B.Tech. BCSE497J - Project-I

INTELLIGENT VISUAL INSPECTION SYSTEM FOR MECHANICAL PART QUALITY ASSURANCE

Submitted in partial fulfillment of the requirements for the degree of

Bachelor of Technology

in

Computer Science & Engineering

by

21BCE2321	**RAHUL JAIN**
21BCE2328	**GAURI SHARMA**
21BCE0111	**OORJIT KOTNALA**

Under the Supervision of

SAIRABANU J

Associate Professor Sr.

School of Computer Science and Engineering (SCOPE)

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

November 2024

DECLARATION

I hereby declare that the project entitled "Intelligent Visual Inspection System For Mechanical Part Quality Assurance" submitted by me, for the award of the degree of *Bachelor of Technology in Computer Science and Engineering* to VIT is a record of bonafide work carried out by me under the supervision of Dr. Sairabanu J.

I further declare that the work reported in this project has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university.

Place : Vellore

Date : 12-11-24

Signature of the Candidate

CERTIFICATE

This is to certify that the project entitled “Intelligent Visual Inspection System For Mechanical Part Quality Assurance” submitted by Rahul Jain (21BCE2321),Gauri Sharma (21BCE2328) and Oorjit Kotnala (21BCE0111), **School of Computer Science and Engineering**, VIT, for the award of the degree of *Bachelor of Technology in Computer Science and Engineering*, is a record of bonafide work carried out by him / her under my supervision during Fall Semester 2024-2025, as per the VIT code of academic and research ethics.

The contents of this report have not been submitted and will not be submitted either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university. The project fulfill the requirements and regulations of the University and in my opinion meets the necessary standards for submission.

Place : Vellore
Date : 12-11-24

Signature of the Guide

Examiner(s)

Head of the Department

SCOPE

ACKNOWLEDGEMENTS

I am deeply grateful to the management of Vellore Institute of Technology (VIT) for providing me with the opportunity and resources to undertake this project. Their commitment to fostering a conducive learning environment has been instrumental in my academic journey. The support and infrastructure provided by VIT have enabled me to explore and develop my ideas to their fullest potential.

My sincere thanks to Dr. Ramesh Babu K, the Dean of the School of Computer Science and Engineering (SCOPE), for his unwavering support and encouragement. His leadership and vision have greatly inspired me to strive for excellence. The Dean's dedication to academic excellence and innovation has been a constant source of motivation for me. I appreciate his efforts in creating an environment that nurtures creativity and critical thinking.

I express my profound appreciation to Dr. Umadevi KS, the Head of Department of Software Systems her insightful guidance and continuous support. Her expertise and advice have been crucial in shaping the direction of my project. The Head of Department's commitment to fostering a collaborative and supportive atmosphere has greatly enhanced my learning experience. His/her constructive feedback and encouragement have been invaluable in overcoming challenges and achieving my project goals.

I am immensely thankful to my project supervisor, Dr. Sairabanu J for her dedicated mentorship and invaluable feedback. His/her patience, knowledge, and encouragement have been pivotal in the successful completion of this project. My supervisor's willingness to share his/her expertise and provide thoughtful guidance has been instrumental in refining my ideas and methodologies. His/her support has not only contributed to the success of this project but has also enriched my overall academic experience.

Thank you all for your contributions and support.

Gauri Sharma
Rahul Jain
Oorjit Kotnala

TABLE OF CONTENTS

List of Figures

List of Abbreviations

AI - Artificial Intelligence

CNN - Convolutional Neural Network

HITL - Human-in-the-Loop

MES - Manufacturing Execution System

ML - Machine Learning

FPS - Frames Per Second

API - Application Programming Interface

DFD - Data Flow Diagram

DRL - Deep Reinforcement Learning (if applicable, based on sample provided)

GPU - Graphics Processing Unit

ABSTRACT

The visual inspection of mechanical and industrial parts is essential to maintain rigorous quality standards across diverse manufacturing sectors. Traditionally, quality control depends heavily on human inspectors to identify and classify defects in products, but manual inspection is labor-intensive, prone to fatigue-induced errors, and subject to subjective inconsistencies. To address these limitations and enhance reliability, this project presents an AI-powered intelligent visual inspection system designed to automate quality control processes using advanced convolutional neural networks (CNNs) and transfer learning.

Inspired by the methodologies from "Automated Visual Inspection of Manufactured Parts Using Deep Learning" by Weiher et al., this project leverages CNNs and transfer learning techniques to effectively classify a wide range of defect types with high precision, even when available labeled data is limited. The CNN model's design, enhanced by transfer learning, allows it to learn from previously labeled data and adapt to new defect types, providing flexibility in its application across various inspection tasks. Moreover, to handle situations with low-confidence predictions, the project incorporates a Human-in-the-Loop (HITL) approach, enabling human inspectors to oversee and validate critical decisions, thus enhancing the system's adaptability and continuous improvement.

The system is engineered to operate in real time, facilitating continuous defect detection and instant feedback, a feature critical in high-throughput manufacturing environments. This real-time capability supports immediate quality adjustments on the production floor, minimizing defective products and reducing downtime. The integration of CNNs, transfer learning, and HITL provides a robust inspection solution that can adapt to variable defect types, lighting conditions, and part orientations.

This project not only aims to reduce the dependency on human labor but also to enhance operational efficiency and inspection accuracy, contributing to lower waste and higher quality standards. By incorporating AI-driven inspection into manufacturing workflows, the system enhances productivity, boosts product quality, and ultimately elevates customer satisfaction.

1. INTRODUCTION

1.1 Background

The visual inspection of mechanical parts is a critical process in ensuring that products meet industry quality standards before they reach consumers. Traditionally, this inspection has been performed manually, with human inspectors assessing parts for defects such as scratches, dents, or inconsistencies in shape and color. However, manual inspection has limitations, often resulting in errors due to fatigue, subjectivity, and inconsistency, which can lead to quality issues and increased operational costs.

With advancements in artificial intelligence (AI) and computer vision, automated visual inspection systems have gained significant traction in recent years. These systems utilize machine learning (ML) algorithms to analyze images or videos of products, detecting defects with high accuracy and consistency. Among these techniques, deep learning, particularly convolutional neural networks (CNNs), has shown promising results, as these models can be trained to identify even subtle imperfections that might be overlooked by the human eye. Moreover, automated inspection systems can operate continuously, providing real-time feedback and ensuring that defects are promptly identified and addressed.

1.2 Motivation

The quality of manufactured products directly impacts customer satisfaction and brand reputation. Traditional visual quality inspection of mechanical and industrial parts relies heavily on human inspectors who often perform subjective, inconsistent, and sometimes erroneous evaluations, particularly in high-volume production settings. As demands for precision and efficiency within manufacturing continue to increase, manual inspection methods have reached their limits.

