# Malaria Detection using CNN – Detailed Documentation

## Introduction

Malaria is a potentially life-threatening disease caused by parasites that are transmitted to people through the bites of infected mosquitoes. Microscopic examination of thin blood smears remains a gold standard for diagnosis. This repository provides a small Python project that trains a convolutional neural network (CNN) to distinguish between parasitized and uninfected red blood cells from microscope images. The project handles data loading and augmentation, trains a simple CNN, visualises several aspects of the dataset and model behaviour, and saves various output plots.

The code lives in a single script (`main.py`) and is intended as a starting point for experimenting with deep learning on cell images. Below is a comprehensive explanation of what each section of the code does and how to reproduce the results on your own machine.

## Repository Overview

The repository contains the following items:

| File | Description |
|---|---|
| `main.py` | Main training script. Contains model definition, data loading/augmentation, training, evaluation and visualisation functions. |
| `requirements.txt` | List of Python dependencies (TensorFlow, NumPy, Matplotlib, Scikit-learn, etc.). |
| `accuracy.png` / `loss.png` | Generated plots of training vs. validation accuracy/loss over epochs. |
| `confusion_matrix.png` | Generated heatmap showing how often the model confuses the two classes. |
| `class_distribution.png` | Bar chart showing how many images belong to each class. |
| `sample_images.png` | Panel of example parasitized and uninfected images. |
| `parasitized_pixel_values.png` / `uninfected_pixel_values.png` | Histograms of pixel intensities for a single image from each class. |

| File | Description |
| --- | --- |
| `readme.txt` | Original notes from the author with basic usage commands. |

In addition, after training the script saves a Keras model ( `malaria_detection_model_keras_v1.h` ), though this file is not stored in the repository by default.

## Environment Setup

1. **Python:** Use Python 3.8–3.11. The project relies on TensorFlow 2.x and other common scientific-computing libraries.
2. **Install dependencies:** Run `pip install -r` to install all required packages. The requirements file includes TensorFlow, NumPy, Matplotlib, Pandas, Scikit-learn and Seaborn, among others.
3. **Virtual environment (optional):** Creating a virtual environment (via `python -m venv .venv` ) helps isolate dependencies. Activate it with `.venv\Scripts\activate` on Windows or `source` on macOS/Linux, then install the requirements.

## Data Preparation

The script expects a folder called `cell_images/cell_images/` in the working directory. Inside this folder there should be two subfolders named **Parasitized** and **Uninfected**, each containing JPEG/PNG images of single red blood cells. This structure matches the NIH Malaria Cell dataset and many Kaggle mirrors.

*Example directory layout:*

```
cell_images/
    cell_images/
        Parasitized/
            C1_thinF_IMG_20150604_104722_cell_9.png
            ...
        Uninfected/
```

If your dataset lives elsewhere, adjust the `data_di` variable in `main.py` accordingly. The code counts files in both folders and plots a bar chart showing the number of samples in each class [1] .

## Data Loading and Augmentation

TensorFlow's `ImageDataGenerat` is used to load images from disk and apply basic augmentations. The generator rescales pixel values to the range `[0,1` and applies random shear, zoom and horizontal flips to

improve generalisation [2]. A validation split of 20 % is defined here so that the same generator can produce both training and validation batches.

Two generators are created:

- **Training generator:** Uses the training subset of the data and applies the augmentations. [3]
- **Validation generator:** Uses the remaining 20 % of the images, with shuffling disabled to preserve label order. [4]

The generators read images on the fly from `Parasitized/` and `Uninfected` folders and produce batches of shape `(batch_size, height, width, channels)`. Labels are automatically inferred from the subfolder names.

### Visualising the Dataset

Several helper functions visualise the dataset:

1. **Class distribution:** The script counts how many images are in each class and plots them as a bar chart [1]. This helps you spot class imbalance.
2. **Sample images:** The `display_samples` function grabs a batch from the training generator and displays six images in a 2×3 grid with their labels [5]. It also saves the panel to `sample_images.png`.
3. **Pixel-value histograms:** The `visualize_pixel_values` function selects one parasitized and one uninfected image from the generator and plots histograms of their pixel intensities [6]. This provides a quick check that the images are properly scaled.

