

TOWARDS AUTOMATED MALARIA DETECTION: A DEEP LEARNING
APPROACH INTEGRATING CUSTOM CONVOLUTIONAL NEURAL
NETWORKS AND PRE-TRAINED MODELS FOR ACCURATE PARASITE
IDENTIFICATION

A THESIS SUBMITTED TO
THE FACULTY OF ARCHITECTURE AND ENGINEERING
OF
EPOKA UNIVERSITY

BY

LORA SHIMA

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR BACHELOR DEGREE
IN COMPUTER ENGINEERING

JUNE, 2024

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name:

Signature:

ABSTRACT

TOWARDS AUTOMATED MALARIA DETECTION: A DEEP LEARNING APPROACH INTEGRATING CUSTOM CONVOLUTIONAL NEURAL NETWORKS AND PRE-TRAINED MODELS FOR ACCURATE PARASITE IDENTIFICATION

Lora Shima
B.Sc., Computer Engineering
Supervisor: M.Sc. Sabrina Begaj

Malaria remains a critical global health challenge, necessitating rapid and accurate diagnosis for effective treatment and control. This study explores the potential of deep learning models for automated malaria parasite detection in thin blood smears. Three distinct approaches were investigated: a custom Convolutional Neural Network (CNN) architecture tailored specifically for malaria parasite features, the efficient MobileNet model, and the deeper GoogLeNet model. The primary objective of this study is to ascertain the model with the highest accuracy while also evaluating the impact of using augmented datasets versus preprocessed ones on model performance.

Each model underwent rigorous training and evaluation using a diverse dataset of thin blood smear images, covering various staining techniques and parasite densities. Performance metrics such as confusion matrices, accuracy and loss plots for both training and validation sets, and detailed classification reports were employed for comprehensive

evaluation. The findings revealed that both the custom CNN and GoogLeNet achieved very similar accuracy levels in malaria parasite detection. In contrast, MobileNet exhibited variability in its performance during validation epochs, indicating fluctuations in its effectiveness compared to the other models.

This research contributes significantly to advancing the application of deep learning in automated malaria screening. It underscores the effectiveness of different model architectures, each offering unique advantages based on specific operational requirements and constraints.

Keywords: Malaria, CNN, MobileNet, GoogLeNet, accuracy.

ABSTRAKT

NË DREJTIM TË ZBULIMIT AUTOMATIK TË MALARIAS: NJË QASJE E MËSIMIT TË THELLË TË STRUKTURUAR QË INTEGRON RRJETET KONVOLUCIONALE TË PERSONLIZUARA DHE MODELET E TRAJNUARA MË PARË PËR IDENTIFIKIMIN E SAKTË TË PARAZITËVE

Lora Shima
B.Sc., Computer Engineering
Supervizor: M.Sc. Sabrina Begaj

Malaria mbetet një sfidë shëndetësore globale kritike, duke kërkuar diagnostifikim të shpejtë dhe të saktë për trajtimin dhe kontrollin efektiv. Ky studim eksploron potencialin e modeleve të mësimit të thellë të strukturuar për zbulimin automatik të parazitëve të malarias në mostra të holla të gjakut. U eksploruan tre qasje të ndryshme: një arkitekturë e përshtatur e rrjetit konvolucional (CNN) specifikisht për veçoritë e parazitëve të malariasë, modeli efikas MobileNet, dhe modeli më i thellë GoogLeNet. Objektivi kryesor i këtij studimi është të zbulohet modeli me shkallë të lartë të saktësisë, dhe gjithashtu të vlerësohet ndikimi i përdorimit të seteve të të dhënave të augmentuara krahasuar me ato të parapërpunuara në performancën e modeleve.

Çdo model u trajtua dhe u vlerësua në mënyrë intensive duke përdorur një set të diversifikuar të imazheve të mostrave të holla të gjakut, duke përfshirë teknikat e

ndryshme të ngjyrimit dhe dendësitë e parazitëve. U përdorën metrika të performancës si matricat e konfuzionit, grafikët e saktësisë dhe humbjeve për të dy setet e trajnimit dhe validimit, si dhe raportet e detajuara të klasifikimit për një vlerësim të plotë.

Studimi tregoi se si CNN-i i personalizuar dhe GoogLeNet arritën nivele shumë të ngjashme të saktësisë në zbulimin e parazitëve të malariasë. Në kontrast, MobileNet tregoi ndryshueshmëri në performancën e saj gjatë epokave të validimit, duke treguar variacione në efikasitetin e saj krahasuar me modelet e tjera.

Ky hulumtim kontribuon në avancimin e aplikimit të mësimin të thellë në skanimin automatik të malariasë. Shfaqet efektivitetin e arkitekturave të ndryshme të modeleve, secila duke ofruar avantazhe unike në bazë të kërkesave dhe kufizimeve specifike operative.

Fjalë kyçe: Malaria, CNN, MobileNet, GoogLeNet, saktësi.

Dedicated to.....

ACKNOWLEDGEMENTS

I am deeply thankful to my supervisor, Sabrina Begaj, whose invaluable guidance, support, and expertise have been instrumental throughout the completion of this thesis. Her insightful feedback, constructive criticism, and encouraging words have significantly shaped the direction of my research and enhanced the quality of my work. I am sincerely grateful for her unwavering commitment, dedication, and mentorship throughout this academic journey.

Additionally, I extend my heartfelt appreciation to the Faculty of Architecture and Engineering at Epoka University for granting me access to their facilities, particularly the computing infrastructure. Their generous support has enabled me to conduct extensive training and testing of deep learning models on large-scale datasets, contributing immensely to the success of this thesis.

TABLE OF CONTENTS

ABSTRACT	iii
ABSTRAKT	Error! Bookmark not defined.
ACKNOWLEDGEMENTS	viii
LIST OF TABLES	xii
LIST OF FIGURES	xiii
LIST OF ABBREVIATIONS	xvii
CHAPTER 1	1
INTRODUCTION	1
1.1. Literature Review	2
1.2. Motivation of the Thesis	3
1.3. Objectives of the Thesis	4
1.3.1. Design and Implementation of a Custom CNN	4
1.3.2. Evaluation of Model Performance	4
1.3.2. Comparative Analysis and Model Selection	4
CHAPTER 2	6
DATASET	6
CHAPTER 3	7
DEEP LEARNING MODELS	7
3.1. Introduction to CNN	7
3.1.1. Convolutional Layers	8
3.1.2. Activation Layers	9
3.1.3. Pooling Layers	10

3.1.4.	Fully-Connected Layers.....	12
3.1.5.	Batch Normalization.....	12
3.1.6.	Dropout Layers.....	13
3.2.	Deep Learning Models.....	14
3.2.1.	Custom CNN.....	14
3.2.2.	MobileNet.....	16
3.2.3.	GoogLeNet.....	18
3.3.	TensorFlow.....	20
CHAPTER 4.....		21
IMPLEMENTATION.....		21
4.1.	Dataset Setup.....	21
4.2	Dataset Split.....	22
4.3	Data Preprocessing and Augmentation	24
4.3.1.	Data Preprocessing.....	24
4.3.2.	Data Augmentation.....	25
4.4	Data Generators.....	27
4.5	Architecture of the models.....	29
4.5.1.	Custom CNN.....	29
4.5.2.	MobileNet.....	29
4.5.3.	GooogLeNet.....	31
4.6	Compilation of the models.....	33
4.6.1.	Optimizer.....	33
4.6.2.	Loss Function.....	34

4.6.3. Compiling the models.....	34
4.7 Training the models.....	35
4.7.1. Epochs.....	35
4.7.2. Training Data.....	35
4.7.3. Validation Data.....	35
4.8 Plotting the graphs for training and validation of the models.....	36
4.9 Testing the models.....	37
4.10 Confusion Matrix and Classification Report.....	37
CHAPTER 5.....	39
RESULTS, COMPARISON AND CONCLUSION.....	39
5.1 Results of the models.....	39
5.1.1. Custom CNN - Preprocess and Augmentation.....	39
5.1.2. Custom CNN - Only Preprocess.....	42
5.1.3. MobileNet - Preprocess and Augmentation.....	45
5.1.4. MobileNet - Only Preprocess.....	48
5.1.5. GoogLeNet - Preprocess and Augmentation.....	51
5.1.6. GoogLeNet - Only Preprocess.....	53
5.2 Comparison.....	56
5.3 Conclusion.....	58
REFERENCES.....	59

LIST OF TABLES

TABLES

Table 3.1. Custom CNN architecture.....16

Table 3.2. MobileNet architecture[16].....18

Table 3.3. GoogLeNet architecture[17].....19

Table 4.1. Class distribution across training, validation and testing
.....23

Table 5.1. The results of accuracy and loss for each case.....56

LIST OF FIGURES

FIGURES

Figure 2.1: Example of parasitized cells.....	6
Figure 2.2: Example of uninfected cells.....	6
Figure 3.1: The general architecture of a CNN model [10].....	8
Figure 3.2: The kernel (small matrix) slides vertically and horizontally across the image (bigger image). It slides over each pixel of input and it convolves with it and the output is stored. [11].....	9
Figure 3.3: Representation of ReLU activation function [12].....	10
Figure 3.4: Pooling layer operation approaches [13].....	11
Figure 3.5: Left: Two FC layers that are connected without dropout. Right: Two FC layers that are connected after dropping 50% of the original connections. [11]	14
Figure 4.1: Visual representation of the dataset in the corresponding subfolders/classes.....	21
Figure 4.2: Graphical representation of the class distribution of Malaria dataset.....	22
Figure 4.3: Part of the code that splits the dataset into 3 folders: train, validation and test	23
Figure 4.4: Graphical representation of class distribution after splitting the dataset.....	23
Figure 4.5: Augmentation and preprocessing techniques for Malaria dataset.....	27
Figure 4.6: Part of the code that demonstrates the setup of data generators for training, validation, and test datasets.....	28

Figure 4.7: The generator for the models with the original images, not augmented ones.....	28
Figure 4.8: The code used for creating CNN architecture.....	29
Figure 4.9: The code used for creating MobileNet architecture.....	31
Figure 4.10: The code used for creating inception_module of GoogLeNet architecture.....	32
Figure 4.11: The code used for creating GoogleNet module.....	33
Figure 4.12: The code used for compiling the models.....	34
Figure 4.13: The code used for training the models.....	36
Figure 4.14: The code used for plotting accuracy and/or loss representation for training and validation dataset.....	37
Figure 4.15: The code used for evaluating the confusion matrix and classification report.....	38
Figure 5.1: Last three epochs of the augmented custom CNN model.....	39
Figure 5.2: Graphical representation of loss function for augmented custom CNN model.....	40
Figure 5.3: Graphical representation of accuracy function for augmented custom CNN model.....	40
Figure 5.4: Confusion matrix for the ‘test’ dataset of augmented Custom CNN model.....	41
Figure 5.5: Classification report for the ‘test’ dataset of augmented Custom CNN model.....	42
Figure 5.6: Last three epochs of the only preprocessed custom CNN model.....	42
Figure 5.7: Graphical representation of loss function for the only preprocessed custom CNN model.....	43
Figure 5.8: Graphical representation of accuracy function for the only preprocessed custom CNN model.....	43
Figure 5.9: Confusion matrix for the ‘test’ dataset of the only preprocessed custom CNN model.....	44

Figure 5.10: Classification report for the ‘test’ dataset of the only preprocessed custom CNN model.....	44
Figure 5.11: Last three epochs of the augmented MobileNet model.....	45
Figure 5.12: Graphical representation of loss function for augmented MobileNet model.....	46
Figure 5.13: Graphical representation of accuracy function for augmented MobileNet model.....	46
Figure 5.14: Confusion matrix for the ‘test’ dataset of augmented MobileNet model..	47
Figure 5.15: Classification report for the ‘test’ dataset of augmented MobileNet model	48
Figure 5.16: Last three epochs of the only preprocessed MobileNet model	48
Figure 5.17: Graphical representation of loss function for the only preprocessed MobileNet model	49
Figure 5.18: Graphical representation of accuracy function for the only preprocessed MobileNet model	49
Figure 5.19: Confusion matrix for the ‘test’ dataset of the only preprocessed MobileNet model	50
Figure 5.20: Classification report for the ‘test’ dataset of the only preprocessed MobileNet model	50
Figure 5.21: Last three epochs of the augmented GoogLeNet model.....	51
Figure 5.22: Graphical representation of loss function for augmented GoogLeNet model.....	51
Figure 5.23: Graphical representation of accuracy function for augmented GoogLeNet model.....	52
Figure 5.24: Confusion matrix for the ‘test’ dataset of augmented GoogLeNet model.....	52
Figure 5.25: Classification report for the ‘test’ dataset of augmented GoogLeNet model.....	52
Figure 5.26: Last three epochs of the only preprocessed GoogLeNet model.....	53

Figure 5.27: Graphical representation of loss function for the only preprocessed
GoogLeNet model.....54

Figure 5.28: Graphical representation of accuracy function for the only preprocessed
GoogLeNet model.....54

Figure 5.29: Confusion matrix for the ‘test’ dataset of the only preprocessed GoogLeNet
model.....55

Figure 5.30: Classification report for the ‘test’ dataset of the only preprocessed
GoogLeNet model.....55

LIST OF ABBREVIATIONS

NLM	National Library of Medicine
NIH	National Institute of Health
CMC Hospital	Christian Medical College
WHO	World Health Organization
CNN	Convolutional Neural Network
ANN	Artificial Neural Network
ECM	Scanning Electron Microscope
ReLU	Rectified Linear Unit
CONV	Convolutional Layer
FC	Fully Connected Layers
RGB	Red, Green and Blue
VGG	Visual Geometry Group

CHAPTER 1

INTRODUCTION

Malaria, a devastating mosquito-borne disease caused by Plasmodium parasites, has plagued humanity for millennia. Historical records indicate its presence in ancient civilizations, with descriptions dating back to ancient Chinese and Indian texts [1]. The disease's global impact is staggering, with the World Health Organization (WHO) estimating 241 million malaria cases and 627,000 deaths in 2020 alone [2]. Despite significant progress in control and prevention efforts, malaria remains a leading cause of morbidity and mortality, particularly in sub-Saharan Africa and other resource-limited regions.

