# Malaria Detector CNN

**F. Pezzuti, P. Tempesti**

**Computational Intelligence & Deep Learning**

MSc in *Artificial Intelligence and Data Engineering*

University of Pisa

**Academic Year 2022/2023**

# Contents

## 1 Introduction

Malaria is an acute febrile illness caused by Plasmodium parasites, which are spread to people through the bites of infected female Anopheles mosquitoes.

The first symptoms – fever, headache and chills – usually appear 10–15 days after the infective mosquito bite and may be mild and difficult to recognize as malaria. Left untreated, P. falciparum malaria can progress to severe illness and death within a period of 24 hours.

In 2021, nearly half of the world's population was at risk of malaria. Some population groups are at considerably higher risk of contracting malaria and developing severe disease: infants, children under 5 years of age, pregnant women and patients with HIV/AIDS, as well as people with low immunity moving to areas with intense malaria transmission such as migrant workers, mobile populations and travellers.

Our aim is to correctly classify as infected or not infected by malaria a cell, using its thin blood smear slide image.

### 1.1 Dataset

We're willing to use the "malaria" dataset from the Tensorflow catalog.

`https://www.tensorflow.org/datasets/catalog/malaria?hl=en`

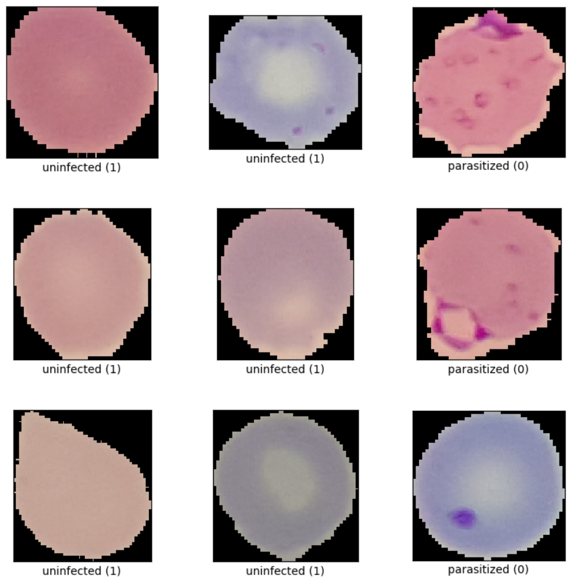

Figure 1: Example of dataset's sample images

### 1.1.1 Properties of the dataset

The dataset is composed by 27558 images, already labeled as '*uninfected*' and '*parasitized*'; the dataset is not split, but the classes are perfectly balanced.

The dataset weight is 317.62MB, so its size is small enough to be completely used in our Colab experiments.

### 1.1.2 Dataset's feature structure

The feature structure of the dataset is the following:

```
FeaturesDict({
    'image': Image(shape=(None, None, 3), dtype=uint8),
    'label': ClassLabel(shape=(), dtype=int64, num_classes=2),
})
```

As we can see from the `Image` shape, our images are colorized but they have different dimensions, we'll need to resize most of them.

## 2 Related works

Classification of malaria-infected blood cells using neural networks is an active area of research in the field of medical image analysis. With the high prevalence of malaria in many parts of the world, accurately and efficiently detecting infected cells is crucial for early diagnosis and effective treatment. In recent years, various machine learning techniques, particularly deep neural networks, have been applied to automate the process of malaria diagnosis using microscopic images of blood smears.

This subsection reviews some of the most relevant works in this area, highlighting the different architectures, data pre-processing techniques, and performance metrics used in these studies. Both of these experiments were performed on the same *malaria* dataset used in this study.

A. H. D. Alnoussairi and A. A. Ibrahim [1] proposed a transfer learning approach using three different pre-trained network as features extractors: VGG-19, ResNet-50 and MobileNetV2; they obtained an accuracy and an f1-score of 1 with all of the networks they used, showing a really high effectiveness of their approach.

Speaking about CNN built from scratch, Z. Liang *et al.* [2] Presented a 4-blocks CNN with 6 convolutional layer and the last block formed by three fully connected layers. They obtained an accuracy and an f1-score of 97.36

# 3 Preprocessing

In our preprocessing pipeline we just had to perform dataset splitting and image rescaling since, as we can see from the histogram reported in image 2, the classes were already balanced, so no data balancing was needed.

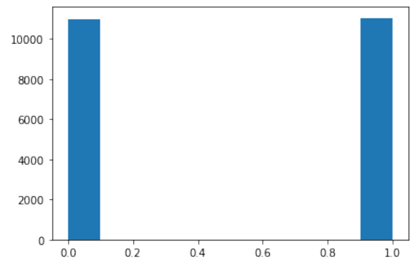

Figure 2: Data balancing histogram

## 3.1 Dataset splits

As first operation of the preprocessing pipeline, we splitted the 27.558 images of the malaria dataset in training set, test set and validation set according to the percentages shown in table 1.

| | **Training set** | **Test set** | **Validation set** |
|---|---|---|---|
| **Percentage** | 80% | 15% | 5% |
| **N. samples** | 22.047 | 4.134 | 1.377 |

Table 1: Dataset splits

The images to be included in each of the splits were randomly chosen in such a way to maintain the same distribution as the original malaria dataset.

## 3.2 Image rescaling

For each split of the dataset we applied image rescaling. In particular, all the images have been resized to a $120x120$ size since most of the images of the malaria dataset were of a dimension of around $120x120$.

## 4 CNN from scratch

The first step of our study was to define some custom architecture for solving our problem. In the first step, we tried a *trial and error* approach to establish what will be the most suitable network structure for our case study.

We have to say that even with the first and smallest model we tried, we reached some really good results, so we performed some addictions/modifications to the network in order to further increase the perormances of the base model.

### 4.1 Basic CNN

In the first experiment, we tried a simple convolutional neural network with 3 convolutional layers and *ReLU* as activation function for the model.

```
Model: "basicCNN"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 120, 120, 3)]     0

 conv2d (Conv2D)             (None, 118, 118, 32)      896

 max_pooling2d (MaxPooling2D  (None, 59, 59, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 57, 57, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 28, 28, 64)       0
 2D)

 conv2d_2 (Conv2D)           (None, 26, 26, 128)       73856

 max_pooling2d_2 (MaxPooling  (None, 13, 13, 128)      0
 2D)

 flatten (Flatten)           (None, 21632)             0

 dense (Dense)               (None, 1)                 21633

=================================================================
Total params: 114,881
Trainable params: 114,881
Non-trainable params: 0
_________________________________________________________________
```

As we can see from the graphs 3 and 4 the model converges in really few epochs, reaching great results in terms of validation accuracy and loss, and without overfitting.

We performed a test of the model against the test set, and the results are shown in the table 2.

Even against the test set, the model performs really well, with an accuracy near to 95%; even the area under the ROC curve is quite perfect, as we can see

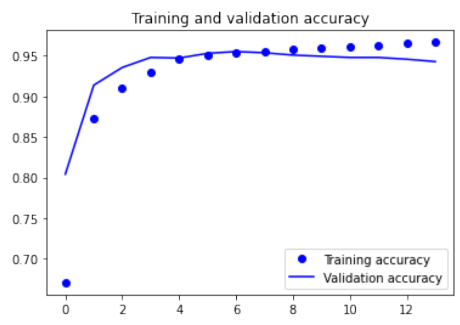

Figure 3: Training/validation accuracy

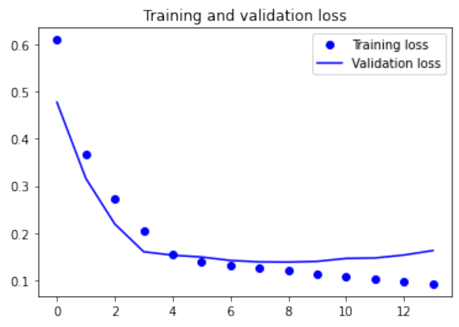

Figure 4: training/validation loss

in figure 6.

| | **Precision** | **Recall** | **F1-score** | **Support** |
|---|---|---|---|---|
| **0** | 0.9781 | 0.9204 | 0.9484 | 2086 |
| **1** | 0.9235 | 0.9790 | 0.9505 | 2048 |
| **Accuracy** | | | 0.9494 | 4134 |
| **Macro Avg** | 0.9508 | 0.9497 | 0.9494 | 4134 |
| **Weighted Avg** | 0.9511 | 0.9494 | 0.9494 | 4134 |

Table 2: Training results obtained using the basic network

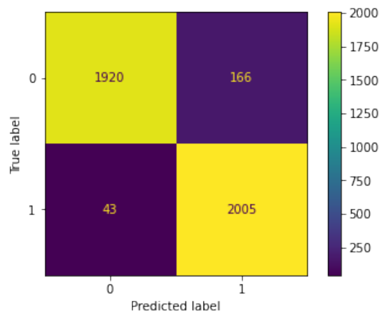

Figure 5: Confusion matrix

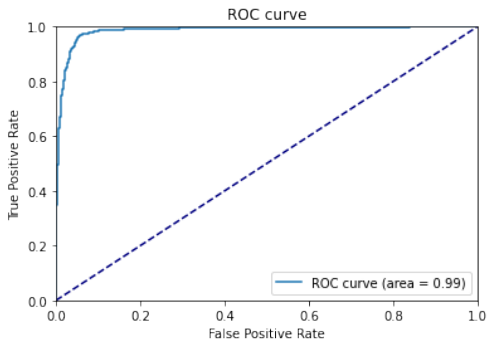

Figure 6: ROC curve

## 4.2 Bigger CNN

The results obtained with the basic model are pretty satisfying, but we wanted to try to improve furhter our performances.

