Mini-Project Report On

BugSenseAI
(An insect bite identification mobile application)

Submitted in partial fulfillment of the requirements for the award of the degree of

Bachelor of Technology

in

Computer Science & Engineering

By

Nikhil Zachariah (U2003153)
Niya Bimal (U2003156)
Noel Joseph Paul (U2003158)
Sanjana Nair (U2003187)

Under the guidance of
Ms. Dincy Paul

Department of Computer Science & Engineering
Rajagiri School of Engineering and Technology (Autonomous)
Rajagiri Valley, Kakkanad, Kochi, 682039

July 2023

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
RAJAGIRI SCHOOL OF ENGINEERING AND TECHNOLOGY
(AUTONOMOUS)
RAJAGIRI VALLEY, KAKKANAD, KOCHI, 682039

CERTIFICATE

This is to certify that the mini-project report entitled ***"BugSenseAI (An insect bite identification mobile application)"*** *is a bonafide work done by* ***Mr. Nikhil Zachariah (U2003153), Ms. Niya Bimal (U2003156), Mr. Noel Joseph Paul (U2003158), Ms. Sanjana Nair (U2003187)****, submitted to the APJ Abdul Kalam Technological University in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology (B. Tech.) in Computer Science and Engineering during the academic year 2022-2023.*

Dr. Preetha K. G.	**Ms. Anita John**	**Ms. Dincy Paul**
Head of Department	Mini-Project Coordinator	Mini-Project Guide
Dept. of CSE	Asst. Professor	Asst. Professor
RSET	Dept. of CSE	Dept. of CSE
	RSET	RSET

ACKNOWLEDGEMENTS

We wish to express our sincere gratitude towards **Dr. P. S. Sreejith**, Principal of RSET, and **Dr. Preetha K. G.**, Head of Department of Computer Science and Engineering for providing us with the opportunity to undertake our mini-project, "BugSenseAI".

We are highly indebted to our mini-project coordinators, **Ms. Anita John**, Assistant Professor, Department of Computer Science and Engineering, **Mr. Sajanraj T.D.**, Assistant Professor, Department of Computer Science and Engineering for their valuable support.

It is indeed our pleasure and a moment of satisfaction for us to express our sincere gratitude to our mini-project guide **Ms.Dincy Paul**, for her patience and all the priceless advice and wisdom she has shared with us.

Last but not least, we would like to express our sincere gratitude towards all other teachers and friends for their continuous support and constructive ideas.

Nikhil Zachariah

Niya Bimal

Noel Joseph Paul

Sanjana Nair

ABSTRACT

BugSenseAI is a mobile application that utilizes a machine learning model trained on the ResNet50 architecture to predict the type of insect bite based on user-uploaded images. By integrating image recognition technology, the app accurately identifies the bite and provides remedies accordingly. The objective of this project is to offer a user-friendly tool for individuals to promptly and accurately identify insect bites, enabling them to take appropriate actions and seek medical attention if necessary. BugSenseAI's methodology involves training the model on a diverse dataset of insect bites, extracting relevant features, and classifying the bites into different categories. The application's contribution to society lies in empowering users to identify bites, alleviate discomfort through suggested remedies, and potentially contribute to entomological research and public health insights.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Traditionally, identifying insect bites has relied on personal experience, limited resources like books or online articles, or seeking professional medical assistance. These approaches are often time-consuming, subjective, and may not always yield accurate results. BugSenseAI seeks to overcome these limitations and provide a convenient, reliable, and accessible solution for bite identification.

1.2 Existing System

The existing systems for insect bite identification are limited in their accuracy and accessibility. While some online resources offer information on different types of insect bites, they often lack interactivity and may not provide personalized suggestions for remedies. Medical professionals can accurately identify bites, but the process requires a visit to a healthcare facility, which may not always be feasible or convenient.

1.3 Problem Statement

The lack of a user-friendly and accurate system for insect bite identification poses a challenge for individuals who need immediate information and remedies. This can lead to confusion, delay in appropriate actions, and potentially worsening health conditions. Therefore, there is a need for a reliable and accessible mobile application that can accurately predict the type of insect bite and provide suitable remedies.

BugSenseAI is a mobile application, making it easily accessible to users anytime, anywhere. Users can conveniently capture and upload images of the insect bite, receive

real-time predictions, and access remedies directly from their mobile devices. This eliminates the need for extensive research or seeking professional advice, providing immediate assistance at the users' fingertips

1.4 Objectives

- **Bug bite identification**: Develop a machine learning model integrated into the BugSenseAI app that accurately predicts and identifies the type of insect bite based on uploaded images, enabling users to understand the specific sting they have encountered.
- **Camera extension/Upload option**: Implement a user-friendly interface within the app that allows users to capture images of the insect bite using their device's camera or upload pre-captured images, ensuring ease of use and accessibility.
- **Suggestion of remedies**: Provide users with tailored suggestions for remedies based on the identified insect bite, offering immediate relief and appropriate actions to alleviate discomfort and prevent further complications.
- **Integration of ResNet50 architecture**: Utilize the ResNet50 architecture in training the machine learning model to enhance accuracy and robustness in identifying and classifying different types of insect bites, ensuring reliable predictions for users.
- **Real-time prediction**: Enable real-time prediction of insect bites within the BugSenseAI app, providing users with instant results and reducing the need for time-consuming research or seeking professional medical advice.
- **User-friendly interface**: Design an intuitive and user-friendly interface for the BugSenseAI app, ensuring ease of navigation, clear instructions, and an overall seamless user experience.
- **Accuracy and reliability**: Aim for a high level of accuracy and reliability in bite identification and remedy suggestions, leveraging the power of machine learning and image recognition technology to provide dependable results.

- **Scalability and future enhancements**: Create a foundation for the BugSenseAI project that allows for future scalability and enhancements, such as incorporating additional insect species, expanding the database of remedies, and integrating feedback mechanisms for continuous improvement.

1.5 Scope

- **Mobile Integration**: The BugSenseAI project aims to develop a mobile application that can be seamlessly integrated with existing mobile technology. The app will leverage the capabilities of smartphones, such as camera functionality, to capture images of insect bites and provide real-time predictions and remedies directly on the user's device.

- **Collaboration with Existing Technology**: BugSenseAI can collaborate with existing technologies, such as image recognition APIs and cloud services, to enhance the accuracy and performance of the application. By leveraging these resources, the app can tap into pre-trained models or utilize cloud-based processing power for more efficient bite identification.

- **User-Friendly Interface**: The project focuses on designing an intuitive and user-friendly interface for the BugSenseAI app. This ensures that users can easily navigate through the application, capture and upload bite images, receive predictions, and access suggested remedies without any technical expertise or difficulties.

- **Scalability**: BugSenseAI has the potential for future scalability. The project can be expanded to include a broader range of insect species and bite types. By continuously training the machine learning model on an extensive dataset, the application can improve its accuracy and reliability in bite identification over time.

- **Database Expansion**: The project can be upscaled by incorporating a comprehensive database of remedies for various insect bites. As more information and research become available, the app's remedy suggestions can be expanded to provide users with a wider range of effective treatments.

- **Integration of Feedback Mechanisms**: To enhance the app's performance and

user experience, feedback mechanisms can be integrated into the project. Users can provide feedback on the accuracy of predictions and the effectiveness of suggested remedies, allowing for continuous improvement and refinement of the application.

- **Collaboration with Public Health Institutions**: BugSenseAI can collaborate with public health institutions, entomologists, and medical professionals to gather valuable insights and improve the understanding of insect bites. By sharing anonymized user data, the project can contribute to public health knowledge and assist in the development of targeted prevention strategies and treatment approaches.

Chapter 2

Literature Review

2.1 Existing methods

1. **VGG (Visual Geometry Group):** The VGG network is a popular CNN architecture that consists of multiple layers with small convolutional filters (3x3). It is known for its simplicity and straightforward structure, with a focus on deepening the network by stacking more layers. However, VGG has a higher number of parameters, which leads to increased computational requirements and memory consumption. Additionally, VGG may struggle with capturing fine-grained details in complex images, such as insect bites.

   Disadvantages compared to ResNet:

   -High computational requirements: VGG models have a significantly higher number of parameters compared to other architectures like ResNet. This results in increased computational requirements during training and inference, making VGG models computationally expensive, especially for large-scale datasets.

   -Limited scalability: The depth and number of parameters in VGG models can limit their scalability. Adding more layers to VGG can further increase computational complexity and memory usage, making it challenging to scale up the architecture.

   -Difficulty in capturing fine-grained details: VGG models consist of multiple stacked convolutional layers with small filters (3x3). While this allows for a rich feature hierarchy, VGG may struggle to capture fine-grained details in images, especially when compared to more advanced architectures like ResNet.

2. **Inception:** The Inception network, also known as GoogLeNet, introduced the concept of "Inception modules" that use different filter sizes within the same layer to

capture features at multiple scales. This architecture is computationally efficient due to the use of 1x1 convolutions to reduce dimensionality. However, the complex inception modules may result in increased model complexity and make training more challenging.

Disadvantages compared to ResNet:

-Increased model complexity: Inception models, such as GoogLeNet, introduce complex inception modules with multiple parallel branches and different filter sizes. This complexity can make training and optimization more challenging compared to ResNet, particularly for smaller datasets or when computational resources are limited.

-Higher computational requirements: While Inception models aim to be computationally efficient, their increased complexity can still require more computational resources compared to ResNet. This can impact both training time and inference speed.

-Reduced interpretability: The intricate structure of Inception models can make them less interpretable compared to ResNet. ResNet's use of residual connections provides a more intuitive understanding of feature extraction and allows for better interpretability.

3. **MobileNet:** MobileNet is designed specifically for mobile and embedded applications, aiming for lightweight models with fewer parameters and reduced computational cost. It employs depth-wise separable convolutions to reduce the number of computations while maintaining accuracy. However, this reduction in complexity may come at the cost of slightly lower accuracy compared to larger models like ResNet or VGG.

Disadvantages compared to ResNet:

-Slightly lower accuracy: MobileNet models prioritize lightweight architectures with fewer parameters and reduced computational cost. This design choice can result in slightly lower accuracy compared to larger models like ResNet or VGG. While

Method	Advantages	Drawbacks
ResNet	- Addresses vanishing gradient problem	- Requires more parameters and memory
	- Accurate and deep representations	- Computationally expensive
VGG	- Simplicity and straightforward structure	- High number of parameters
	- Easy to understand and implement	- Struggles with fine-grained details
Inception	- Efficient due to inception modules	- Increased model complexity
	- Multiscale feature extraction	- Challenging training
MobileNet	- Lightweight and suitable for mobile devices	- Slightly lower accuracy compared to larger models
	- Reduced computational cost	

Table 2.1: Comparison of Existing Methods

MobileNet is suitable for mobile and embedded applications, it may not achieve the same level of accuracy as more complex architectures.

-Reduced representation power: The use of depth-wise separable convolutions in MobileNet reduces the computational cost but also limits the model's representation power. This may impact its ability to capture complex and intricate features in images compared to more complex architectures like ResNet.

-Sensitivity to hyperparameters: MobileNet's performance can be more sensitive to hyperparameter choices due to the specific design of the architecture. Careful tuning of hyperparameters is required to achieve optimal accuracy and performance.

In conclusion, the ResNet architecture, while having higher computational requirements, stands out by effectively addressing the vanishing gradient problem and achieving accurate and deep representations. VGG is simpler but struggles with fine-grained details. Inception offers efficient feature extraction with increased complexity, and MobileNet focuses on lightweight models suitable for mobile devices with slightly lower accuracy. The choice of the method depends on the specific requirements, trade-offs, and available resources for the BugSenseAI project.

Chapter 3

System Analysis

3.1 Expected System Requirements

The system of user which is a smart phone is expected to have the following features:

- Android platform with a version above 4.
- In-built camera
- A storage space of approximate 100 MB for the app and additional storage to save image files to be uploaded to app for identification
- A minimum Ram size of 2GB is required in the device.

3.2 Feasibility Analysis

3.2.1 Technical Feasibility

The project is technically feasible since majority of the population is in possession of smartphones. The app only requires minimum requirements to run on a smartphone.

3.2.2 Operational Feasibility

The operations are built in a simple and easy-to-use manner. Installation of the app is the only prerequisite operation to be done.

3.2.3 Economic Feasibility

The BugSenseAI mobile application has strong economic feasibility. It can generate revenue through app purchases, in-app advertisements, and partnerships. The app's accurate bite identification and remedy suggestions reduce unnecessary healthcare visits, resulting

in cost savings. Its convenience and potential for monetization make it an economically viable solution. The development of the app is also zero budget as it was built using free resources.

3.3 Hardware Requirements

The following are the system requirements to develop the Android App.

- Processor: Intel Core i5
- Hard Disk: Minimum 100GB
- RAM: Minimum 8GB

3.4 Software Requirements

The following are the softwares used in the development of the app.

Operating System: Windows or Linux

3.4.1 Android Studio for the Kotlin app development

Android Studio is a popular Integrated Development Environment (IDE) for developing Java, Kotlin and Flutter apps, as it provides a wide range of tools and features that can help you build high-quality apps faster. Some of the key features of Android Studio for App development in Kotlin include:

A rich set of tools for debugging, testing, and profiling your app. A powerful code editor with support for code completion, refactoring, and more A flexible build system with support for building, testing, and deploying your app. Integration with popular version control systems like Git. A visual layout editor for building attractive user interfaces

3.4.2 Google Colab

Google Colab is an open-source online python IDE that allows you to create and share documents that contain live code, equations, visualizations, and narrative text. It is commonly used for data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.

In a Google Colab notebook, you can write and execute code, as well as add text and images to create a rich document that combines code, output, and documentation. You can use colab notebooks for a wide range of tasks, including data visualization, machine learning, and scientific computing.

3.4.3 TFLite

TFLite, short for TensorFlow Lite, is a lightweight and efficient framework developed by Google to deploy machine learning models on resource-constrained devices, such as mobile phones, embedded devices, and IoT devices. It is designed to provide fast inference and minimal memory footprint, making it well-suited for edge computing scenarios.

TFLite offers several key features and optimizations:

1. Model Optimization: TFLite employs various techniques to optimize the size and performance of machine learning models. This includes quantization, which reduces the precision of model weights and activations to 8 bits or lower, resulting in smaller model sizes and faster computations. TFLite also supports model compression techniques like pruning and weight sharing to further reduce the model size.

2. Interpreter: TFLite provides an interpreter that allows the execution of trained machine learning models on target devices. The interpreter optimizes the execution of the model by leveraging hardware-specific acceleration, such as using the GPU or specialized neural network accelerators.

3. Conversion Tools: TFLite includes conversion tools that facilitate the conversion of models trained with popular deep learning frameworks, such as TensorFlow and Keras, into a format compatible with the TFLite interpreter. This conversion process ensures that models can be efficiently deployed and executed using TFLite.

4. Hardware Acceleration: TFLite takes advantage of hardware acceleration when available, optimizing the inference process for specific devices. This includes leveraging GPU compute capabilities or using specialized hardware like Neural Processing Units (NPUs) or Tensor Processing Units (TPUs) for faster and more efficient computations.

5. Platform Support: TFLite provides support for various platforms, including Android, iOS, Linux, and microcontrollers. This wide range of platform support enables developers to deploy TFLite models across different device types and operating systems.

Overall, TFLite offers a comprehensive framework for deploying machine learning

models on resource-constrained devices. Its model optimization techniques, efficient inference engine, hardware acceleration support, and platform compatibility make it an ideal choice for deploying machine learning models at the edge, enabling real-time inference and on-device intelligence.

Chapter 4

Methodology

4.1 Proposed Method

- Develop a mobile application that can recognize insect bites and suggest remedies for it.
- We plan to build a model using Resnet-50 CNN architecture.
- User gives a photo as input and application gives the name of the recognized insect bite as output.
- Application also contains features like remedy suggestions for the identified insect bite.