The life cycle of the malaria parasite involves both human and mosquito hosts. When an infected female mosquito bites a human, it injects Plasmodium parasites into the bloodstream. These parasites travel to the liver, where they multiply and mature. Subsequently, they infect red blood cells, causing them to rupture and release more parasites, leading to the characteristic symptoms of malaria, such as fever, chills, and fatigue [3].

Traditionally, malaria diagnosis relies on microscopic examination of Giemsa-stained blood smears, where skilled technicians identify and quantify parasites within red blood cells [4]. While considered the gold standard, this method is time-consuming, labor-intensive, and requires specialized expertise, making it impractical in many settings.

Moreover, the accuracy of microscopic diagnosis can be affected by factors such as the examiner's experience and the parasite density in the blood sample.

Recent advancements in deep learning, a subfield of artificial intelligence, offer a promising avenue for automating and enhancing malaria diagnosis [5]. By leveraging large datasets of annotated blood smear images, deep learning models can learn to identify and classify malaria parasites with high accuracy, potentially surpassing human performance. However, existing models often prioritize accuracy over efficiency and interpretability, hindering their widespread adoption, particularly in resource-limited settings.

1.1 Literature Review

The application of deep learning techniques to medical imaging, particularly for disease diagnosis, has gained significant traction in recent years. In the context of malaria detection, several studies have explored the potential of Convolutional Neural Networks (CNNs) to automate and enhance the analysis of microscopic blood smear images [6].

Mishra (2021) explored the use of transfer learning and snapshot ensembles for malaria parasite detection in thin blood smears [6]. The study utilized pre-trained models like EfficientNet-B0, DenseNet121, Inception-v3, and ResNet50-V2, among others [6]. The snapshot ensemble of EfficientNet-B0 achieved the highest F1 score of 99.37%, demonstrating the effectiveness of ensemble techniques in improving model performance [6].

Sarkar et al. (2020) focused on developing shallow CNN architectures for malaria detection, aiming to reduce computational complexity while maintaining high accuracy [7]. Their 3-layered CNN model achieved an accuracy of 95.32%, comparable to that of deeper models like VGG-16 and ResNet-50 [7]. This study highlighted the potential of simpler models for efficient malaria screening, particularly in resource-constrained environments [7].

Schwarz Schuler et al. (2022) proposed an enhanced scheme for reducing the complexity of pointwise convolutions in CNNs for image classification [8]. Their approach, based on interleaved grouped filters, was applied to EfficientNet-B0 and tested on various datasets, including a malaria dataset [8]. The modified model, kEffNet-B0 V2, achieved comparable accuracy to the baseline EfficientNet-B0 while significantly reducing the number of parameters and computations [8].

These studies collectively demonstrate the potential of deep learning in revolutionizing malaria diagnosis. While Mishra (2021) and Schwarz Schuler et al. (2022) focused on improving model efficiency and reducing complexity [6,8], Sarkar et al. (2020) emphasized the importance of interpretability in medical image analysis [7]. The highest accuracy was achieved by Mishra (2021) using a snapshot ensemble of EfficientNet-B0, underscoring the effectiveness of ensemble techniques in this domain [6]. However, the choice of the optimal model depends on various factors, including the specific requirements of the application, available computational resources, and the desired balance between accuracy, efficiency, and interpretability.

1.2 Motivation of the Thesis

The urgent need for accessible, accurate, and efficient malaria diagnostic tools, particularly in resource-limited regions where the disease burden is highest, serves as the driving force behind this thesis. Current diagnostic methods, while effective, are often hindered by logistical challenges and the need for specialized expertise. Deep learning, with its demonstrated potential in image analysis, offers a promising avenue for developing automated screening tools that can overcome these limitations.

This study aims to explore the potential of different deep learning architectures for malaria parasite detection in thin blood smears. By investigating a custom-designed Convolutional Neural Network (CNN) alongside established models like MobileNet and GoogLeNet, we

seek to identify an optimal model that balances high diagnostic accuracy with computational efficiency and interpretability. Additionally, the objective of this study is to assess whether using augmented training datasets enhances the model's performance in malaria parasite detection. The ultimate goal is to contribute to the development of robust, accessible, and deployable automated malaria screening tools that can significantly enhance diagnostic capabilities in diverse healthcare settings, especially those with limited resources and expertise.

1.3 Objectives of the Thesis

1.2.1 Design and Implementation of a Custom CNN:

A custom CNN architecture will be designed and implemented, specifically tailored to capture the unique morphological features of malaria parasites in microscopic images. This involves careful consideration of network depth, filter sizes, activation functions, and other hyperparameters to optimize the model's performance.

1.2.2 Evaluation of Model Performance:

The diagnostic performance of the custom CNN, MobileNet, and GoogLeNet models will be rigorously evaluated using a diverse dataset of thin blood smear images. Performance metrics will include accuracy, precision, recall, F1 score and support. Computational efficiency will also be assessed by measuring inference time and memory usage.

1.2.3 Comparative Analysis and Model Selection:

The strengths and weaknesses of each model will be thoroughly analyzed and compared. The most suitable architecture for real-world malaria screening applications will be identified based on its performance, efficiency, and interpretability, considering the specific needs and constraints of different healthcare settings. Specifically, the study will

examine whether using augmented training datasets improves the models' performance metrics compared to models trained on only preprocessed datasets.

By addressing these objectives, this thesis aims to contribute to the development of automated malaria screening tools that are accurate, efficient, interpretable, and accessible, ultimately improving malaria diagnosis and control efforts globally.

CHAPTER 2

DATASET

The malaria dataset, sourced from the National Library of Medicine (NLM), part of the National Institutes of Health (NIH), is a crucial resource for research in the field of automated malaria diagnosis [6,8]. It comprises 27,558 segmented red blood cell images derived from Giemsa-stained thin blood smears. These images are equally divided into two classes: parasitized (infected) and uninfected, providing a balanced dataset for training machine learning models [6,8]. The samples were collected from 150 infected and 50 uninfected individuals at the Malaria Screener research activity in CMC Hospital, Bangladesh [6].

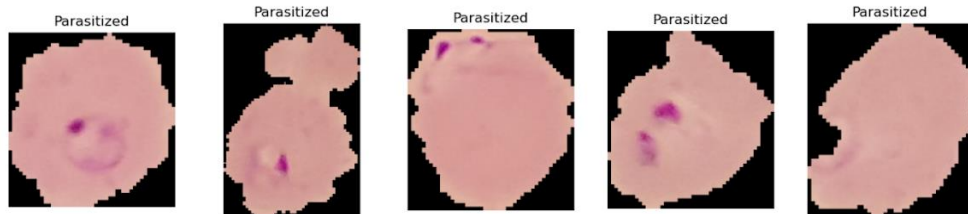


Figure 2.1: Example of parasitized cells

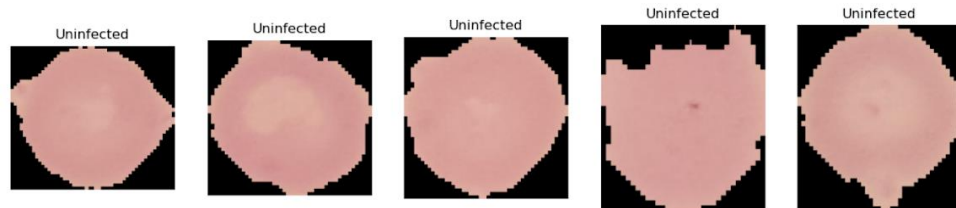


Figure 2.2: Example of uninfected cells

CHAPTER 3

DEEP LEARNING MODELS

3.1. Introduction to CNN

Convolutional Neural Networks (CNNs) are a specialized type of Artificial Neural Network (ANN) designed for efficient image and video recognition [9]. Similar to ANNs, CNNs consist of neurons that learn through optimization. Each neuron receives input, performs a dot product, and optionally follows it with a non-linearity [9]. The power of CNNs lies in their ability to automatically learn and extract relevant features from images, eliminating the need for manual feature engineering [7].

CNNs are specifically tailored for image data, with their architecture designed to process input in the form of multiple arrays, where each array corresponds to a color channel (e.g., red, green, blue) [9]. A key characteristic of CNNs is their local connectivity, where neurons in one layer connect only to a small region of neurons in the previous layer, enabling them to capture local patterns and spatial hierarchies effectively [9].

As information flows through the layers of a CNN, which typically include convolutional layers, activation functions, pooling layers, and fully connected layers, the network learns to detect increasingly complex features [7]. For example, early layers might identify edges and corners, while deeper layers recognize more abstract patterns like shapes and objects. This hierarchical feature extraction allows CNNs to achieve remarkable performance in image classification tasks, making them a cornerstone of modern computer vision.

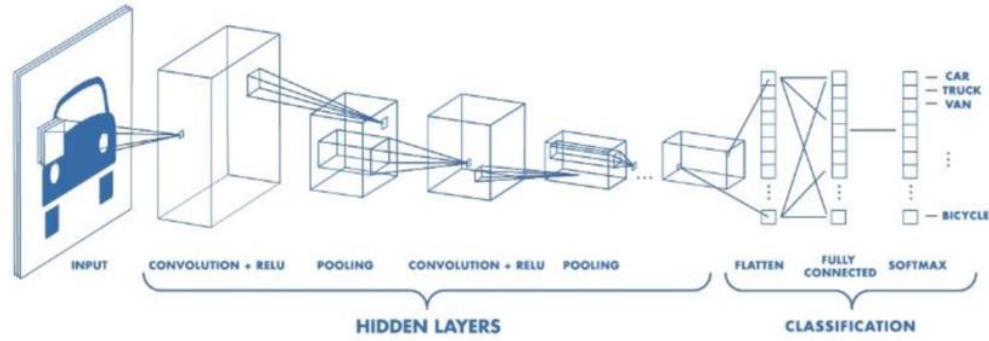


Figure 3.1: The general architecture of a CNN model [10]

3.1.1 Convolutional Layers:

The fundamental element in a Convolutional Neural Network is the CONV layer, serving as a key component in the network's architecture [11]. Within the CONV layer, a set of K learnable filters, known as "kernels," are employed. Each kernel is characterized by its width and height, which are typically small and square in shape, e.g., 3x3 or 5x5 pixels. However, the depth of each kernel matches the depth of the input volume, meaning it extends through all the channels of the input image.

During the convolution operation, each kernel slides across the image from left to right and top to bottom, one step at a time. At each position, the kernel's values are multiplied element-wise with the corresponding pixel values in the image, and the results are summed to produce a single output value. This process is repeated for each kernel at every position, resulting in a feature map that highlights specific patterns or features detected by the kernel.

131	162	232	84	91	207
104	-1	10	+1	237	109
243	-2	20	+2	135	26
185	-1	20	+1	61	225
157	124	25	14	102	108
5	155	116	218	232	249

Figure 3.2: The kernel (small matrix) slides vertically and horizontally across the image (bigger image). It slides over each pixel of input and it convolves with it and the output is stored. [11]

By employing multiple kernels, each tuned to detect different features, the CNN can learn to recognize a wide variety of visual patterns, from simple edges and corners to more complex shapes and textures. The depth of the output volume after a CONV layer is equal to the number of kernels used, reflecting the multiple feature maps generated by the convolution process.

This hierarchical feature extraction process continues as the data flows through subsequent layers of the CNN, with deeper layers learning to recognize increasingly abstract and complex features.

3.1.2 Activation Layers:

Following each convolutional (CONV) layer within a CNN, a non-linear activation function is applied element-wise to the output feature maps [11]. This activation layer is crucial for introducing non-linearity into the network, enabling it to learn complex patterns

and relationships in the data [11]. While often represented as ReLU (Rectified Linear Unit) in network diagrams due to its widespread use, other activation functions like sigmoid, tanh, or swish can also be employed [6, 11].

In the context of malaria detection, the choice of activation function can impact the model's performance. Although activation layers do not have learnable parameters, they are integral to the CNN architecture. They operate on the input volume, maintaining the same dimensions ($W_{input} = W_{output}$, $H_{input} = H_{output}$, $D_{input} = D_{output}$) while applying the chosen activation function to each element [11]. This non-linear transformation allows the network to model complex decision boundaries and ultimately improve its ability to classify malaria-infected cells accurately.

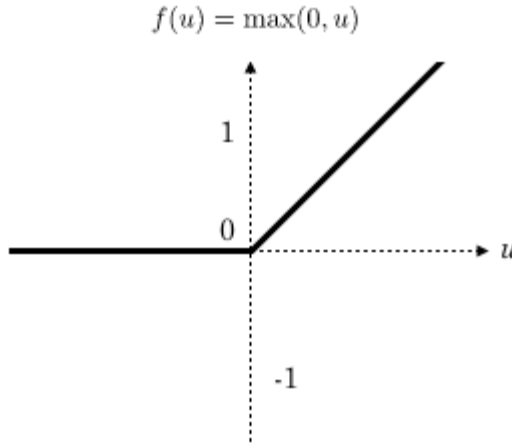


Figure 3.3: Representation of ReLU activation function [12]

3.1.3 Pooling Layers:

Pooling layers are essential components of Convolutional Neural Networks (CNNs) that serve to progressively reduce the spatial dimensions (width and height) of the input volume [11]. This reduction in dimensionality not only decreases the computational burden and memory requirements of the network but also helps to mitigate overfitting by introducing a degree of translational invariance [11].

Pooling layers operate independently on each depth slice of the input volume, typically applying either the max or average function [11]. Max pooling, which selects the maximum value within a local neighborhood, is often preferred in earlier layers to extract dominant features [11]. Average pooling, on the other hand, calculates the average value within a neighborhood and is sometimes used in later layers or as a replacement for fully connected layers in certain architectures [11].

In the context of malaria detection, pooling layers play a crucial role in reducing the complexity of the feature maps while retaining essential information for parasite identification. For instance, Mishra (2021) employed max-pooling layers in their custom CNN architecture to downsample the feature maps and improve the model's efficiency [6].