This project is motivated by the potential of AI-driven solutions to revolutionize quality control, particularly for mechanical and industrial components. By incorporating computer vision and deep learning, an automated inspection system can precisely detect defects in various parts, even when their geometry or structure is complex. Such a system is expected to reduce the occurrence of defective items reaching consumers, minimize downtime, and reduce waste, thus fostering better operational efficiency and higher product quality.

1.3 Scope of the Project

The core objective of this project is to apply computer vision and artificial intelligence to perform visual quality inspection of mechanical and industrial parts autonomously. The project will begin by analyzing the challenges associated with manual visual inspections, including

human error, inconsistency, and inefficiency in high-volume production environments. By addressing these pain points, the project will demonstrate the advantages of an automated inspection system.

The development will include AI-based models capable of detecting defects such as surface anomalies, dimensional inaccuracies, and other critical issues with high precision and reliability. The system will incorporate defect classification, pattern recognition, and real-time feedback using ML algorithms. Additionally, the project aims to create a framework that integrates seamlessly with existing manufacturing workflows, ensuring smooth operation without disrupting production lines.

The system's performance will be evaluated across different manufacturing scenarios to ensure reliability and effectiveness. Ultimately, this work will develop a robust framework for visual quality inspection in manufacturing, enhancing product quality, reducing waste, and increasing operational efficiency.

2.PROJECT DESCRIPTION AND GOALS

2.1 Literature Review

The integration of machine learning (ML) and artificial intelligence (AI) into inspection and quality assurance processes has shown considerable promise, particularly for identifying complex defects in manufacturing. Recent research offers valuable insights into the strengths and limitations of these technologies across diverse industrial applications.

Machine Learning in Cybersecurity: Apruzzese et al. (2023) and Dasgupta et al. (2022) examine ML's role in cybersecurity, focusing on models that improve anomaly detection and threat recognition. Their findings highlight ML's ability to identify complex, non-linear patterns, a feature that is beneficial for quality inspection systems where defects can vary widely in shape and size.

AI and Cybersecurity Innovation: Kumar et al. (2023) discuss AI's impact on cybersecurity, specifically on real-time threat detection and AI-driven automation. Although cybersecurity differs from manufacturing, their insights on real-time decision-making and automation are applicable to real-time quality assurance in industrial settings, where quick response times are essential for maintaining quality.

AI for Threat Detection in Cybersecurity: Salih et al. (2021) explore deep learning's role in detecting cybersecurity threats, particularly for systems that require rapid response times. Their study provides foundational insights for designing visual inspection systems that require quick, automated defect detection in high-speed production environments.

Machine Learning for Visual Inspection: Zakaria et al. (2023) investigate the application of real-time ML for bridge inspections using edge devices, which demonstrates how ML can provide high-speed, consistent analyses in complex environments. Their findings are valuable for implementing edge devices in manufacturing, ensuring localized quality inspection without relying on centralized processing.

Unsupervised Deep Learning for Anomaly Detection: Zipfel et al. (2023) present a comparative study of unsupervised deep learning models for quality assurance in industry. Their research emphasizes the value of unsupervised techniques for defect detection, particularly when large labeled datasets are not available, which is relevant for cost-effective applications in manufacturing.

Deep CNNs and Transfer Learning for Visual Inspection: Weiher et al. (2023) explore deep CNNs and transfer learning for automated visual inspection, focusing on defect detection in manufacturing. This study is particularly pertinent to this project, as it demonstrates the adaptability of transfer learning in handling diverse defect types, forming a basis for the project's methodology.

Causal Deep Learning for Explainable Quality Inspection: Liang et al. (2023) propose causal deep learning models for quality inspection, which improve interpretability under challenging visual conditions. This approach is relevant to manufacturing, where explainable models increase trust and transparency in defect classifications.

Edge-Cloud Integration for Quality Assurance: Safari et al. (2023) examine edge-cloud integration for quality inspection, showcasing how distributed systems can handle high-volume, real-time defect detection. Their work provides insights into balancing computational loads across edge and cloud, supporting scalable inspection solutions for large-scale manufacturing.

Biological Vision Modeling in Fabric Defect Detection: Li et al. (2023) adapt principles of biological vision to create a defect detection model for fabrics. This innovative approach mimics human perception, enhancing model accuracy, which could be beneficial for detecting complex defect patterns in various industrial products.

Low-Rank Subspace Clustering for Integrated Circuit Verification: Tan et al. (2023) apply low-rank subspace clustering and high-frequency texture analysis for verifying integrated circuit images, demonstrating a method for detecting subtle defects. Their approach has potential applications in manufacturing, particularly for identifying fine, high-frequency defects in complex components.

2.2 Research Gap

Despite advances in automated inspection using CNNs, certain critical challenges and gaps remain unaddressed. Many studies and solutions focus on narrow applications, targeting specific defect types or product categories, which limits their broader applicability. Such solutions often lack the adaptability needed for complex mechanical parts that exhibit varied geometries and a wide range of defect types. Furthermore, real-world conditions pose additional challenges, as fluctuating lighting, background variations, and inconsistent part orientation can significantly impact model performance. Existing solutions often struggle to maintain reliability and accuracy under these dynamic conditions, limiting their effectiveness in real production settings.

This project seeks to address these gaps by developing a versatile CNN-based inspection system capable of detecting diverse defect types across a variety of mechanical components. By employing transfer learning, the system aims to achieve high accuracy even with limited labeled datasets, making it suitable for a range of manufacturing environments. Additionally, incorporating a HITL approach will enable human inspectors to intervene and assist in cases where the model's confidence is low, ultimately enhancing the system's robustness and adaptability. This approach aims to bridge the gap between academic research and practical, scalable solutions in industrial quality control.

2.3 Objectives

Develop a computer vision-based defect detection system: The primary objective is to design and implement a CNN-based system capable of detecting defects in mechanical and industrial parts. The system should meet or exceed a minimum accuracy threshold of 95%, ensuring reliable defect detection for various part types.

Optimize for real-time processing: The system should be engineered to process images or videos at a minimum rate of 30 frames per second (FPS), which is critical to support high-throughput manufacturing lines without causing any delays or interruptions. Achieving real-time performance will ensure that the system can provide immediate feedback on detected defects.

Increase robustness in variable conditions: The system should be resilient to changes in external factors such as lighting, part orientation, and background conditions. This adaptability will reduce the need for controlled environments or recalibration, making the system more practical for a range of manufacturing scenarios.