In the second experiment we added a new convolutional layer on top of the basic architecture, in order to refine the classification leveraging on the absence of overfitting.

```
Model: "biggerCNN"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 120, 120, 3)]     0

 conv2d_3 (Conv2D)           (None, 118, 118, 32)      896

 max_pooling2d_3 (MaxPooling  (None, 59, 59, 32)       0
 2D)

 conv2d_4 (Conv2D)           (None, 57, 57, 64)        18496

 max_pooling2d_4 (MaxPooling  (None, 28, 28, 64)       0
 2D)

 conv2d_5 (Conv2D)           (None, 26, 26, 128)       73856

 max_pooling2d_5 (MaxPooling  (None, 13, 13, 128)      0
 2D)

 conv2d_6 (Conv2D)           (None, 11, 11, 256)       295168

 max_pooling2d_6 (MaxPooling  (None, 5, 5, 256)        0
 2D)

 flatten_1 (Flatten)         (None, 6400)              0

 dense_1 (Dense)             (None, 1)                 6401

=================================================================
Total params: 394,817
Trainable params: 394,817
Non-trainable params: 0
_________________________________________________________________
```

Either with this model the convergence is reached in few epochs, and as after 10 epochs the model started to overfit (figure 8.

We tested the model against the test set, and the results are shown in the table 3.

All the metrics are slightly lower than the basic model, but the overall behaviour is the same.

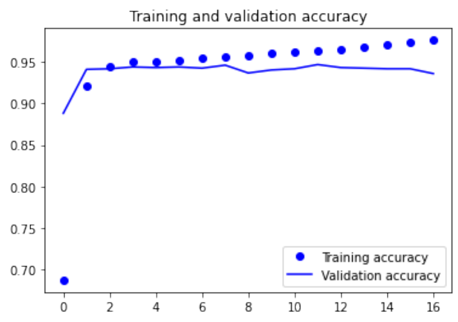

Figure 7: Training/validation accuracy

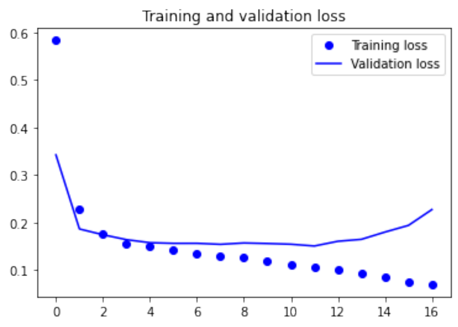

Figure 8: training/validation loss

| | **Precision** | **Recall** | **F1-score** | **Support** |
|---|---|---|---|---|
| **0** | 0.9735 | 0.9147 | 0.9432 | 2086 |
| **1** | 0.9181 | 0.9746 | 0.9455 | 2048 |
| **Accuracy** | | | 0.9444 | 4134 |
| **Macro Avg** | 0.9458 | 0.9446 | 0.9443 | 4134 |
| **Weighted Avg** | 0.9461 | 0.9444 | 0.9443 | 4134 |

Table 3: Training results obtained using the bigger network

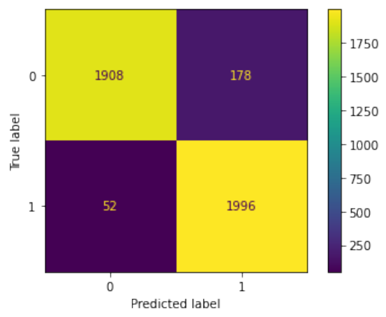

Figure 9: Confusion matrix

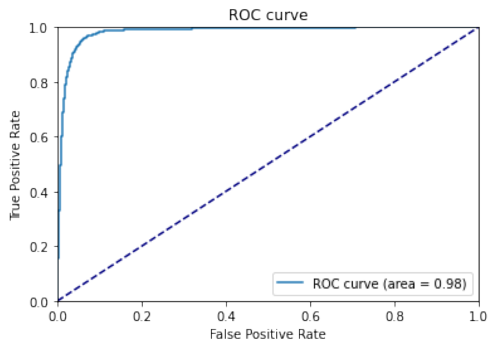

Figure 10: ROC curve

## 4.3 Bigger CNN with a 256-neuron Dense Layer

In order to improve the performances of the bigger network, we decided to introduce a 256-neuron dense layer on top of that architecture. We wanted to see if the additional dense layer is capable to better generalize the network output, which is decreased adding a convolutional layer.

```
Model: "256denseCNN"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 120, 120, 3)]     0

 conv2d_7 (Conv2D)           (None, 118, 118, 32)      896

 max_pooling2d_7 (MaxPooling  (None, 59, 59, 32)       0
 2D)

 conv2d_8 (Conv2D)           (None, 57, 57, 64)        18496

 max_pooling2d_8 (MaxPooling  (None, 28, 28, 64)       0
 2D)

 conv2d_9 (Conv2D)           (None, 26, 26, 128)       73856

 max_pooling2d_9 (MaxPooling  (None, 13, 13, 128)      0
 2D)

 conv2d_10 (Conv2D)          (None, 11, 11, 256)       295168

 max_pooling2d_10 (MaxPoolin  (None, 5, 5, 256)        0
 g2D)

 flatten_2 (Flatten)         (None, 6400)              0

 dense_2 (Dense)             (None, 256)               1638656

 dense_3 (Dense)             (None, 1)                 257

=================================================================
Total params: 2,027,329
Trainable params: 2,027,329
Non-trainable params: 0
_________________________________________________________________
```

With the dense layer, the model reached a stable validation loss after more epochs than the previous experiments, but even if the validation loss starts increasing importantly right after the best epoch (epoch 11), the validation accuracy remains stable.

We tested also this configuration, and the table 4 summarizes them.

In this experiment we reached the maximum values for each metric, without changing the overal behaviour of the classifier.

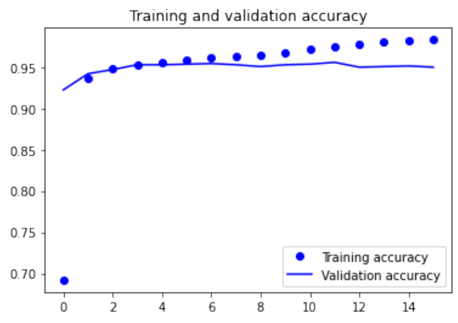

Figure 11: Training/validation accuracy

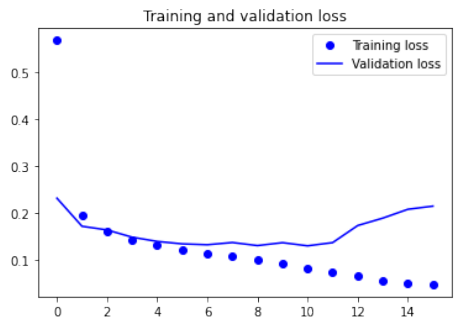

Figure 12: training/validation loss

| | **Precision** | **Recall** | **F1-score** | **Support** |
|---|---|---|---|---|
| **0** | 0.9749 | 0.9319 | 0.9529 | 2086 |
| **1** | 0.9336 | 0.9756 | 0.9542 | 2048 |
| **Accuracy** | | | 0.9536 | 4134 |
| **Macro Avg** | 0.9543 | 0.9538 | 0.9535 | 4134 |
| **Weighted Avg** | 0.9545 | 0.9536 | 0.9535 | 4134 |

Table 4: Training results obtained using the bigger network with 256-neuron dense layer