4.1.1 Image Detection

The ResNet (Residual Network) model architecture is a deep convolutional neural network (CNN) that was introduced to address the vanishing gradient problem encountered in training very deep networks. The core idea behind ResNet is the use of residual connections, also known as skip connections, which allow the network to learn residual mappings

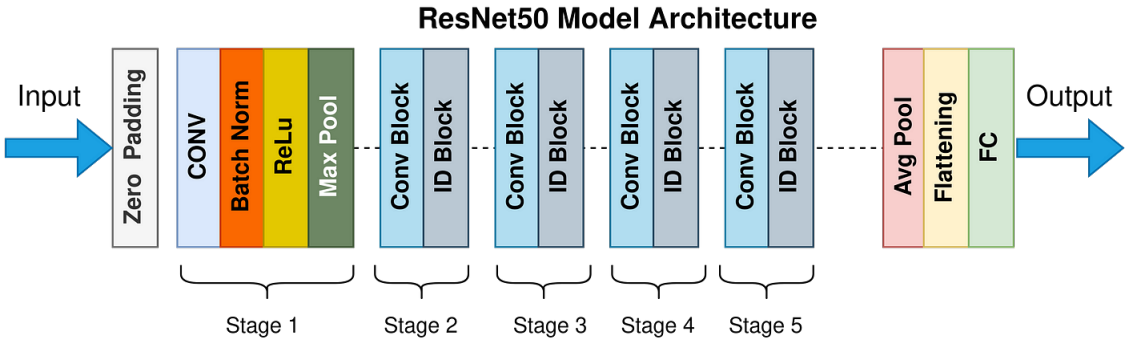

Figure 4.1: model structure overview

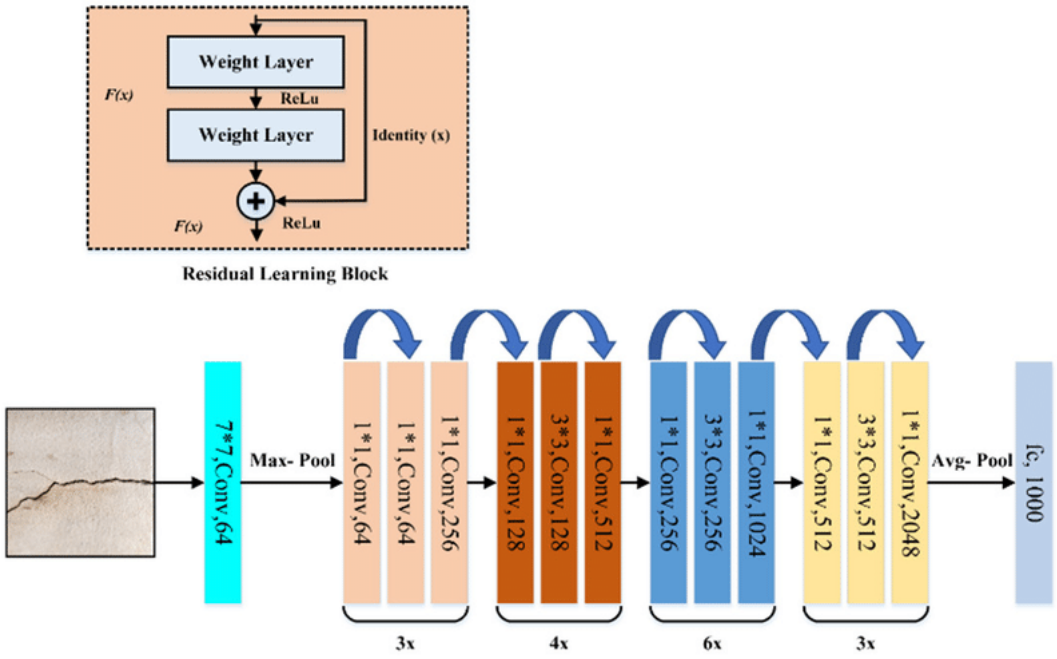

Figure 4.2: model detailed structure

instead of attempting to learn the desired underlying mappings directly.

The ResNet architecture consists of several layers, including convolutional layers, batch normalization layers, activation functions (such as ReLU), and fully connected layers for classification. The key component of ResNet is the residual block, which contains one or more convolutional layers.

Residual connections

The basic building block of a residual block consists of three main steps:

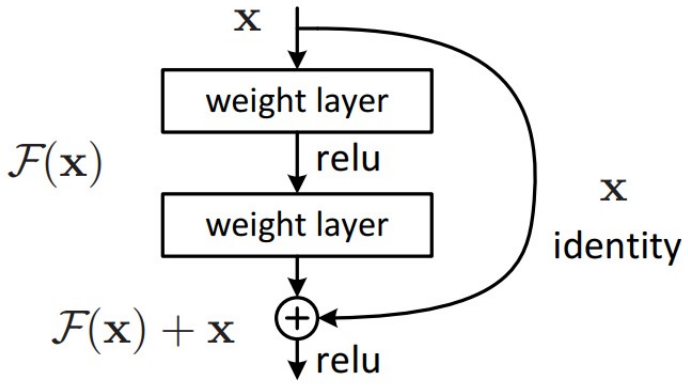

Figure 4.3: Residual learning: a building block

1. Identity Path: The input data is passed through a series of convolutional layers, batch normalization, and activation functions to extract features.

2. Shortcut Connection: A shortcut connection, also called a skip connection, directly connects the input of the residual block to its output. This shortcut connection bypasses the convolutional layers and allows the gradient to flow directly through the block without attenuation.

3. Merge: The output from the identity path and the shortcut connection are added element-wise. This merging operation combines the learned residual mapping from the convolutional layers with the original input, producing the final output of the residual block.

The introduction of residual connections addresses the vanishing gradient problem by allowing the gradients to bypass multiple layers and flow directly to earlier layers. This helps to preserve and propagate gradients effectively during the backpropagation process, making it easier to train very deep networks.

The residual connections also enable the network to learn residual mappings, capturing the difference between the input and the desired output. This residual learning focuses on learning the small, residual changes required to transform the input into the desired output, which is typically easier to optimize.

To predict images, the ResNet model utilizes its deep architecture to extract hierarchical features from the input image. As the image passes through the layers of convolutional filters, it undergoes downsampling, which progressively reduces the spatial dimensions while increasing the number of filters. This allows the model to capture both low-level and high-level features, learning representations that are increasingly abstract and complex.

The learned features are then fed into fully connected layers for classification, where the model assigns probabilities to different classes or outputs the predicted label based on the given task, such as insect bite identification in the case of BugSenseAI.

Overall, the ResNet model's use of residual connections and its deep architecture enables it to effectively learn and extract features from images, overcoming the challenges of training deep networks and achieving high accuracy in image prediction tasks.

Backpropagation and Vanishing Gradient Problem

Backpropagation is a key algorithm used to train neural networks, including the ResNet (Residual Network) architecture. It involves computing gradients with respect to the network's parameters during the training phase.

During backpropagation, the gradients are calculated by propagating the error from the output layer back to the input layer. The gradients indicate the direction and magnitude of adjustments needed to minimize the loss function and improve the network's performance. These gradients are then used to update the weights of the network through an optimization algorithm, such as stochastic gradient descent (SGD).

In the case of ResNet, the vanishing gradient problem is particularly relevant due to the network's depth. ResNet models typically consist of numerous layers (e.g., ResNet-50 has 50 layers), making them very deep neural networks. As the gradients are backpropagated through these layers, they can become extremely small or even vanish, resulting in the vanishing gradient problem.

The vanishing gradient problem in ResNet can occur due to the multiplicative nature of the gradients. As gradients are backpropagated through multiple layers, they are multiplied by the weights of each layer. If these weights are less than 1, the gradients can diminish rapidly as they propagate through each layer, eventually becoming negligible.

Vanishing gradient problem - consequences:

1. Slow Training: When gradients become extremely small, the updates to the network's weights are also small. This slow rate of change hampers the learning process, as it takes longer for the model to converge to an optimal solution.

2. Difficulty in Learning Long-Range Dependencies: Deep networks are designed to capture complex relationships and dependencies in the input data. However, when gradients vanish, the network fails to effectively propagate error signals across long sequences or layers, limiting its ability to learn these long-range dependencies.

3. Limited Depth Utilization: Deep networks with many layers have the potential to learn more intricate representations. However, the vanishing gradient problem restricts the effective utilization of the depth of the network, as the gradients do not reach the initial layers with sufficient magnitude to facilitate significant updates.

Solution using residual connections:

The introduction of residual connections in ResNet helps alleviate the vanishing gradient problem. Residual connections provide shortcuts for the gradients to bypass multiple layers, allowing them to flow directly to earlier layers. By bypassing these layers, the gradients can reach the initial layers without being significantly attenuated. This enables better gradient flow and mitigates the vanishing gradient problem in deep ResNet models.

The residual connections in ResNet provide an identity mapping, where the original input is added to the output of the residual block. This way, the model can learn the residual changes required to transform the input into the desired output, rather than attempting to learn the entire mapping directly. The gradients can easily flow through this identity mapping, which helps combat the vanishing gradient problem and facilitates the training of deep ResNet models.

Chapter 5

System Design

5.1 Architecture Diagram

5.1.1 Overall architecture

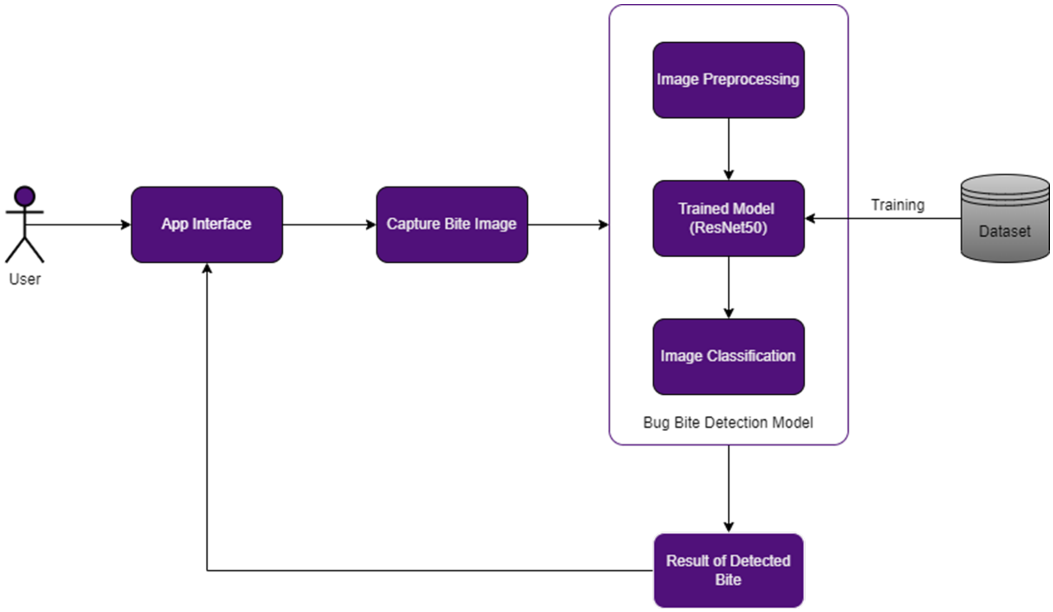

Figure 5.1: Architecture diagram

A user who is affected by an insect bite opens the BugSenseAI application on his/her mobile. The application prompts the user to capture or upload an image of the affected area. The application then forwards the image to the pre-trained model which was trained on a large dataset consisting of more than 4000 images divided into 3 subsets training, validation and testing each of which contained 5 insect classes: Bee, Mosquito, Tick, Spider, and None. The model predicts the bite and returns the results back to the application which displays it on its UI along with remedies.

5.1.2 Mobile architecture

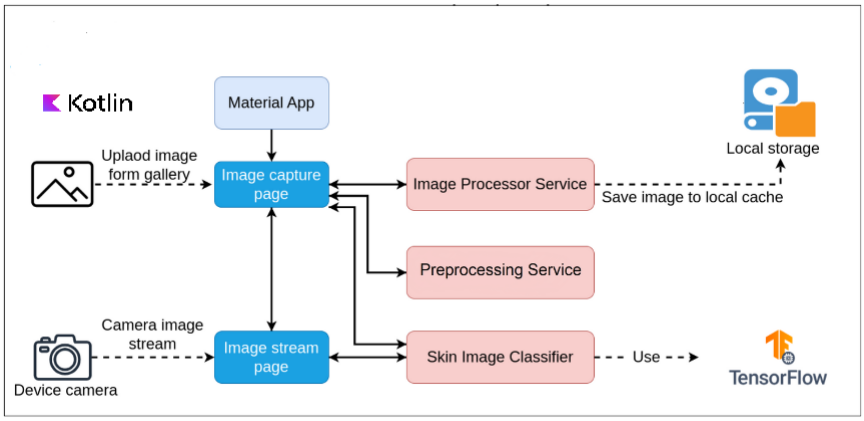

Figure 5.2: Mobile architecture

The user uploads an image on the image capture page of the BugSense AI app whose UI was made using Kotlin. The user can either take a picture through the camera module provided in the app or upload an image from his/her gallery. The image is forwarded to the tensorflow model which returns the result of the predicted insect bite

5.2 Sequence diagram

5.2.1 Overview

Figure 5.3 shows an overview or a high-level view of the application's working. The user captures or uploads an image of the affected skin area to the application's interface. The application forwards the image to the TFLite model which was converted TFLite from a .h5 model version. The image is initially preprocessed using techniques like shearing, zooming etc. The model which was trained on ResNet architecture classifies the image into any one of the 5 classes: Tick, Mosquito, Spider, Bee or None and returns the result to the mobile application. The mobile application takes the result and displays it to the user along with generating remedies for the predicted insect bite.

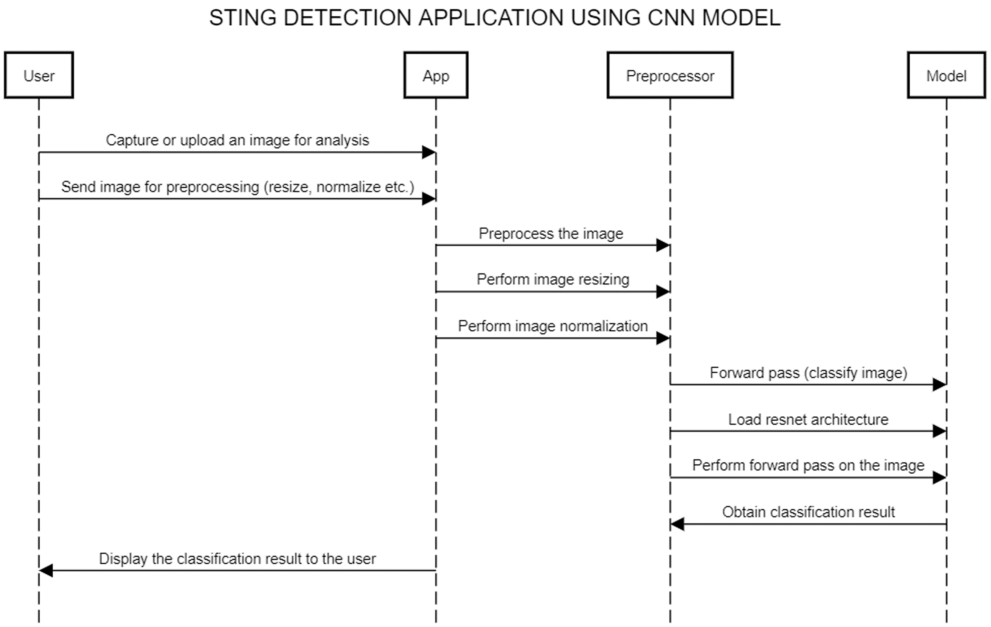

Figure 5.3: Overview

5.2.2 Mobile Application

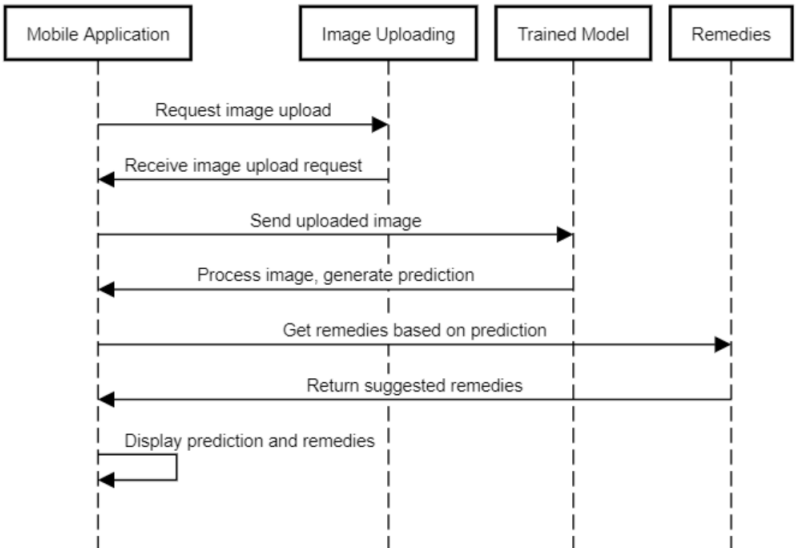

Figure 5.4: Mobile Application

The mobile application which was developed on Kotlin on Android Studio IDE consists of 3 activity pages, the initial splash page followed by an image capture page with a camera module. This page initially asks the user to grant permissions to the application for using camera and accessing files from gallery. The user proceeds to upload or capture an image. The results are displayed on the same page along with a button to proceed to the next

activity ie. remedies page which shows remedies for the predicted insect bite.

5.2.3 Image Uploading

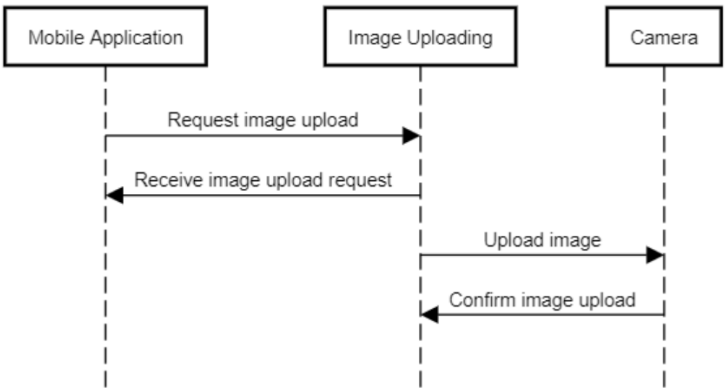

Figure 5.5: Image Uploading

The mobile application requests user to upload an image of the affected area. The app provides users with the option to capture or upload an image using the camera module.