While max pooling has been the dominant pooling operation in CNNs, recent research has explored alternative pooling strategies and more sophisticated micro-architectures to further enhance the network's performance and efficiency [8]. These advancements in pooling mechanisms contribute to the ongoing evolution of CNNs for various image recognition tasks, including the critical domain of malaria detection.

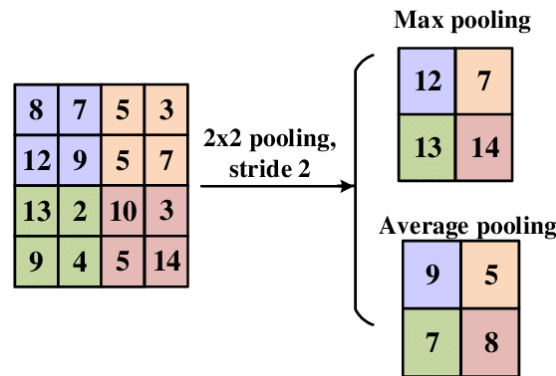


Figure 3.4: Pooling layer operation approaches [13]

3.1.4 Fully-Connected Layers:

Fully connected (FC) layers, also known as dense layers, are an integral part of many neural network architectures, including Convolutional Neural Networks (CNNs) [11]. In FC layers, each neuron is connected to every activation in the previous layer, akin to traditional feedforward neural networks [11,13]. These layers typically reside towards the end of the network, following the convolutional and pooling layers, and are responsible for aggregating the extracted features from earlier layers to produce the final classification output [11].

In the context of malaria detection using CNNs, FC layers play a crucial role in interpreting the high-level features learned by the convolutional layers and mapping them to the final decision of whether a cell is parasitized or not. For example, in the study by Mishra (2021), the EfficientNet-B0 architecture utilized FC layers to combine the extracted features and produce the final classification probabilities for each class [6].

The number of neurons and layers in the FC section can vary depending on the complexity of the task and the desired model capacity [11]. Deeper FC layers with more neurons can potentially capture more intricate relationships in the data but might also increase the risk of overfitting [11]. Therefore, careful consideration of the FC layer configuration is essential to achieve optimal performance in malaria detection tasks.

3.1.5 Batch Normalization:

Batch normalization is a technique commonly employed in training deep neural networks, including Convolutional Neural Networks (CNNs) [11,14]. It involves normalizing the activations of each layer for each mini-batch during training [11,14]. This normalization helps to stabilize and accelerate the training process, allowing for faster convergence and potentially improved performance [11,14].

In the context of malaria detection, batch normalization can be beneficial in several ways. It can help to mitigate the vanishing gradient problem, which can hinder the training of deep networks [14]. Additionally, it can reduce the sensitivity of the model to the initialization of weights and the choice of learning rate, making the training process more robust [11,14].

Mishra (2021) utilized batch normalization in their custom CNN model for malaria parasite detection, likely contributing to the model's good convergence and performance [6]. While batch normalization does not eliminate the need for careful hyperparameter tuning, it can simplify the process and make the training more stable [11,14].

The benefits of batch normalization extend beyond faster training. It can also lead to improved generalization performance, as the normalized activations tend to be more robust to variations in the input data [11,14]. This can be particularly advantageous in malaria detection, where the appearance of parasites can vary depending on the staining technique and image quality.

3.1.6 Dropout Layers:

Dropout is a regularization technique that operates by randomly deactivating a proportion of neurons during each training iteration, effectively creating an ensemble of different network architectures [11, 15]. This process forces the network to learn redundant representations of the data, reducing its reliance on any single neuron and improving generalization to unseen examples [11, 15].

In the context of malaria detection, dropout can be particularly beneficial due to the limited size of available datasets and the potential for overfitting. By applying dropout, the model becomes less sensitive to the specific training examples and learns to generalize better to new blood smear images. Mishra (2021) employed dropout in their custom CNN architecture, likely contributing to the model's ability to generalize well to the test set [6].

The probability of a neuron being dropped out, denoted as 'p', is a hyperparameter that needs to be tuned for optimal performance [11]. While dropout may slightly decrease training accuracy, it often leads to significant improvements in testing accuracy, indicating better generalization [11, 15].

In addition to preventing overfitting, dropout can also act as a form of implicit model averaging, where the predictions of multiple thinned networks are combined [15]. This can further enhance the model's robustness and performance in classifying malaria-infected cells.

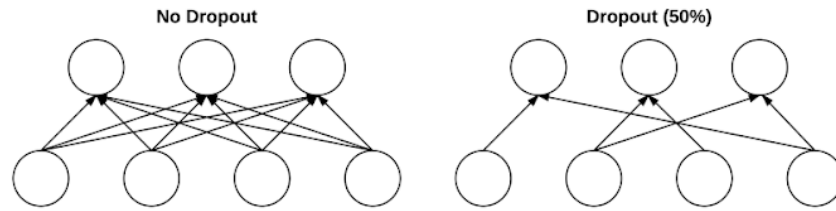


Figure 3.5: **Left:** Two FC layers that are connected without dropout. **Right:** Two FC layers that are connected after dropping 50% of the original connections. [11]

3.2. Deep Learning Models

In this section, we will discuss the deep learning models utilized in my research for malaria classification. I used three different models, namely CustomCNN, MobileNet, and GoogLeNet which have shown remarkable performance in various computer vision tasks.

3.2.1 Custom CNN:

The Convolutional Neural Network (CNN) model that I have modeled is for binary image classification, specifically designed to discern malaria-infected cells within microscopic

images. It adheres to a standard CNN architecture, comprising convolutional layers, pooling layers, and fully connected layers.

The model accepts input images with dimensions of `img_width` x `img_height` pixels and three colour channels (RGB). These channels correspond to the red, green, and blue components of the image.

The initial layer is a convolutional layer equipped with 32 filters, each possessing a 3x3 kernel size. This layer applies the filters across the input image, effectively extracting localized features like edges and textures [11]. Subsequently, the Rectified Linear Unit (ReLU) activation function is employed to introduce non-linearity, enhancing the model's capacity to learn intricate patterns [11].

Max pooling, with a 2x2 pool size, is then implemented to reduce the spatial dimensions of the feature maps, preserving the most noteworthy information while minimizing computational demands.

This sequence of convolution, activation, and pooling is repeated three additional times. Notably, the number of filters in each subsequent convolutional layer increases (64, 128, and 128, respectively), enabling the model to learn progressively more abstract and complex features at varying scales.

Following the convolutional and pooling layers, the output is flattened into a one-dimensional vector. This transformation prepares the extracted features for processing by the fully connected layers.

A dense layer comprising 64 neurons and ReLU activation is introduced to further refine the extracted features [10]. The final dense layer, consisting of a single neuron with a sigmoid activation function, generates a probability score ranging from 0 to 1. This score signifies the model's confidence in classifying the input image as containing a malaria-infected cell.

Layer (type)	Output Shape	Param #
malaria_cells (InputLayer)	(None, 224, 224, 3)	0
conv2d_12 (Conv2D)	(None, 224, 224, 32)	896
max_pooling2d_12 (MaxPooling2D)	(None, 112, 112, 32)	0
conv2d_13 (Conv2D)	(None, 112, 112, 64)	18,496
max_pooling2d_13 (MaxPooling2D)	(None, 56, 56, 64)	0
conv2d_14 (Conv2D)	(None, 56, 56, 128)	73,856
max_pooling2d_14 (MaxPooling2D)	(None, 28, 28, 128)	0
conv2d_15 (Conv2D)	(None, 28, 28, 128)	147,584
max_pooling2d_15 (MaxPooling2D)	(None, 14, 14, 128)	0
flatten_3 (Flatten)	(None, 25088)	0
dense_3 (Dense)	(None, 64)	1,605,696
cell_classes (Dense)	(None, 1)	65

Table 3.1: Custom CNN architecture

3.2.2 MobileNet

MobileNets, introduced by Howard et al. (2017), are a class of convolutional neural network (CNN) architectures specifically designed for mobile and embedded vision applications [16]. The primary objective behind MobileNets is to develop lightweight and efficient models capable of delivering strong performance on devices with constrained computational resources, while minimizing the sacrifice in accuracy.

1. **MobileNet Architecture:** The backbone of MobileNet revolves around depthwise separable convolutions, a pivotal architectural choice that markedly reduces computational demands compared to traditional convolutional approaches [16]. By decoupling the filtering and combination stages, this model achieves a substantial reduction in parameters and operations, enhancing its speed and efficiency.

The network commences with a standard convolutional layer, followed by a sequence of depthwise separable convolutions [16]. This initial convolutional layer extracts fundamental low-level features from the input image. Subsequent depthwise separable convolutions then refine these features and extract higher-level representations, all while minimizing computational overhead.

Downsampling, crucial for reducing spatial dimensions and enlarging receptive fields in CNNs, is strategically integrated into the MobileNet architecture using strided convolutions within the depthwise layers [16]. This approach facilitates efficient downsampling without necessitating additional pooling layers, thereby further enhancing the overall efficiency of the model.

The concluding layers of the MobileNet model include a global average pooling layer, which aggregates features across spatial dimensions, and a FC layer that generates the final classification output [16]. This streamlined design eliminates the need for computationally expensive fully connected layers throughout the network, contributing significantly to reducing the model's size and complexity.

In essence, the MobileNet architecture represents a meticulously balanced fusion of depthwise separable convolutions, pointwise convolutions, and strategic downsampling. This design not only enables the model to achieve an impressive balance between accuracy, efficiency, and model size but also renders it highly suitable for mobile and embedded vision applications where computational resources are limited.

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5 \times$	Conv dw / s1	$3 \times 3 \times 512$ dw
	Conv / s1	$1 \times 1 \times 512 \times 512$
	Conv dw / s2	$3 \times 3 \times 512$ dw
	Conv / s1	$1 \times 1 \times 512 \times 1024$
	Conv dw / s2	$3 \times 3 \times 1024$ dw
	Conv / s1	$1 \times 1 \times 1024 \times 1024$
	Avg Pool / s1	Pool 7×7
	FC / s1	1024×1000
	Softmax / s1	Classifier

Table 3.2: MobileNet architecture [16]

3.2.3 GoogLeNet

[17] Introduced in 2014 by Szegedy et al., GoogLeNet represents a significant advancement in the field of deep learning and computer vision. This deep convolutional neural network (CNN) gained prominence for its innovative architecture aimed at improving computational efficiency while maintaining high accuracy in image classification and object detection tasks. GoogLeNet's design principles have influenced subsequent developments in CNN architectures, making it a foundational model in the pursuit of efficient and effective deep learning solutions for visual recognition tasks.

1. **GoogLeNet Architecture:** At its core lies the Inception module, designed to optimize computational resources while capturing features at multiple scales. This module achieves efficiency by employing parallel convolutional filters of different sizes (1x1, 3x3, and 5x5) alongside a max-pooling path. To manage the computational complexity associated with larger filter sizes, dimensionality reduction using 1x1 convolutions is integrated before applying the more computationally intensive convolutions.

Szegedy et al.'s paper [17] provides a detailed architectural overview of GoogLeNet, including layer types, sizes, strides, and computational metrics, underscoring its efficient design. By stacking multiple Inception modules with periodic max-pooling layers, GoogLeNet scales effectively without overwhelming computational demands. This strategic approach allows the network to process visual information across various scales, enhancing its ability to abstract features simultaneously from different perspectives.

The design philosophy of GoogLeNet reflects an intuitive understanding that effective visual processing entails capturing information across multiple scales and integrating it seamlessly. This approach, coupled with its computational efficiency, has positioned GoogLeNet as a pioneering model in the field of computer vision, setting a benchmark for subsequent advancements in deep learning architectures.

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Table 3.3: GoogLeNet architecture [17]

3.3 Tensor Flow

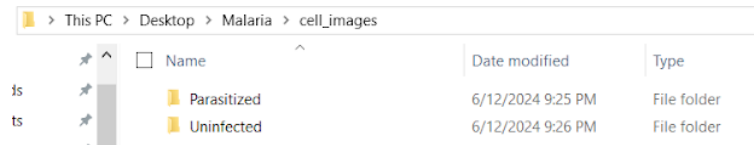
TensorFlow, developed by Google, stands as a cornerstone in modern machine learning frameworks due to its extensive capabilities and robust infrastructure [18]. It offers unparalleled advantages for deploying machine learning models, making it an ideal choice for my research on malaria detection. TensorFlow's key strengths lie in its ability to efficiently handle complex computations, scale seamlessly across distributed systems, and facilitate rapid prototyping and deployment of models. Leveraging pre-trained models like MobileNet and GoogLeNet, which have been trained on the ImageNet dataset, within TensorFlow is particularly advantageous. These models provide a solid foundation through transfer learning, allowing me to harness their learned features for accurate malaria detection using my dataset. This approach not only accelerates model development but also ensures that the models are optimized for performance on resource-constrained environments typical in healthcare applications.

CHAPTER 4

IMPLEMENTATION

4.1 Dataset Setup

[19] The malaria dataset is publicly available through TensorFlow Datasets, which is a collection of datasets ready to use with TensorFlow and Keras machine learning frameworks. The dataset can be easily accessed and loaded using the TensorFlow Datasets API, allowing researchers and practitioners to seamlessly integrate it into their deep learning workflows. As mentioned in Chapter 2, the dataset has 27,558 single-cell images, each labeled as either ‘Parasitized’ or ‘Uninfected’ (see Figure 4.1). This dataset has a balanced distribution of classes which ensures that deep learning models trained for this dataset are not biased. Figure 4.2 illustrates the class distribution of the dataset.