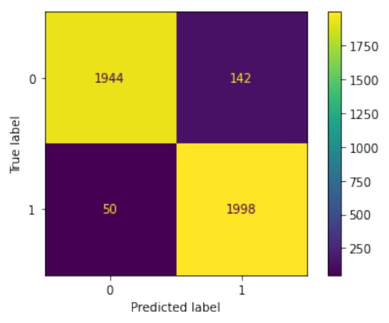

Figure 13: Confusion matrix

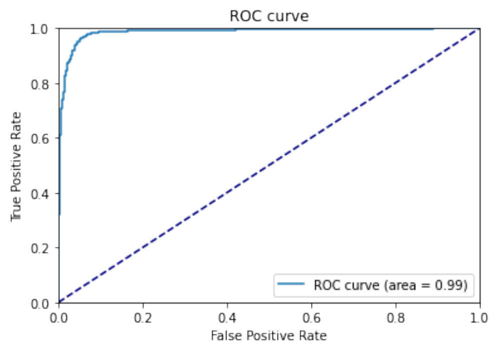

Figure 14: ROC curve

### 4.4 Bigger CNN with a 128-neuron Dense Layer

Adding a 256-neuron dense layer increased significantly the number of model parmaeters (more than 5x parameters), so we tried to reduce the dense layer in order to reduce the total parameter without losing performance.

```
Model: "128denseCNN"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_4 (InputLayer)        [(None, 120, 120, 3)]     0         
                                                                 
 conv2d_11 (Conv2D)          (None, 118, 118, 32)      896       
                                                                 
 max_pooling2d_11 (MaxPoolin  (None, 59, 59, 32)       0         
 g2D)                                                            
                                                                 
 conv2d_12 (Conv2D)          (None, 57, 57, 64)        18496     
                                                                 
 max_pooling2d_12 (MaxPoolin  (None, 28, 28, 64)       0         
 g2D)                                                            
                                                                 
 conv2d_13 (Conv2D)          (None, 26, 26, 128)       73856     
                                                                 
 max_pooling2d_13 (MaxPoolin  (None, 13, 13, 128)      0         
 g2D)                                                            
                                                                 
 conv2d_14 (Conv2D)          (None, 11, 11, 256)       295168    
                                                                 
 max_pooling2d_14 (MaxPoolin  (None, 5, 5, 256)        0         
 g2D)                                                            
                                                                 
 flatten_3 (Flatten)         (None, 6400)              0         
                                                                 
 dense_4 (Dense)             (None, 128)               819328    
                                                                 
 dense_5 (Dense)             (None, 1)                 129       
                                                                 
=================================================================
Total params: 1,207,873
Trainable params: 1,207,873
Non-trainable params: 0
_________________________________________________________________
```

The model behaved similarly to the previous experiment, but the validation accuracy (figure 15 started to slightly decrease after the best epoch (epoch 10), showing an overfitting tendency.

The testing results are similar to the previous experiment, as we can see from the table 5, but all the metrics are slightly lower.

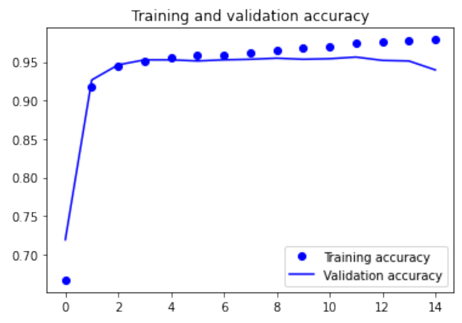

Figure 15: Training/validation accuracy

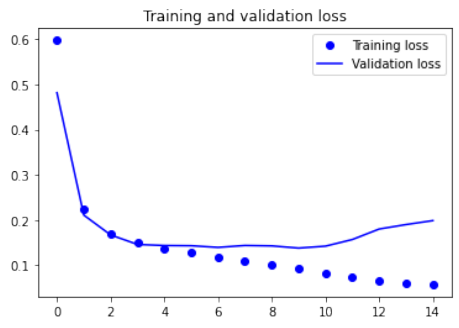

Figure 16: training/validation loss

| | **Precision** | **Recall** | **F1-score** | **Support** |
|---|---|---|---|---|
| **0** | 0.9744 | 0.9314 | 0.9525 | 2086 |
| **1** | 0.9332 | 0.9751 | 0.9537 | 2048 |
| **Accuracy** | | | 0.9531 | 4134 |
| **Macro Avg** | 0.9538 | 0.9533 | 0.9531 | 4134 |
| **Weighted Avg** | 0.9540 | 0.9531 | 0.9531 | 4134 |

Table 5: Training results obtained using the bigger network with 128-neuron dense layer

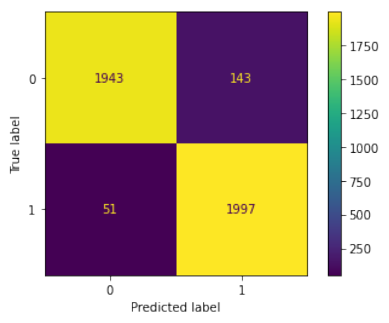

Figure 17: Confusion matrix

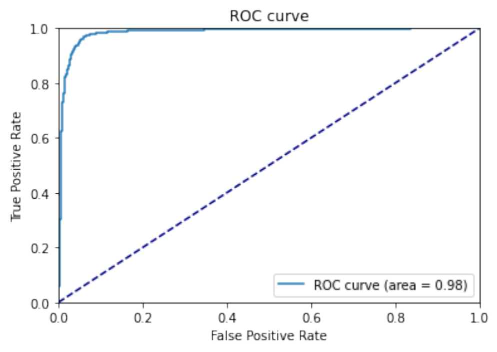

Figure 18: ROC curve

## 4.5 Basic CNN with a dense layer

As a last experiment, we wanted to add the dense layer on top of the basic architecture, because it performed better than the bigger one without the dense layer.

Obviously, the generalization degree of the basic architecture was higher than the bigger network by construction, and we wanted to see if the dense layer can improve also these performances or if the smaller network cannot have benefits from a dense layer.

```
Model: "256denseSmallCNN"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_5 (InputLayer)        [(None, 120, 120, 3)]     0

 conv2d_15 (Conv2D)          (None, 118, 118, 32)      896

 max_pooling2d_15 (MaxPoolin  (None, 59, 59, 32)       0
 g2D)

 conv2d_16 (Conv2D)          (None, 57, 57, 64)        18496

 max_pooling2d_16 (MaxPoolin  (None, 28, 28, 64)       0
 g2D)

 conv2d_17 (Conv2D)          (None, 26, 26, 128)       73856

 max_pooling2d_17 (MaxPoolin  (None, 13, 13, 128)      0
 g2D)

 flatten_4 (Flatten)         (None, 21632)             0

 dense_6 (Dense)             (None, 256)               5538048

 dense_7 (Dense)             (None, 1)                 257

=================================================================
Total params: 5,631,553
Trainable params: 5,631,553
Non-trainable params: 0
_________________________________________________________________
```

This configuration was the fastest to reach the convergence, (the selected epoch was the epoch 6), but it suffered of a slightly constant decrease of the validation accuracy (figure 19) and an increase of the validation loss (figure 20).

In the table 6 the testing results are shown: the behaviour is tha same of the other networks, but the performance are slightly poorer than both the experiments on the bigger network with a dense layer.

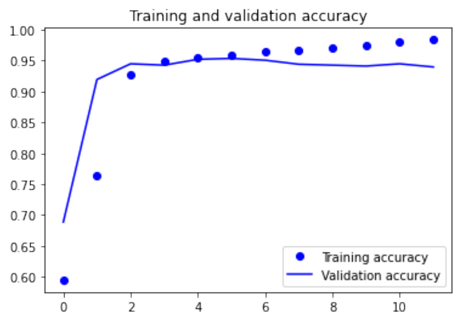

Figure 19: Training/validation accuracy

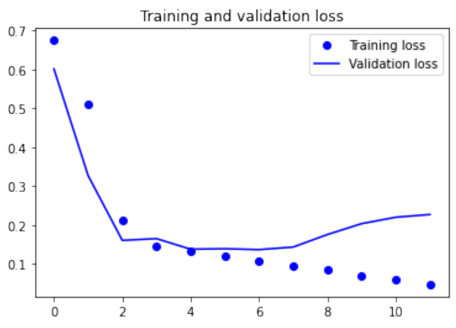

Figure 20: training/validation loss

| | **Precision** | **Recall** | **F1-score** | **Support** |
|---|---|---|---|---|
| **0** | 0.9753 | 0.9262 | 0.9501 | 2086 |
| **1** | 0.9285 | 0.9761 | 0.9517 | 2048 |
| **Accuracy** | | | 0.9509 | 4134 |
| **Macro Avg** | 0.9519 | 0.9511 | 0.9509 | 4134 |
| **Weighted Avg** | 0.9521 | 0.9509 | 0.9509 | 4134 |

Table 6: Training results obtained using the basic network with 256-neuron dense layer

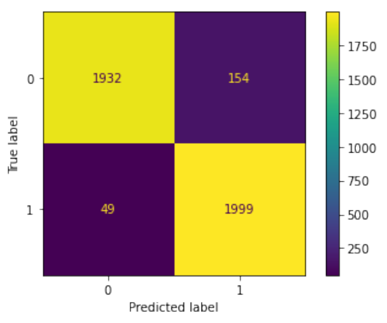

Figure 21: Confusion matrix

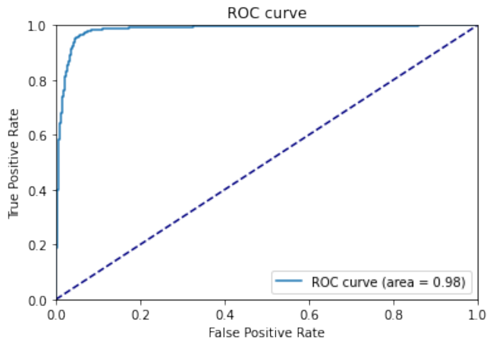

Figure 22: ROC curve

## 4.6 Chosen model

After all the experiments, we decided to choose as the best model the *bigger network with 256-neuron dense layer*.