5.2.4 Trained model

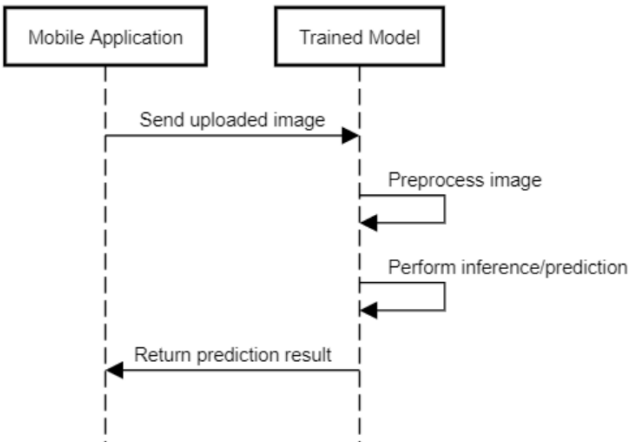

Figure 5.6: Trained Model

The image uploaded is sent to the pre-trained model. The model preprocesses the image and then performs prediction with a training accuracy of 96.46%. The results are sent back to the mobile application.

5.2.5 Remedies

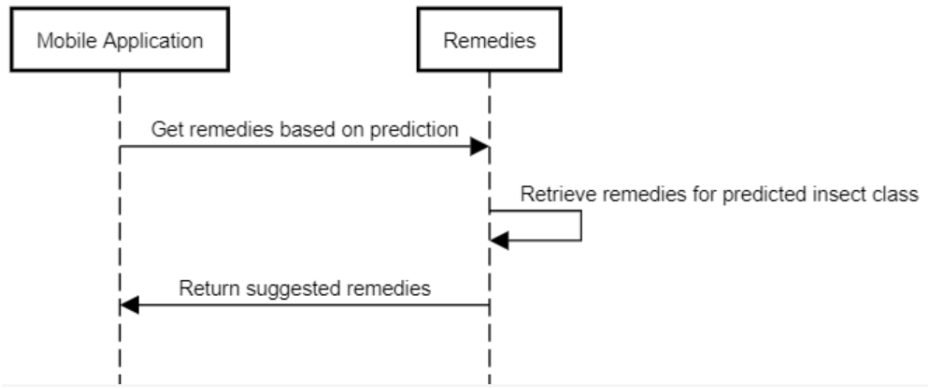

Figure 5.7: Remedies

The predicted results from the model is sent to the mobile application. The mobile application then maps the insect class to its corresponding remedy text file. The file is then processed and its contents are displayed to the user on clicking the remedies button on the app UI. No remedies button would be displayed for 'None' case.

Chapter 6

System Implementation

6.1 Bug Bite Recognition

6.1.1 Model Architecture

The ResNet-50 architecture is a deep convolutional neural network (CNN) that consists of 50 layers, including convolutional layers, pooling layers, and fully connected layers. It is widely used for image classification tasks due to its ability to effectively capture intricate features.

In the case of the BugSenseAI project, ResNet-50 is utilized to train a model capable of classifying images into five insect classes: Tick, Mosquito, Spider, Bee, and None (if the image does not belong to any of the specified classes). To adapt ResNet-50 for this task, five additional dense layers are added at the end of the network.

The ResNet-50 architecture has several notable components:

1. Input Layer: This layer accepts an image as input, typically with dimensions of 224x224 pixels and three color channels (RGB).

2. Convolutional Layers: The initial layers of ResNet-50 consist of convolutional layers that extract low-level features from the input image. These layers apply a set of learnable filters to capture various visual patterns such as edges, corners, and textures.

3. Residual Blocks: ResNet-50 comprises several residual blocks, which are the building blocks of the architecture. Each residual block consists of multiple convolutional layers, batch normalization, and skip connections. These skip connections, also known as residual connections, enable the network to learn residual mappings, allowing for efficient gradient flow during training.

4. Pooling Layers: After the residual blocks, ResNet-50 incorporates pooling layers, typically max pooling, to reduce the spatial dimensions of the feature maps while retaining

the most relevant information. Pooling helps in reducing computational complexity and achieving some degree of translation invariance.

5. Fully Connected Layers: The original ResNet-50 architecture concludes with a fully connected layer that performs the final classification. However, in the BugSenseAI project, five additional dense layers are appended to the ResNet-50 architecture to accommodate the five insect classes. These dense layers consist of multiple neurons and are responsible for learning class-specific representations.

During the training phase, the BugSenseAI model uses a large dataset of labeled images, including samples from the five insect classes. The ResNet-50 architecture, including the appended dense layers, is trained using techniques such as backpropagation and gradient descent optimization. The model learns to identify distinctive features associated with each insect class through an iterative process of adjusting the weights and biases of the network's parameters.

Once the model is trained, it can classify images uploaded by users through the BugSenseAI mobile application. The uploaded image is passed through the trained ResNet-50 model, including the additional dense layers. The model performs forward propagation, extracting high-level features and making predictions based on the learned representations. The output of the model provides the probability distribution of the image belonging to each of the five insect classes. The class with the highest probability is selected as the predicted class for the uploaded image, thereby identifying the type of insect bite detected by the BugSenseAI model.

By incorporating the ResNet-50 architecture with additional dense layers, the BugSenseAI model effectively learns to classify images into the specified insect classes, enabling accurate identification of different types of insect bites for the benefit of the users.

Softmax Activation Function

Activation functions play a crucial role in neural networks, including in the BugSenseAI project. They introduce non-linearity to the network, allowing it to learn complex relationships and make predictions on nonlinear data. One commonly used activation function is softmax.

Softmax is an activation function commonly used in multi-class classification tasks. It takes a vector of real numbers as input and produces a probability distribution over

multiple classes. The output of softmax is a vector of values between 0 and 1, where the values sum up to 1, indicating the probability of the input belonging to each class.

Mathematically, the softmax function for a vector of inputs $z = [z_1, z_2, ..., z_n]$ is defined as follows:

$$softmax(z) = \left[\frac{\exp(z_1)}{\sum_{i=1}^{n} \exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{n} \exp(z_i)}, \ldots, \frac{\exp(z_n)}{\sum_{i=1}^{n} \exp(z_i)} \right] \tag{6.1}$$

The softmax function exponentiates each element of the input vector and divides it by the sum of the exponentiated values. This normalization ensures that the output vector represents a valid probability distribution.

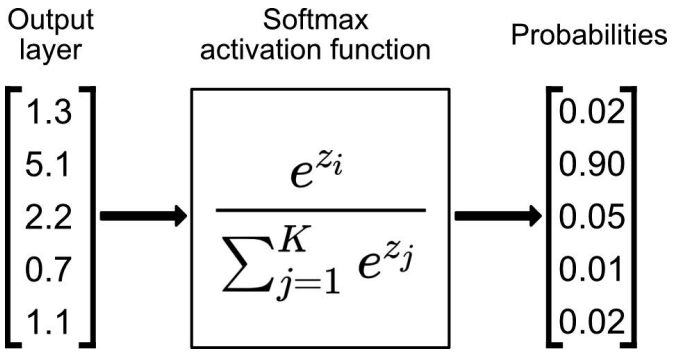

Figure 6.1: Softmax Activation Function-example

In the BugSenseAI project, softmax is used as the activation function in the final layer of the modified ResNet-50 model. This final layer is responsible for producing the class probabilities for the five insect classes: Tick, Mosquito, Spider, Bee, and None.

During the inference phase, when a user uploads an image, the trained ResNet-50 model processes the image through the network's layers and extracts the features. These features are then fed into the final layer, where the softmax activation function is applied. The softmax function converts the raw output of the network into class probabilities.

The class probability vector obtained from the softmax function is then used to determine the predicted class for the uploaded image. The class with the highest probability is selected as the predicted class, indicating the type of insect bite identified by the BugSenseAI model.

By using softmax, the BugSenseAI project ensures that the model's output is a valid probability distribution over the five insect classes, allowing for reliable and interpretable predictions.

ReLU activation function

ResNet-50 architecture and its dense layers in the BugSenseAI project utilize the Rectified Linear Unit (ReLU) activation function.

ReLU is a widely used activation function in deep learning models, including ResNet-50. It introduces non-linearity by setting all negative values to zero and leaving positive values unchanged. This helps the network learn complex relationships and enables faster training compared to other activation functions like sigmoid or tanh.

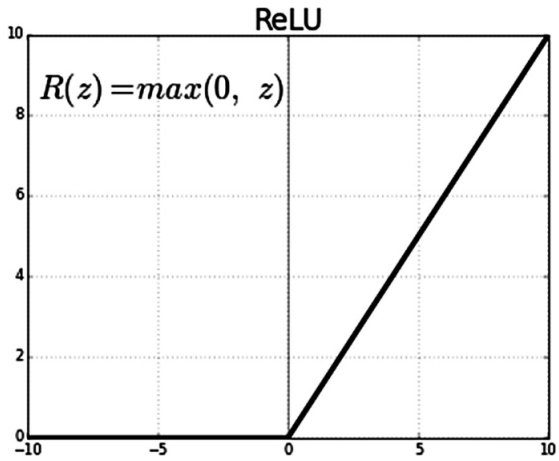

Figure 6.2: ReLU activation function

In ResNet-50, ReLU activation is applied after each convolutional and dense layer, including the appended dense layers for the BugSenseAI model. This non-linear activation allows the network to capture more expressive features and make more informed decisions during the classification process.

During training, the ReLU activation function promotes sparse and efficient representations by disregarding negative values. This can help mitigate the vanishing gradient problem and accelerate the learning process. By using ReLU, the BugSenseAI model benefits from improved performance, faster convergence, and increased capacity to capture intricate features of insect bites.

To summarize, the ResNet-50 architecture, including the appended dense layers, in

the BugSenseAI project utilizes the ReLU activation function after each convolutional and dense layer. This activation function enhances the model's ability to learn complex patterns and contributes to accurate classification of images uploaded by users.

6.1.2 Dataset

The dataset used for the BugSenseAI project is the "Processed Data" dataset. It was subdivided into three main subfolders: "train," "test," and "val," in the ratio 80:10:10, each consisting of five subfolders representing the five insect classes: bee, mosquito, none, spider, and tick.

The "train" folder contains a total of 3,649 images distributed among the five classes. These images were used for training the model, allowing it to learn and recognize patterns associated with each insect class.

The "val" folder contains 453 images, again distributed among the five classes. These images were used for validation during the training process to monitor the model's performance on unseen data and prevent overfitting.

The "test" folder comprises 459 images, also distributed among the five classes. These images were not used during model training or validation but can be employed to evaluate the final trained model's performance and assess its accuracy on unseen data.

To process the dataset, various functions and techniques were applied. Here is a summary of the operations performed on the dataset:

1. ImageDataGenerator: The Keras 'ImageDataGenerator' class was utilized to generate augmented images for the training data. It applied transformations such as zooming, shearing, and horizontal flipping to increase the dataset's diversity and improve the model's ability to generalize.

2. Preprocessing: Preprocessing for augmentation such as as zooming, shearing, flipping etc. was done on the dataset. The images in the dataset were preprocessed using the 'preprocess input' function from the 'tensorflow.keras.applications.resnet50' module. This function applies specific preprocessing steps, such as mean subtraction and scaling, to align the images with the requirements of the ResNet-50 model.

3. Data Flow: The 'flow from directory' method from the 'ImageDataGenerator' class was employed to load and generate batches of images directly from the directory structure. This facilitated the training and validation processes by automatically assigning labels to

the images based on their respective subfolders.

4. Model Building: The ResNet-50 architecture was used as the base model for the BugSenseAI project. The model was initialized with pre-trained weights from the ImageNet dataset, except for the top layers, which were customized for the specific insect classification task. Dense layers were appended to the base model, followed by a softmax activation function to produce class probabilities.

5. Training and Evaluation: The model was compiled using the Adam optimizer and categorical cross-entropy loss function. The training process involved fitting the model to the training data, with early stopping and model checkpointing techniques applied to monitor and save the best performing model based on validation accuracy. The training progress was recorded in a history object, allowing for visualization of the accuracy and loss trends over epochs.

6. Evaluation on Validation Set: After training, the model's performance was evaluated using the evaluation generator ('evaluate generator') on the validation data. The accuracy metric was used to assess the model's ability to correctly classify images from the validation set.

In summary, the "Processed Data" dataset used in the BugSenseAI project consists of training, validation, and test sets. Various functions and techniques were applied, including data augmentation, preprocessing, and model training, to prepare the dataset for training the ResNet-50 model.

6.2 Version History

Our initial trained model had a validation accuracy of 34.2 %. On investigation of the reason for low accuracy, we found out that the dataset wasn't properly divided into 80:10:10 ratio. The dataset was then gathered again and properly divided into the ratio of 80:10:10 (80% for training set, 10% for testing set and 10% for validation set.). The model was trained again and the accuracy increased to 87.77%. To further increase the accuracy, we added some more preprocessing functions to the training set such as zooming, shearing , flipping etc. apart from the default preprocess input function of ResNet50 which boosted the validation accuracy to 98.45%.

In order to use the model in the app, the initial .H5 model had to be converted to .TFLite

quantized format since TFLite uses less memory in the App. The converted TFLite model demonstrated a validation accuracy of 96.46%.

6.3 Modules

The BugSenseAI project consists of various modules that work together to provide an efficient and user-friendly insect bite identification system. These modules include the mobile application, the image uploading feature, the trained model, and the display of the prediction results along with suggested remedies.

(i) Mobile Application

The BugSenseAI mobile application serves as the interface for users to interact with the system. It provides a user-friendly environment where users can upload images of insect bites and receive predictions and remedies based on the uploaded images.

Implementation: - The mobile application is typically developed using frameworks such as React Native, Flutter, or native Android/iOS development tools. - The code for the mobile application involves creating user interface components, handling user interactions, and integrating with other modules. - It includes features such as image uploading, displaying prediction results, and providing remedy suggestions.

(ii) Image Uploading

Within the BugSenseAI mobile application, users can upload an image of an insect bite. The image uploading feature allows users to capture or select an existing image from their device's gallery. The uploaded image is then sent to the server for processing and prediction.

Implementation: - The code for image uploading depends on the mobile application framework and the chosen method for image selection. - It involves implementing functionality to capture images using the device's camera or selecting images from the device's gallery. - This code typically handles permissions, camera access, or file selection, and sends the selected image to the server for processing.

(iii) Trained Model

The trained model is a critical component of BugSenseAI. It is built using the ResNet-50 architecture, a deep CNN model that has been trained on a large dataset of labeled insect bite images. The model has learned to recognize and classify five insect classes: Tick, Mosquito, Spider, Bee, and None. It takes the uploaded image as input and performs inference using the learned weights and parameters to predict the insect class.

Implementation: - The code for the trained model includes building and training the ResNet-50 model using a deep learning framework such as TensorFlow or Keras. - It involves loading the ResNet-50 architecture and its pre-trained weights. - The code also includes modifying the model's final layers to match the insect classes (Tick, Mosquito, Spider, Bee, and None) and retraining or fine-tuning the model with the insect bite dataset.

(iv) Prediction Results

Once the trained model processes the uploaded image, it generates prediction results. The model predicts the insect class that best matches the characteristics of the uploaded image. For example, if the uploaded image corresponds to a tick bite, the model will predict the "Tick" class. The prediction result is then displayed to the user in the mobile application.

Implementation: - Once the model receives the uploaded image, the code passes the image through the model for inference. - It involves preprocessing the image, such as resizing and normalizing the pixel values, to match the input requirements of the trained model. - The code then uses the trained model to predict the insect class of the uploaded image. - Finally, the code formats and returns the prediction result to be displayed in the mobile application.

(v) Remedies

In addition to the prediction result, the BugSenseAI mobile application also suggests remedies based on the identified insect bite. The application provides information and guidance on how to alleviate the symptoms, offer first aid, or seek medical attention, depending on the insect bite classification. These remedies aim to provide helpful recom-

mendations to users who have encountered insect bites.

Implementation: - The code for suggesting remedies is typically implemented within the mobile application. - It involves associating each insect class (Tick, Mosquito, Spider, Bee) with a set of corresponding remedies. - The code retrieves the predicted insect class from the model's prediction results and retrieves the associated remedies. - Finally, the code formats and displays the suggested remedies in the mobile application along with the prediction result.

By integrating these modules, BugSenseAI offers a comprehensive solution for insect bite identification. Users can conveniently upload images of insect bites through the mobile application, receive accurate predictions from the trained model, and access remedies tailored to the specific insect bite classification. This combination of image recognition technology and remedy suggestions enhances user awareness and aids in addressing insect bites effectively.

Chapter 7

Testing

7.1 Testing Objectives

The primary objectives of the BugSenseAI app testing were as follows: 1. Verify the accuracy of insect bite identification by the app's trained model. 2. Validate the proper functioning of the image uploading feature. 3. Evaluate the delivery of accurate remedy suggestions based on the identified insect bite. 4. Assess the overall performance, responsiveness, and usability of the app.