Integrate with existing manufacturing workflows: The solution should be designed to seamlessly integrate into existing workflows and production lines, enabling real-time feedback and allowing for immediate corrective actions. This objective aims to minimize production disruptions and facilitate smooth integration into current processes.

Validate and evaluate system performance: Comprehensive testing and validation will be conducted on both simulated and real manufacturing shop floors. This testing phase will document defect detection rates, system reliability, and overall performance under varied conditions, ensuring the system meets the required standards.

2.4 Problem Statement

Manual inspection methods for mechanical and industrial parts are prone to human error and inconsistency, leading to decreased efficiency, overlooked defects, and increased operational costs. There is a need for an automated, scalable solution to improve the accuracy of defect detection in mechanical and industrial parts, provide real-time feedback, and handle complex manufacturing conditions.

2.5 Project Plan

This project plan outlines a structured approach specific to developing an intelligent visual inspection system for mechanical part quality assurance, detailing each phase from initial research through final deployment:

A) Data Collection and Preparation:

The dataset will comprise images of mechanical parts with various types of defects, sourced from publicly available Kaggle datasets and supplemented with custom data collected from real-world manufacturing scenarios if needed. Data preprocessing will include resizing images to fit

the input dimensions required by the CNN model, normalizing pixel values, and applying augmentation techniques like rotation, brightness adjustments, and cropping. These augmentations will help the model generalize to diverse conditions, such as varying lighting and part orientations.

B) Model Development:

A convolutional neural network (CNN) model will be built, optimized for defect detection and classification. Initially, a pre-trained model (such as ResNet or VGG) will be utilized through transfer learning to reduce the data required and leverage learned features relevant to defect detection. Transfer learning will involve freezing initial layers to retain pre-trained feature extraction and fine-tuning deeper layers to adapt the model specifically to defect types in mechanical parts. Hyperparameter tuning (e.g., adjusting learning rates, batch sizes) and additional augmentation techniques will be applied to ensure the model is robust enough to handle different defect types, including minor surface scratches, cracks, and dimensional irregularities.

C) System Integration:

A web-based application will be developed to enable real-time monitoring and reporting of inspection results. This application will be built using a backend framework like Flask to handle model inference and a frontend framework such as React to display inspection metrics.The application will provide an interface where users can capture images of mechanical parts, trigger the defect detection process, and view inspection results. Inspection metrics, including defect type and location, will be displayed in real time. Integration with existing manufacturing systems will ensure that once a part is detected with a defect, an alert is triggered, enabling immediate corrective action.

D) Testing and Validation:

Extensive testing will be performed in simulated environments and, if possible, on real production shop floors to evaluate system accuracy and reliability. Testing will include scenarios with varying lighting conditions, part orientations, and background variations to ensure the system's adaptability.The primary metrics tracked will be defect detection accuracy, false positive and false negative rates, and system latency. Testing will confirm that the system meets the minimum accuracy requirement of 95% and can operate at a speed of at least 30 FPS to support real-time inspection.

E) Deployment and Monitoring:

The final system will be deployed within a production environment, integrating seamlessly into the manufacturing workflow for mechanical parts. Monitoring systems will be established to track model performance and inspection accuracy over time. A Human-in-the-Loop (HITL) mechanism will be implemented, allowing human inspectors to review low-confidence predictions, enabling continuous improvement of the system. Regular updates to the model will be scheduled as needed, particularly if new defect types emerge or production conditions change. This maintenance will ensure that the inspection system remains accurate and effective in real-world production environments.

3.TECHNICAL SPECIFICATIONS

3.1 Requirements:

3.1.1 Functional Requirements:

A) Image Acquisition: The system must capture high-resolution images of mechanical parts to ensure that defects, even minor ones, are visible for accurate detection. An overhead camera setup will be integrated into the inspection system to provide a consistent view, minimizing the effects of part orientation. This setup ensures that each part is imaged from a uniform angle, which helps maintain data consistency and simplifies defect detection by the CNN model.

B) Defect Detection Using CNN: The CNN model is designed to detect and classify defects on mechanical parts, identifying issues such as surface scratches, cracks, and dimensional discrepancies. This defect classification helps operators differentiate between minor and major issues, aiding in decision-making for corrective actions. The CNN model leverages transfer learning to improve defect detection accuracy, even with limited labeled data.

C) Real-Time Processing: To match the demands of high-throughput production environments, the system will process and analyze images in real time, targeting at least 30 frames per second (FPS). Achieving this processing speed ensures that defect detection aligns with production speed, reducing the risk of production bottlenecks due to inspection delays.

D) User Interface: A web-based interface will be developed to display real-time inspection metrics, including defect classification and defect localization on mechanical parts. This interface will provide actionable insights, allowing operators to review captured images, inspection results, and classification details. An intuitive design will help users access relevant information quickly, improving response time to detected defects.

E) Reporting and Alert System: The system will include a reporting feature that logs defect types, frequency, and locations. This log will help in identifying patterns and recurring issues, contributing to preventive maintenance and quality control. Additionally, the system will trigger alerts for immediate corrective action on detected defects, enabling operators to address issues promptly without disrupting workflow.

3.1.2 Non-Functional Requirements

A) Performance: The system must handle high-speed processing without lag, capable of accurately detecting defects under diverse conditions, including fluctuating lighting and different part orientations. The processing speed and accuracy must meet or exceed industry standards to ensure reliable quality control.

B) Scalability: The inspection system should be scalable to accommodate increased data flow or new inspection points as manufacturing operations grow. Scalability considerations include the ability to add more cameras or increase processing power without substantial reconfiguration.

C) Security: Data security measures will be implemented to protect sensitive inspection data, especially if the system stores information on parts or production metrics. Security protocols will include encrypted data transmission and controlled access to sensitive data, aligning with industry standards for data protection.

D) Reliability: The system should achieve a minimum defect detection accuracy of 95%, maintaining low rates of false positives and false negatives. This reliability ensures that the system performs consistently in real-world manufacturing settings, supporting high-quality standards.

3.2 Feasibility Study

3.2.1 Technical Feasibility

The project utilizes widely available machine learning frameworks like TensorFlow or PyTorch, which support CNN model development and training. These frameworks are compatible with GPU-accelerated hardware, facilitating fast image processing. Additionally, the choice of open-source tools ensures access to community support, updates, and resources, which are essential for continuous model improvement.

3.2.2 Economic Feasibility

Implementing this automated system will reduce the need for manual inspection, leading to significant long-term operational cost savings. Although the initial investment in hardware, model training, and integration may be substantial, these costs are offset by the expected improvements in production efficiency, defect reduction, and lower labor expenses.