Since the difference between the two experiments with the dense layer are really low, we decided also to tune the number of neuron in the dense layer during the hyperparameter optimization, in addition to the other hyperparameters.

## 5 Hyper-parameters' optimization

We tried to boost the performances of our classifer by **fine-tuning the hyperparameters**. To perform fine-tuning with *Keras* we used the library called *Keras Tuner*.

As already said, the CNN from scratch that we decided to optimize is the bigger network with 256-neuron dense layer.

### 5.1 Hyper-parameters

The **hyper-parameters** that we decided to tune are:

- number of units in the dense layer → from 128 to 256 in steps of 32
- activation function → *ReLu*, *Tanh*
- learning rate → 0.01, 0.001, 0.0001

### 5.2 Hyperband tuning

To fine tune the CNN from scratch model with *Keras Tuner*, we decided to use the **Hyperband algorithm** with 3 hyperband iterations and early stopping callback in such a way to prevent overfitting.

#### 5.2.1 Search results

The best validation binary accuracy we obtained is of 0.9589, and the set of hyper-parameters that gave us this result, the best set of hyper-parameters, is the following:

- number of units in the dense layer → 160
- activation function → *ReLu*
- learning rate → 0.001

In particular, the best set of hyper-parameters gave us the following results:

```
Trial summary
Hyperparameters:
activation_function: relu
n_units_dl: 160
learning_rate: 0.001
tuner/epochs: 15
tuner/initial_epoch: 5
tuner/bracket: 1
tuner/round: 1
tuner/trial_id: 0083
Score: 0.958635687828064
```

## 5.3 Training of the optimized model

The architecture of the optimized model is the following:

```
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 120, 120, 3)]     0

 conv2d_4 (Conv2D)           (None, 118, 118, 32)      896

 max_pooling2d_4 (MaxPooling  (None, 59, 59, 32)       0
 2D)

 conv2d_5 (Conv2D)           (None, 57, 57, 64)        18496

 max_pooling2d_5 (MaxPooling  (None, 28, 28, 64)       0
 2D)

 conv2d_6 (Conv2D)           (None, 26, 26, 128)       73856

 max_pooling2d_6 (MaxPooling  (None, 13, 13, 128)      0
 2D)

 conv2d_7 (Conv2D)           (None, 11, 11, 256)       295168

 max_pooling2d_7 (MaxPooling  (None, 5, 5, 256)        0
 2D)

 flatten_1 (Flatten)         (None, 6400)              0

 dense_2 (Dense)             (None, 160)               1024160

 dense_3 (Dense)             (None, 1)                 161

=================================================================
Total params: 1,412,737
Trainable params: 1,412,737
Non-trainable params: 0
```

### 5.3.1 Training results

We trained our CNN from scratch model using the optimal set of hyper-parameters and an early stopping callback and we obtained the results reported in table 7.

| | **Precision** | **Recall** | **F1-score** | **Support** |
|---|---|---|---|---|
| **0** | 0.9721 | 0.9348 | 0.9531 | 2086 |
| **1** | 0.9361 | 0.9727 | 0.9540 | 2048 |
| **Accuracy** | | | 0.9536 | 4134 |
| **Macro Avg** | 0.9541 | 0.9537 | 0.9536 | 4134 |
| **Weighted Avg** | 0.9543 | 0.9536 | 0.9535 | 4134 |

Table 7: Training results obtained using the optimized model

In images 23 and 5.3.1 are shown graphically the results obtained.

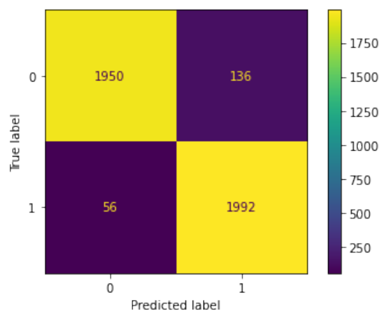

Figure 23: Confusion matrix

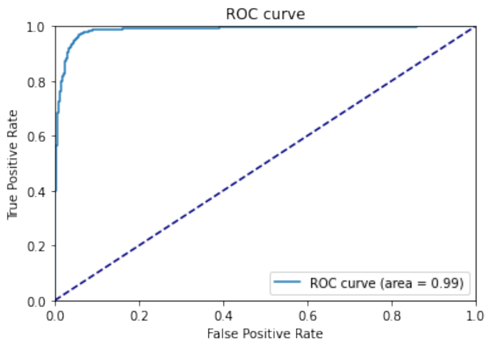

Figure 24: ROC curve

## 5.4 Comments

As we can see from the classification reports, the confusion matrix, and the graphs, the optimized model has almost the same accuracy as the unoptimized *bigger network with 256-neuron dense layer*.

The reason why the optimized model is preferrable than the unoptimized, is that the optimal value for the hyper parameter ”number of units of the dense layer” is of 160 instead of 256, so the network is much smaller; in particular, the original network has $2,027,329$ trainable parameters, while the optimized model has $1,412,737$ parameters. So, thanks to hyper-parameters’ optimization we have almost the same performances, but with almost half of the parameters.

# 6 Pre training

Despite the performances obtained with our CNN from scratch model were quite satisfactory, we decided to try to achieve better performances by exploiting some pretrained networks. In particular, we exploited the pretrained networks following both the two standard approaches:

- feature extraction
- fine-tuning

During **feature extraction** we freezed all the layers of the convolutional base since we used the pre-trained network just to to extract meaningful features. On top of pre-trained networks's output we trained a new classifier our malaria dataset in order to correctly classify blood cells as infected by malaria or uninfected.

During **fine-tuning** instead we gradually unfreezed some of the last layers of the convolutional base of the model we considered as being the best model produced using the **feature extraction** approach. We trained the networks on our malaria dataset in order to adjust some of the pre-trained network's weights and the weights of the new model.

The pre-trained networks that we used are *VGG16* and *InceptionV3*.

## 6.1 VGG16

*VGG16* is a convolutional neural network (CNN) proposed in 2014 and trained on the *ImageNet* dataset for image classification tasks.

The network is composed by 16 layers, takes an input of fixed dimension of 224×224×3 and generates as output a vector of 1000 probabilities corresponding to the 1000 *ImageNet*'s classes; *VGG16*'s architecture is shown in image 25:

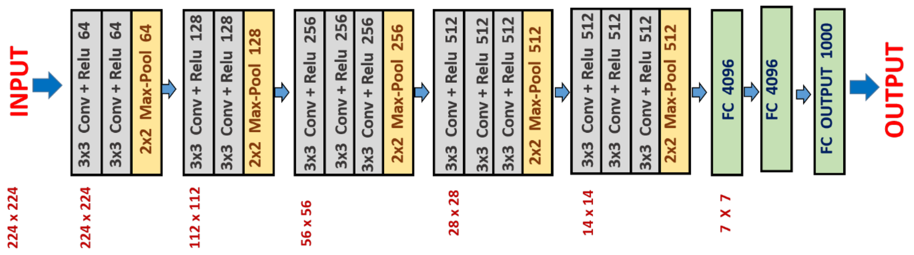

Figure 25: *VGG16*'s architecture

### 6.1.1 Feature extraction

As first approach, we exploited *VGG16* simply as **feature extractor**.

Since *ImageNet* is a dataset of images belonging to 1000 different classes which are far different from the images of the dataset we are considering, we didn't expect to obtain excellent results.