7.2 Testing Approach

The testing process involved the following steps:

1. Test Case Preparation: Test cases were designed to cover various scenarios, including uploading different insect bite images, validating correct identification, and verifying accurate remedy suggestions.

2. Test Environment Setup: The app was installed on different devices via emulators. The App was also tested on different physical devices that runs Android such as Samsung, Nothing and Xiaomi.

3. Functional Testing: The functionality of the app was rigorously tested, including image uploading, processing, prediction accuracy, and remedy suggestion retrieval.

4. Performance Testing: The app's performance was evaluated under different hardware specifications and it was found to be free from any type of drop in performance.

5. User Experience Testing: A group of users tested the app to provide feedback on the user interface, ease of use, and overall experience.

7.3 Testing Results

The BugSenseAI app testing produced the following results:

1. Insect Bite Identification Accuracy: The app demonstrated an accuracy of 96.46% in identifying insect bites. It correctly classified a significant majority of uploaded images into the appropriate insect classes (Tick, Mosquito, Spider, Bee, None).

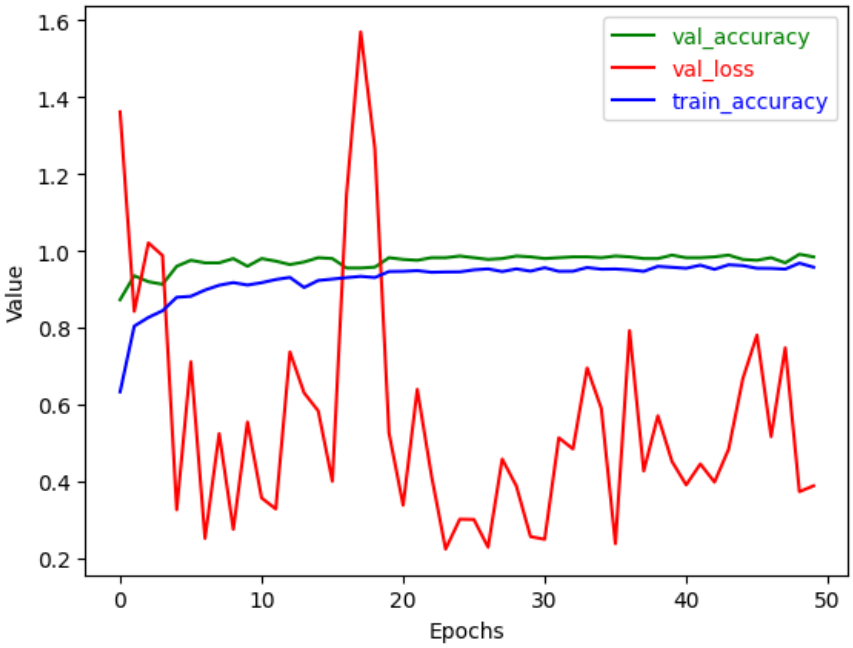

Figure 7.1: Plotting of validation accuracy, training accuracy, validation loss

2. Image Uploading Functionality: The image uploading feature worked flawlessly across multiple devices, allowing users to capture or select images from their device's gallery and successfully upload them for processing.

3. Remedy Suggestion Accuracy: The app consistently provided relevant remedy suggestions based on the identified insect bite. The suggested remedies aligned with standard practices and offered appropriate guidance for alleviating symptoms and seeking medical attention when necessary.

4. Performance and Responsiveness: The app exhibited satisfactory performance with minimal latency during image processing and prediction. It maintained a responsive and smooth user interface, even when handling larger image sizes.

5. User Experience: User feedback indicated that the app was intuitive, easy to navigate, and provided clear instructions for image uploading. Users appreciated the

prompt and accurate identification results and found the remedy suggestions helpful and informative.

Chapter 8

Results

The BugSenseAI mobile application underwent testing among a group of students, with impressively high insect bite identification accuracy of 96.46%. Users praised its consistent and accurate classification of uploaded images into various insect classes (Tick, Mosquito, Spider, Bee, or None). The user-friendly interface, smooth performance, and clear remedy recommendations contributed to a positive experience. BugSenseAI proves to be a valuable tool, empowering users to swiftly make informed decisions about insect bites and enhance their well-being.

The following screenshots show an example use case scenario of the working of BugSenseAI mobile application from a user's point of view.

When the user opens the app he/she is initially brought to the splash page (Fig 8.1). After loading the main activity page as shown in Fig 8.2 is displayed where the user on his initial use would be asked to grant permissions to the app. Then the user is asked to upload an image of the affected skin area. He/she can upload an image from the gallery ('SELECT IMAGE' button in fig 8.3) or take a picture using the camera module ('CAMERA' button in Fig 8.3). After uploading the image the user clicks the 'SENSE' button and then the result from the pre-trained model is returned and the predicted insect bite is displayed on the screen along with a 'VIEW REMEDIES' button as shown in fig 8.4. On clicking this button the user is taken to another page that shows the remedies for the predicted insect bite (fig 8.5).

Fig 8.6 and 8.7 shows another use case where the user is not affected by an insect bite, ie. the 'None' case. The user captures an image of the non-affected skin area using the camera module as shown in Fig 8.6. The result is returned as None and displayed to the

user as shown in fig 8.7. In 'None' case the 'VIEW REMEDIES' button would not appear as the user was not affected by an insect bite.

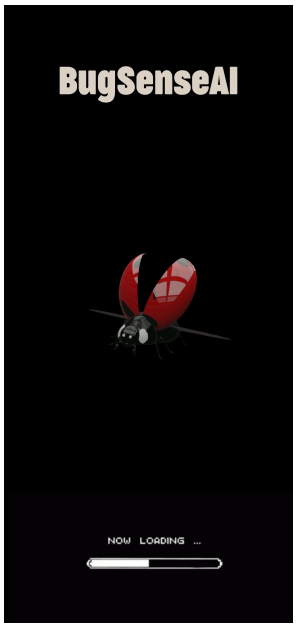

Figure 8.1: Splash Page

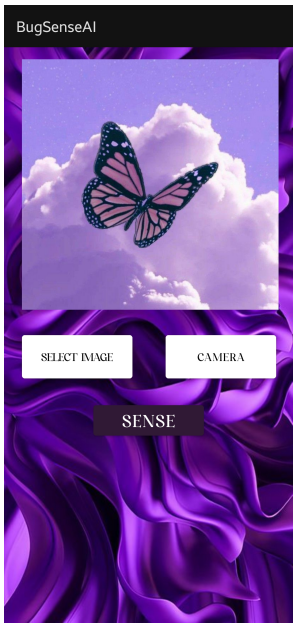

Figure 8.2: Main Activity Page

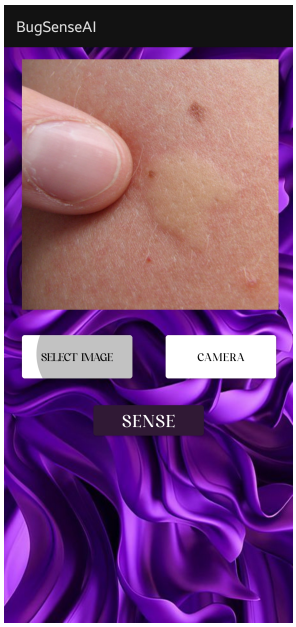

Figure 8.3: Image Upload from Storage

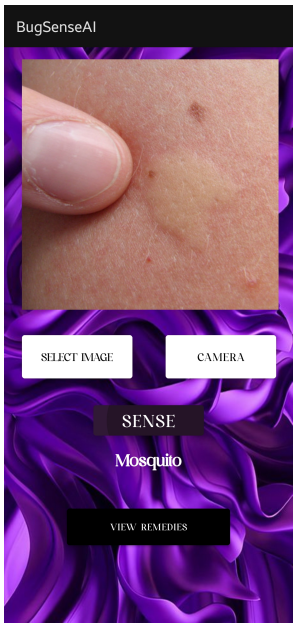

Figure 8.4: Prediction of Image Uploaded

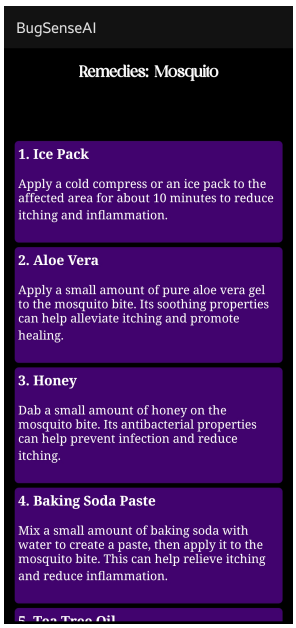

Figure 8.5: Remedies based on Prediction

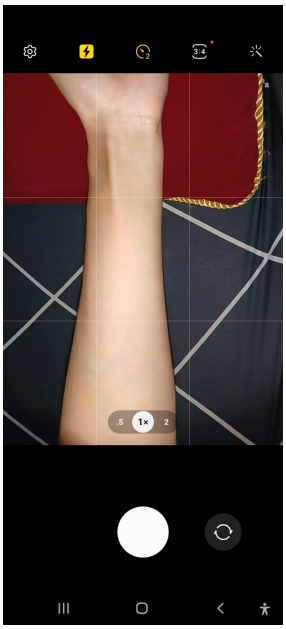

Figure 8.6: Camera module

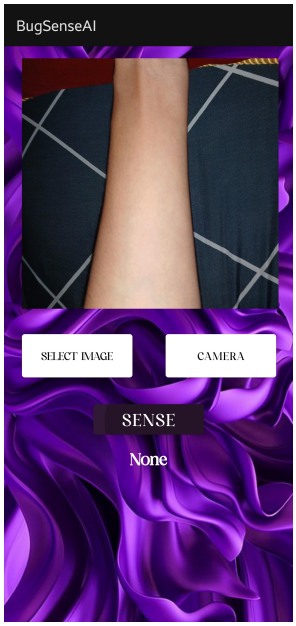

Figure 8.7: Result of image uploaded from camera module

Chapter 9

Risks and Challenges

1. Accuracy Issues: The problem occurred during the conversion from .H5 to TFLite model which was essential for integrating with the app that runs only with TFLite model This conversion process led to deviations in accuracy, impacting the overall performance of the model.

2. Device Permissions: Access must be granted on the device for features to run, such as camera for capturing images of affected area.

3. App Familiarization: User needs to get familiar with the various interfaces of the app.

4. Effectiveness of Remedies: The remedies provided for each bite might not be effective in all cases.

Chapter 10

Conclusion

We have developed an android application to identify the insect from a sting bite of the patient. The app consists of the following features: Menu consisting of options to upload sting bite image or click real-time image of the bite,Image Recognition and Remedy Display for each bite. All of these features have been verified to be working as intended. We first collected and preprocessed a dataset of insect bite images. Then, fine-tuned the pre-trained ResNet-50 model on the dataset by freezing the initial layers and training the remaining layers. Then we added a few fully connected layers for classification. During training,we utilized residual connections to propagate gradients effectively and enable the model to learn complex features related to insect bites. Evaluate the trained model on a test set and fine-tune hyper parameters if necessary.

Recommendations for Future Enhancements

- Increasing insect classes and improving accuracy: The model can be modified to accomodate more than the 5 insect classes that are currently implemented. Different techniques can be tested to improve accuracy of the model further.

- Continuous Model Training: Regular updates to the trained model can further improve identification accuracy by incorporating more diverse insect bite images. The model can be retrained based on newly received image data that were uploaded by users with each use. (ie. setting up of live training feature.)

- Integration of User Feedback: Incorporating user feedback in future updates can enhance the user experience and address any potential usability issues.

References

[1] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 2016, pp. 770-778, doi: 10.1109/CVPR.2016.90.

[2] Razzak, M.I.; Naz, S.; Zaib, A. Deep Learning for Medical Image Processing: Overview, Challenges and the Future. In Classification in BioApps: Automation of Decision Making; Springer International Publishing: Cham, Switzerland, 2018; pp. 323–350.

[3] V. Sharma and N. Singh, "Deep Convolutional Neural Network with ResNet-50 Learning algorithm for Copy-Move Forgery Detection," 2021 7th International Conference on Signal Processing and Communication (ICSC), Noida, India, 2021, pp. 146-150, doi: 10.1109/ICSC53193.2021.9673422.

[4] R. Chauhan, K. K. Ghanshala and R. C. Joshi, "Convolutional Neural Network (CNN) for Image Detection and Recognition," 2018 First International Conference on Secure Cyber Computing and Communication (ICSCCC), Jalandhar, India, 2018, pp. 278-282, doi: 10.1109/ICSCCC.2018.8703316.

[5] Xin, M., Wang, Y. Research on image classification model based on deep convolution neural network. J Image Video Proc. 2019, 40 (2019). https://doi.org/10.1186/s13640-019-0417-8

[6] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 2016, pp. 779-788, doi: 10.1109/CVPR.2016.91.

[7] Simonyan, Karen & Zisserman, Andrew. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv 1409.1556.

[8] W. Liu, G. Wu, F. Ren and X. Kang, "DFF-ResNet: An insect pest recognition model based on residual networks," in Big Data Mining and Analytics, vol. 3, no. 4, pp. 300-310, Dec. 2020, doi: 10.26599/BDMA.2020.9020021.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In Advances in Neural Information Processing Systems 25, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1097–1105.

[10] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, Densely connected convolutional networks, in Proc. 2017 IEEE Conf. Computer Vision and Pattern Recognition, Honolulu, HI, USA, 2017, pp. 2261–2269

Appendix A: Base Paper

Deep Residual Learning for Image Recognition

Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun
Microsoft Research
{kahe, v-xiangz, v-shren, jiansun}@microsoft.com

Abstract

Deeper neural networks are more difficult to train. We present a residual learning framework to ease the training of networks that are substantially deeper than those used previously. We explicitly reformulate the layers as learning residual functions with reference to the layer inputs, instead of learning unreferenced functions. We provide comprehensive empirical evidence showing that these residual networks are easier to optimize, and can gain accuracy from considerably increased depth. On the ImageNet dataset we evaluate residual nets with a depth of up to 152 layers—8× deeper than VGG nets [40] but still having lower complexity. An ensemble of these residual nets achieves 3.57% error on the ImageNet test *set. This result won the 1st place on the ILSVRC 2015 classification task. We also present analysis on CIFAR-10 with 100 and 1000 layers.*

The depth of representations is of central importance for many visual recognition tasks. Solely due to our extremely deep representations, we obtain a 28% relative improvement on the COCO object detection dataset. Deep residual nets are foundations of our submissions to ILSVRC & COCO 2015 competitions[1]*, where we also won the 1st places on the tasks of ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation.*

1. Introduction

Deep convolutional neural networks [22, 21] have led to a series of breakthroughs for image classification [21, 49, 39]. Deep networks naturally integrate low/mid/high-level features [49] and classifiers in an end-to-end multi-layer fashion, and the "levels" of features can be enriched by the number of stacked layers (depth). Recent evidence [40, 43] reveals that network depth is of crucial importance, and the leading results [40, 43, 12, 16] on the challenging ImageNet dataset [35] all exploit "very deep" [40] models, with a depth of sixteen [40] to thirty [16]. Many other non-trivial visual recognition tasks [7, 11, 6, 32, 27] have also greatly benefited from very deep models.

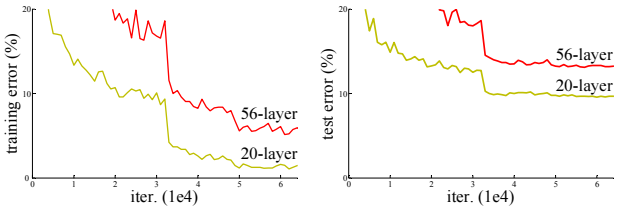

Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

Driven by the significance of depth, a question arises: *Is learning better networks as easy as stacking more layers?* An obstacle to answering this question was the notorious problem of vanishing/exploding gradients [14, 1, 8], which hamper convergence from the beginning. This problem, however, has been largely addressed by normalized initialization [23, 8, 36, 12] and intermediate normalization layers [16], which enable networks with tens of layers to start converging for stochastic gradient descent (SGD) with back-propagation [22].

When deeper networks are able to start converging, a *degradation* problem has been exposed: with the network depth increasing, accuracy gets saturated (which might be unsurprising) and then degrades rapidly. Unexpectedly, such degradation is *not caused by overfitting*, and adding more layers to a suitably deep model leads to *higher training error*, as reported in [10, 41] and thoroughly verified by our experiments. Fig. 1 shows a typical example.

The degradation (of training accuracy) indicates that not all systems are similarly easy to optimize. Let us consider a shallower architecture and its deeper counterpart that adds more layers onto it. There exists a solution *by construction* to the deeper model: the added layers are *identity* mapping, and the other layers are copied from the learned shallower model. The existence of this constructed solution indicates that a deeper model should produce no higher training error than its shallower counterpart. But experiments show that our current solvers on hand are unable to find solutions that

[1] http://image-net.org/challenges/LSVRC/2015/ and http://mscoco.org/dataset/#detections-challenge2015.