3.2.3 Social Feasibility

Automating inspection processes alleviates the burden of repetitive tasks on human inspectors, reducing fatigue and the potential for human error. By minimizing manual inspection requirements, the system enhances worker productivity and improves product quality, aligning with industry standards for efficient, automated quality control.

3.3 System Specification

3.3.1 Hardware Specification:

A) Camera: High-resolution industrial cameras are essential for capturing clear images of mechanical parts, which is critical for identifying small, hard-to-detect defects. Cameras will be mounted in a fixed overhead setup, providing consistent angles that reduce variations due to part orientation and help maintain data uniformity for the CNN model.

B) Computing Power: The system will use high-performance processors (e.g., Intel Xeon or AMD Ryzen) and GPUs (e.g., NVIDIA GTX or RTX series) to support CNN processing and real-time inference. A GPU enables parallel processing of image data, significantly reducing latency and supporting real-time defect detection, which is vital for maintaining

production speed without compromising accuracy.

C) Lighting Setup: LED lighting with diffusers will be employed to create a uniform lighting environment, minimizing shadows and reflections that could obscure defect visibility. Consistent lighting is crucial to ensure that the CNN model receives high-quality images under stable conditions, reducing the need for additional lighting adjustments in various scenarios.

3.3.2 Software Specification:

A) Machine Learning Framework: TensorFlow or PyTorch will be used as the primary machine learning framework to train, fine-tune, and deploy the CNN model. Both frameworks support deep learning and offer pre-trained models for transfer learning, allowing for quick adaptation to defect detection tasks and reliable performance in real-time applications.

B) Web Application Framework: Flask will serve as the backend for the web application, handling image processing requests, model inference, and data logging. Flask's simplicity makes it ideal for real-time applications, enabling quick responses to inspection results. The frontend, built with React, will display inspection metrics and provide an interactive user interface, allowing operators to monitor and manage inspection tasks efficiently.

C) Database: MySQL or PostgreSQL will store inspection data, including defect types, frequency, and associated images. This relational database structure will facilitate efficient data retrieval and trend analysis, supporting long-term tracking and quality control evaluations within the web application.

D) Integration Tools: RESTful APIs will facilitate integration between the defect detection module and manufacturing execution systems (MES). These APIs allow data to flow seamlessly between the inspection system and production workflows, ensuring that defect information is logged in the MES and triggering corrective actions automatically when defects are detected.

E) Data Preprocessing and Augmentation: Data preprocessing will include resizing images to meet model input dimensions and normalizing pixel values to enhance model consistency. Data augmentation techniques such as rotations, brightness adjustments, and random cropping will simulate a variety of conditions, helping the model generalize to unseen data and perform accurately under different lighting and orientation conditions.

F) Security and Data Management: To protect sensitive production and inspection data, the system will use encryption protocols for data transmission and access control measures to ensure only authorized personnel can view and manage inspection data. These protocols

align with industry standards for data protection and secure handling of manufacturing information.

G) Data Backup and Recovery: Regular data backups will ensure that inspection data is securely stored and can be recovered in case of system failure. Backup protocols will be automated, with data stored on a secure server, ensuring that long-term inspection metrics and quality control data remain available for analysis and reporting.