The model that gave us the best results exploiting *VGG16* as feature extractor is reported below:

```
 Model: "VGG16_feature_extraction_simple_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 224, 224, 3)]     0

 tf.__operators__.getitem (S  (None, 224, 224, 3)      0
 licingOpLambda)

 tf.nn.bias_add (TFOpLambda)  (None, 224, 224, 3)      0

 vgg16 (Functional)          (None, 7, 7, 512)         14714688

 flatten (Flatten)           (None, 25088)             0

 dense (Dense)               (None, 256)               6422784

 dense_1 (Dense)             (None, 1)                 257
=================================================================
Total params: 21,137,729
Trainable params: 6,423,041
Non-trainable params: 14,714,688
```

The classification report obtained training this model is reported in table 8.

| | **Precision** | **Recall** | **F1-score** | **Support** |
|---|---|---|---|---|
| **0** | 0.6633 | 0.5355 | 0.5926 | 2086 |
| **1** | 0.6045 | 0.7231 | 0.6585 | 2048 |
| **Accuracy** | | | 0.6284 | 4134 |
| **Macro Avg** | 0.6339 | 0.6293 | 0.6255 | 4134 |
| **Weighted Avg** | 0.6342 | 0.6284 | 0.6252 | 4134 |

Table 8: Training results obtained training the simple model network

The results of the training are shown in images 26 and 27, the train/validation accuracy and loss can be seen in images 28, 29.

### 6.1.2 Comments on VGG16's results

Since the results obtained using *VGG16* simply as feature extractor were not so good, we decided not to exploit it with the fine-tuning technique.

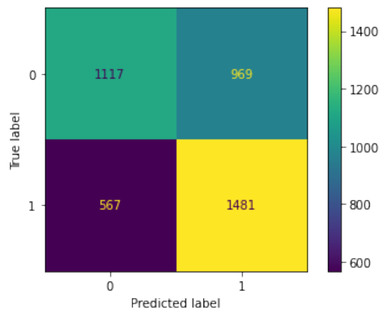

Figure 26: Confusion matrix

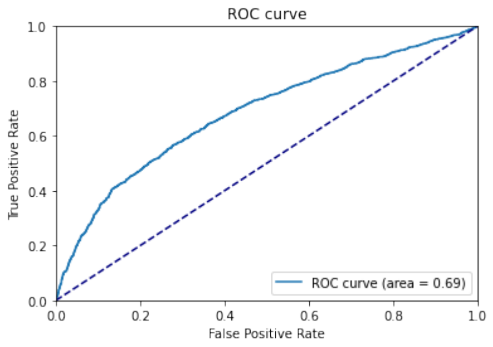

Figure 27: ROC curve

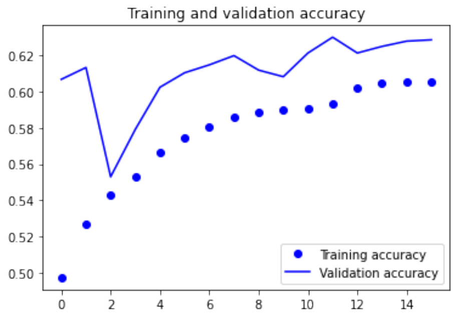

Figure 28: Training/validation accuracy

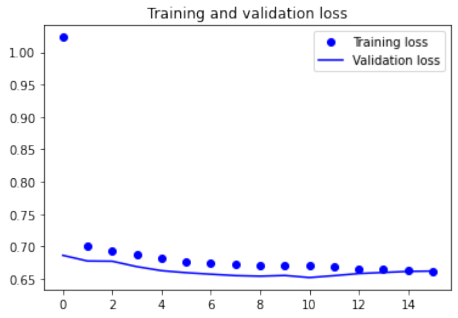

Figure 29: Training/validation loss

## 6.2 InceptionV3

*InceptionV3* is a convolutional neural network (CNN) proposed in 2015. This network is more computationally efficient than the *VGG16*, both in terms of number of parameters generated by the network, memory and other resources.

The network has a total of 42 layers, input size fixed to 299×299×3; In image 30 is shown *InceptionV3*'s architecture.

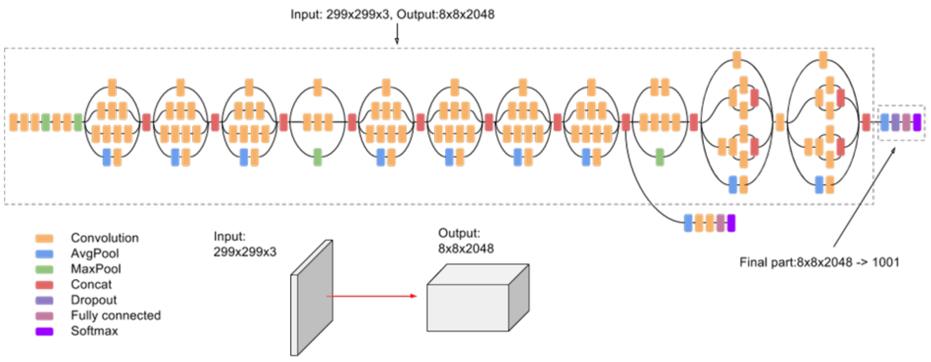

Figure 30: *InceptionV3*'s architecture

### 6.2.1 Feature extraction

As first approach, we exploited *InceptionV3* simply as **feature extractor**.

Exactly as for *VGG16*, since *ImageNet* is a dataset of images belonging to 1000 different classes which are far different from the images of the dataset we are considering, we didn't expect to obtain excellent results; we still expect the results to be better than *VGG16*'s ones since *InceptionV3* is a more powerful model.

To apply the feature extraction approach, we freezed all the layers of *InceptionV3* as follows:

```
# freeze the convolutional basis
conv_base.trainable = False
```

We performed various experiments, but in this report we decided just to show the two experiments that gave us the best results.

**Global average pooling model**

The simple model that gave us the best results exploiting *InceptionV3* as feature extractor is reported below:

```
Model: "inceptionV3_feature_extraction_global_avg_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_2 (InputLayer)        [(None, 299, 299, 3)]     0         
                                                                 
 tf.math.truediv (TFOpLambda  (None, 299, 299, 3)      0         
 )                                                               
                                                                 
 tf.math.subtract (TFOpLambd  (None, 299, 299, 3)      0         
 a)                                                              
                                                                 
 inception_v3 (Functional)   (None, 8, 8, 2048)        21802784  
                                                                 
 global_average_pooling2d (G  (None, 2048)             0         
 lobalAveragePooling2D)                                          
                                                                 
 dense (Dense)               (None, 256)               524544    
                                                                 
 dropout (Dropout)           (None, 256)               0         
                                                                 
 dense_1 (Dense)             (None, 1)                 257       
                                                                 
=================================================================
Total params: 22,327,585
Trainable params: 524,801
Non-trainable params: 21,802,784
```

The classification report obtained from the training of this model is reported in table 9.

| | **Precision** | **Recall** | **F1-score** | **Support** |
|---|---|---|---|---|
| **0** | 0.7607 | 0.7953 | 0.7776 | 2086 |
| **1** | 0.7814 | 0.7451 | 0.7628 | 2048 |
| **Accuracy** | | | 0.7704 | 4134 |
| **Macro Avg** | 0.7710 | 0.7702 | 0.7702 | 4134 |
| **Weighted Avg** | 0.7709 | 0.7704 | 0.7703 | 4134 |

Table 9: Training results obtained training the global avg network

The results of the training are shown in images 31 and 32, the train/validation accuracy and loss can be seen in images 33, 34.

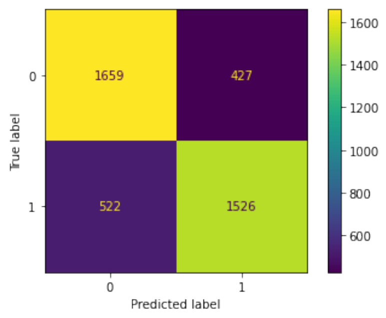

Figure 31: Confusion matrix

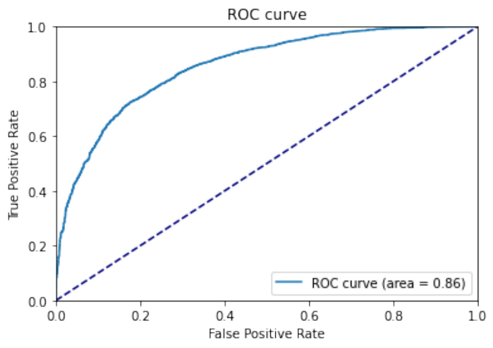

Figure 32: ROC curve

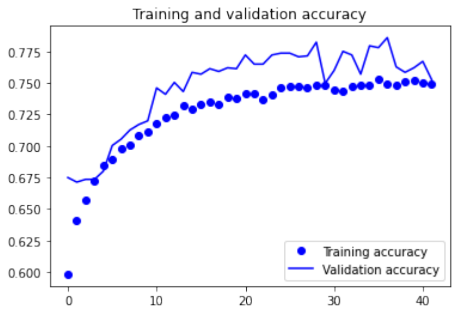

Figure 33: Training/validation accuracy

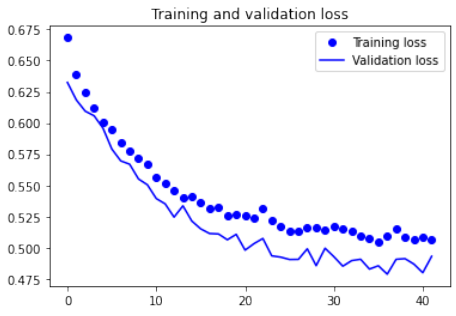

Figure 34: Training/validation loss

## Complex model

The only difference between the architecture of the *global average pooling model* and this model, is on the number of neurons in the dense layer: in this model we use 1024 instead of 256.