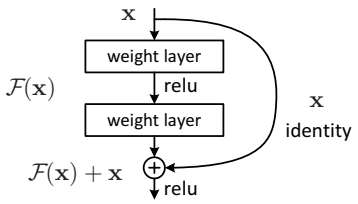

Figure 2. Residual learning: a building block.

are comparably good or better than the constructed solution (or unable to do so in feasible time).

In this paper, we address the degradation problem by introducing a *deep residual learning* framework. Instead of hoping each few stacked layers directly fit a desired underlying mapping, we explicitly let these layers fit a residual mapping. Formally, denoting the desired underlying mapping as $\mathcal{H}(\mathbf{x})$, we let the stacked nonlinear layers fit another mapping of $\mathcal{F}(\mathbf{x}) := \mathcal{H}(\mathbf{x}) - \mathbf{x}$. The original mapping is recast into $\mathcal{F}(\mathbf{x}) + \mathbf{x}$. We hypothesize that it is easier to optimize the residual mapping than to optimize the original, unreferenced mapping. To the extreme, if an identity mapping were optimal, it would be easier to push the residual to zero than to fit an identity mapping by a stack of nonlinear layers.

The formulation of $\mathcal{F}(\mathbf{x}) + \mathbf{x}$ can be realized by feedforward neural networks with "shortcut connections" (Fig. 2). Shortcut connections [2, 33, 48] are those skipping one or more layers. In our case, the shortcut connections simply perform *identity* mapping, and their outputs are added to the outputs of the stacked layers (Fig. 2). Identity shortcut connections add neither extra parameter nor computational complexity. The entire network can still be trained end-to-end by SGD with backpropagation, and can be easily implemented using common libraries (*e.g.*, Caffe [19]) without modifying the solvers.

We present comprehensive experiments on ImageNet [35] to show the degradation problem and evaluate our method. We show that: 1) Our extremely deep residual nets are easy to optimize, but the counterpart "plain" nets (that simply stack layers) exhibit higher training error when the depth increases; 2) Our deep residual nets can easily enjoy accuracy gains from greatly increased depth, producing results substantially better than previous networks.

Similar phenomena are also shown on the CIFAR-10 set [20], suggesting that the optimization difficulties and the effects of our method are not just akin to a particular dataset. We present successfully trained models on this dataset with over 100 layers, and explore models with over 1000 layers.

On the ImageNet classification dataset [35], we obtain excellent results by extremely deep residual nets. Our 152-layer residual net is the deepest network ever presented on ImageNet, while still having lower complexity than VGG nets [40]. Our ensemble has **3.57%** top-5 error on the ImageNet *test* set, and *won the 1st place in the ILSVRC 2015 classification competition*. The extremely deep representations also have excellent generalization performance on other recognition tasks, and lead us to further *win the 1st places on: ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation* in ILSVRC & COCO 2015 competitions. This strong evidence shows that the residual learning principle is generic, and we expect that it is applicable in other vision and non-vision problems.

2. Related Work

Residual Representations. In image recognition, VLAD [18] is a representation that encodes by the residual vectors with respect to a dictionary, and Fisher Vector [30] can be formulated as a probabilistic version [18] of VLAD. Both of them are powerful shallow representations for image retrieval and classification [4, 47]. For vector quantization, encoding residual vectors [17] is shown to be more effective than encoding original vectors.

In low-level vision and computer graphics, for solving Partial Differential Equations (PDEs), the widely used Multigrid method [3] reformulates the system as subproblems at multiple scales, where each subproblem is responsible for the residual solution between a coarser and a finer scale. An alternative to Multigrid is hierarchical basis preconditioning [44, 45], which relies on variables that represent residual vectors between two scales. It has been shown [3, 44, 45] that these solvers converge much faster than standard solvers that are unaware of the residual nature of the solutions. These methods suggest that a good reformulation or preconditioning can simplify the optimization.

Shortcut Connections. Practices and theories that lead to shortcut connections [2, 33, 48] have been studied for a long time. An early practice of training multi-layer perceptrons (MLPs) is to add a linear layer connected from the network input to the output [33, 48]. In [43, 24], a few intermediate layers are directly connected to auxiliary classifiers for addressing vanishing/exploding gradients. The papers of [38, 37, 31, 46] propose methods for centering layer responses, gradients, and propagated errors, implemented by shortcut connections. In [43], an "inception" layer is composed of a shortcut branch and a few deeper branches.

Concurrent with our work, "highway networks" [41, 42] present shortcut connections with gating functions [15]. These gates are data-dependent and have parameters, in contrast to our identity shortcuts that are parameter-free. When a gated shortcut is "closed" (approaching zero), the layers in highway networks represent *non-residual* functions. On the contrary, our formulation always learns residual functions; our identity shortcuts are never closed, and all information is always passed through, with additional residual functions to be learned. In addition, high-

way networks have not demonstrated accuracy gains with extremely increased depth (*e.g.*, over 100 layers).

3. Deep Residual Learning

3.1. Residual Learning

Let us consider $\mathcal{H}(\mathbf{x})$ as an underlying mapping to be fit by a few stacked layers (not necessarily the entire net), with $\mathbf{x}$ denoting the inputs to the first of these layers. If one hypothesizes that multiple nonlinear layers can asymptotically approximate complicated functions[2], then it is equivalent to hypothesize that they can asymptotically approximate the residual functions, *i.e.*, $\mathcal{H}(\mathbf{x}) - \mathbf{x}$ (assuming that the input and output are of the same dimensions). So rather than expect stacked layers to approximate $\mathcal{H}(\mathbf{x})$, we explicitly let these layers approximate a residual function $\mathcal{F}(\mathbf{x}) := \mathcal{H}(\mathbf{x}) - \mathbf{x}$. The original function thus becomes $\mathcal{F}(\mathbf{x}) + \mathbf{x}$. Although both forms should be able to asymptotically approximate the desired functions (as hypothesized), the ease of learning might be different.

This reformulation is motivated by the counterintuitive phenomena about the degradation problem (Fig. 1, left). As we discussed in the introduction, if the added layers can be constructed as identity mappings, a deeper model should have training error no greater than its shallower counterpart. The degradation problem suggests that the solvers might have difficulties in approximating identity mappings by multiple nonlinear layers. With the residual learning reformulation, if identity mappings are optimal, the solvers may simply drive the weights of the multiple nonlinear layers toward zero to approach identity mappings.

In real cases, it is unlikely that identity mappings are optimal, but our reformulation may help to precondition the problem. If the optimal function is closer to an identity mapping than to a zero mapping, it should be easier for the solver to find the perturbations with reference to an identity mapping, than to learn the function as a new one. We show by experiments (Fig. 7) that the learned residual functions in general have small responses, suggesting that identity mappings provide reasonable preconditioning.

3.2. Identity Mapping by Shortcuts

We adopt residual learning to every few stacked layers. A building block is shown in Fig. 2. Formally, in this paper we consider a building block defined as:

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + \mathbf{x}. \tag{1}$$

Here $\mathbf{x}$ and $\mathbf{y}$ are the input and output vectors of the layers considered. The function $\mathcal{F}(\mathbf{x}, \{W_i\})$ represents the residual mapping to be learned. For the example in Fig. 2 that has two layers, $\mathcal{F} = W_2\sigma(W_1\mathbf{x})$ in which σ denotes ReLU [29] and the biases are omitted for simplifying notations. The operation $\mathcal{F} + \mathbf{x}$ is performed by a shortcut connection and element-wise addition. We adopt the second nonlinearity after the addition (*i.e.*, $\sigma(\mathbf{y})$, see Fig. 2).

The shortcut connections in Eqn.(1) introduce neither extra parameter nor computation complexity. This is not only attractive in practice but also important in our comparisons between plain and residual networks. We can fairly compare plain/residual networks that simultaneously have the same number of parameters, depth, width, and computational cost (except for the negligible element-wise addition).

The dimensions of $\mathbf{x}$ and $\mathcal{F}$ must be equal in Eqn.(1). If this is not the case (*e.g.*, when changing the input/output channels), we can perform a linear projection W_s by the shortcut connections to match the dimensions:

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + W_s\mathbf{x}. \tag{2}$$

We can also use a square matrix W_s in Eqn.(1). But we will show by experiments that the identity mapping is sufficient for addressing the degradation problem and is economical, and thus W_s is only used when matching dimensions.

The form of the residual function $\mathcal{F}$ is flexible. Experiments in this paper involve a function $\mathcal{F}$ that has two or three layers (Fig. 5), while more layers are possible. But if $\mathcal{F}$ has only a single layer, Eqn.(1) is similar to a linear layer: $\mathbf{y} = W_1\mathbf{x} + \mathbf{x}$, for which we have not observed advantages.

We also note that although the above notations are about fully-connected layers for simplicity, they are applicable to convolutional layers. The function $\mathcal{F}(\mathbf{x}, \{W_i\})$ can represent multiple convolutional layers. The element-wise addition is performed on two feature maps, channel by channel.

3.3. Network Architectures

We have tested various plain/residual nets, and have observed consistent phenomena. To provide instances for discussion, we describe two models for ImageNet as follows.

Plain Network. Our plain baselines (Fig. 3, middle) are mainly inspired by the philosophy of VGG nets [40] (Fig. 3, left). The convolutional layers mostly have 3×3 filters and follow two simple design rules: (i) for the same output feature map size, the layers have the same number of filters; and (ii) if the feature map size is halved, the number of filters is doubled so as to preserve the time complexity per layer. We perform downsampling directly by convolutional layers that have a stride of 2. The network ends with a global average pooling layer and a 1000-way fully-connected layer with softmax. The total number of weighted layers is 34 in Fig. 3 (middle).

It is worth noticing that our model has *fewer* filters and *lower* complexity than VGG nets [40] (Fig. 3, left). Our 34-layer baseline has 3.6 billion FLOPs (multiply-adds), which is only 18% of VGG-19 (19.6 billion FLOPs).

[2]This hypothesis, however, is still an open question. See [28].

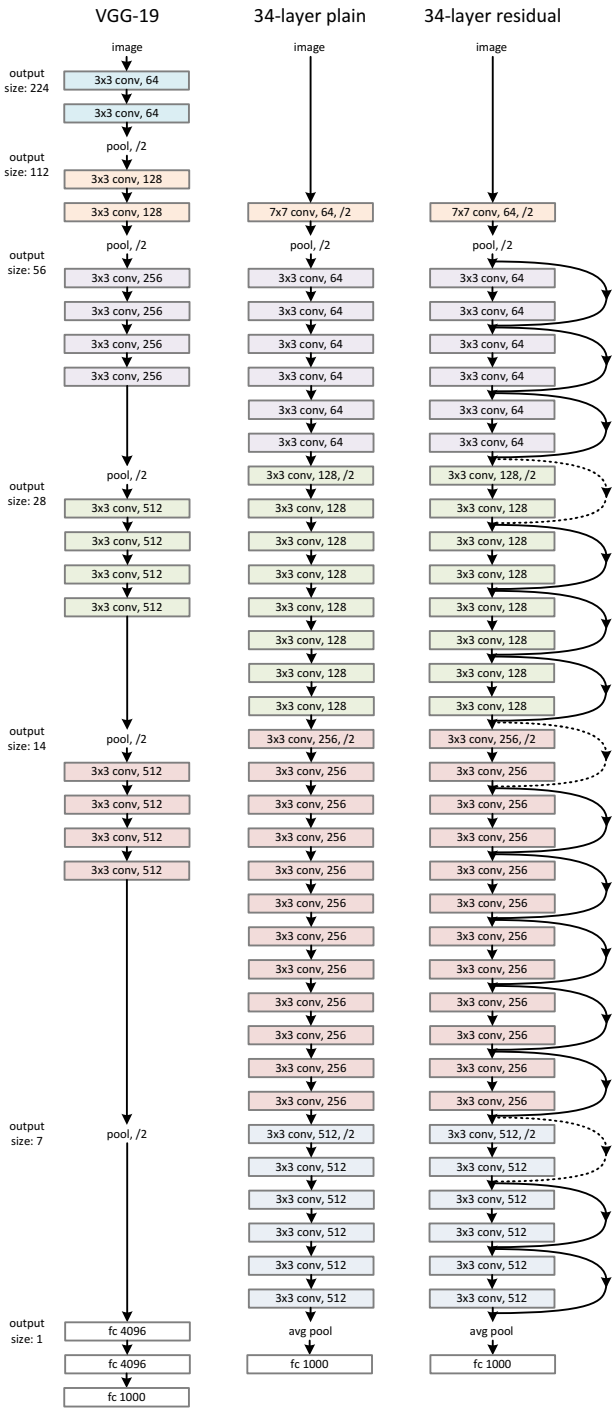

Figure 3. Example network architectures for ImageNet. **Left**: the VGG-19 model [40] (19.6 billion FLOPs) as a reference. **Middle**: a plain network with 34 parameter layers (3.6 billion FLOPs). **Right**: a residual network with 34 parameter layers (3.6 billion FLOPs). The dotted shortcuts increase dimensions. **Table 1** shows more details and other variants.

Residual Network. Based on the above plain network, we insert shortcut connections (Fig. 3, right) which turn the network into its counterpart residual version. The identity shortcuts (Eqn.(1)) can be directly used when the input and output are of the same dimensions (solid line shortcuts in Fig. 3). When the dimensions increase (dotted line shortcuts in Fig. 3), we consider two options: (A) The shortcut still performs identity mapping, with extra zero entries padded for increasing dimensions. This option introduces no extra parameter; (B) The projection shortcut in Eqn.(2) is used to match dimensions (done by 1×1 convolutions). For both options, when the shortcuts go across feature maps of two sizes, they are performed with a stride of 2.

3.4. Implementation

Our implementation for ImageNet follows the practice in [21, 40]. The image is resized with its shorter side randomly sampled in $[256, 480]$ for scale augmentation [40]. A 224×224 crop is randomly sampled from an image or its horizontal flip, with the per-pixel mean subtracted [21]. The standard color augmentation in [21] is used. We adopt batch normalization (BN) [16] right after each convolution and before activation, following [16]. We initialize the weights as in [12] and train all plain/residual nets from scratch. We use SGD with a mini-batch size of 256. The learning rate starts from 0.1 and is divided by 10 when the error plateaus, and the models are trained for up to 60×10^4 iterations. We use a weight decay of 0.0001 and a momentum of 0.9. We do not use dropout [13], following the practice in [16].

In testing, for comparison studies we adopt the standard 10-crop testing [21]. For best results, we adopt the fully-convolutional form as in [40, 12], and average the scores at multiple scales (images are resized such that the shorter side is in $\{224, 256, 384, 480, 640\}$).

4. Experiments

4.1. ImageNet Classification

We evaluate our method on the ImageNet 2012 classification dataset [35] that consists of 1000 classes. The models are trained on the 1.28 million training images, and evaluated on the 50k validation images. We also obtain a final result on the 100k test images, reported by the test server. We evaluate both top-1 and top-5 error rates.

Plain Networks. We first evaluate 18-layer and 34-layer plain nets. The 34-layer plain net is in Fig. 3 (middle). The 18-layer plain net is of a similar form. See Table 1 for detailed architectures.

The results in Table 2 show that the deeper 34-layer plain net has higher validation error than the shallower 18-layer plain net. To reveal the reasons, in Fig. 4 (left) we compare their training/validation errors during the training procedure. We have observed the degradation problem - the

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Table 1. Architectures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of blocks stacked. Downsampling is performed by conv3_1, conv4_1, and conv5_1 with a stride of 2.

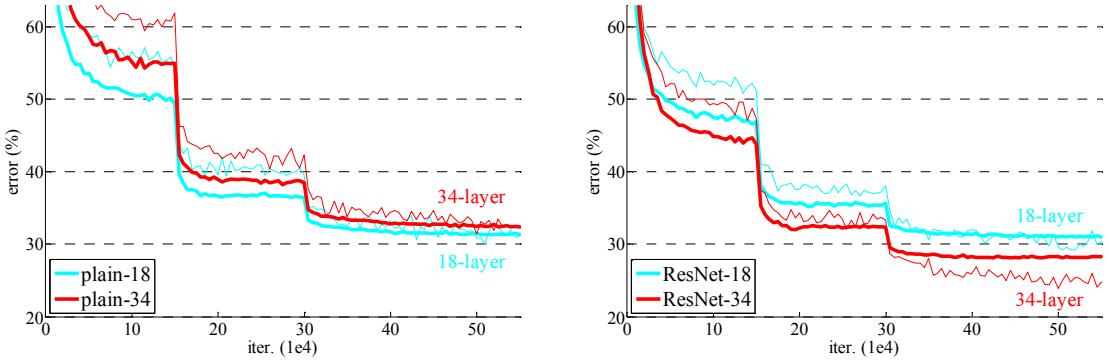

Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

	plain	ResNet
18 layers	27.94	27.88
34 layers	28.54	**25.03**

Table 2. Top-1 error (%, 10-crop testing) on ImageNet validation. Here the ResNets have no extra parameter compared to their plain counterparts. Fig. 4 shows the training procedures.