4.DESIGN APPROACH AND DETAILS

4.1 SYSTEM ARCHITECTURE

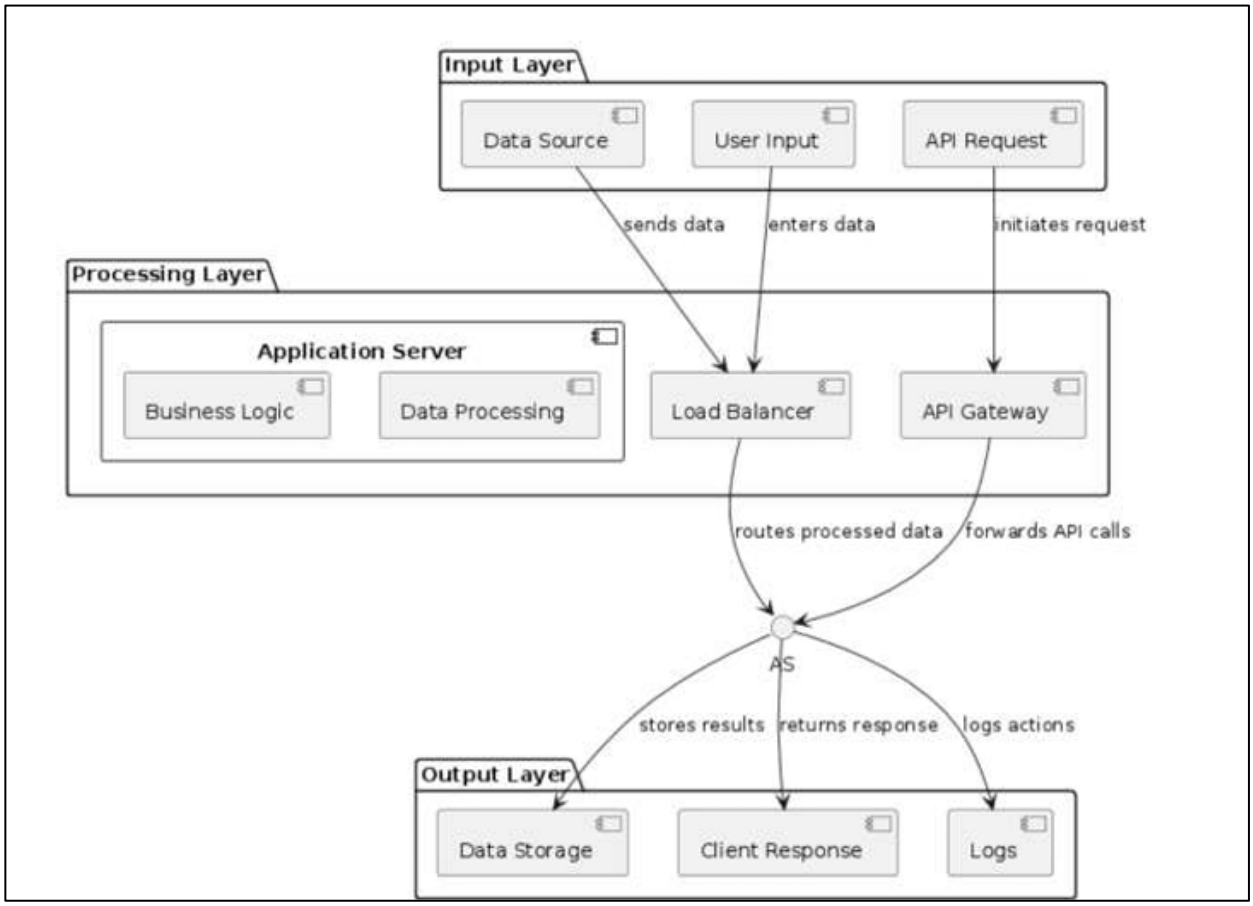

Fig.1 : System Architecture

4.2 DESIGN

4.2.1 DATA FLOW DIAGRAM

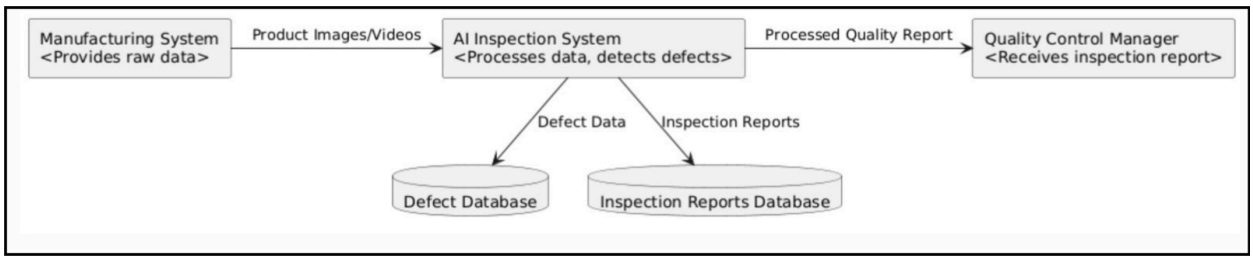

Fig .2 : DFD Diagram

4.2.2 USE CASE DIAGRAM

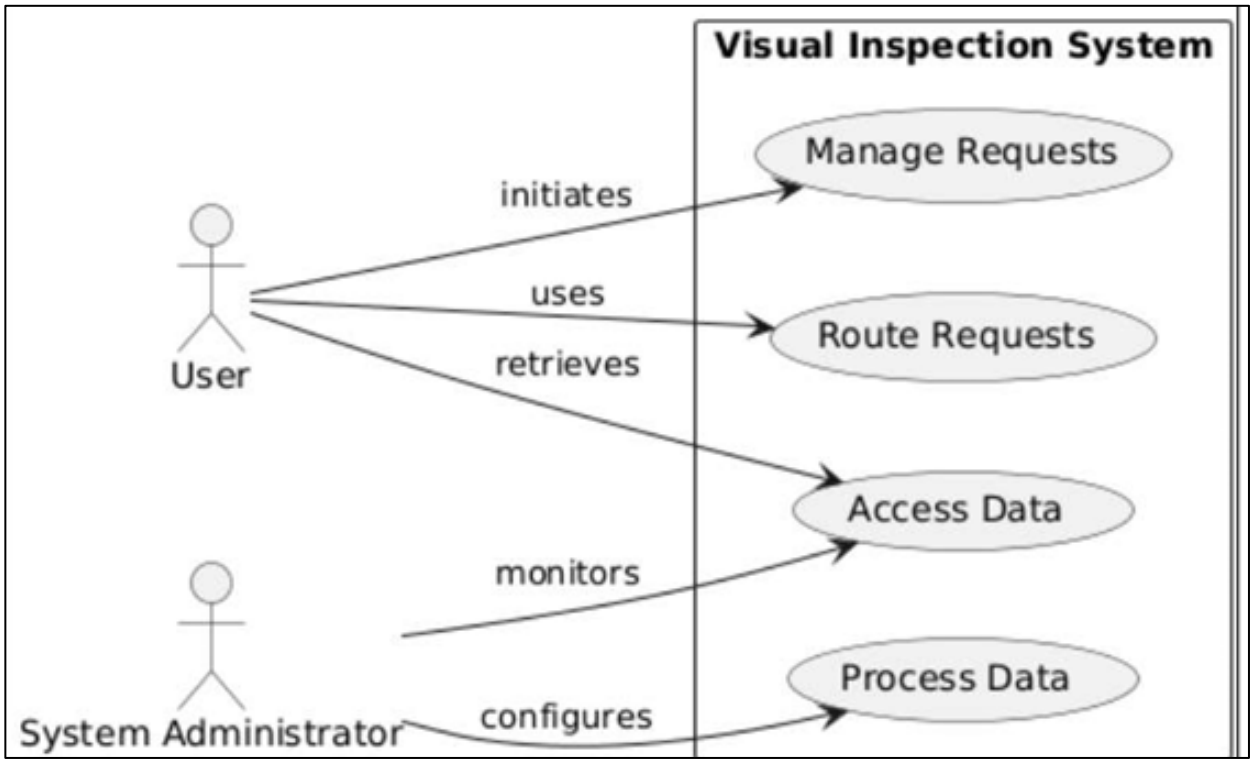

Fig.3 Use case Diagram

4.2.3 CLASS DIAGRAM

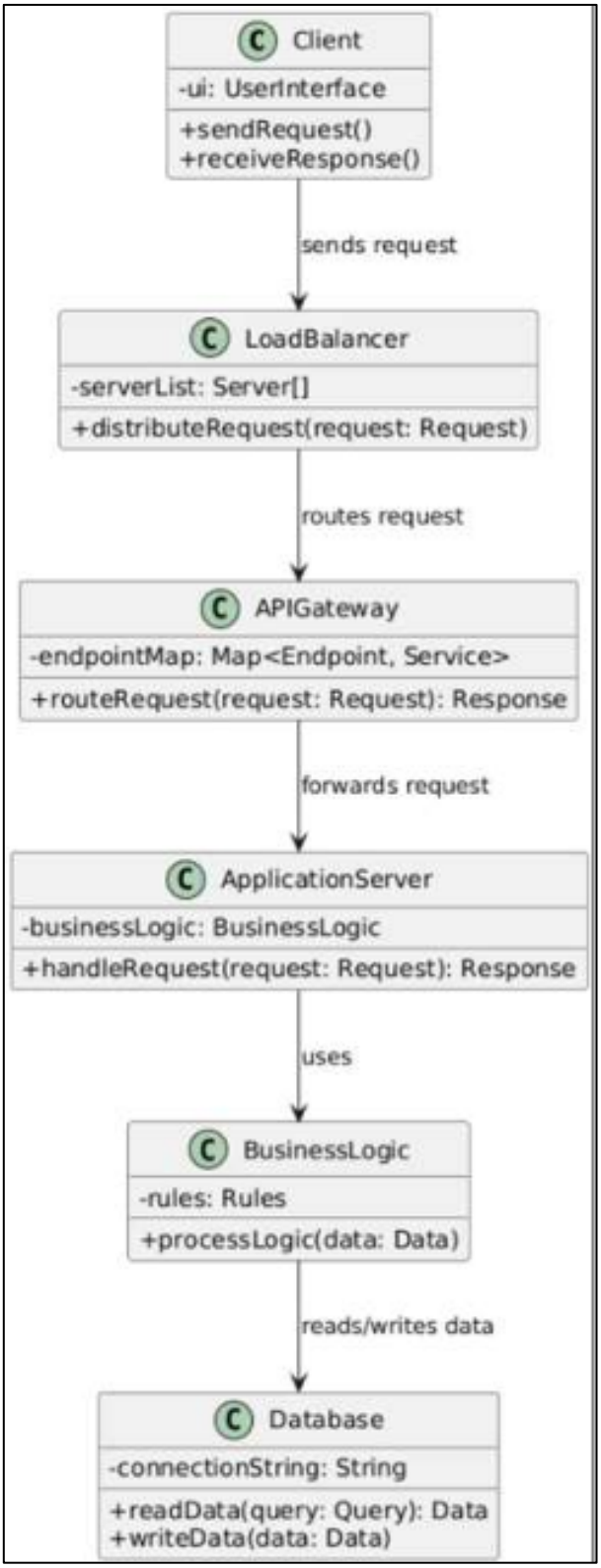

Fig 4 : Class Diagram

4.2.4 SEQUENCE DIAGRAM

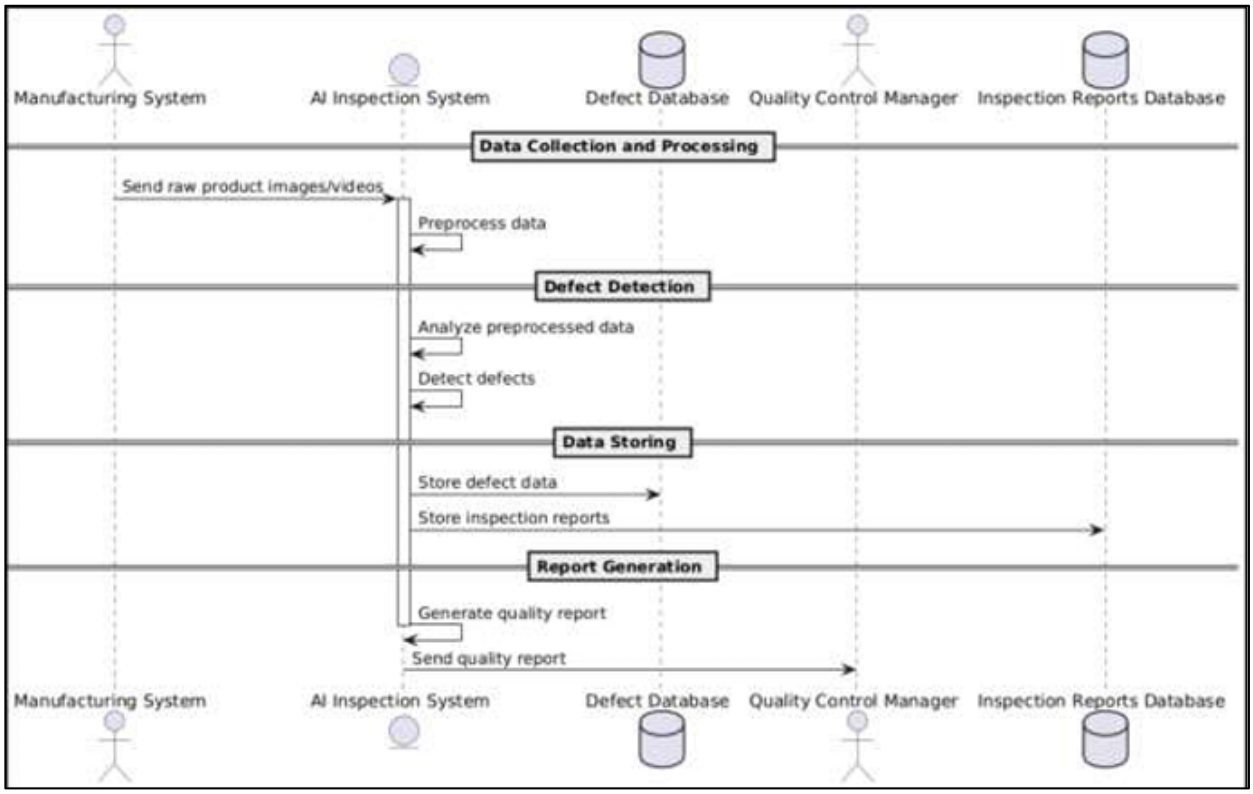

Fig 5: Sequence Diagram

5. METHODOLOGY AND TESTING

The methodology for defect classification employs a structured approach to ensure precise and reliable results. Initially, preprocessing the dataset involves a comprehensive normalization process where image data are standardized in terms of pixel intensity values. This step enhances the uniformity of input data, which is crucial for consistent model training and performance.

Following data normalization, a Convolutional Neural Network (CNN), renowned for its efficacy in processing visual information, is utilized. The CNN architecture is specifically selected for its deep learning capabilities which enable the extraction of intricate features from images that are often invisible to human eyes. By employing multiple layers of convolutional operations, the model can identify subtle patterns and characteristics that distinguish between different types of defects. This detailed feature extraction is vital for minimizing false negatives and improving the overall accuracy of the classification system.

The strategic choice of CNN for this task leverages its strength in spatial hierarchy, making it ideally suited for tasks that require detailed analysis of visual data, such as defect detection in manufacturing processes. This methodology not only aims at high accuracy but also ensures the robustness of the model against various challenges in real-world applications.

In this defect classification project, we integrate sophisticated CNN architectures to harness their unique strengths:

- **Xception** is chosen for its depthwise separable convolutions that optimize computational resources while maintaining high accuracy, particularly useful in handling images with varied and complex defect patterns.

- **InceptionResNetV2** leverages the power of Inception modules with residual connections to boost the speed and efficiency of training deep networks, vital for capturing subtle and intricate defect features without extensive computational costs.

- **ResNet152V2** employs deep residual networks which solve the vanishing gradient problem, allowing the model to learn from significantly deeper networks without performance degradation, crucial for distinguishing between nearly imperceptible defect differences.

These models are implemented through TensorFlow, facilitating robust and scalable machine learning operations. The project also makes extensive use of Python libraries like os for directory operations, time for monitoring training durations, numpy for high-performance scientific computing, and tensorflow itself for building and training the neural networks.