The architecture of the *complex model* is reported below:

```
Model: "inceptionV3_feature_extraction_complex_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 299, 299, 3)]     0

 tf.math.truediv_1 (TFOpLamb  (None, 299, 299, 3)      0
 da)

 tf.math.subtract_1 (TFOpLam  (None, 299, 299, 3)      0
 bda)

 inception_v3 (Functional)   (None, 8, 8, 2048)        21802784

 global_average_pooling2d_1   (None, 2048)             0
 (GlobalAveragePooling2D)

 dense (Dense)               (None, 1024)              2098176

 dropout (Dropout)           (None, 1024)              0

 dense_1 (Dense)             (None, 1)                 1025
=================================================================
Total params: 23,901,985
Trainable params: 2,099,201
Non-trainable params: 21,802,784
```

The classification report obtained training this model is reported in table 10.

| | **Precision** | **Recall** | **F1-score** | **Support** |
|---|---|---|---|---|
| **0** | 0.7687 | 0.8063 | 0.7871 | 2086 |
| **1** | 0.7924 | 0.7529 | 0.7722 | 2048 |
| **Accuracy** | | | 0.7799 | 4134 |
| **Macro Avg** | 0.7806 | 0.7796 | 0.7796 | 4134 |
| **Weighted Avg** | 0.7805 | 0.7799 | 0.7797 | 4134 |

Table 10: Training results obtained training the complex network

The results of the training are shown in images 35 and 36, the train/validation accuracy and loss can be seen in images 37, 38.

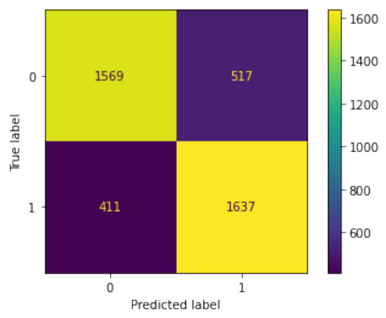

Figure 35: Confusion matrix

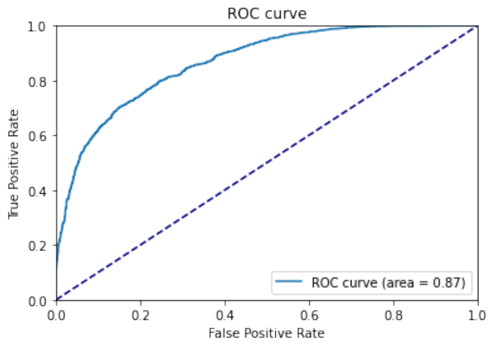

Figure 36: ROC curve

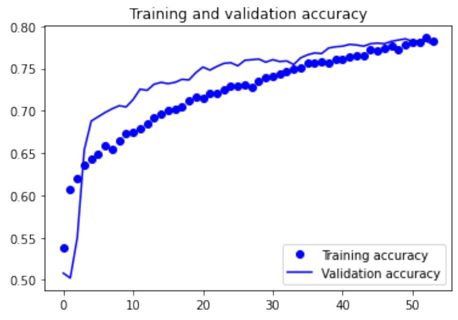

Figure 37: Training/validation accuracy

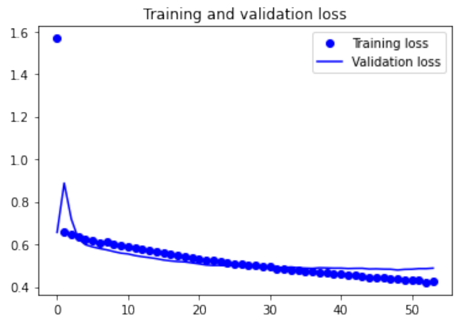

Figure 38: Training/validation loss

### 6.2.2 Fine-tuning

To fine-tune the two best networks obtained using *InceptionV3* as feature extractor, we unfreezed the top two inception blocks (in practice we kept freezed the first 249 layers, and we unfreezed the remaining layers). We used a learning rate of 0.0001 since we don't want to change too much the weights of the pre-trained network, and as optimizer we used *SGD* since it is the best optimizer for *InceptionV3*.

```
for layer in global_avg_model.get_layer('inception_v3').layers[:249]:
   layer.trainable = False
for layer in global_avg_model.get_layer('inception_v3').layers[249:]:
   layer.trainable = True
```

**Global average pooling model**

As we can see from the report shown below, now the trainable parameters are $11,639,681$ instead of the $524,801$ we had in the feature-extraction approach for the same model.

```
Model: "inceptionV3_finetuning_global_avg_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 299, 299, 3)]     0

 tf.math.truediv (TFOpLambda  (None, 299, 299, 3)      0
 )

 tf.math.subtract (TFOpLambd  (None, 299, 299, 3)      0
 a)

 inception_v3 (Functional)   (None, 8, 8, 2048)        21802784

 global_average_pooling2d (G  (None, 2048)             0
 lobalAveragePooling2D)

 dense (Dense)               (None, 256)               524544

 dropout (Dropout)           (None, 256)               0

 dense_1 (Dense)             (None, 1)                 257

=================================================================
Total params: 22,327,585
Trainable params: 11,639,681
Non-trainable params: 10,687,904
```

The classification report obtained from the training of this model is reported in table 11.

The results of the training are shown in images 39 and 40, the train/validation accuracy and loss can be seen in images 41, 42.

|  | **Precision** | **Recall** | **F1-score** | **Support** |
|---|---|---|---|---|
| **0** | 0.7924 | 0.7522 | 0.7718 | 2086 |
| **1** | 0.7600 | 0.7993 | 0.7792 | 2048 |
| **Accuracy** |  |  | 0.7755 | 4134 |
| **Macro Avg** | 0.7762 | 0.7757 | 0.7755 | 4134 |
| **Weighted Avg** | 0.7764 | 0.7755 | 0.7754 | 4134 |

Table 11: Training results obtained fine-tuning the global avg network

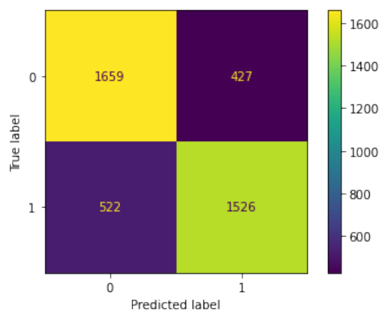

Figure 39: Confusion matrix

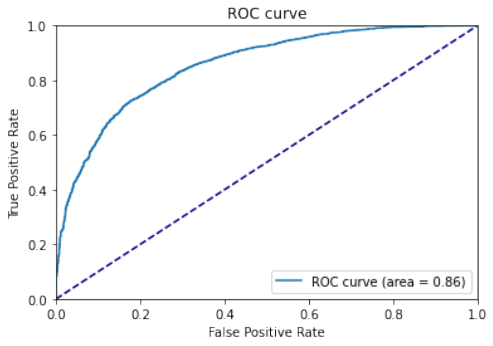

Figure 40: ROC curve

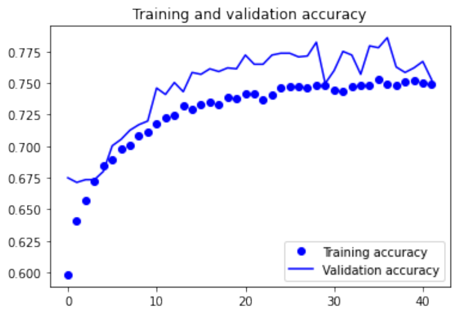

Figure 41: Training/validation accuracy

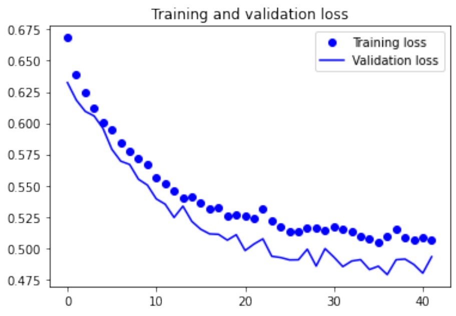

Figure 42: Training/validation loss

**Complex model**
This model is the one that gave us the better results.

The architecture of this model is summarized in the following report; as we can see, the number of trainable parameters now is $13,214,081$ while in the feature-extraction approach for the same model we had $2,099,201$ trainable parameters.

```
Model: "inceptionV3_finetuning_complex_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 299, 299, 3)]     0

 tf.math.truediv_1 (TFOpLamb  (None, 299, 299, 3)      0
 da)

 tf.math.subtract_1 (TFOpLam  (None, 299, 299, 3)      0
 bda)

 inception_v3 (Functional)   (None, 8, 8, 2048)        21802784

 global_average_pooling2d_1   (None, 2048)             0
 (GlobalAveragePooling2D)

 dense (Dense)               (None, 1024)              2098176

 dropout (Dropout)           (None, 1024)              0

 dense_1 (Dense)             (None, 1)                 1025

=================================================================
Total params: 23,901,985
Trainable params: 13,214,081
Non-trainable params: 10,687,904
```