	Name	Date modified	Type
ts	Parasitized	6/12/2024 9:25 PM	File folder
ts	Uninfected	6/12/2024 9:26 PM	File folder

Figure 4.1: Visual representation of the dataset in the corresponding subfolders/classes

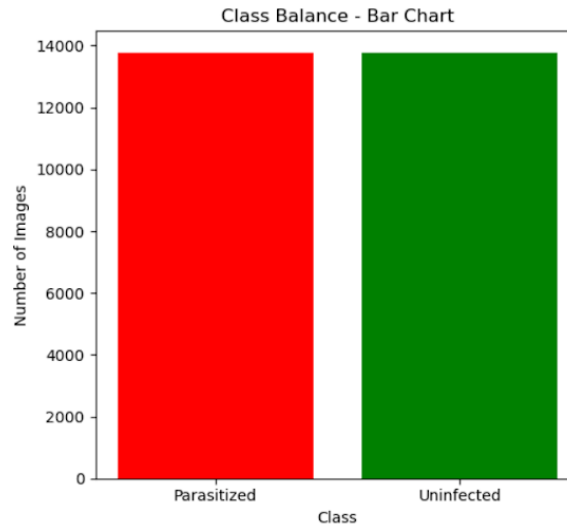


Figure 4.2: Graphical representation of the class distribution of Malaria dataset.

4.2 Dataset Split

To prepare the malaria dataset for model training and evaluation, it was divided into three distinct subsets: 70% for training, 15% for validation, and 15% for testing. This division ensures that the model learns from a substantial portion of the data, is validated on unseen samples during training, and is ultimately evaluated on a completely independent test set. The splitting process was performed using the 'splitfolder' library from Keras, which preserves the original class structure by maintaining the 'parasitized' and 'uninfected' subfolders within each subset (see Figure 4.3). This approach ensures that the class distribution remains balanced across the training, validation, and test sets, preventing any potential biases in the model's learning process.

```

# Define input and output directories
input_folder = pathlib.Path("cell_images/")
output_folder = pathlib.Path("split_data/")

# Check if the output directory already exists
if not output_folder.exists():
    # Split with a fixed ratio
    splitfolders.ratio(
        input=str(input_folder),
        output=str(output_folder),
        seed=1337,
        ratio=(0.7, 0.15, 0.15), # 70% train, 15% val, 15% test
        group_prefix=None,
        move=False # copy files instead of moving them
    )
    print("Dataset has been split into train, val, and test sets.")
else:
    print("The split_data directory already exists. No splitting performed.")

```

Figure 4.3: Part of the code that splits the dataset into 3 folders: train, validation and test

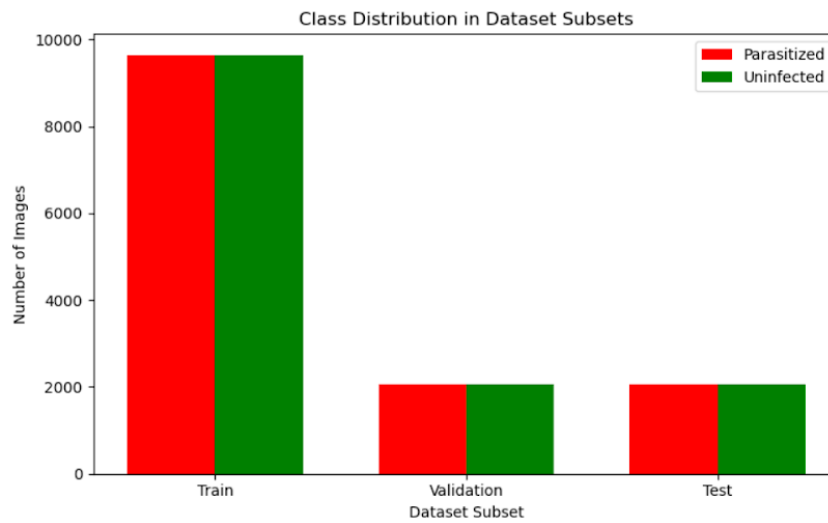


Figure 4.4: Graphical representation of class distribution after splitting the dataset

	<i>Parasitized</i>	<i>Uninfected</i>
<i>Train</i>	9645	9645
<i>Validation</i>	2066	2066
<i>Test</i>	2068	2068

Table 4.1: Class distribution across training, validation and testing

4.3 Dataset Preprocessing and Augmentation

Data preprocessing and data augmentation are essential techniques in machine learning and computer vision that aim to enhance the quality and diversity of training data, thereby improving the performance and robustness of machine learning models.

1. Data Preprocessing

Data preprocessing process involves a series of operations performed on raw data to make it suitable for training a model. This typically includes steps such as normalization (scaling data to a standard range), handling missing values, feature scaling, and data transformation (e.g., converting categorical variables to numerical ones) [20]. The goal is to ensure that the data is in a consistent format and range that allows the model to learn effectively from it.

In this dataset preprocessing pipeline, resizing and normalizing images are crucial steps that ensure consistent input dimensions and efficient training. I resize all images to 224x224 pixels, which standardizes their size regardless of their original dimensions. This resizing is essential for maintaining uniformity in the input data, enabling batch processing and compatibility with neural network architectures, which often require fixed input sizes. Additionally, resizing reduces the computational load and memory usage, facilitating faster and more efficient model training.

Alongside resizing, I normalize the pixel values of the images by scaling them by a factor of $1/255$. This normalization step converts pixel values from their original range of $[0, 255]$ to a normalized range of $[0, 1]$. Normalization is critical as it ensures that the input data has a consistent scale, which helps in accelerating the convergence of the neural network during training. By having pixel values in a smaller range, the model can process the images more effectively, reducing the risk of issues related to varying scales in the

input data. Together, resizing and normalization create a well-preprocessed dataset that enhances the model's ability to learn and generalize from the training data, leading to improved performance on unseen images.

2. Data Augmentation

Data augmentation process involves generating new synthetic data from existing data by applying random transformations such as rotations, translations, flips, zooms, and crops [21]. This technique helps to increase the diversity of the training data without collecting additional samples. By exposing the model to variations in the data during training, data augmentation improves the model's ability to generalize to unseen examples and enhances its robustness against overfitting [21].

For Malaria dataset, image augmentation plays a pivotal role in enhancing the robustness and generalization capability of our neural network. We implement augmentation using the `ImageDataGenerator` class from Keras, which applies a variety of random transformations to the training images.

1. **Rotation:** Rotates images randomly within a specified degree range (in this case, up to 30 degrees). This helps the model learn to recognize objects from different viewpoints, enhancing its ability to generalize.
2. **Shear:** Applies a shearing transformation to images. Shear distorts the shape of the image along a given axis, which can simulate effects like tilting or skewing. This augmentation helps the model handle variations in object angles and perspectives.

3. **Zoom:** Randomly zooms into or out of images by a specified percentage (up to 7%). Zoom augmentation helps the model learn to focus on different parts of an object, improving its robustness to varying object scales.
4. **Horizontal Flip:** Flips images horizontally with a probability of 50%. This augmentation introduces variations in the orientation of objects, ensuring the model can recognize objects regardless of their left-right orientation.
5. **Brightness Adjustment:** Randomly adjusts the brightness of images within a specified range (here, between 0.3 and 0.6). This augmentation mimics changes in lighting conditions, helping the model become more invariant to brightness variations.
6. **Width Shift and Height Shift:** Shifts images horizontally and vertically by a fraction of the total width and height (up to 7%). These augmentations simulate translations of objects within the image, improving the model's ability to recognize objects in different positions.

By incorporating these augmentations, we effectively increase the diversity of our training dataset without actually increasing the number of images. This approach helps prevent overfitting, as the model is exposed to slightly altered versions of the images in each epoch, promoting better learning of invariant features. Furthermore, the augmentation process is performed in real-time, ensuring that each batch of training data is unique. Overall, the implementation of augmentation significantly improves the model's ability to generalize from the training data to unseen data, leading to more accurate and reliable predictions in real-world scenarios.

```

image_gen_train = ImageDataGenerator(
    rotation_range=30,
    shear_range=0.08,
    zoom_range=0.07,
    horizontal_flip=True,
    validation_split = 0.3,
    rescale = 1/255,
    brightness_range=[0.3, 0.6],
    width_shift_range=0.07,
    height_shift_range=0.07
)

image_gen = ImageDataGenerator(
    rescale = 1/255,
)

```

Figure 4.5: Augmentation and preprocessing techniques for Malaria dataset

4.4 Data Generators

In Figure 4.6, a snippet of code demonstrates the setup of data generators for training, validation, and test datasets using the ‘ImageDataGenerator’ class from Keras. For training data, the images are loaded from the ‘train’ directory using ‘image_gen_train.flow_from_directory’. This function applies augmentation techniques and preprocessing steps, resizing each image to 224x224 pixels, and organizing them into batches of 32. To maintain reproducibility, a random seed of 42 is employed, and ‘class_mode’ is set to “binary” for this binary classification problem.

For validation and test datasets, images are loaded using ‘image_gen.flow_from_directory’ from the ‘val’ and ‘test’ directories, respectively. Here, the images are resized to 224x224 pixels and normalized for consistent preprocessing, without additional augmentation. Notably, for the test data, the ‘shuffle’ parameter is set to ‘False’ to ensure the evaluation occurs in a consistent and ordered manner. The seed value of 42 is also used for reproducibility across all datasets. This meticulous setup

guarantees that all datasets are appropriately preprocessed and ready for training, validation, and testing the neural network.

```
batch_size=32

training_data = image_gen_train.flow_from_directory(
    input_folder / 'train',
    class_mode="binary",
    target_size = (224, 224),
    batch_size = batch_size,
    seed = 42
)

validation_data = image_gen.flow_from_directory(
    input_folder / 'val',
    class_mode="binary",
    target_size = (224, 224),
    batch_size = batch_size,
    seed = 42
)

test_data = image_gen.flow_from_directory(
    input_folder / 'test',
    class_mode="binary",
    target_size = (224, 224),
    batch_size = batch_size,
    shuffle = False,
    seed = 42
)
```

Figure 4.6: Part of the code that demonstrates the setup of data generators for training, validation, and test datasets

This is when we are applying augmentation for the images in the ‘training’ folder. When we are not applying augmentation, only preprocessing techniques to the ‘training’ folder, the ‘image_gen’ generator will only normalize the image, as seen below in Figure 4.7:

```
#Only normalizing
image_gen = ImageDataGenerator(
    rescale = 1/255
)
```

Figure 4.7: The generator for the models with the original images, not augmented ones.

4.5 Architecture of the models

1. Custom CNN:

The custom CNN architecture employed in this study is detailed in Table 3.1. As a model developed by me, all layers within this architecture are trainable. This means that during the training process, the parameters (weights and biases) of every layer in the CNN can be adjusted or optimized based on the available training data. Figure 4.8 below illustrates the code that specifies this particular architecture, providing a visual reference for its implementation and configuration in the context of the study. This setup allows for fine-tuning and optimization of the model's performance through iterative learning from the training data.

```
input_shape = (224, 224, 3)
input_tensor = Input(shape=input_shape, dtype=tf.float32, name="malaria_cells")

# Convolutional Layers
X = Conv2D(filters=32, kernel_size=(3, 3), activation='relu', padding='same')(input_tensor)
X = MaxPooling2D(pool_size=(2, 2))(X)

X = Conv2D(filters=64, kernel_size=(3, 3), activation='relu', padding='same')(X)
X = MaxPooling2D(pool_size=(2, 2))(X)

X = Conv2D(filters=128, kernel_size=(3, 3), activation='relu', padding='same')(X)
X = MaxPooling2D(pool_size=(2, 2))(X)

X = Conv2D(filters=128, kernel_size=(3, 3), activation='relu', padding='same')(X)
X = MaxPooling2D(pool_size=(2, 2))(X)

# Flattening
X = Flatten()(X)

# Fully connected layers
X = Dense(units=64, activation='relu')(X)
output_tensor = Dense(units=1, activation='sigmoid', name="cell_classes")(X)

# Create model
model = Model(inputs=input_tensor, outputs=output_tensor)
```

Figure 4.8: The code used for creating CNN architecture

2. MobileNet:

The second model I have chosen is a pre-trained MobileNet model, which is a powerful convolutional neural network architecture originally trained on the ImageNet dataset. This

model has learned to extract a diverse set of features from images, making it highly effective for various computer vision tasks. By initializing it with 'imagenet' weights and setting 'include_top=False', it ensures that only the convolutional layers are imported, omitting the dense classification layers specific to ImageNet's 1000 classes.

To tailor this MobileNet for the binary classification of Malaria dataset, I freeze all but the last four layers using a simple loop in Python, as seen in Figure 4.9. Freezing these layers prevents their weights from being updated during training, preserving the learned features. This approach is crucial because it allows to exploit the high-level representations learned by MobileNet while customizing the final layers to suit my unique classification requirements.

On top of the frozen MobileNet base, I construct a new model using Keras's 'Sequential' API. I add a 'GlobalAveragePooling2D()' layer to condense the output of the MobileNet into a flat feature vector, effectively summarizing the spatial information. This is followed by 'Dropout' layers to prevent overfitting by randomly deactivating some neurons during training. Then, I incorporate 'Dense' layers with ReLU activation to further distill and learn intricate patterns from the pooled features. Finally, a sigmoid-activated dense layer ensures that the model outputs a probability score for binary classification.

This setup not only maximizes the use of pre-trained knowledge in MobileNet but also fine-tunes the model to excel in my specific task, achieving both efficiency and effectiveness in image classification.

```

# Load MobileNet model without top layers and pre-trained weights
base_mobilenet_model = MobileNet(weights='imagenet', include_top=False, input_shape=input_shape)

# Freeze initial layers
for layer in base_mobilenet_model.layers[:-4]: # Freeze all layers except the last 4
    layer.trainable = False

# Create new model on top of the base MobileNet
model = Sequential()
model.add(base_mobilenet_model)
model.add(GlobalAveragePooling2D()) # Pool features to reduce dimensionality
model.add(Dropout(0.5)) # Dropout for regularization
model.add(Dense(512, activation='relu')) # Additional dense layer for feature learning
model.add(Dropout(0.5)) # More dropout
model.add(Dense(1, activation='sigmoid')) # Output layer for binary classification

# Summary of the model
model.summary()

```

Figure 4.9: The code used for creating MobileNet architecture

3. GoogLeNet

The third model I have chosen is GoogLeNet whose architecture has been described in Chapter 3 but below will be the implementation used for Malaria dataset. Although, this is a pretrained model, in this case, we are training from the very start.