## Model Architecture

The `create_model(` function defines a simple CNN using Keras's Sequential API [7]. It consists of:

1. A 2D convolutional layer with 32 filters of size 3×3 and ReLU activation.
2. A max-pooling layer with pool size 2×2 to downsample feature maps.
3. Another convolutional layer with 64 filters followed by another max-pooling layer.
4. A flatten layer to convert the feature maps to a 1-D vector.
5. A dense (fully connected) layer with 512 units and ReLU activation.
6. A dropout layer with rate 0.5 to mitigate overfitting.
7. A final dense layer with one unit and sigmoid activation, producing a probability of the cell being parasitized.

The model is compiled with the Adam optimiser, binary cross-entropy loss and an accuracy metric [8]. This architecture is intentionally small for demonstration purposes; you can experiment with deeper networks or transfer-learning backbones (e.g. ResNet50) by modifying this function.

## Training the Model

After defining the model, the script calls `model.fit` with the training and validation generators. By default the model trains for 10 epochs [9]. The `steps_per_epoch` and `validation_steps` arguments

are computed from the number of samples in each generator. Training prints progress on each epoch and returns a `history` object that records accuracy and loss over time.

### Monitoring Training

The training history is plotted in two separate figures:

1. **Accuracy curves:** The script plots training versus validation accuracy for each epoch and saves the result to `accuracy.pn` [10].
2. **Loss curves:** A similar plot shows training versus validation loss and is saved to `loss.png` [11].

Monitoring these curves helps determine whether the model is overfitting (e.g. training accuracy high but validation accuracy stagnant) and whether more epochs or different augmentation strategies are warranted.

## Saving, Loading and Evaluating the Model

Once training finishes, the model is saved to `malaria_detection_model_keras_v1.h5` [12]. The code
demonstrates how to load the saved model using `tf.keras.models.load_model` and evaluates its performance on the validation set by calling `model.evaluate` [13].

Predicted probabilities are flattened and thresholded at 0.5 to obtain binary predictions. A confusion matrix is computed from the true labels and predicted labels and visualised as a heatmap[14]. Additionally, a scikit-learn classification report is printed to summarise precision, recall and F1-score for each class[15]. These outputs help you understand where the model performs well and where it might be misclassifying images.

## How to Run the Script

Follow these steps after preparing the environment and dataset:

1. **Install dependencies:** `pip install -r requirements.txt`.
2. **Ensure the dataset is in** `cell_images/cell_images/Parasitized` **and** `cell_images cell_images/Uninfecte`.
3. **Run the training script:** Execute `python main.py` from the project root. This will load the data, train the model, generate plots and save the trained model. Do not close the plotting windows prematurely; the script will block until you close each figure.

The original `readme.tx` mentions a Streamlit application, but this documentation focuses only on the standalone training script. For interactive deployment or model serving you can build your own application once you have a trained model.

## Customisation and Tips

- **Modify hyperparameters:** You can change the `image_shape`, `batch_size`, number of epochs and other parameters defined near the top of `main.py`. Increasing the input resolution or the number of epochs may yield better performance at the cost of training time.
- **Improve the architecture:** Try adding more convolutional layers, increasing the number of filters, or integrating popular architectures like VGG16 or MobileNet via `tf.keras.application`.
- **Regularisation:** To combat overfitting, consider adding more dropout layers, L2 regularisation or early stopping based on validation loss.
- **Class imbalance:** If your dataset is unbalanced (e.g. many more uninfected images), you can compute class weights and pass them to `model.fit` or oversample the minority class.
- **Alternative augmentations:** `ImageDataGenerator` supports a variety of augmentations such as rotation, brightness adjustments and vertical flips. Augmentation generally improves robustness by exposing the model to more varied samples.
- **Evaluation metrics:** Beyond accuracy, consider tracking precision, recall, F1-score, ROC-AUC and confusion matrices. Scikit-learn and TensorFlow provide convenient functions for these metrics.

## Conclusion

This repository offers a compact example of using convolutional neural networks to detect malaria parasites from blood-smear images. It demonstrates how to load and augment image data, build a simple Keras model, train it while tracking metrics, evaluate its performance and visualise the results. Feel free to extend this baseline by experimenting with more sophisticated architectures, fine-tuning on additional datasets, or integrating the model into a web or mobile application.

---

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 raw.githubusercontent.com

https://raw.githubusercontent.com/huma918/Malaria-Detection-/main/main.py