The classification report obtained from the training of this model is reported in table 12.

| | **Precision** | **Recall** | **F1-score** | **Support** |
|---|---|---|---|---|
| **0** | 0.7921 | 0.7196 | 0.7541 | 2086 |
| **1** | 0.7387 | 0.8076 | 0.7716 | 2048 |
| **Accuracy** | | | 0.7632 | 4134 |
| **Macro Avg** | 0.7654 | 0.7636 | 0.7629 | 4134 |
| **Weighted Avg** | 0.7656 | 0.7632 | 0.7628 | 4134 |

Table 12: Training results obtained fine-tuning the complex network

The results of the training are shown in images 43, 44 the train/validation accuracy and loss can be seen in images 45, 46.

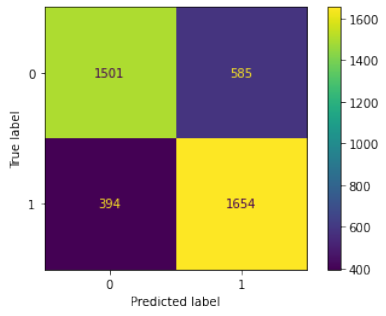

Figure 43: Confusion matrix

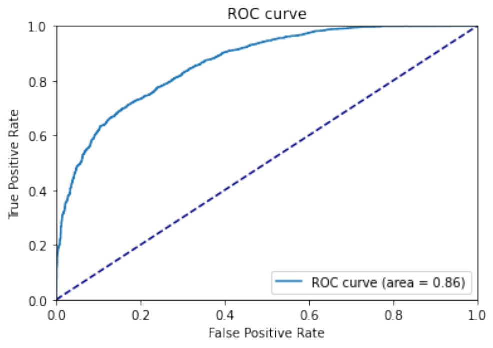

Figure 44: ROC curve

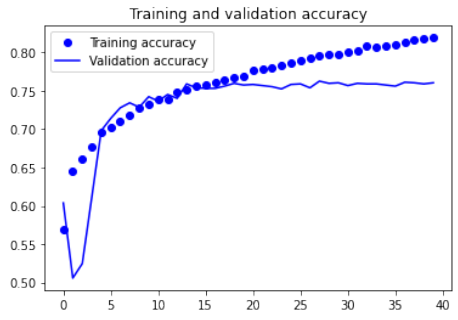

Figure 45: Training/validation accuracy

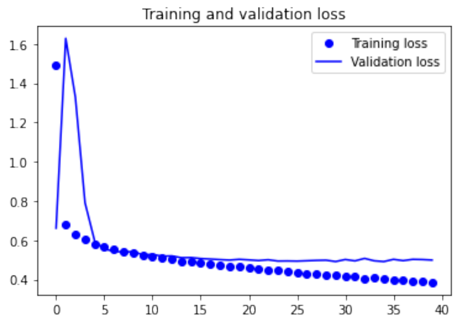

Figure 46: Training/validation loss

### 6.2.3 Comments on InceptionV3's results

The two fine-tuned networks behave almost the same, so of course is preferrable the *global average pooling model*, which is the one having a minor number of parameters.

The results obtained are not satisfactory: we expected a great increase on the accuracy, while actually it almost remained the same. Probably, since *ImageNet* is a dataset of images very different from the ones of the malaria dataset, the *InceptionV3* pre-trained netowork solves problems that are very different from our malaria classification task.

Of course, the performances of the network would increase by unfreezing much more layers of the pre-trained model, but training pre-trained networks with a lot of unfreezed layers has a very high-computational cost compared to the computational cost of training our CNN from scratch.

An interesting experiment would be to try to use other pre-trained networks.

## 7 ExplainableAI

The introduction of explainable AI (XAI) aimed to increase the interpretability of our models and to provide explanations for their decisions or recommendations. This is important because many AI applications, such as medical diagnosis, have high-stakes consequences, and it is essential to ensure that their decisions are fair, unbiased, and trustworthy.

### 7.1 CAM visualization

We decided to perform an XAI task using the Class Activation Map (CAM) visualization.

CAM visualization is a popular technique for interpreting the decisions of CNNs and can help identify which parts of an image the network is attending to for classification.

In a medical scenario this is really important to detect possible misclassifications, or to better address some treatment directly to the infected parts of our blood cells.

### 7.2 CAM workflow

We decided to perform CAM leveraging our optimized model built from scratch, because it is the model with the best performances.

```
Model: "optimized_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 120, 120, 3)]     0

 conv2d_4 (Conv2D)           (None, 118, 118, 32)      896

 max_pooling2d_4 (MaxPooling  (None, 59, 59, 32)       0
 2D)

 conv2d_5 (Conv2D)           (None, 57, 57, 64)        18496

 max_pooling2d_5 (MaxPooling  (None, 28, 28, 64)       0
 2D)

 conv2d_6 (Conv2D)           (None, 26, 26, 128)       73856

 max_pooling2d_6 (MaxPooling  (None, 13, 13, 128)      0
 2D)

 conv2d_7 (Conv2D)           (None, 11, 11, 256)       295168

 max_pooling2d_7 (MaxPooling  (None, 5, 5, 256)        0
 2D)

 flatten_1 (Flatten)         (None, 6400)              0
 dense_2 (Dense)              (None, 160)               1024160

 dense_3 (Dense)              (None, 1)                 161

=================================================================
Total params: 1,412,737
Trainable params: 1,412,737
Non-trainable params: 0
_________________________________________________________________
```

We created the feature map from output of the last convolutional layer (*conv2d_7*), and then we multiplied the feature map with the weights of the final dense layer (*dense_3*), obtaining the class activation map.

We upsample the CAM to the original input size to obtain the heatmap that highlights the zones of the images most used by the model to make a prediction.

Finally, we superimposed the heatmap to the original image to visualize which regions of the images are important in classification: this superimposition allowed the experts to make some considerations about the effectiveness of the classification, and about which elements of the image can lead to a correct diagnosis.

## 7.3 CAM visualization results

We performed CAM visualization over 3 different parasitized samples of the dataset, in order to identify some pattern in the model's predictions.