First, it is defined the ‘inception_module’ function (see Figure 4.10). This function creates an Inception module, which is a core component of the GoogLeNet architecture. The Inception module performs multiple convolutions with different kernel sizes (1x1, 3x3, and 5x5) in parallel, followed by a max pooling operation. The outputs of these convolutions and the pooling operation are then concatenated along the depth dimension. This design allows the network to capture features at various scales and is inspired by the inception architecture described in the GoogLeNet paper [17].

Then, the ‘create_googlenet’ function constructs the overall architecture of the network (see Figure 4.11). It begins with an input layer and processes the input through several stages:

1. **Stage 1:** A 7x7 convolutional layer with 64 filters, followed by a max pooling layer and batch normalization. This stage aims to reduce the spatial dimensions of the input while preserving important features.
2. **Stage 2:** This stage consists of a 1x1 convolutional layer followed by a 3x3 convolutional layer with 192 filters, batch normalization, and max pooling. The 1x1 convolution helps to reduce the dimensionality before the 3x3 convolution, similar to the approach used in the original GoogLeNet architecture.
3. **Stage 3:** Two Inception modules are applied, capturing multi-scale features. This is followed by a max pooling layer.
4. **Stage 4:** Five Inception modules are used in this stage. Each module captures a variety of features using different filter sizes. After the last module in this stage, a max pooling layer further reduces the spatial dimensions.
5. **Stage 5:** Two more Inception modules are applied, followed by global average pooling. Global average pooling reduces each feature map to a single value by taking the average of all the elements in the feature map, which helps to prevent overfitting.

In the final stage, dropout is applied to reduce overfitting, followed by a dense layer with a sigmoid activation function, which produces the binary classification output.

```
# Define the inception module
def inception_module(x, filters):
    # 1x1
    path1 = Conv2D(filters=filters[0], kernel_size=(1,1), strides=1, padding='same', activation='relu')(x)

    # 1x1 -> 3x3
    path2 = Conv2D(filters=filters[1][0], kernel_size=(1,1), strides=1, padding='same', activation='relu')(x)
    path2 = Conv2D(filters=filters[1][1], kernel_size=(3,3), strides=1, padding='same', activation='relu')(path2)

    # 1x1 -> 5x5
    path3 = Conv2D(filters=filters[2][0], kernel_size=(1,1), strides=1, padding='same', activation='relu')(x)
    path3 = Conv2D(filters=filters[2][1], kernel_size=(5,5), strides=1, padding='same', activation='relu')(path3)

    # 3x3 -> 1x1
    path4 = MaxPooling2D(pool_size=(3,3), strides=1, padding='same')(x)
    path4 = Conv2D(filters=filters[3], kernel_size=(1,1), strides=1, padding='same', activation='relu')(path4)

    return Concatenate(axis=-1)([path1, path2, path3, path4])
```

Figure 4.10: The code used for creating inception_module of GoogLeNet architecture

```

# Define the GoogLeNet architecture
def create_googlenet(input_shape, num_classes):
    input_layer = Input(shape=input_shape)

    # Stage 1
    x = Conv2D(filters=64, kernel_size=(7,7), strides=2, padding='same', activation='relu')(input_layer)
    x = MaxPooling2D(pool_size=(3,3), strides=2, padding='same')(x)
    x = BatchNormalization()(x)

    # Stage 2
    x = Conv2D(filters=64, kernel_size=(1,1), strides=1, padding='same', activation='relu')(x)
    x = Conv2D(filters=192, kernel_size=(3,3), strides=1, padding='same', activation='relu')(x)
    x = BatchNormalization()(x)
    x = MaxPooling2D(pool_size=(3,3), strides=2, padding='same')(x)

    # Stage 3
    x = inception_module(x, [64, (96, 128), (16, 32), 32])
    x = inception_module(x, [128, (128, 192), (32, 96), 64])
    x = MaxPooling2D(pool_size=(3,3), strides=2, padding='same')(x)

    # Stage 4
    x = inception_module(x, [192, (96, 208), (16, 48), 64])
    x = inception_module(x, [160, (112, 224), (24, 64), 64])
    x = inception_module(x, [128, (128, 256), (24, 64), 64])
    x = inception_module(x, [112, (144, 288), (32, 64), 64])
    x = inception_module(x, [256, (160, 320), (32, 128), 128])
    x = MaxPooling2D(pool_size=(3,3), strides=2, padding='same')(x)

    # Stage 5
    x = inception_module(x, [256, (160, 320), (32, 128), 128])
    x = inception_module(x, [384, (192, 384), (48, 128), 128])
    x = GlobalAveragePooling2D()(x)

    # Stage 6
    x = Dropout(0.5)(x)
    output = Dense(1, activation='sigmoid')(x)

    model = Model(inputs=input_layer, outputs=output)
    return model

```

Figure 4.11: The code used for creating GoogLeNet module

4.6 Compilation of the models

In Figure 4.12, the code snippet illustrates the crucial steps involved in preparing each of our models for training in TensorFlow/Keras. The components seen are:

1. Optimizer:

[22] The Adam optimizer is initialized here using ‘tf.keras.optimizers.Adam()’. Adam stands for Adaptive Moment Estimation and is an optimization algorithm known for its effectiveness in training deep neural networks. It dynamically adjusts the learning rate throughout training based on the gradient's first and second moments, which helps in efficiently navigating the complex landscapes of loss functions. By default, Adam's

parameters are set to reasonable values, making it a popular choice for various deep learning tasks.

2. Loss Function:

[23] The chosen loss function for this model is ‘BinaryCrossentropy’, specified by ‘tf.keras.losses.BinaryCrossentropy()’. This particular loss function is tailored for binary classification problems, where the model predicts probabilities for two classes (typically 0 and 1). Binary cross-entropy quantifies the difference between the predicted probability distribution and the actual distribution (one-hot encoded), providing a measure of how well the model's predictions match the true labels. It is particularly suited for tasks where the goal is to minimize the discrepancy between predicted and actual outcomes, guiding the training process towards optimal model performance.

3. Compiling the model:

After defining the optimizer and loss function, `model.compile()` combines these components to configure the model for training. By specifying ‘metrics=["accuracy"]’, the ‘`model.compile()`’ function ensures that during training, the model's accuracy on both the training and validation data will be computed and displayed, allowing for ongoing assessment and refinement of the model's predictive capabilities. This metric serves as a benchmark to gauge how well the model is learning to distinguish between the two classes in the binary classification problem at hand.

```
optim = tf.keras.optimizers.Adam()
loss = tf.keras.losses.BinaryCrossentropy()

model.compile(optimizer=optim, loss=loss, metrics=["accuracy"])
```

Figure 4.12: The code used for compiling the models

4.7 Training the models

In the realm of deep learning with TensorFlow/Keras, the `'model.fit()'` function is fundamental for training neural network models. In Figure 4.13, `'epochs = 10'` specifies the number of complete passes through the entire training dataset that the model will undergo during training. Each epoch consists of forward and backward passes through the network, where weights are adjusted based on the error calculated by the loss function. The parameters within `'model_fit()'` are as follows:

1. Epochs:

This parameter (`'epochs=10'`) defines the number of times the model will iterate over the training dataset. It controls the duration and intensity of the training process, influencing how thoroughly the model learns from the data. The choice of epochs balances between underfitting (too few epochs) and overfitting (too many epochs), ensuring the model achieves optimal generalization on unseen data. [24]

2. Training Data:

This parameter specifies the dataset used for training the model. It typically consists of input features and corresponding target labels used to update the model's weights during each epoch.

3. Validation Data:

This optional parameter provides another dataset (separate from the training set) used to evaluate the model's performance after each epoch. It helps monitor overfitting and ensures the model generalizes well to new data.

This setup is standard across most neural network models and serves as a foundational step in deep learning workflows. By specifying `epochs`, `model.fit()` controls the depth of learning and optimization, ensuring the model's parameters are tuned to minimize error effectively over multiple iterations. This iterative process is essential for achieving robust and accurate predictions in various machine learning tasks.

```
epochs = 10  
|  
history = model.fit(training_data, epochs=epochs, validation_data=validation_data)
```

Figure 4.13: The code used for training the models

4.8 Plotting the graphs for training and validation of the models

After training the models using TensorFlow/Keras, it is customary to assess their performance by plotting graphs that visualize the training and validation accuracy, as well as the training and validation loss. The plot will display two curves: one representing the training accuracy or loss and the other the validation accuracy or loss, each plotted against the number of epochs. By comparing these curves, one can gauge how well the model generalizes to unseen data (validation loss) and how effectively it minimizes error during training (training loss). This graphical representation is crucial for diagnosing overfitting or underfitting issues and for making informed decisions regarding further training iterations or model adjustments.

```
plt.figure(figsize=(7, 4))
ax = plt.axes()
ax.plot(range(1, epochs+1), history.history["loss"], marker="o", label="Training loss")
ax.plot(range(1, epochs+1), history.history["val_loss"], marker="*", ls="--", label="Validation loss")
ax.legend()
plt.show()

plt.figure(figsize=(7, 4))
ax = plt.axes()
ax.plot(range(1, epochs+1), history.history["accuracy"], marker="o", label="Training accuracy")
ax.plot(range(1, epochs+1), history.history["val_accuracy"], marker="*", ls="--", label="Validation accuracy")
ax.legend()
plt.show()
```

Figure 4.14: The code used for plotting accuracy and/or loss representation for training and validation dataset

4.9 Testing the models

The testing phase using ‘model.evaluate(test_data)’ serves as a critical step in assessing the performance of a trained machine learning model. This function evaluates the model's performance on a designated test dataset by computing predefined metrics such as loss and accuracy, which were specified during the model compilation phase. Specifically, ‘model.evaluate()’ calculates and returns the model's loss value and any additional metrics defined in the model's configuration. This quantitative assessment provides valuable insights into how well the model generalizes to unseen data, helping to gauge its overall effectiveness and informing decisions regarding model deployment or further refinement. By comparing the evaluation results across different models or iterations, practitioners can systematically optimize and improve the model's predictive capabilities for real-world applications.

4.10 Confusion Matrix and Classification Report

[11] When we talk about evaluating machine learning models, the confusion matrix and classification report are essential tools for understanding the model's performance on a test dataset, as depicted in Figure 4.15. The confusion matrix provides a detailed tabular representation of actual versus predicted class labels, showing counts of true positives,

true negatives, false positives, and false negatives. It helps identify where the model excels in making correct predictions and where it struggles with misclassifications. On the other hand, the classification report presents a summary of key metrics—precision, recall, F1-score, and support—for each class in a structured format. Together, these tools offer comprehensive insights into the model's strengths and weaknesses, aiding in informed decision-making for model optimization and deployment strategies in real-world applications.

```
# Get predictions on the test data
predictions = model.predict(test_data)
predicted_classes = np.where(predictions > 0.5, 1, 0).flatten()

# Get true classes
true_classes = test_data.classes

# Create confusion matrix
conf_matrix = confusion_matrix(true_classes, predicted_classes)

# Print confusion matrix
print("Confusion matrix:\n", conf_matrix)

# Plot confusion matrix
plt.figure(figsize=(10, 7))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=test_data.class_indices, yticklabels=test_data.class_indices)

for i, value in enumerate(conf_matrix[1]):
    plt.text(i + 0.5, 1.5, str(value), ha="center", va="center", color="black")

plt.xlabel('Predicted Labels', fontsize=14) # Increase label font size for better visibility
plt.ylabel('True Labels', fontsize=14)
plt.title('Confusion Matrix', fontsize=16)

# Ensure tight layout to prevent labels from overlapping
plt.tight_layout()
plt.show()

# Print classification report
print(classification_report(true_classes, predicted_classes, target_names=test_data.class_indices.keys()))
```

Figure 4.15: The code used for evaluating the confusion matrix and classification report

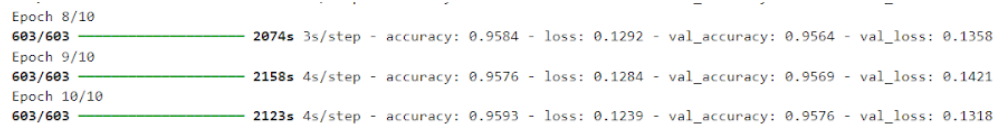
CHAPTER 5

RESULTS, COMPARISON AND CONCLUSION

5.1 Result of the models

Based on the procedural steps outlined in Chapter 4, the forthcoming figures will showcase the final epochs during the training phase for all models using augmented images or only preprocessed ones. These figures will encapsulate each model's convergence over epochs and highlight their respective performance metrics, crucial for assessing efficacy. Additionally, the evaluation results of all models on the test dataset will be illustrated, emphasizing accuracy and other metrics essential for real-world applicability. Subsequent figures will present visual representations of training and validation accuracy and loss curves, providing insights into each model's learning dynamics and convergence patterns. Furthermore, there will also be displayed the confusion matrix and classification report, respectively, offering a detailed breakdown of predictive performance across different classes. Together, these figures offer a comprehensive view of each model's training progression, evaluation outcomes, and performance metrics crucial for validation and interpretation in classification tasks.

1. Custom CNN – Preprocessed and Augmented



Epoch 8/10				
603/603	2074s	3s/step	- accuracy: 0.9584 - loss: 0.1292	- val_accuracy: 0.9564 - val_loss: 0.1358
Epoch 9/10				
603/603	2158s	4s/step	- accuracy: 0.9576 - loss: 0.1284	- val_accuracy: 0.9569 - val_loss: 0.1421
Epoch 10/10				
603/603	2123s	4s/step	- accuracy: 0.9593 - loss: 0.1239	- val_accuracy: 0.9576 - val_loss: 0.1318

Figure 5.1: Last three epochs of the augmented custom CNN model

Figure 5.1 shows the last three epochs of a custom CNN model. The graph shows the training and validation accuracy increasing with each epoch, while the training and validation loss decreases. This indicates that the model is learning and improving its performance on both the training and validation sets.