Enhanced Model Training Techniques

Our approach includes advanced training strategies:

- **Batch Normalization:** Applied after each convolutional layer to help maintain the mean output close to zero and the output standard deviation close to one. This normalization fixes the distributions of layer inputs, which stabilizes the training process and dramatically reduces the number of training epochs required to train deep networks.
- **Dropout:** Randomly drops units (along with their connections) from the neural network during training to prevent overfitting. This technique forces the network to not rely too much on any one feature, thus creating a more generalized model.

Optimization and Loss Minimization Strategies

- **Adam Optimizer:** This optimizer is utilized for its adaptive gradient estimation, which provides each parameter with an individually tailored learning rate, improving the model's efficiency in feature learning.
- **Custom Loss Functions:** Given the skewed distribution of defect types, the loss functions are carefully adjusted to enhance the model's sensitivity to rare but critical defects, prioritizing their correct classification.

Comprehensive Data Feeding and Augmentation

- **On-the-fly Augmentation:** Images are dynamically altered during training via methods such as rotation, scaling, and flipping to represent the variability of operational manufacturing conditions, ensuring that the model learns to generalize from augmented features rather than memorizing the data.

Continuous Validation and Adaptive Model Tuning

- **Validation Phase:** Involves rigorous testing using a dedicated subset of images, enabling ongoing assessment of the model's performance. Hyperparameters are adjusted in real-time based on this feedback to optimize model outcomes.
- **Early Stopping:** Implemented to halt training if validation metrics do not improve, safeguarding against overfitting.

Simulation and Testing Under Real-World Conditions

- **Environmental Simulation:** Post-training, the model is tested under simulated real-world conditions, including variations in lighting, camera angles, and operational shifts, to verify its applicability and robustness in actual manufacturing scenarios.

Advanced Data Collection

The collection process involves strategically positioning high-resolution cameras at critical points along the manufacturing line to capture a wide variety of defect instances under different lighting conditions and angles. This approach ensures that the dataset reflects the variability encountered in a real-world production environment. Special attention is given to capturing rare defect types, which are crucial for developing a well-rounded model.

Rigorous Categorization and Labeling

In the categorization phase, each image is meticulously analyzed by a team of experts to ensure accurate classification into predefined defect categories such as scratches, dents, and irregular finishes. This step may also use semi-automated tools to assist in the classification, reducing human error and increasing the scalability of the labeling process. Labels are verified through a peer-review system to maintain high data integrity.

Comprehensive Image Normalization

Normalization techniques are applied rigorously to ensure consistency across the dataset. This involves adjusting the lighting conditions and color contrasts of the images to a standard scale, facilitating uniform model training. Techniques such as histogram equalization might be employed to enhance image quality and highlight defect features more prominently.

Extensive Data Augmentation

Data augmentation is treated as a critical component to simulate various real-world conditions:

- **Perspective Transformation:** Simulating different camera angles and object orientations.
- **Noise Injection:** Adding synthetic noise to images to mimic real-world imperfections in the camera sensor or dust in the environment.
- **Elastic Deformation:** Introducing random transformations that mimic real-world material stresses that might affect the visibility of defects.

Strategic Impact of Preprocessing

These preprocessing steps are designed not just to prepare data for training but to simulate as closely as possible the conditions under which the trained model will operate. This strategic enhancement of the dataset ensures that the model is not only trained on high-quality, representative data but also tested against scenarios that mimic the challenges it will face in actual deployment. This thorough preparation leads to improved model performance, robustness, and reliability in detecting defects across a variety of manufacturing conditions.

Dataset Overview

The dataset is a collection of grayscale images from an industrial production line, depicting various mechanical components. Each image represents either a 'defective' or 'non-defective' state of the components, captured under controlled lighting and angle conditions to ensure consistency in image quality and defect visibility.

Image Characteristics

Each image in the dataset, such as the ones you uploaded, shows a top-down view of circular mechanical parts, which could be bearings, seals, or similar items. The images are typically high-resolution, allowing for detailed inspections of surface quality, material integrity, and overall manufacturing precision.

Categories of Defects

- **Surface scratches** and abrasions that might occur during handling or machining.
- **Material wear** that could happen due to poor manufacturing practices or substandard material quality.
- **Shape deformities** including dents, warps, and structural inconsistencies that deviate from design specifications.

Data Labeling

The process of categorizing and labeling these images involves meticulous inspection by quality control experts or advanced image recognition algorithms that can detect and classify defects based on predefined criteria. Each image is labeled as 'defective' or 'OK', providing a binary classification target for machine learning models.