34-layer plain net has higher *training* error throughout the whole training procedure, even though the solution space of the 18-layer plain network is a subspace of that of the 34-layer one.

We argue that this optimization difficulty is *unlikely* to be caused by vanishing gradients. These plain networks are trained with BN [16], which ensures forward propagated signals to have non-zero variances. We also verify that the backward propagated gradients exhibit healthy norms with BN. So neither forward nor backward signals vanish. In fact, the 34-layer plain net is still able to achieve competitive accuracy (Table 3), suggesting that the solver works to some extent. We conjecture that the deep plain nets may have exponentially low convergence rates, which impact the reducing of the training error[3]. The reason for such optimization difficulties will be studied in the future.

Residual Networks. Next we evaluate 18-layer and 34-layer residual nets (*ResNets*). The baseline architectures are the same as the above plain nets, expect that a shortcut connection is added to each pair of 3×3 filters as in Fig. 3 (right). In the first comparison (Table 2 and Fig. 4 right), we use identity mapping for all shortcuts and zero-padding for increasing dimensions (option A). So they have *no extra parameter* compared to the plain counterparts.

We have three major observations from Table 2 and Fig. 4. First, the situation is reversed with residual learning – the 34-layer ResNet is better than the 18-layer ResNet (by 2.8%). More importantly, the 34-layer ResNet exhibits considerably lower training error and is generalizable to the validation data. This indicates that the degradation problem is well addressed in this setting and we manage to obtain accuracy gains from increased depth.

Second, compared to its plain counterpart, the 34-layer

[3]We have experimented with more training iterations (3×) and still observed the degradation problem, suggesting that this problem cannot be feasibly addressed by simply using more iterations.

model	top-1 err.	top-5 err.
VGG-16 [40]	28.07	9.33
GoogLeNet [43]	-	9.15
PReLU-net [12]	24.27	7.38
plain-34	28.54	10.02
ResNet-34 A	25.03	7.76
ResNet-34 B	24.52	7.46
ResNet-34 C	24.19	7.40
ResNet-50	22.85	6.71
ResNet-101	21.75	6.05
ResNet-152	**21.43**	**5.71**

Table 3. Error rates (%, **10-crop** testing) on ImageNet validation. VGG-16 is based on our test. ResNet-50/101/152 are of option B that only uses projections for increasing dimensions.

method	top-1 err.	top-5 err.
VGG [40] (ILSVRC'14)	-	8.43†
GoogLeNet [43] (ILSVRC'14)	-	7.89
VGG [40] (v5)	24.4	7.1
PReLU-net [12]	21.59	5.71
BN-inception [16]	21.99	5.81
ResNet-34 B	21.84	5.71
ResNet-34 C	21.53	5.60
ResNet-50	20.74	5.25
ResNet-101	19.87	4.60
ResNet-152	**19.38**	**4.49**

Table 4. Error rates (%) of **single-model** results on the ImageNet validation set (except † reported on the test set).

method	top-5 err. (**test**)
VGG [40] (ILSVRC'14)	7.32
GoogLeNet [43] (ILSVRC'14)	6.66
VGG [40] (v5)	6.8
PReLU-net [12]	4.94
BN-inception [16]	4.82
ResNet (ILSVRC'15)	**3.57**

Table 5. Error rates (%) of **ensembles**. The top-5 error is on the test set of ImageNet and reported by the test server.

ResNet reduces the top-1 error by 3.5% (Table 2), resulting from the successfully reduced training error (Fig. 4 right *vs*. left). This comparison verifies the effectiveness of residual learning on extremely deep systems.

Last, we also note that the 18-layer plain/residual nets are comparably accurate (Table 2), but the 18-layer ResNet converges faster (Fig. 4 right *vs*. left). When the net is "not overly deep" (18 layers here), the current SGD solver is still able to find good solutions to the plain net. In this case, the ResNet eases the optimization by providing faster convergence at the early stage.

Identity *vs*. Projection Shortcuts. We have shown that parameter-free, identity shortcuts help with training. Next we investigate projection shortcuts (Eqn.(2)). In Table 3 we compare three options: (A) zero-padding shortcuts are used for increasing dimensions, and all shortcuts are parameter-free (the same as Table 2 and Fig. 4 right); (B) projection shortcuts are used for increasing dimensions, and other shortcuts are identity; and (C) all shortcuts are projections.

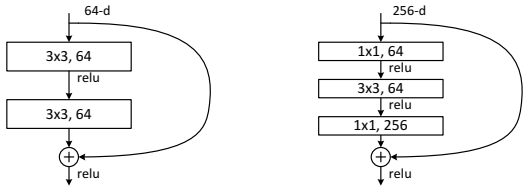

Figure 5. A deeper residual function $\mathcal{F}$ for ImageNet. Left: a building block (on 56×56 feature maps) as in Fig. 3 for ResNet-34. Right: a "bottleneck" building block for ResNet-50/101/152.

Table 3 shows that all three options are considerably better than the plain counterpart. B is slightly better than A. We argue that this is because the zero-padded dimensions in A indeed have no residual learning. C is marginally better than B, and we attribute this to the extra parameters introduced by many (thirteen) projection shortcuts. But the small differences among A/B/C indicate that projection shortcuts are not essential for addressing the degradation problem. So we do not use option C in the rest of this paper, to reduce memory/time complexity and model sizes. Identity shortcuts are particularly important for not increasing the complexity of the bottleneck architectures that are introduced below.

Deeper Bottleneck Architectures. Next we describe our deeper nets for ImageNet. Because of concerns on the training time that we can afford, we modify the building block as a *bottleneck* design[4]. For each residual function $\mathcal{F}$, we use a stack of 3 layers instead of 2 (Fig. 5). The three layers are 1×1, 3×3, and 1×1 convolutions, where the 1×1 layers are responsible for reducing and then increasing (restoring) dimensions, leaving the 3×3 layer a bottleneck with smaller input/output dimensions. Fig. 5 shows an example, where both designs have similar time complexity.

The parameter-free identity shortcuts are particularly important for the bottleneck architectures. If the identity shortcut in Fig. 5 (right) is replaced with projection, one can show that the time complexity and model size are doubled, as the shortcut is connected to the two high-dimensional ends. So identity shortcuts lead to more efficient models for the bottleneck designs.

50-layer ResNet: We replace each 2-layer block in the

[4]Deeper *non*-bottleneck ResNets (*e.g*., Fig. 5 left) also gain accuracy from increased depth (as shown on CIFAR-10), but are not as economical as the bottleneck ResNets. So the usage of bottleneck designs is mainly due to practical considerations. We further note that the degradation problem of plain nets is also witnessed for the bottleneck designs.

34-layer net with this 3-layer bottleneck block, resulting in a 50-layer ResNet (Table 1). We use option B for increasing dimensions. This model has 3.8 billion FLOPs.

101-layer and 152-layer ResNets: We construct 101-layer and 152-layer ResNets by using more 3-layer blocks (Table 1). Remarkably, although the depth is significantly increased, the 152-layer ResNet (11.3 billion FLOPs) still has *lower complexity* than VGG-16/19 nets (15.3/19.6 billion FLOPs).

The 50/101/152-layer ResNets are more accurate than the 34-layer ones by considerable margins (Table 3 and 4). We do not observe the degradation problem and thus enjoy significant accuracy gains from considerably increased depth. The benefits of depth are witnessed for all evaluation metrics (Table 3 and 4).

Comparisons with State-of-the-art Methods. In Table 4 we compare with the previous best single-model results. Our baseline 34-layer ResNets have achieved very competitive accuracy. Our 152-layer ResNet has a single-model top-5 validation error of 4.49%. This single-model result outperforms all previous ensemble results (Table 5). We combine six models of different depth to form an ensemble (only with two 152-layer ones at the time of submitting). This leads to **3.57%** top-5 error on the test set (Table 5). *This entry won the 1st place in ILSVRC 2015.*

4.2. CIFAR-10 and Analysis

We conducted more studies on the CIFAR-10 dataset [20], which consists of 50k training images and 10k testing images in 10 classes. We present experiments trained on the training set and evaluated on the test set. Our focus is on the behaviors of extremely deep networks, but not on pushing the state-of-the-art results, so we intentionally use simple architectures as follows.

The plain/residual architectures follow the form in Fig. 3 (middle/right). The network inputs are 32×32 images, with the per-pixel mean subtracted. The first layer is 3×3 convolutions. Then we use a stack of $6n$ layers with 3×3 convolutions on the feature maps of sizes $\{32, 16, 8\}$ respectively, with $2n$ layers for each feature map size. The numbers of filters are $\{16, 32, 64\}$ respectively. The subsampling is performed by convolutions with a stride of 2. The network ends with a global average pooling, a 10-way fully-connected layer, and softmax. There are totally $6n$+2 stacked weighted layers. The following table summarizes the architecture:

output map size	32×32	16×16	8×8
# layers	1+2n	2n	2n
# filters	16	32	64

When shortcut connections are used, they are connected to the pairs of 3×3 layers (totally $3n$ shortcuts). On this dataset we use identity shortcuts in all cases (*i.e.*, option A), so our residual models have exactly the same depth, width, and number of parameters as the plain counterparts.

method			error (%)
Maxout [9]			9.38
NIN [25]			8.81
DSN [24]			8.22
	# layers	# params	
FitNet [34]	19	2.5M	8.39
Highway [41, 42]	19	2.3M	7.54 (7.72±0.16)
Highway [41, 42]	32	1.25M	8.80
ResNet	20	0.27M	8.75
ResNet	32	0.46M	7.51
ResNet	44	0.66M	7.17
ResNet	56	0.85M	6.97
ResNet	110	1.7M	**6.43** (6.61±0.16)
ResNet	1202	19.4M	7.93

Table 6. Classification error on the **CIFAR-10** test set. All methods are with data augmentation. For ResNet-110, we run it 5 times and show "best (mean±std)" as in [42].

We use a weight decay of 0.0001 and momentum of 0.9, and adopt the weight initialization in [12] and BN [16] but with no dropout. These models are trained with a mini-batch size of 128 on two GPUs. We start with a learning rate of 0.1, divide it by 10 at 32k and 48k iterations, and terminate training at 64k iterations, which is determined on a 45k/5k train/val split. We follow the simple data augmentation in [24] for training: 4 pixels are padded on each side, and a 32×32 crop is randomly sampled from the padded image or its horizontal flip. For testing, we only evaluate the single view of the original 32×32 image.

We compare $n = \{3, 5, 7, 9\}$, leading to 20, 32, 44, and 56-layer networks. Fig. 6 (left) shows the behaviors of the plain nets. The deep plain nets suffer from increased depth, and exhibit higher training error when going deeper. This phenomenon is similar to that on ImageNet (Fig. 4, left) and on MNIST (see [41]), suggesting that such an optimization difficulty is a fundamental problem.

Fig. 6 (middle) shows the behaviors of ResNets. Also similar to the ImageNet cases (Fig. 4, right), our ResNets manage to overcome the optimization difficulty and demonstrate accuracy gains when the depth increases.

We further explore $n = 18$ that leads to a 110-layer ResNet. In this case, we find that the initial learning rate of 0.1 is slightly too large to start converging[5]. So we use 0.01 to warm up the training until the training error is below 80% (about 400 iterations), and then go back to 0.1 and continue training. The rest of the learning schedule is as done previously. This 110-layer network converges well (Fig. 6, middle). It has *fewer* parameters than other deep and thin

[5]With an initial learning rate of 0.1, it starts converging (<90% error) after several epochs, but still reaches similar accuracy.

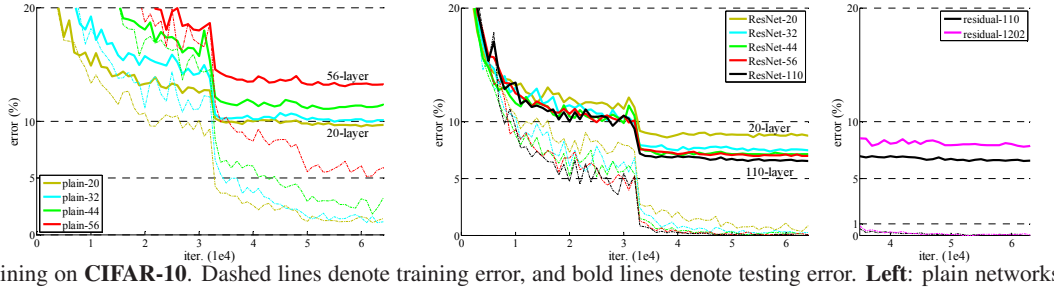

Figure 6. Training on **CIFAR-10**. Dashed lines denote training error, and bold lines denote testing error. **Left**: plain networks. The error of plain-110 is higher than 60% and not displayed. **Middle**: ResNets. **Right**: ResNets with 110 and 1202 layers.

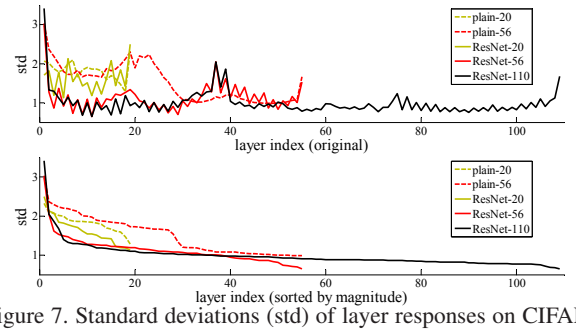

Figure 7. Standard deviations (std) of layer responses on CIFAR-10. The responses are the outputs of each 3×3 layer, after BN and before nonlinearity. **Top**: the layers are shown in their original order. **Bottom**: the responses are ranked in descending order.

networks such as FitNet [34] and Highway [41] (Table 6), yet is among the state-of-the-art results (6.43%, Table 6).

Analysis of Layer Responses. Fig. 7 shows the standard deviations (std) of the layer responses. The responses are the outputs of each 3×3 layer, after BN and before other nonlinearity (ReLU/addition). For ResNets, this analysis reveals the response strength of the residual functions. Fig. 7 shows that ResNets have generally smaller responses than their plain counterparts. These results support our basic motivation (Sec.3.1) that the residual functions might be generally closer to zero than the non-residual functions. We also notice that the deeper ResNet has smaller magnitudes of responses, as evidenced by the comparisons among ResNet-20, 56, and 110 in Fig. 7. When there are more layers, an individual layer of ResNets tends to modify the signal less.

Exploring Over 1000 layers. We explore an aggressively deep model of over 1000 layers. We set $n = 200$ that leads to a 1202-layer network, which is trained as described above. Our method shows *no optimization difficulty*, and this 10^3-layer network is able to achieve *training error* <0.1% (Fig. 6, right). Its test error is still fairly good (7.93%, Table 6).

But there are still open problems on such aggressively deep models. The testing result of this 1202-layer network is worse than that of our 110-layer network, although both have similar training error. We argue that this is because of overfitting. The 1202-layer network may be unnecessarily large (19.4M) for this small dataset. Strong regularization such as maxout [9] or dropout [13] is applied to obtain the best results ([9, 25, 24, 34]) on this dataset. In this paper, we use no maxout/dropout and just simply impose regularization via deep and thin architectures by design, without distracting from the focus on the difficulties of optimization. But combining with stronger regularization may improve results, which we will study in the future.

training data	07+12	07++12
test data	VOC 07 test	VOC 12 test
VGG-16	73.2	70.4
ResNet-101	**76.4**	**73.8**

Table 7. Object detection mAP (%) on the PASCAL VOC 2007/2012 test sets using **baseline** Faster R-CNN. See also appendix for better results.

metric	mAP@.5	mAP@[.5, .95]
VGG-16	41.5	21.2
ResNet-101	**48.4**	**27.2**

Table 8. Object detection mAP (%) on the COCO validation set using **baseline** Faster R-CNN. See also appendix for better results.

4.3. Object Detection on PASCAL and MS COCO

Our method has good generalization performance on other recognition tasks. Table 7 and 8 show the object detection baseline results on PASCAL VOC 2007 and 2012 [5] and COCO [26]. We adopt *Faster R-CNN* [32] as the detection method. Here we are interested in the improvements of replacing VGG-16 [40] with ResNet-101. The detection implementation (see appendix) of using both models is the same, so the gains can only be attributed to better networks. Most remarkably, on the challenging COCO dataset we obtain a 6.0% increase in COCO's standard metric (mAP@[.5, .95]), which is a 28% relative improvement. This gain is solely due to the learned representations.

Based on deep residual nets, we won the 1st places in several tracks in ILSVRC & COCO 2015 competitions: ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation. The details are in the appendix.

References

[1] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[2] C. M. Bishop. *Neural networks for pattern recognition*. Oxford university press, 1995.

[3] W. L. Briggs, S. F. McCormick, et al. *A Multigrid Tutorial*. Siam, 2000.

[4] K. Chatfield, V. Lempitsky, A. Vedaldi, and A. Zisserman. The devil is in the details: an evaluation of recent feature encoding methods. In *BMVC*, 2011.

[5] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman. The Pascal Visual Object Classes (VOC) Challenge. *IJCV*, pages 303–338, 2010.

[6] R. Girshick. Fast R-CNN. In *ICCV*, 2015.

[7] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *CVPR*, 2014.