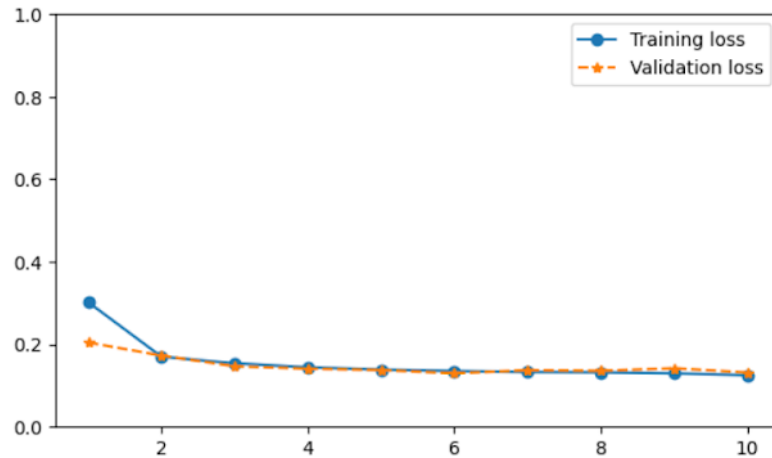


Figure 5.2: Graphical representation of loss function for augmented custom CNN model

Figure 5.2 shows the graphical representation of the loss function of the CNN model. The graph shows that the training loss and validation loss decrease over time, indicating that the model is learning and improving its performance. The validation loss is slightly higher than the training loss, which is expected as the model is being evaluated on unseen data.

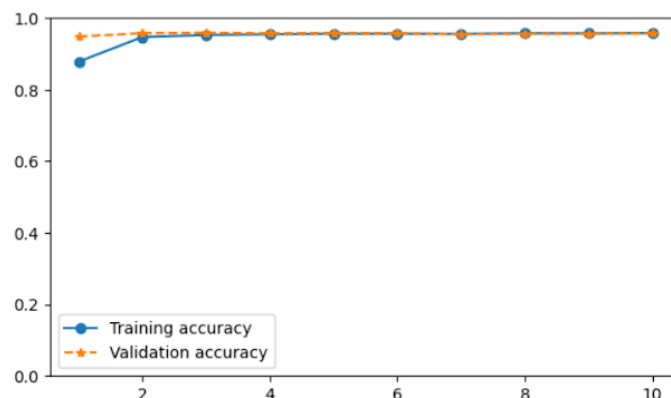


Figure 5.3: Graphical representation of accuracy function for augmented custom CNN model

Figure 5.3 shows the graphical representation of the accuracy function of the CNN model. The graph shows that the training accuracy and validation accuracy increase over time, indicating that the model is learning and improving its performance. The validation accuracy is slightly lower than the training accuracy, which is expected as the model is being evaluated on unseen data.

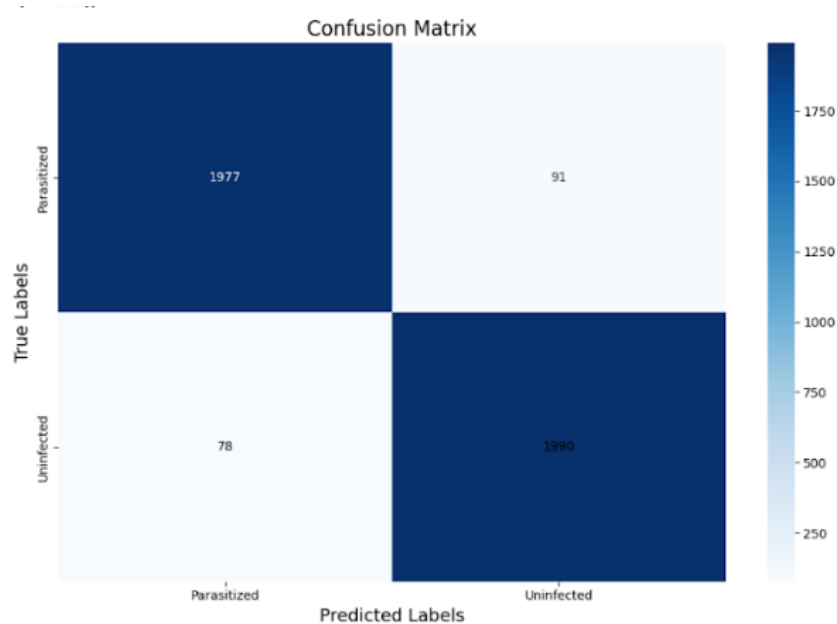


Figure 5.4: Confusion matrix for the ‘test’ dataset of augmented Custom CNN model

Figure 5.4 presents a confusion matrix, a table that summarizes the performance of a classification model. The rows of the matrix correspond to the actual classes, while the columns represent the predicted classes. The values in the matrix indicate the number of instances that fall into each combination of actual and predicted classes. In this case, the model demonstrates high accuracy, with most instances correctly classified along the diagonal of the matrix (true positives and true negatives). However, there are some misclassifications.

	precision	recall	f1-score	support
Parasitized	0.96	0.96	0.96	2068
Uninfected	0.96	0.96	0.96	2068
accuracy			0.96	4136
macro avg	0.96	0.96	0.96	4136
weighted avg	0.96	0.96	0.96	4136

Figure 5.5: Classification report for the ‘test’ dataset of augmented Custom CNN model

Figure 5.5 presents a classification report, a concise summary of various evaluation metrics for a classification model. The report includes precision, recall, and F1-score for each class, as well as overall accuracy and macro averages. Precision measures the accuracy of positive predictions, recall measures the ability to find all positive instances, and the F1-score balances precision and recall. In this context, the model exhibits high precision and recall for both classes, indicating its effectiveness in accurately identifying instances of each class. The overall accuracy further confirms the model's strong performance on the test dataset.

2. Custom CNN - Only preprocessed

```

Epoch 8/10
603/603 ————— 1994s 3s/step - accuracy: 0.9825 - loss: 0.0512 - val_accuracy: 0.9511 - val_loss: 0.1807
Epoch 9/10
603/603 ————— 2062s 3s/step - accuracy: 0.9856 - loss: 0.0395 - val_accuracy: 0.9514 - val_loss: 0.2360
Epoch 10/10
603/603 ————— 2037s 3s/step - accuracy: 0.9889 - loss: 0.0333 - val_accuracy: 0.9514 - val_loss: 0.2931

```

Figure 5.6: Last three epochs of the only preprocessed custom CNN model

Figure 5.6 illustrates the last three epochs of a preprocessed custom CNN model's training process. The graph displays the training accuracy increasing with each epoch, while the training loss decreases. However, the validation accuracy and validation loss fluctuate, indicating the model might be overfitting the training data.

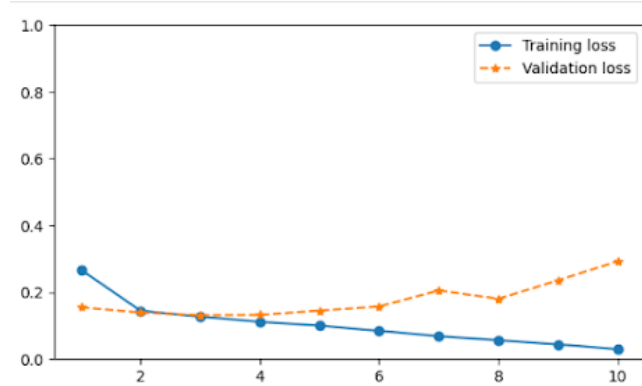


Figure 5.7: Graphical representation of loss function for the only preprocessed custom CNN model

Figure 5.7 presents a graphical representation of the loss function for the preprocessed custom CNN model. The graph demonstrates a decrease in training loss over time, signifying the model's learning. However, the validation loss fluctuates, suggesting the model might be overfitting the training data.

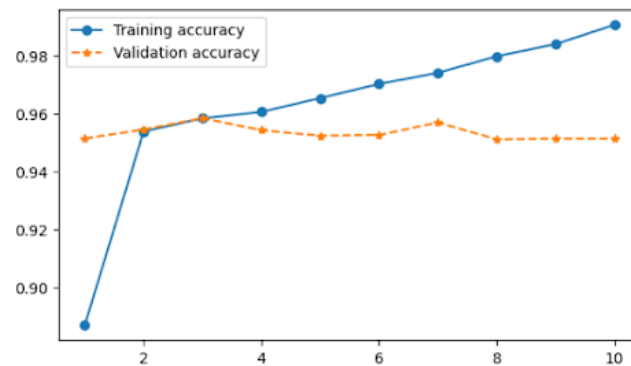


Figure 5.8: Graphical representation of accuracy function for the only preprocessed custom CNN model

Figure 5.8 illustrates the accuracy function for the preprocessed custom CNN model. The graph showcases an increase in training accuracy over time, indicating the model's

learning. However, the validation accuracy fluctuates, suggesting the model might be overfitting the training data.

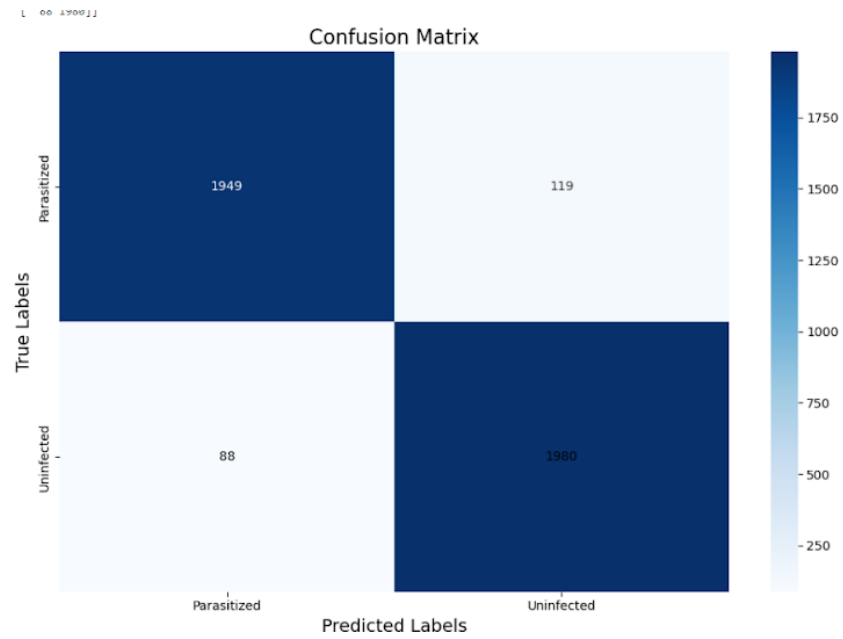


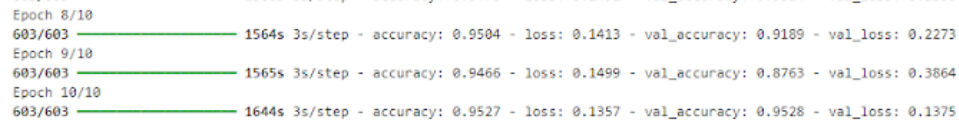
Figure 5.9: Confusion matrix for the ‘test’ dataset of the only preprocessed custom CNN model

	precision	recall	f1-score	support
Parasitized	0.96	0.94	0.95	2068
Uninfected	0.94	0.96	0.95	2068
accuracy			0.95	4136
macro avg	0.95	0.95	0.95	4136
weighted avg	0.95	0.95	0.95	4136

Figure 5.10: Classification report for the ‘test’ dataset of the only preprocessed custom CNN model

Figure 5.10 presents a classification report for the 'test' dataset of the preprocessed custom CNN model, showing an overall accuracy of 95% and relatively high precision, recall, and F1-score for both classes.

3. MobileNet – Preprocessed and Augmented



Epoch	Steps	Time/Step	Accuracy	Loss	Val Accuracy	Val Loss
Epoch 8/10	603/603	1564s	0.9504	0.1413	0.9189	0.2273
Epoch 9/10	603/603	1565s	0.9466	0.1499	0.8763	0.3864
Epoch 10/10	603/603	1644s	0.9527	0.1357	0.9528	0.1375

Figure 5.11: Last three epochs of the augmented MobileNet model

Figure 5.11 illustrates the last three epochs of an augmented MobileNet model's training process. The graph displays the training and validation accuracy increasing with each epoch, while the training and validation loss decreases. This trend indicates that the model is effectively learning and enhancing its performance on both the training and validation datasets. The validation loss is slightly higher than the training loss, which is expected due to the model's evaluation on unseen data. Also, we can see that the validation loss can sometimes get higher and then lower and then high again, which is a common pattern observed during the training of machine learning models, particularly deep learning models like CNNs and MobileNets. This pattern arises due to the interplay between the model's learning capacity and the complexity of the validation dataset.

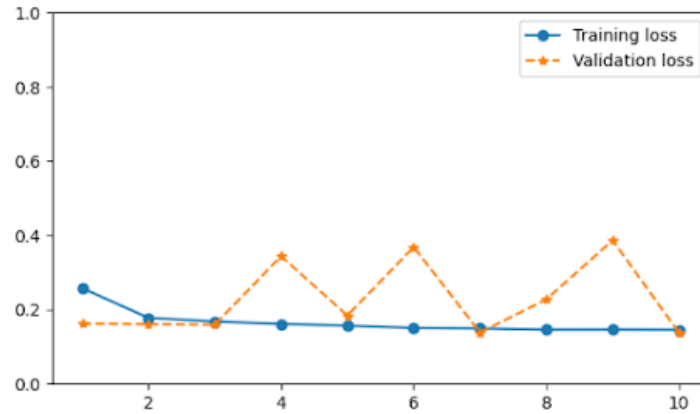


Figure 5.12: Graphical representation of loss function for augmented MobileNet model

Figure 5.12 presents a graphical representation of the loss function for the augmented MobileNet model. The graph demonstrates a consistent decrease in both training and validation loss over time, signifying the model's ongoing learning and improvement. The validation loss is marginally higher than the training loss, which is expected due to the model's evaluation on unseen data.

