss: 0.1673 - val_acc: 0.9446
Epoch 6/10
65/65 [==============================] - 10s 159ms/step - loss: 0.
1631 - acc: 0.9482
151/151 [==============================] - 41s 272ms/step - loss:
0.1371 - acc: 0.9544 - val_loss: 0.1631 - val_acc: 0.9482
Epoch 7/10
65/65 [==============================] - 10s 157ms/step - loss: 0.
1577 - acc: 0.9462
151/151 [==============================] - 41s 272ms/step - loss:
0.1310 - acc: 0.9559 - val_loss: 0.1577 - val_acc: 0.9462
Epoch 8/10
65/65 [==============================] - 9s 145ms/step - loss: 0.1
540 - acc: 0.9517
151/151 [==============================] - 40s 267ms/step - loss:
0.1296 - acc: 0.9582 - val_loss: 0.1540 - val_acc: 0.9517
Epoch 9/10
65/65 [==============================] - 10s 155ms/step - loss: 0.
1378 - acc: 0.9539
151/151 [==============================] - 41s 271ms/step - loss:
0.1270 - acc: 0.9580 - val_loss: 0.1378 - val_acc: 0.9539
Epoch 10/10
65/65 [==============================] - 9s 145ms/step - loss: 0.1
340 - acc: 0.9549
151/151 [==============================] - 40s 263ms/step - loss:
0.1246 - acc: 0.9584 - val_loss: 0.1340 - val_acc: 0.9549
```

In [0]:
```
test_steps = N_test // BATCH_SIZE
predictions = model.predict_generator(test_generator, test_steps) #
probabilities
predictions = np.array(predictions>0.5, dtype=np.int32) # class lab
els

cm = confusion_matrix(true_labels, predictions) #,['Paratized','Uni
nfected'])

print_confusion_matrix(cm, ['Paratized','Uninfected'], figsize = (1
0,7), fontsize=14);
```



Results and Conclusions

Results

The validation errors of all the four models are listed below:

Model	Validation error
Resnet	0.9193
VGG	0.9201
2x2 Max Pooling	0.9112
Fractional Max Pooling	0.9549

Conclusions

In this analysis we discovered whether adoption of fractional max pooling provides distinguisable improvements to the state of art solutions in the context of malaria detection from cells.

The results of our experiments clearly show that the fractional max pooling architecture overcomes traditional 2x2 max pooling, VGGNet and Resnet in the context of malaria detection. Fractional max pooling has 4.39 %-points better predictive power than 2x2 max pooling, and thus it should be favored in malaria detection. Rather surprisingly, the model perfomed better than any of the state-of-art solutions included in the analysis.

It is noteworthy that fractional max pooling did have substantially smaller fraction of cases that were identified as uninfected even though they were infected (upper right corner of confusion matrix). This error has most relevance to individual patients as their medical examinations might be terminated due to this classification error. Similarly, the fraction of cases evaluated as paratized that were really uninfected (lower left corner of confusion matrix) was significantly lower with fractional max pooling than any other model. In practice, this type of errors are likely to cause waste of resources as unnecessary treatments are initiated. Thus, utilizing fractional max pooling over any other method is likely lead to more effective use of resources and it should be encouraged.