[8] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, 2010.

[9] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio. Maxout networks. *arXiv:1302.4389*, 2013.

[10] K. He and J. Sun. Convolutional neural networks at constrained time cost. In *CVPR*, 2015.

[11] K. He, X. Zhang, S. Ren, and J. Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. In *ECCV*, 2014.

[12] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, 2015.

[13] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv:1207.0580*, 2012.

[14] S. Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Diploma thesis, TU Munich*, 1991.

[15] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[16] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

[17] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *TPAMI*, 33, 2011.

[18] H. Jegou, F. Perronnin, M. Douze, J. Sanchez, P. Perez, and C. Schmid. Aggregating local image descriptors into compact codes. *TPAMI*, 2012.

[19] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv:1408.5093*, 2014.

[20] A. Krizhevsky. Learning multiple layers of features from tiny images. *Tech Report*, 2009.

[21] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.

[22] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1989.

[23] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient backprop. In *Neural Networks: Tricks of the Trade*, pages 9–50. Springer, 1998.

[24] C.-Y. Lee, S. Xie, P. Gallagher, Z. Zhang, and Z. Tu. Deeply-supervised nets. *arXiv:1409.5185*, 2014.

[25] M. Lin, Q. Chen, and S. Yan. Network in network. *arXiv:1312.4400*, 2013.

[26] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft COCO: Common objects in context. In *ECCV*. 2014.

[27] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *CVPR*, 2015.

[28] G. Montúfar, R. Pascanu, K. Cho, and Y. Bengio. On the number of linear regions of deep neural networks. In *NIPS*, 2014.

[29] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *ICML*, 2010.

[30] F. Perronnin and C. Dance. Fisher kernels on visual vocabularies for image categorization. In *CVPR*, 2007.

[31] T. Raiko, H. Valpola, and Y. LeCun. Deep learning made easier by linear transformations in perceptrons. In *AISTATS*, 2012.

[32] S. Ren, K. He, R. Girshick, and J. Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *NIPS*, 2015.

[33] B. D. Ripley. *Pattern recognition and neural networks*. Cambridge university press, 1996.

[34] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio. Fitnets: Hints for thin deep nets. In *ICLR*, 2015.

[35] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *arXiv:1409.0575*, 2014.

[36] A. M. Saxe, J. L. McClelland, and S. Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv:1312.6120*, 2013.

[37] N. N. Schraudolph. Accelerated gradient descent by factor-centering decomposition. Technical report, 1998.

[38] N. N. Schraudolph. Centering neural network gradient factors. In *Neural Networks: Tricks of the Trade*, pages 207–226. Springer, 1998.

[39] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. In *ICLR*, 2014.

[40] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[41] R. K. Srivastava, K. Greff, and J. Schmidhuber. Highway networks. *arXiv:1505.00387*, 2015.

[42] R. K. Srivastava, K. Greff, and J. Schmidhuber. Training very deep networks. *1507.06228*, 2015.

[43] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *CVPR*, 2015.

[44] R. Szeliski. Fast surface interpolation using hierarchical basis functions. *TPAMI*, 1990.

[45] R. Szeliski. Locally adapted hierarchical basis preconditioning. In *SIGGRAPH*, 2006.

[46] T. Vatanen, T. Raiko, H. Valpola, and Y. LeCun. Pushing stochastic gradient towards second-order methods–backpropagation learning with transformations in nonlinearities. In *Neural Information Processing*, 2013.

[47] A. Vedaldi and B. Fulkerson. VLFeat: An open and portable library of computer vision algorithms, 2008.

[48] W. Venables and B. Ripley. Modern applied statistics with s-plus. 1999.

[49] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional neural networks. In *ECCV*, 2014.

Appendix B: Sample Code

Model Building

Training ResNet50 Model

```
# Define the paths to your dataset
train_dir = '/content/drive/MyDrive/Processed_Data/train'
validation_dir = '/content/drive/MyDrive/Processed_Data/val'

# Define the image dimensions and batch size
image_size = (224, 224)
batch_size = 32

# Create an ImageDataGenerator instance
train_datagen = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    preprocessing_function=preprocess_input
)

validation_datagen = ImageDataGenerator(preprocessing_function=preprocess_input)

# Load and preprocess the training dataset
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=image_size,
    batch_size=batch_size,
    class_mode='categorical'
)

# Load and preprocess the validation dataset
validation_generator = validation_datagen.flow_from_directory(
    validation_dir,
    target_size=image_size,
    batch_size=batch_size,
    class_mode='categorical'
)

from tensorflow.keras.applications import ResNet50
from tensorflow.keras import layers
from tensorflow.keras.models import Model
```

```
# Load the ResNet50 model without the top layer
base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
# Add a flatten layer
x = base_model.output
x = layers.Flatten()(x)
x = layers.Dropout(0.5)(x)
# Add your own classification layers on top
predictions = layers.Dense(5, activation='softmax')(x)

# Create the final model
model = Model(inputs=base_model.input, outputs=predictions)

model.summary()

# Freeze the base model layers
for layer in base_model.layers:
    layer.trainable = False

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Set the number of training and validation steps
train_steps_per_epoch = train_generator.n // batch_size
validation_steps = validation_generator.n // batch_size

# Train the model
history = model.fit(
    train_generator,
    steps_per_epoch=train_steps_per_epoch,
    epochs=50,
    verbose=1,
    validation_data=validation_generator,
    validation_steps=validation_steps
)

h=history.history
h.keys()

import matplotlib.pyplot as plt
plt.plot(h['val_accuracy'],c='green')
plt.plot(h['val_loss'],c='red')
plt.plot(h['accuracy'],c='blue')
plt.xlabel("Epochs")
```

```
plt.ylabel("Value")

legend = plt.legend(['val_accuracy', 'val_loss', 'train_accuracy'], loc='upper right')
legend.get_texts()[0].set_color('green')
legend.get_texts()[1].set_color('red')
legend.get_texts()[2].set_color('blue')

plt.show()

acc= model.evaluate_generator(validation_generator)[1]
print(f"The accuracy of the model is {acc*100}% ")

model.save("/content/drive/MyDrive/best_model_v8.h5")
```

Conversion from .H5 to Tflite Quantized Model

```
import os
import tensorflow as tf
import numpy as np
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications.resnet50 import preprocess_input

# configuration parameters
TEST_DATA_DIR = '/content/drive/MyDrive/Processed_Data/val'
MODEL_PATH = "/content/drive/MyDrive/best_model_v8.h5"
TEST_SAMPLES = 453
image_size = 224

# create an image generator with a batch size of 1
test_data_generator = ImageDataGenerator(preprocessing_function=preprocess_input)
validation_generator = test_data_generator.flow_from_directory(
    "/content/drive/MyDrive/Processed_Data/val",
    target_size=(image_size, image_size),
    batch_size=1,
    class_mode='categorical',# classes to predict
    shuffle =False
    )

def represent_data_gen():
    """ it yields an image one by one """
    for ind in range(len(validation_generator.filenames)):
        img_with_label = validation_generator.next() # it returns (image and label) tuple
        yield [np.array(img_with_label[0], dtype=np.float32, ndmin=2)] # return only image
```

```
keras_model = tf.keras.models.load_model("/content/drive/MyDrive/best_model_v8.h5")
converter = tf.lite.TFLiteConverter.from_keras_model(keras_model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]

# assign the custom image generator fn to representative_dataset
converter.representative_dataset = represent_data_gen

# Ensure that if any ops can't be quantized, the converter throws an error
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
# Set the input and output tensors to uint8 (APIs added in r2.3)
converter.inference_input_type = tf.uint8
converter.inference_output_type = tf.uint8
tflite_model = converter.convert()

# write the model to a tflite file as binary file
tflite_both_quant_file = "/content/drive/MyDrive/best_model_v8_quant_uint.tflite"
with open(tflite_both_quant_file, "wb") as f:
    f.write(tflite_model)
```

Testing validation accuracy of quantized Tflite model

```
import tensorflow as tf
import numpy as np
import os

# Load the TFLite model
model_path = '/content/drive/MyDrive/best_model_v8_quant_uint.tflite'
interpreter = tf.lite.Interpreter(model_path=model_path)
interpreter.allocate_tensors()

# Load the labels
labels_path = '/content/drive/MyDrive/labels.txt'
with open(labels_path, 'r') as f:
    labels = f.read().splitlines()

# Define the image size expected by the model
input_details = interpreter.get_input_details()[0]
input_shape = input_details['shape'][1:3]

# Function to preprocess the image
def preprocess_image(image_path):
    img = tf.keras.preprocessing.image.load_img(image_path, target_size=input_shape)
    img_array = tf.keras.preprocessing.image.img_to_array(img)
```

```
    img_array = np.expand_dims(img_array, axis=0)
    img_array = img_array.astype(np.uint8)

    return img_array

# Function to make predictions
def predict(image_path):
    input_data = preprocess_image(image_path)

    # Set the input tensor
    input_index = input_details['index']
    interpreter.set_tensor(input_index, input_data)

    # Run the inference
    interpreter.invoke()

    # Get the output tensor
    output_index = interpreter.get_output_details()[0]['index']
    output = interpreter.get_tensor(output_index)

    # Get the predicted class label
    predicted_class = labels[np.argmax(output)]

    return predicted_class

# Function to evaluate the model on the validation dataset
def evaluate_model(validation_dir):
    num_correct = 0
    total_samples = 0

    # Iterate over the subfolders (class labels)
    for label in labels:
        label_dir = os.path.join(validation_dir, label)
        images = os.listdir(label_dir)

        # Iterate over the images in each subfolder
        for image_file in images:
            image_path = os.path.join(label_dir, image_file)
            predicted_label = predict(image_path)

            # Check if the predicted label matches the true label
            if predicted_label == label:
                num_correct += 1
```

```
            total_samples += 1

    # Calculate the validation accuracy
    accuracy = num_correct / total_samples

    return accuracy

# Path to the validation dataset directory
validation_dir = '/content/drive/MyDrive/Processed_Data/val'

# Evaluate the model
validation_accuracy = evaluate_model(validation_dir)
print('Validation Accuracy:', validation_accuracy*100,"%")
```

Testing an the Tflite model with the image

```
import numpy as np
import tensorflow as tf

# Load the float32 TensorFlow Lite model
model_path = '/content/drive/MyDrive/best_model_v8_quant_uint.tflite'
interpreter = tf.lite.Interpreter(model_path=model_path)
interpreter.allocate_tensors()

# Load the class labels
labels_path = '/content/drive/MyDrive/labels.txt'
with open(labels_path, 'r') as f:
    class_labels = [line.strip() for line in f.readlines()]

# Preprocess the image
def preprocess_image(image):
    # Resize the image to match the input requirements of the model
    input_shape = interpreter.get_input_details()[0]['shape'][1:3]
    image = tf.image.resize(image, input_shape)

    image = tf.cast(image, tf.uint8)

    # Add batch and channel dimensions to the image
    image = tf.expand_dims(image, axis=0)

    return image

# Load and preprocess the image
```

```
image_path =
'/content/drive/MyDrive/Processed_Data/val/tick/tick-0_jpg.rf.a4e81dae549d4d4f6b878369158e
15dd.jpg'
image = tf.io.read_file(image_path)
image = tf.image.decode_jpeg(image, channels=3)
preprocessed_image = preprocess_image(image)

# Run inference on the preprocessed image
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

interpreter.set_tensor(input_details[0]['index'], preprocessed_image)
interpreter.invoke()
output_tensor = interpreter.get_tensor(output_details[0]['index'])
predictions = np.squeeze(output_tensor)

# Get the predicted class label
predicted_class_index = np.argmax(predictions)
predicted_class = class_labels[predicted_class_index]

print("Predicted class:", predicted_class)
```

App Development in Kotlin

MainActivity.kt

```
package com.example.imageclassification

import android.app.Activity
import android.content.Intent
import android.content.pm.PackageManager
import android.graphics.Bitmap
import android.net.Uri
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.preference.PreferenceManager
import android.provider.MediaStore
import android.util.Log
import android.view.View
import android.widget.Button
import android.widget.ImageView
import android.widget.TextView
```

```
import android.widget.Toast
import com.example.imageclassification.ml.BestModelV5QuantUint
import org.tensorflow.lite.DataType
import org.tensorflow.lite.support.image.TensorImage
import org.tensorflow.lite.support.tensorbuffer.TensorBuffer
import java.util.jar.Manifest
import android.os.Handler
import android.os.Looper

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_start)
        Handler(Looper.getMainLooper()).postDelayed({
            val intent = Intent(this, middle_activity::class.java)
            startActivity(intent)
            finish() // Optional: If you want to close the current activity after starting the next one.
        }, 5000) // 2000 milliseconds = 2 seconds
        }
    }
```

middle_activity.kt

```
package com.example.imageclassification

import android.app.Activity
import android.content.Intent
import android.content.pm.PackageManager
import android.graphics.Bitmap
import android.net.Uri
import android.os.Bundle
import android.provider.MediaStore
import android.util.Log
import android.view.View
import android.widget.*
import androidx.appcompat.app.AppCompatActivity
import com.example.imageclassification.ml.BestModelV5QuantUint
import org.tensorflow.lite.DataType
import org.tensorflow.lite.support.image.TensorImage
import org.tensorflow.lite.support.tensorbuffer.TensorBuffer

class middle_activity : AppCompatActivity() {
```

```
    lateinit var select_image_button : Button
    lateinit var make_prediction : Button
    lateinit var img_view : ImageView
    lateinit var text_view : TextView
    lateinit var bitmap: Bitmap
    lateinit var camerabtn : Button
    lateinit var view_remedy: Button


    var passing_string = String()

    public fun checkandGetpermissions(){
        if(checkSelfPermission(android.Manifest.permission.CAMERA) ==
PackageManager.PERMISSION_DENIED){
            requestPermissions(arrayOf(android.Manifest.permission.CAMERA), 100)
        }
        else{
            Toast.makeText(this, "Camera permission granted", Toast.LENGTH_SHORT).show()
        }
    }

    override fun onRequestPermissionsResult(
        requestCode: Int,
        permissions: Array<out String>,
        grantResults: IntArray
    ) {
        super.onRequestPermissionsResult(requestCode, permissions, grantResults)
        if(requestCode == 100){
            if(grantResults[0] == PackageManager.PERMISSION_GRANTED)
            {
                Toast.makeText(this, "Camera permission granted", Toast.LENGTH_SHORT).show()
            }
            else{
                Toast.makeText(this, "Permission Denied", Toast.LENGTH_SHORT).show()
            }
        }
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        select_image_button = findViewById(R.id.button)
```

```
        make_prediction = findViewById(R.id.button2)
        img_view = findViewById(R.id.imageView2)
        text_view = findViewById(R.id.textView)
        camerabtn = findViewById<Button>(R.id.camerabtn)
        view_remedy = findViewById(R.id.button3)
        view_remedy.visibility = Button.GONE

        // handling permissions
        checkandGetpermissions()

        val labels = application.assets.open("labels.txt").bufferedReader().use { it.readText()
}.split("\n")



        select_image_button.setOnClickListener(View.OnClickListener {
            Log.d("mssg", "button pressed")
            var intent : Intent = Intent(Intent.ACTION_GET_CONTENT)
            intent.type = "image/*"

            startActivityForResult(intent, 250)
        })

        make_prediction.setOnClickListener(View.OnClickListener {
            var resized = Bitmap.createScaledBitmap(bitmap, 224, 224, true)
            val model = BestModelV5QuantUint.newInstance(this)

            var tbuffer = TensorImage.fromBitmap(resized)
            var byteBuffer = tbuffer.buffer

// Creates inputs for reference.
            val inputFeature0 = TensorBuffer.createFixedSize(intArrayOf(1, 224, 224, 3),
DataType.UINT8)
            inputFeature0.loadBuffer(byteBuffer)

// Runs model inference and gets result.
            val outputs = model.process(inputFeature0)
            val outputFeature0 = outputs.outputFeature0AsTensorBuffer

            var max = getMax(outputFeature0.floatArray)
```

```
            text_view.setText(labels[max])
            passing_string = labels[max]
            //Log.d("Tag", passing_string)
            if(passing_string!="None")
                view_remedy.visibility = Button.VISIBLE
            else
                view_remedy.visibility = Button.GONE

// Releases model resources if no longer used.
            model.close()
        })

        camerabtn.setOnClickListener(View.OnClickListener {
            var camera : Intent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)
            startActivityForResult(camera, 200)
        })
        view_remedy.setOnClickListener {
            //val inputValue = passing_string // The variable you want to pass
            val intent = Intent(this, NewActivity_Remedies::class.java)
            intent.putExtra("inputValue", passing_string.toString()) // Pass the variable to the next
activity
            startActivity(intent)
        }




    }

    override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
        super.onActivityResult(requestCode, resultCode, data)

        if(requestCode == 250){
            img_view.setImageURI(data?.data)

            var uri : Uri?= data?.data
            bitmap = MediaStore.Images.Media.getBitmap(this.contentResolver, uri)
        }
        else if(requestCode == 200 && resultCode == Activity.RESULT_OK){
            bitmap = data?.extras?.get("data") as Bitmap
            img_view.setImageBitmap(bitmap)
        }
```

```
    }

    fun getMax(arr:FloatArray) : Int{
        var ind = 0;
        var min = 0.0f;

        for(i in 0..4)
        {
            if(arr[i] > min)
            {
                min = arr[i]
                ind = i;
            }
        }
        return ind
    }
}
```