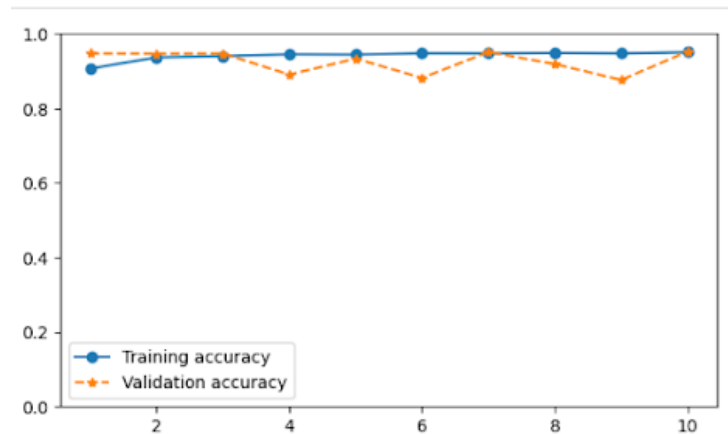


Figure 5.13: Graphical representation of accuracy function for augmented MobileNet model

Figure 5.13 illustrates the accuracy function for the augmented MobileNet model. The graph showcases a continuous increase in both training and validation accuracy over time,

indicating the model's progressive learning and performance enhancement. The validation accuracy is slightly lower than the training accuracy, which is anticipated as the model is assessed on data it hasn't encountered before. It can be seen that the validation accuracy has kind of the same pattern as the validation loss and this is because of the complexity, as explained above.

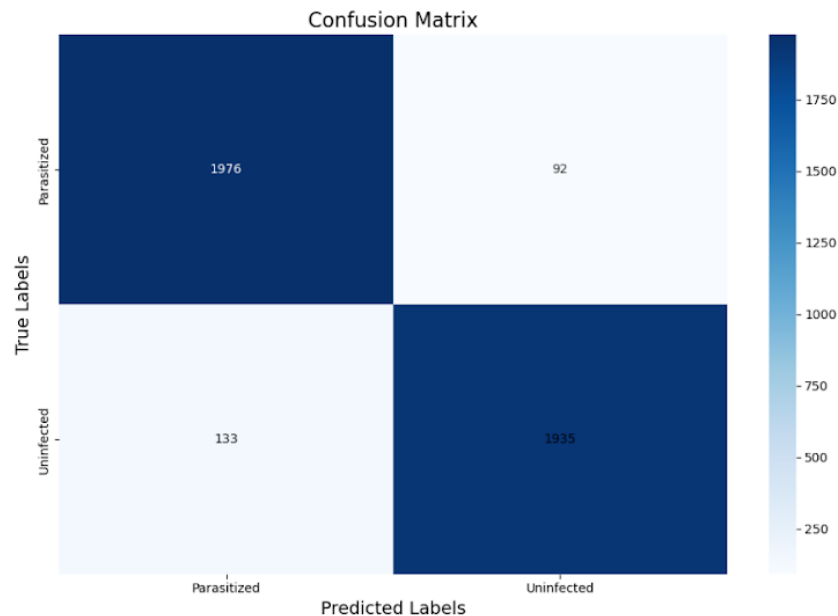


Figure 5.14: Confusion matrix for the ‘test’ dataset of augmented MobileNet model

Figure 5.14 displays a confusion matrix for the ‘test’ dataset of the MobileNet model. The matrix provides a summary of the model's predictions, with the rows representing actual classes and the columns representing predicted classes. The values within the matrix denote the number of instances falling into each combination of actual and predicted classes. The model exhibits high accuracy, as evidenced by the concentration of values along the diagonal (true positives and true negatives). However, some misclassifications are present, indicated by the off-diagonal values (false positives and false negatives).

	precision	recall	f1-score	support
Parasitized	0.94	0.96	0.95	2068
Uninfected	0.95	0.94	0.95	2068
accuracy			0.95	4136
macro avg	0.95	0.95	0.95	4136
weighted avg	0.95	0.95	0.95	4136

Figure 5.15: Classification report for the ‘test’ dataset of augmented MobileNet model

Figure 5.15 presents a classification report for the ‘test’ dataset of the MobileNet model, summarizing its performance with key metrics. The model achieves a high overall accuracy of 0.95, meaning it correctly classifies 95% of the instances in the test set. For both classes, the precision, recall, and F1-score are also notably high, indicating the model's effectiveness in accurately identifying and classifying instances of each class. This strong performance across all metrics underscores the model's robustness and reliability in handling the test dataset.

4. MobileNet - Only preprocessed

```

Epoch 8/10
603/603 ————— 1646s 3s/step - accuracy: 0.9648 - loss: 0.0945 - val_accuracy: 0.9163 - val_loss: 0.2270
Epoch 9/10
603/603 ————— 1651s 3s/step - accuracy: 0.9647 - loss: 0.0956 - val_accuracy: 0.9378 - val_loss: 0.1572
Epoch 10/10
603/603 ————— 1613s 3s/step - accuracy: 0.9695 - loss: 0.0813 - val_accuracy: 0.9550 - val_loss: 0.1576

```

Figure 5.16: Last three epochs of the only preprocessed MobileNet model

Figure 5.16 illustrates the last three epochs of a preprocessed MobileNet model's training process. The graph displays the training accuracy increasing with each epoch, while the training loss decreases.

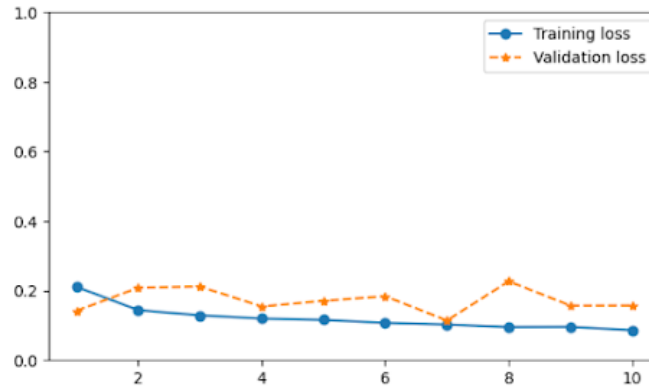


Figure 5.17: Graphical representation of loss function for the only preprocessed MobileNet model

Figure 5.17 presents a graphical representation of the loss function for the preprocessed MobileNet model. The graph demonstrates a decrease in training loss over time, signifying the model's learning. However, the validation loss fluctuates, suggesting the model might be overfitting the training data.

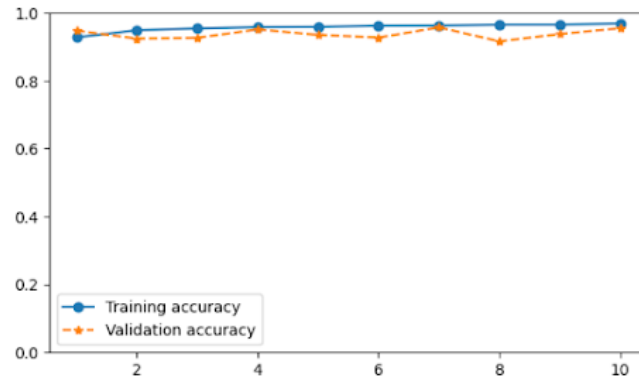


Figure 5.18: Graphical representation of accuracy function for the only preprocessed MobileNet model

Figure 5.18 illustrates the accuracy function for the preprocessed MobileNet model. The graph showcases an increase in training accuracy over time, indicating the model's learning. However, the validation accuracy fluctuates, suggesting the model might be overfitting the training data.

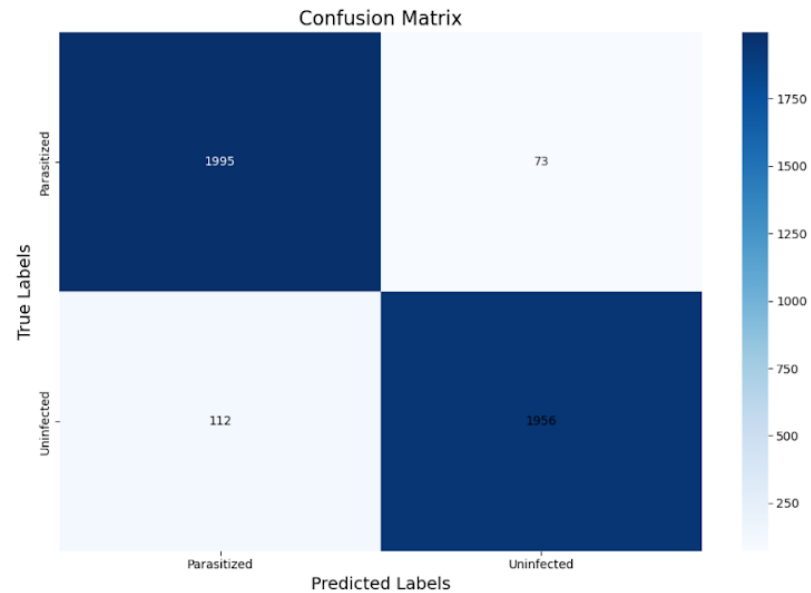


Figure 5.19: Confusion matrix for the ‘test’ dataset of the only preprocessed MobileNet model

	precision	recall	f1-score	support
Parasitized	0.95	0.96	0.96	2068
Uninfected	0.96	0.95	0.95	2068
accuracy			0.96	4136
macro avg	0.96	0.96	0.96	4136
weighted avg	0.96	0.96	0.96	4136

Figure 5.20: Classification report for the ‘test’ dataset of the only preprocessed MobileNet model

Figure 5.20 presents a classification report for the 'test' dataset of the preprocessed MobileNet model, showing an overall accuracy of 96% and relatively high precision, recall, and F1-score for both classes. However, the model's performance is slightly better on the negative class compared to the positive class.

5. GoogLeNet - Preprocessed and Augmented

```
Epoch 8/10
603/603 ————— 2296s 4s/step - accuracy: 0.9585 - loss: 0.1342 - val_accuracy: 0.9589 - val_loss: 0.1485
Epoch 9/10
603/603 ————— 2342s 4s/step - accuracy: 0.9599 - loss: 0.1293 - val_accuracy: 0.9593 - val_loss: 0.1313
Epoch 10/10
603/603 ————— 2362s 4s/step - accuracy: 0.9591 - loss: 0.1317 - val_accuracy: 0.9627 - val_loss: 0.1143
```

Figure 5.21: Last three epochs of the augmented GoogLeNet model

Figure 5.21 shows the last three epochs of the augmented GoogLeNet model, indicating the model is learning and improving its performance.

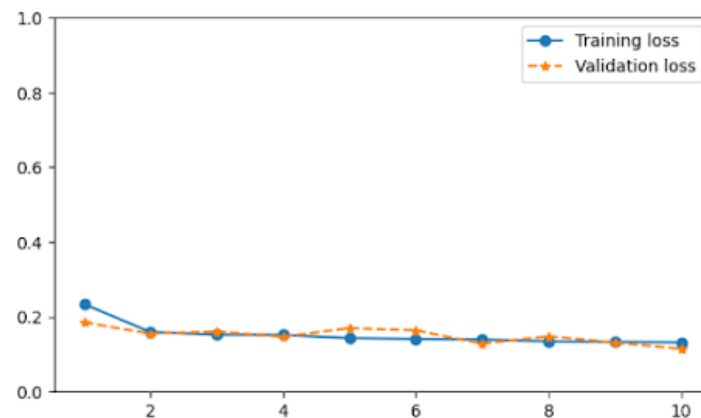


Figure 5.22: Graphical representation of loss function for augmented GoogLeNet model

Figure 5.22 shows the graphical representation of the loss function for the augmented GoogLeNet model, indicating the model's ongoing learning and improvement.

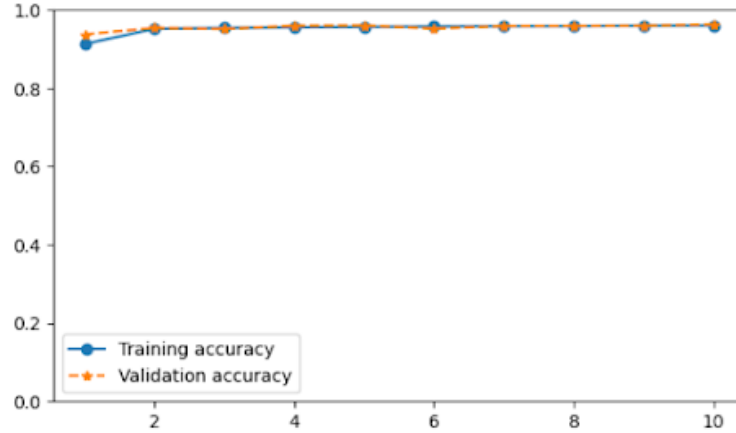


Figure 5.23: Graphical representation of accuracy function for augmented GoogLeNet model

Figure 5.23 shows the graphical representation of the accuracy function for the augmented GoogLeNet model, indicating the model's progressive learning and performance enhancement.