### 7.3.1 Sample 1

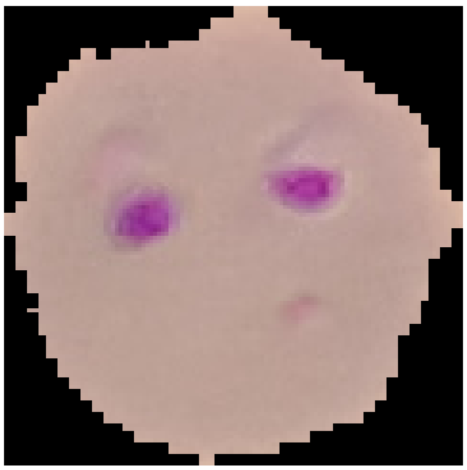

Figure 47: Sample 1

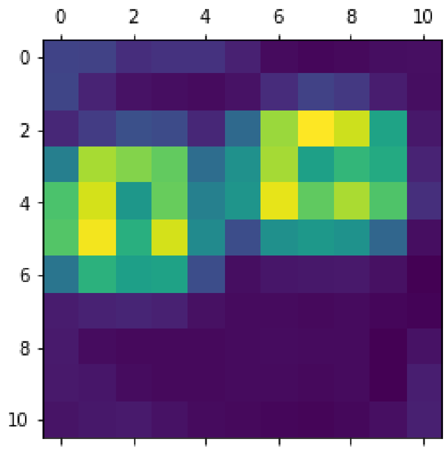

Figure 48: Heatmap of sample 1

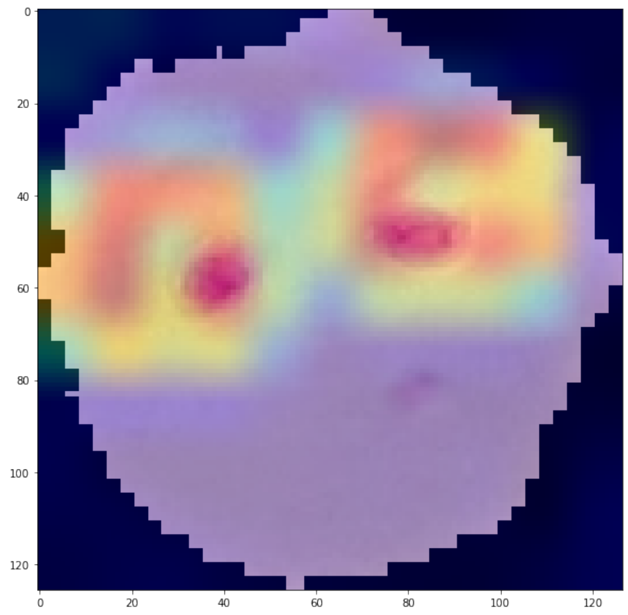

Figure 49: Heatmap superimposed to the sample 1

### 7.3.2 Sample 2

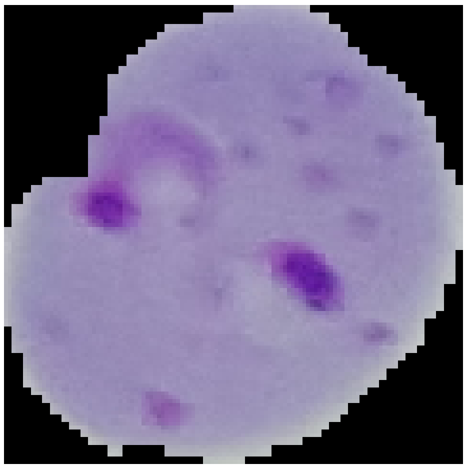

Figure 50: Sample 2

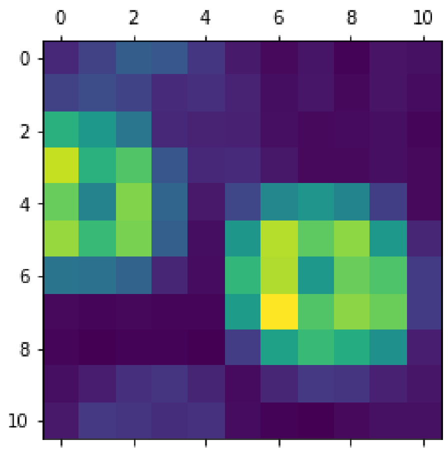

Figure 51: Heatmap of sample 2

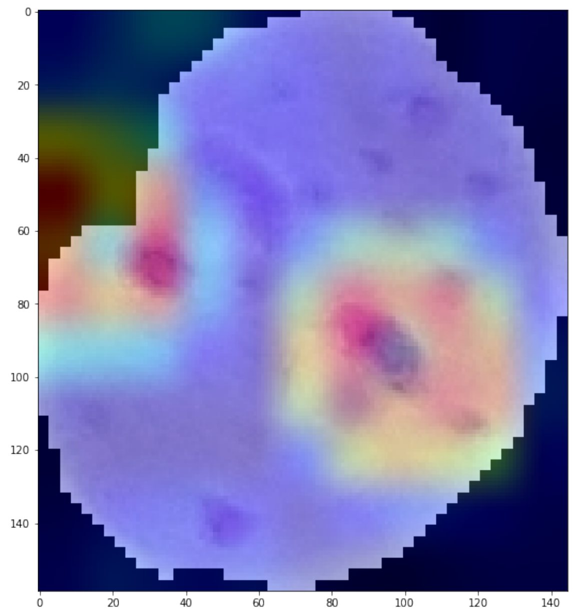

Figure 52: Heatmap superimposed to the sample 2

### 7.3.3 Sample 3

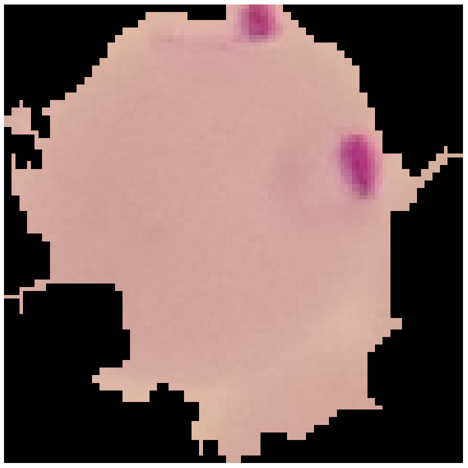

Figure 53: Sample 3

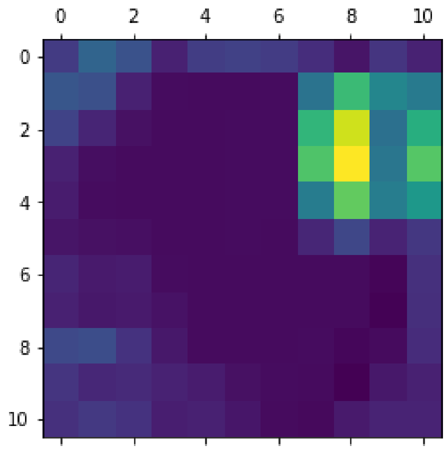

Figure 54: Heatmap of sample 3

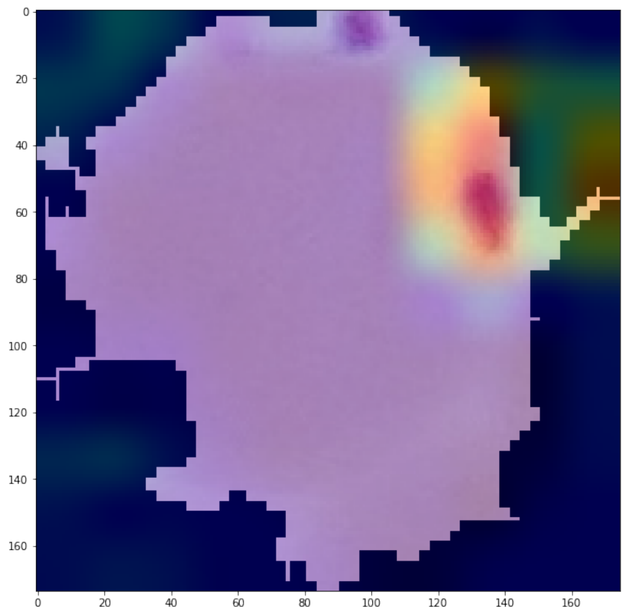

Figure 55: Heatmap superimposed to the sample 3

## 7.4 Findings

As we can see, we used 3 samples with different colors, and the classifier is not biased by that: so the cell color is not important during the classification task.

We used also a sample pretty irregular (*sample 3*) and its shape was not recognized as an important pattern during classification, so we can deduce that also the cell shape is not a possible hint for a correct diagnosis.

In all the three samples the classifier leverages the different color spots inside the blood cell to make its decision: so, we can deduce that identify these spots can be the main task to perform during a malaria diagnosis.

# 8 Conclusions

We are really satisfied by the results obtained using the CNN from scratch, which are comparable to state-of-the-art approaches more and more complex.

Also, the explainable AI task conducted on the network created from scratch allowed us to learn a possible classification pattern, which can be useful for medical experts in order to better assess their diagnoses.

The less satisfying part of our experiments are the results obtained leveraging the pre-trained models.

A possible explanation of the poor performances obtained by VGG-16 and InceptionV3 can be the peculiar dataset used: our dataset was composed by really simple and low-definition images, that can be different from the training base of these models.

Also, our problem is a binary classification, which is not the speciality of these pre-trained architectures.

The improvement of the classification performance leveraging pre-trained models can be the most obvious further improvement to our study, maybe using the models proposed in the related works.

We can also try to refine our classifier from scratch, looking for new improvement areas.

## References

[1] Muqdad Hanoon Dawood Alnussairi and Abdullahi Abdu İbrahim. Malaria parasite detection using deep learning algorithms based on (cnns) technique. *Computers and Electrical Engineering*, 103:108316, 2022.

[2] Zhaohui Liang, Andrew Powell, Ilker Ersoy, Mahdieh Poostchi, Kamolrat Silamut, Kannappan Palaniappan, Peng Guo, Md Amir Hossain, Antani Sameer, Richard James Maude, Jimmy Xiangji Huang, Stefan Jaeger, and George Thoma. Cnn-based image analysis for malaria diagnosis. pages 493–496, 2016.