NewActivity_Remedies.kt

```
package com.example.imageclassification

import android.content.res.AssetManager
import android.graphics.Color
import android.graphics.Typeface
import android.graphics.drawable.GradientDrawable
import android.os.Bundle
import android.webkit.WebView
import android.widget.TextView
import androidx.appcompat.app.AppCompatActivity
import android.os.Handler
import android.text.Spannable
import android.text.SpannableStringBuilder
import android.text.style.AbsoluteSizeSpan
import android.text.style.StyleSpan
import android.util.TypedValue
import android.view.ViewGroup
import android.widget.LinearLayout
import android.widget.ProgressBar
import androidx.core.content.ContextCompat
import androidx.core.view.marginBottom
import java.io.BufferedReader
import java.io.IOException
```

```
import java.io.InputStream
import java.io.InputStreamReader

class NewActivity_Remedies : AppCompatActivity() {
    lateinit var text_view2 : TextView
    lateinit var progress_bar:ProgressBar
    lateinit var remedyTextView:TextView
    lateinit var containerLayout: LinearLayout
    lateinit var assetManager: AssetManager



    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.newactivity_remedies)
        progress_bar = findViewById(R.id.progressBar3)

        progress_bar.visibility = ProgressBar.VISIBLE
        containerLayout = findViewById(R.id.containerLayout)
        assetManager = applicationContext.assets


        Handler().postDelayed({
            // Display the actual content after 1 seconds

            showContent()
        }, 1000)

    }

    private fun showContent() {

        progress_bar.visibility = ProgressBar.GONE

        val inputValue = intent.getStringExtra("inputValue").toString()

        text_view2 = findViewById(R.id.textView2)
        text_view2.setText("Remedies: "+inputValue)

        // remedyTextView = findViewById(R.id.textView5)



        val fileName = when (inputValue) {
```

```
        "Bee" -> "bee_remedies.txt"
        "Mosquito" -> "mosquito_remedies.txt"
        "Spider" -> "spider_remedies.txt"
        "Tick" -> "tick_remedies.txt"
        else -> ""
    }

    if (fileName.isNotEmpty()) {
        val remedies = loadTextFromFile(fileName)
        displayRemedies(remedies)
    }
}


private fun loadTextFromFile(fileName: String): List<String> {
    val remedies = mutableListOf<String>()

    try {
        val inputStream = assetManager.open(fileName)
        val inputStreamReader = InputStreamReader(inputStream)
        val bufferedReader = BufferedReader(inputStreamReader)

        var line: String?
        while (bufferedReader.readLine().also { line = it } != null) {
            remedies.add(line ?: "")
        }

        bufferedReader.close()
    } catch (e: IOException) {
        e.printStackTrace()
    }

    return remedies
}

private fun displayRemedies(remedies: List<String>) {
    val layoutParams = LinearLayout.LayoutParams(
        LinearLayout.LayoutParams.MATCH_PARENT,
        LinearLayout.LayoutParams.WRAP_CONTENT
    )
    layoutParams.setMargins(0, 17, 0, 0)

    var currentTextView: TextView? = null
```

```
        for (remedy in remedies) {
            if (remedy.isNotBlank()) {
                if (remedy.startsWith("*")) {
                    // Create a new TextView for the next remedy
                    currentTextView = TextView(this)
                    currentTextView.typeface = Typeface.create(Typeface.SERIF, Typeface.NORMAL)
                    currentTextView.setTextColor(Color.WHITE)
                   // currentTextView.setTextSize(TypedValue.COMPLEX_UNIT_SP, 16f)
                    currentTextView.layoutParams = layoutParams
                    currentTextView.background = getRoundedBackground()
                    currentTextView.setPadding(13,13,13,13)
                    containerLayout.addView(currentTextView)
                    // Remove the leading "*" from the remedy text
                    val heading = remedy.substring(1).trim()
                    val formattedText = SpannableStringBuilder(heading)
                    formattedText.setSpan(StyleSpan(Typeface.BOLD), 0, heading.length,
Spannable.SPAN_EXCLUSIVE_EXCLUSIVE)
                    currentTextView.text = formattedText
                    currentTextView.setTextSize(TypedValue.COMPLEX_UNIT_SP, 20f)
                } else {
                    // Append the explanation to the current TextView
                    val explanation = SpannableStringBuilder("\n$remedy")
                    explanation.setSpan(
                        AbsoluteSizeSpan(16, true), // Set the size to 16sp
                        0,
                        explanation.length,
                        Spannable.SPAN_EXCLUSIVE_EXCLUSIVE
                    )
                    currentTextView?.append(explanation)

                }
            }
            else{
                currentTextView?.append("\n")
            }
        }
    }


    private fun formatRemedy(remedyNumber: Int, remedyText: String): SpannableStringBuilder {
        val formattedText = SpannableStringBuilder()
        val boldStyle = StyleSpan(Typeface.BOLD)

        val heading = "Remedy $remedyNumber:"
```

```
        formattedText.append(heading)
        formattedText.append("\n")
        formattedText.setSpan(boldStyle, 0, heading.length,
Spannable.SPAN_EXCLUSIVE_EXCLUSIVE)
        formattedText.append(remedyText)
        formattedText.append("\n")

        return formattedText
    }
    private fun getRoundedBackground(): GradientDrawable {
        val radius = 15f // Adjust the value to change the corner radius
        val shape = GradientDrawable()
        shape.cornerRadius = radius
        shape.setColor(Color.parseColor("#41046e"))
        return shape
    }
}
```

Appendix C: CO-PO And CO-PSO Mapping

COURSE OUTCOMES:

After completion of the course the student will be able to

SL. NO	DESCRIPTION	Blooms' Taxonomy Level
CO1	Identify technically and economically feasible problems (Cognitive Knowledge Level: Apply)	Level 3: Apply
CO2	Identify and survey the relevant literature for getting exposed to related solutions and get familiarized with software development processes (Cognitive Knowledge Level: Apply)	Level 3: Apply
CO3	Perform requirement analysis, identify design methodologies and develop adaptable & reusable solutions of minimal complexity by using modern tools & advanced programming techniques (Cognitive Knowledge Level: Apply)	Level 3: Apply
CO4	Prepare technical report and deliver presentation (Cognitive Knowledge Level: Apply)	Level 3: Apply
CO5	Apply engineering and management principles to achieve the goal of the project (Cognitive Knowledge Level: Apply)	Level 3: Apply

CO-PO AND CO-PSO MAPPING

	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO 1	PSO 2	PSO3
CO1	3	3	3	3		2	2	3	2	2	2	3	2	2	2
CO2	3	3	3	3	3	2		3	2	3	2	3	2	2	2
CO3	3	3	3	3	3	2	2	3	2	2	2	3			2
CO4	2	3	2	2	2			3	3	3	2	3	2	2	2
CO5	3	3	3	2	2	2	2	3	2		2	3	2	2	2

3/2/1: high/medium/low

JUSTIFICATIONS FOR CO-PO MAPPING

MAPPING	LOW/ MEDIUM/ HIGH	JUSTIFICATION
100003/CS622T.1-PO1	**HIGH**	Identify technically and economically feasible problems by applying the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
100003/CS622T.1-PO2	**HIGH**	Identify technically and economically feasible problems by analysing complex engineering problems reaching substantiated conclusions using first principles of mathematics.
100003/CS622T.1-PO3	**HIGH**	Design solutions for complex engineering problems by identifying technically and economically feasible problems.
100003/CS622T.1-PO4	**HIGH**	Identify technically and economically feasible problems by analysis and interpretation of data.
100003/CS622T.1-PO6	**MEDIUM**	Responsibilities relevant to the professional engineering practice by identifying the problem.
100003/CS622T.1-PO7	**MEDIUM**	Identify technically and economically feasible problems by understanding the impact of the professional engineering solutions.
100003/CS622T.1-PO8	**HIGH**	Apply ethical principles and commit to professional ethics to identify technically and economically feasible problems.
100003/CS622T.1-PO9	**MEDIUM**	Identify technically and economically feasible problems by working as a team.
100003/CS622T.1-PO10	**MEDIUM**	Communicate effectively with the engineering community by identifying technically and economically feasible problems.
100003/CS622T.1-P011	**MEDIUM**	Demonstrate knowledge and understanding of engineering and management principles by selecting the technically and economically feasible problems.
100003/CS622T.1-PO12	**HIGH**	Identify technically and economically feasible problems for long term learning.
100003/CS622T.1-PSO1	**MEDIUM**	Ability to identify, analyze and design solutions to identify technically and economically feasible problems.
100003/CS622T.1-PSO2	**MEDIUM**	By designing algorithms and applying standard practices in software project development and Identifying technically and economically feasible problems.
100003/CS622T.1-PSO3	**MEDIUM**	Fundamentals of computer science in competitive research can be applied to Identify technically and economically feasible problems.
100003/CS622T.2-PO1	**HIGH**	Identify and survey the relevant by applying the knowledge of mathematics, science, engineering fundamentals.

100003/CS622T.2-PO2	**HIGH**	Identify, formulate, review research literature, and analyze complex engineering problems get familiarized with software development processes.
100003/CS622T.2-PO3	**HIGH**	Design solutions for complex engineering problems and design based on the relevant literature.
100003/CS622T.2-PO4	**HIGH**	Use research-based knowledge including design of experiments based on relevant literature.
100003/CS622T.2-PO5	**HIGH**	Identify and survey the relevant literature for getting exposed to related solutions and get familiarized with software development processes by using modern tools.
100003/CS622T.2-PO6	**MEDIUM**	Create, select, and apply appropriate techniques, resources, by identifying and surveying the relevant literature.
100003/CS622T.2-PO8	**HIGH**	Apply ethical principles and commit to professional ethics based on the relevant literature.
100003/CS622T.2-PO9	**MEDIUM**	Identify and survey the relevant literature as a team.
100003/CS622T.2-PO10	**HIGH**	Identify and survey the relevant literature for a good communication to the engineering fraternity.
100003/CS622T.2-PO11	**MEDIUM**	Identify and survey the relevant literature to demonstrate knowledge and understanding of engineering and management principles.
100003/CS622T.2-PO12	**HIGH**	Identify and survey the relevant literature for independent and lifelong learning.
100003/CS622T.2-PSO1	**MEDIUM**	Design solutions for complex engineering problems by Identifying and survey the relevant literature.
100003/CS622T.2-PSO2	**MEDIUM**	Identify and survey the relevant literature for acquiring programming efficiency by designing algorithms and applying standard practices.
100003/CS622T.2-PSO3	**MEDIUM**	Identify and survey the relevant literature to apply the fundamentals of computer science in competitive research.
100003/CS622T.3-PO1	**HIGH**	Perform requirement analysis, identify design methodologies by using modern tools & advanced programming techniques and by applying the knowledge of mathematics, science, engineering fundamentals.
100003/CS622T.3-PO2	**HIGH**	Identify, formulate, review research literature for requirement analysis, identify design methodologies and develop adaptable & reusable solutions.

100003/CS622T.3-PO3	**HIGH**	Design solutions for complex engineering problems and perform requirement analysis, identify design methodologies.
100003/CS622T.3-PO4	**HIGH**	Use research-based knowledge including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
100003/CS622T.3-PO5	**HIGH**	Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools.
100003/CS622T.3-PO6	**MEDIUM**	Perform requirement analysis, identify design methodologies and assess societal, health, safety, legal, and cultural issues.
100003/CS622T.3-PO7	**MEDIUM**	Understand the impact of the professional engineering solutions in societal and environmental contexts and Perform requirement analysis, identify design methodologies and develop adaptable & reusable solutions.
100003/CS622T.3-PO8	**HIGH**	Perform requirement analysis, identify design methodologies and develop adaptable & reusable solutions by applying ethical principles and commit to professional ethics.
100003/CS622T.3-PO9	**MEDIUM**	Function effectively as an individual, and as a member or leader in teams, and in multidisciplinary settings.
100003/CS622T.3-PO10	**MEDIUM**	Communicate effectively with the engineering community and with society at large to perform requirement analysis, identify design methodologies.
100003/CS622T.3-PO11	**MEDIUM**	Demonstrate knowledge and understanding of engineering requirement analysis by identifying design methodologies.
100003/CS622T.3-PO12	**HIGH**	Recognize the need for, and have the preparation and ability to engage in independent and lifelong learning in the broadest context of technological change by analysis, identify design methodologies and develop adaptable & reusable solutions.
100003/CS622T.3-PSO3	**MEDIUM**	The ability to apply the fundamentals of computer science in competitive research and prior to that perform requirement analysis, identify design methodologies.
100003/CS622T.4-PO1	**MEDIUM**	Prepare technical report and deliver presentation by applying the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
100003/CS622T.4-PO2	**HIGH**	Identify, formulate, review research literature, and analyze complex engineering problems by preparing technical report and deliver presentation.

100003/CS622T.4-PO3	**MEDIUM**	Prepare Design solutions for complex engineering problems and create technical report and deliver presentation.
100003/CS622T.4-PO4	**MEDIUM**	Use research-based knowledge including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions and prepare technical report and deliver presentation.
100003/CS622T.4-PO5	**MEDIUM**	Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools and Prepare technical report and deliver presentation.
100003/CS622T.4-PO8	**HIGH**	Prepare technical report and deliver presentation by applying ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
100003/CS622T.4-PO9	**HIGH**	Prepare technical report and deliver presentation effectively as an individual, and as a member or leader in teams, and in multidisciplinary settings.
100003/CS622T.4-PO10	**HIGH**	Communicate effectively with the engineering community and with society at large by prepare technical report and deliver presentation.
100003/CS622T.4-PO11	**MEDIUM**	Demonstrate knowledge and understanding of engineering and management principles and apply these to one's own work by prepare technical report and deliver presentation.
100003/CS622T.4-PO12	**HIGH**	Recognize the need for, and have the preparation and ability to engage in independent and lifelong learning in the broadest context of technological change by prepare technical report and deliver presentation.
100003/CS622T.4-PSO1	**MEDIUM**	Prepare a technical report and deliver presentation to identify, analyze and design solutions for complex engineering problems in multidisciplinary areas.
100003/CS622T.4-PSO2	**MEDIUM**	To acquire programming efficiency by designing algorithms and applying standard practices in software project development and to prepare technical report and deliver presentation.
100003/CS622T.4-PSO3	**MEDIUM**	To apply the fundamentals of computer science in competitive research and to develop innovative products to meet the societal needs by preparing technical report and deliver presentation.
100003/CS622T.5-PO1	**HIGH**	Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
100003/CS622T.5-PO2	**HIGH**	Identify, formulate, review research literature, and analyze complex engineering problems by applying engineering and management principles to achieve the goal of the project.

100003/CS622T.5-PO3	**HIGH**	Apply engineering and management principles to achieve the goal of the project and to design solutions for complex engineering problems and design system components or processes that meet the specified needs.
100003/CS622T.5-PO4	**MEDIUM**	Apply engineering and management principles to achieve the goal of the project and use research-based knowledge including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
100003/CS622T.5-PO5	**MEDIUM**	Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools and to apply engineering and management principles to achieve the goal of the project.
100003/CS622T.5-PO6	**MEDIUM**	Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal, and cultural issues and the consequent responsibilities by applying engineering and management principles to achieve the goal of the project.
100003/CS622T.5-PO7	**MEDIUM**	Understand the impact of the professional engineering solutions in societal and environmental contexts, and apply engineering and management principles to achieve the goal of the project.
100003/CS622T.5-PO8	**HIGH**	Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice and to use the engineering and management principles to achieve the goal of the project.
100003/CS622T.5-PO9	**MEDIUM**	Function effectively as an individual, and as a member or leader in teams, and in multidisciplinary settings and to apply engineering and management principles to achieve the goal of the project.
100003/CS622T.5-PO11	**MEDIUM**	Demonstrate knowledge and understanding of engineering and management principles and apply these to one's own work, as a member and leader in a team. Manage projects in multidisciplinary environments and to apply engineering and management principles to achieve the goal of the project.
100003/CS622T.5-PO12	**HIGH**	Recognize the need for, and have the preparation and ability to engage in independent and lifelong learning in the broadest context of technological change and to apply engineering and management principles to achieve the goal of the project.
100003/CS622T.5-PSO1	**MEDIUM**	The ability to identify, analyze and design solutions for complex engineering problems in multidisciplinary areas. Apply engineering and management principles to achieve the goal of the project.

100003/CS622T.5-PSO2	**MEDIUM**	The ability to acquire programming efficiency by designing algorithms and applying standard practices in software project development to deliver quality software products meeting the demands of the industry and to apply engineering and management principles to achieve the goal of the project.
100003/CS622T.5-PSO3	**MEDIUM**	The ability to apply the fundamentals of computer science in competitive research and to develop innovative products to meet the societal needs thereby evolving as an eminent researcher and entrepreneur and apply engineering and management principles to achieve the goal of the project.