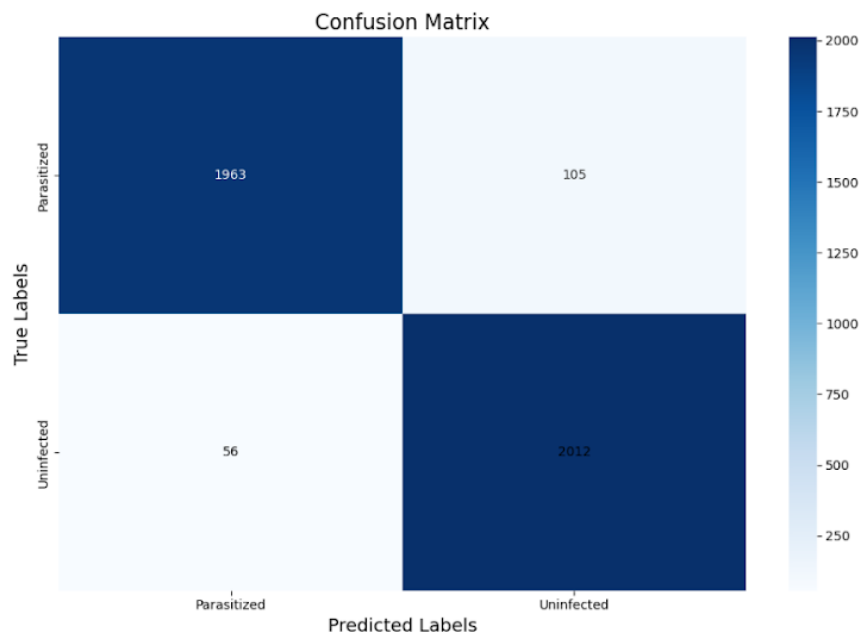


Figure 5.24: Confusion matrix for the 'test' dataset of augmented GoogLeNet model

Figure 5.24 shows the confusion matrix for the 'test' dataset of the GoogLeNet model, summarizing the model's predictions. As the other models, it can be seen that the performance is very well in GoogLeNet too.

	precision	recall	f1-score	support
Parasitized	0.97	0.95	0.96	2068
Uninfected	0.95	0.97	0.96	2068
accuracy			0.96	4136
macro avg	0.96	0.96	0.96	4136
weighted avg	0.96	0.96	0.96	4136

Figure 5.25: Classification report for the 'test' dataset of augmented GoogLeNet model

Figure 5.25 presents a classification report for the 'test' dataset of the GoogLeNet model, showing an overall accuracy of 96% and high precision, recall, and F1-score for both classes.

6. GoogLeNet - Only preprocessed

```
Epoch 8/10
603/603 ————— 2245s 4s/step - accuracy: 0.9667 - loss: 0.1004 - val_accuracy: 0.9613 - val_loss: 0.1108
Epoch 9/10
603/603 ————— 2259s 4s/step - accuracy: 0.9627 - loss: 0.1087 - val_accuracy: 0.9656 - val_loss: 0.1148
Epoch 10/10
603/603 ————— 2283s 4s/step - accuracy: 0.9633 - loss: 0.1050 - val_accuracy: 0.9659 - val_loss: 0.1004
```

Figure 5.26: Last three epochs of the only preprocessed GoogLeNet model

Figure 5.26 shows the last three epochs of the only preprocessed GoogLeNet model, indicating the model is learning and improving its performance.

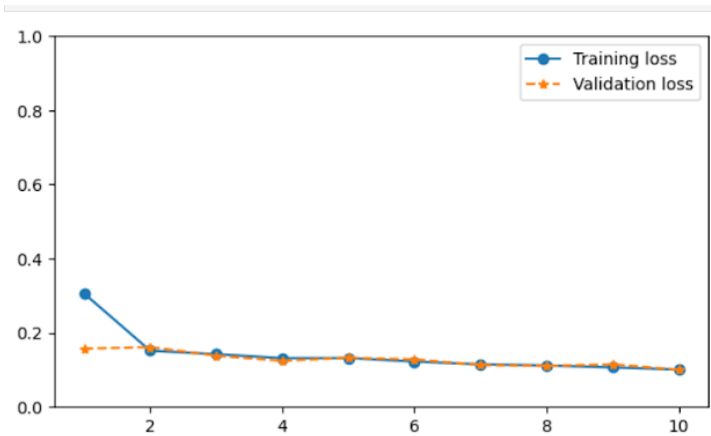


Figure 5.27: Graphical representation of loss function for the only preprocessed GoogLeNet model

Figure 5.27 shows the graphical representation of the loss function for the only preprocessed GoogLeNet model, indicating the model's ongoing learning and improvement

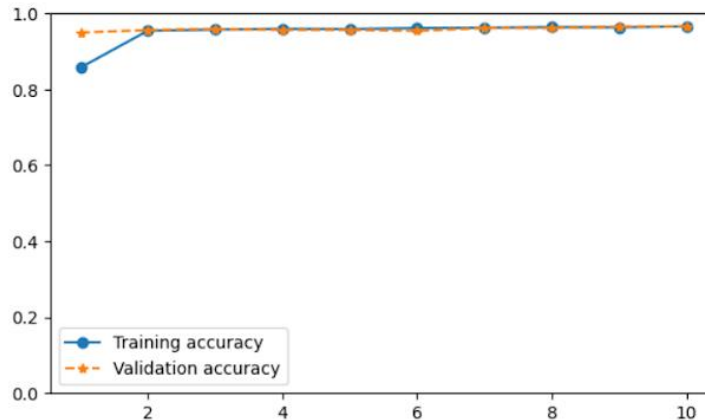


Figure 5.28: Graphical representation of accuracy function for the only preprocessed GoogLeNet model

Figure 5.28 shows the graphical representation of the accuracy function for the only preprocessed GoogLeNet model, indicating the model's progressive learning and performance enhancement.

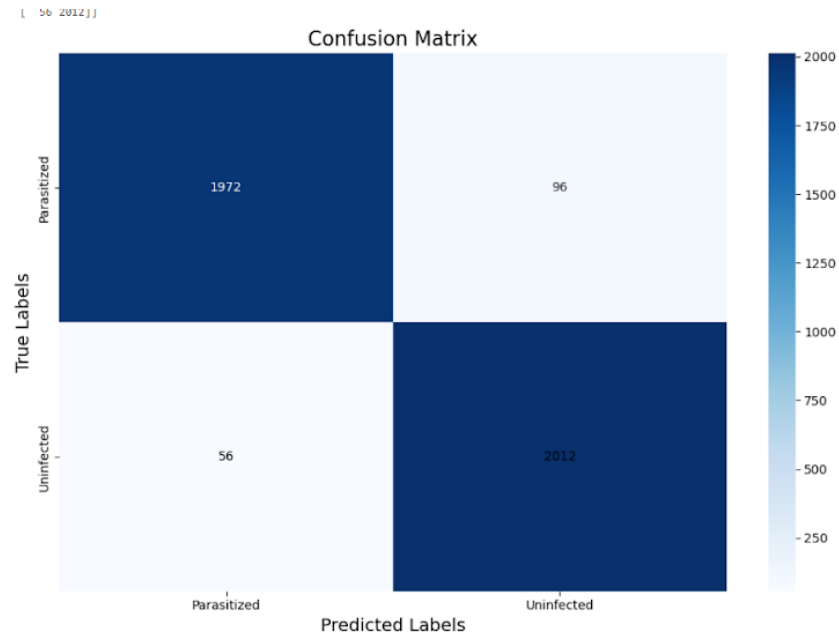


Figure 5.29: Confusion matrix for the ‘test’ dataset of the only preprocessed GoogLeNet model

	precision	recall	f1-score	support
Parasitized	0.97	0.95	0.96	2068
Uninfected	0.95	0.97	0.96	2068
accuracy			0.96	4136
macro avg	0.96	0.96	0.96	4136
weighted avg	0.96	0.96	0.96	4136

Figure 5.30: Classification report for the ‘test’ dataset of the only preprocessed GoogLeNet model

Figure 5.30 presents a classification report for the 'test' dataset of the GoogLeNet model, showing an overall accuracy of 96% and high precision, recall, and F1-score for both classes.

5.2 Comparing the results of the models

Table 5.1 below will provide a comprehensive comparison of the accuracy and loss metrics for the test dataset, evaluated using the `evaluate()` function, across both augmented and only preprocessed datasets for all the models studied in this research. This comparison is essential for assessing the impact of dataset augmentation on the performance of each model in malaria parasite detection. The results will highlight whether the inclusion of augmented data enhances the models' accuracy and reduces loss compared to models trained solely on preprocessed datasets. This analysis aims to provide insights into the effectiveness of dataset augmentation strategies in improving the robustness and reliability of automated malaria screening tools.

	<i>Augmented</i>		<i>Only Preprocessed</i>	
	<i>Accuracy</i>	<i>Loss</i>	<i>Accuracy</i>	<i>Loss</i>
<i>Custom CNN</i>	0.9550	0.1219	0.9438	0.3076
<i>MobileNet</i>	0.9505	0.1306	0.9608	0.1333
<i>GoogLeNet</i>	0.9520	0.1378	0.9594	0.1081

Table 5.1: The results of accuracy and loss for each case

Based on the evaluation results presented in the table, it is evident that each model demonstrates strong performance in malaria parasite detection, albeit with varying degrees of accuracy and loss across augmented and only preprocessed datasets. For the Custom CNN, the augmented dataset yields an accuracy of 0.9550 and a loss of 0.1219, slightly outperforming the accuracy of 0.9438 and loss of 0.3076 achieved with the preprocessed

dataset. This indicates that augmenting the dataset enhances the Custom CNN's ability to generalize and classify malaria parasites accurately.

Similarly, the MobileNet model achieves an accuracy of 0.9505 and a loss of 0.1306 with the augmented dataset, showing robust performance comparable to the accuracy of 0.9608 and significantly lower loss of 0.1333 observed with the preprocessed dataset. Here, the model's performance remains consistently high across both datasets, with a marginal advantage in loss when trained on the preprocessed data.

In contrast, GoogLeNet achieves an accuracy of 0.9520 and a loss of 0.1378 with the augmented dataset, slightly below the accuracy of 0.9594 and notably lower loss of 0.1081 achieved with the preprocessed dataset. This suggests that for GoogLeNet, the preprocessed dataset leads to marginally higher accuracy and significantly lower loss compared to the augmented dataset.

Overall, while each model demonstrates strong performance in detecting malaria parasites, the results indicate that the choice between augmented and preprocessed datasets impacts model performance differently. The Custom CNN benefits slightly more from dataset augmentation, achieving its highest accuracy with the augmented dataset. In contrast, MobileNet shows consistent performance across both datasets, while GoogLeNet achieves marginally better results with the preprocessed dataset in terms of accuracy and significantly lower loss. These findings underscore the importance of dataset preprocessing strategies in optimizing model performance for automated malaria screening applications.

5.3 Conclusion

In conclusion, this study has delved into the potential of deep learning models for automated malaria parasite detection in thin blood smears, a critical step towards enhancing diagnostic capabilities and disease control efforts. By investigating a custom Convolutional Neural Network (CNN) architecture alongside pre-trained models like MobileNet and GoogLeNet, we have explored diverse approaches to tackle this challenging task. The results obtained from rigorous training and evaluation on a diverse dataset of thin blood smear images have shed light on the strengths and weaknesses of each model.

The custom CNN and GoogLeNet models demonstrated remarkable accuracy in malaria parasite detection, achieving comparable performance levels. This highlights the effectiveness of both tailored architectures and pre-trained models in capturing intricate features and patterns associated with malaria parasites. In contrast, MobileNet, while efficient, exhibited variability in its performance during validation epochs, suggesting potential limitations in its generalization capabilities compared to the other models.

This research contributes significantly to the ongoing efforts in leveraging deep learning for automated malaria screening. The findings underscore the importance of selecting appropriate model architectures based on specific operational requirements and constraints. While the custom CNN and GoogLeNet models offer high accuracy, MobileNet's efficiency might be advantageous in resource-limited settings. Further research could explore ensemble techniques or hybrid models that combine the strengths of different architectures to achieve even higher accuracy and robustness. Additionally, investigating the interpretability of these models could enhance their acceptance and adoption in clinical practice.

REFERENCES

- [1] Cox, F. E. (2010). History of the discovery of the malaria parasites and their vectors. *Parasites & vectors*, 3(1), 5.
- [2] World Health Organization. (2021). World malaria report 2021.
- [3] White, N. J. (2004). Antimalarial drug resistance. *Journal of Clinical Investigation*, 113(8), 1084-1092.
- [4] Tangpukdee, N., Duangdee, C., Wilairatana, P., & Krudsood, S. (2009). Malaria diagnosis: A brief review. *The Korean Journal of Parasitology*, 47(2), 93-102.
- [5] Dong, J., Jiang, Z., Tong, Z., Li, H., & Qin, J. (2021). Deep learning for malaria parasite detection: A review. *Journal of Medical Imaging and Health Informatics*, 11(1), 18-27.
- [6] Mishra, S. (2021). Malaria Parasite Detection using Efficient Neural Ensembles. *Journal of Electronics, Electromedical Engineering, and Medical Informatics (JEEEMI)*, 3(3), 119-133.
- [7] Sarkar, S., Sharma, R., & Shah, K. (2020). Malaria detection from RBC images using shallow Convolutional Neural Networks. *arXiv preprint arXiv:2010.11521*.
- [8] Schwarz Schuler, J.P., Romani, S., Puig, D., Rashwan, H., & Abdel-Nasser, M. (2022). An Enhanced Scheme for Reducing the Complexity of Pointwise Convolutions in CNNs for Image Classification Based on Interleaved Grouped Filters without Divisibility Constraints. *Entropy*, 24(9), 1264.

- [9] O'Shea, K., & Nash, R. (2015). An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*.
- [10] Khan, A., Sohail, A., Zahoor, U., & Qureshi, A. S. (2020). Understanding of convolutional neural network (CNN) — A review. *Cognitive Systems Research*, 64, 128-149.
- [11] Rosebrock, A. (2017). *Deep Learning for Computer Vision with Python: Starter Bundle*. PyImageSearch.
- [12] Wu, R., Feng, J., & Li, H. (2019). Pooling layer operation approaches: 1. Pooling layers. For the function of decreasing the computational cost and avoiding overfitting, pooling layers are often added between convolutional layers. *Deep learning for cognitive computing*, 105-116.
- [13] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.
- [14] Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning* (pp. 448-456). PMLR.
- [15] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1), 1929-1958.
- [16] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., ... & Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*.

- [17] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *arXiv*, 2014.
- [18] Abadi, M. et al. (2016). TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, 265-283.
- [19] TensorFlow. (n.d.). TensorFlow Datasets: Malaria. Retrieved June 18, 2024, from <https://www.tensorflow.org/datasets/catalog/malaria>
- [20] Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems* (2nd ed.). O'Reilly Media.
- [21] Shorten, C., & Khoshgoftaar, T. M. (2019). A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1), 60.
- [22] D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *3rd International Conference on Learning Representations, ICLR 2015*, San Diego, CA, USA, 2015.
- [23] S. Kullback and R. A. Leibler, "On Information and Sufficiency," *Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79-86, 1951.
- [24] - Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," *Nature*, vol. 521, no. 7553, pp. 436-444, 2015.