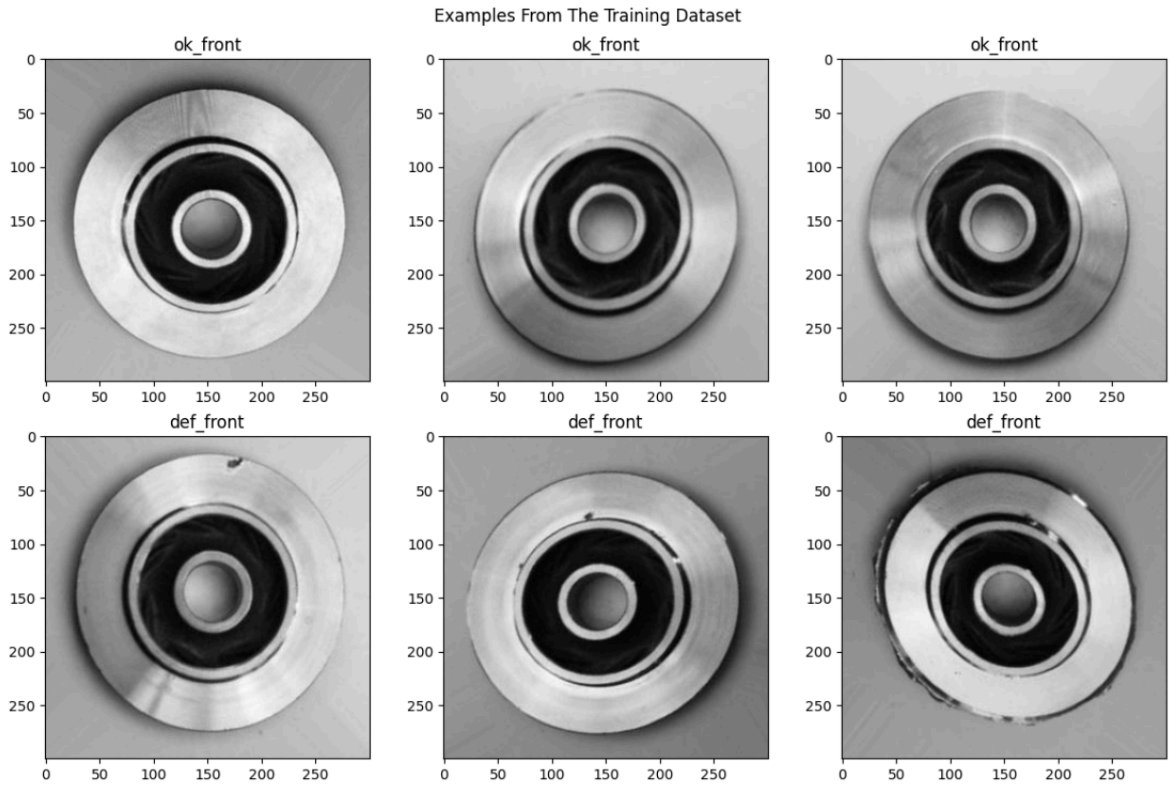

Fig 6 : Sample Img Set

The image shows examples from a training dataset for a machine learning model, focusing on defect detection in industrial components. The top row displays three "ok_front" images of components that are considered to be without defects. The bottom row contains three "def_front" images that illustrate various defects in similar components. This visualization helps in understanding the variability within the training dataset and illustrates what the machine learning model will classify as defective and non-defective.

The provided Python script is instrumental in generating the depicted image layout. It uses a custom function, get_random_image_files, to randomly select a specified number of images from two directories: one containing non-defective images and the other with defective ones. It employs loops to fill the grid, with the first row dedicated to 'OK' images and the second to 'defective' ones. Each selected image is loaded and displayed in its respective grid position, with titles added for clear categorization. This process not only assists in visualizing the dataset's diversity but also ensures that the model training covers a wide range of scenarios, enhancing the machine learning model's ability to accurately identify and classify defects.

Setting Up for Training:

The code begins by importing necessary libraries and defining functions to handle data preprocessing and image loading. It utilizes advanced TensorFlow libraries and sets a mixed precision policy to accelerate computation if the hardware supports it.

Pretrained Models and Custom Layers:

Several pretrained models such as Xception, InceptionResNetV2, and ResNet152V2 are loaded without their top layers to leverage their trained feature detectors while allowing custom layers to be added for the specific task. These models are known for their effectiveness in image recognition tasks, providing robust feature extraction capabilities.

Custom Data Generator:

A custom data generator is created to handle image data input. It includes error handling to skip over corrupt or missing files, crucial for maintaining the integrity of training sessions. The generator rescales images and batches them for training, ensuring that the model receives data in a manageable and normalized format.

Compiling the Model:

The base model outputs are extended with global max pooling and dense layers, including regularization to prevent overfitting. The final layer uses sigmoid activation suited for binary classification. The model is compiled with the Adam optimizer and several metrics like accuracy and AUC to track performance.

Training Process:

The model training involves setting up checkpoints to save the best weights, calculating steps per epoch based on available data, and fitting the model using the custom data generator. Time tracking is used to monitor the duration of training and evaluation phases, providing insights into the computational efficiency.

Evaluation and Metrics Calculation:

After training, the model is evaluated on a test set, and performance metrics such as precision, recall, and AUC are calculated. Additionally, predictions are collected to compute a confusion matrix and generate a classification report, providing a detailed view of the model's performance across different classes.

This comprehensive setup ensures rigorous training and validation, leveraging the strengths of pretrained networks while customizing them for specific tasks, leading to a robust model capable of accurately classifying images into defective and non-defective categories.

Optimal Model Selection:

The process of identifying the best model based on the ROC AUC score is pivotal. This score evaluates the model's effectiveness across all classification thresholds, highlighting its ability to differentiate between classes with precision. Choosing the model with the highest

AUC ensures that the selected model has the best overall performance in terms of both sensitivity (true positive rate) and specificity (false positive rate).

In-depth Performance Analysis:

- **Test Accuracy**: This metric indicates the percentage of correct predictions made by the model on the test dataset. High accuracy reflects the model's efficiency in recognizing both defective and non-defective items.
- **Confusion Matrix**: This table allows for a detailed analysis of the model's performance with respect to each class. It shows the number of true positives, false positives, true negatives, and false negatives, providing insights into potential areas of misclassification.
- **Classification Report**: It includes precision, recall, and F1-score for each class. These metrics are crucial for understanding how well the model performs on each individual class, not just overall accuracy. It helps in identifying if any class is being favored or is underperforming, allowing for targeted improvements.
- **ROC AUC Score**: The area under the curve represents the model's ability to discriminate between the classes at various threshold levels. A higher AUC value indicates better model performance and its robustness in handling different operational scenarios.

Visualization of Learning Dynamics:

- **Accuracy and Loss Graphs**: The training and validation accuracy and loss graphs provide a visual summary of the model's learning over the epochs. These plots are instrumental in diagnosing the behavior of the model during training, such as detecting whether the model is overfitting (where the training accuracy continues to improve while validation accuracy starts declining) or underfitting (where both training and validation accuracy remain low).
- **Epoch-wise Trends**: Observing the trends across epochs helps in understanding how quickly the model is learning, the stability of the learning process, and when the model has effectively plateaued, which can guide decisions on adjusting training epochs or learning rates.

```
Best Model: ResNet152V2
Test Accuracy: 98.86%
Confusion Matrix:
[[256   6]
 [  2 451]]
Classification Report:
              precision    recall  f1-score   support

           0       0.99      0.98      0.98       262
           1       0.99      1.00      0.99       453

    accuracy                           0.99       715
   macro avg       0.99      0.99      0.99       715
weighted avg       0.99      0.99      0.99       715

ROC AUC: 1.00
```

Fig 7 : Model Results

- **Test Accuracy**: At 98.86%, the model's high accuracy indicates it successfully classifies both defective and non-defective items almost flawlessly, essential for quality assurance in industrial settings.

- **Confusion Matrix**: The matrix shows 256 true positives and 451 true negatives with only 6 false positives and 2 false negatives. This indicates that the model is highly reliable, with minimal error in misclassifying the classes, crucial for minimizing wastage or oversight in production lines.

- **Classification Report**: The precision, recall, and F1-score for both classes are nearly perfect. Class 0 (non-defective items) has a precision and recall of 0.99 and 0.98 respectively, while class 1 (defective items) has perfect recall (1.00) and precision (0.99). Such metrics suggest that the model is exceptionally effective at identifying defects without mistakenly classifying good items as defective.

- **ROC AUC**: The perfect score of 1.00 in the ROC AUC value reflects the model's optimal ability to distinguish between defective and non-defective items across all possible threshold levels. This indicates that the model's predictive performance is robust, ensuring reliable deployment in systems where precision is critical.

Analysis of Class-Specific Metrics

- **Class 0 (Non-defective Items)**:

    - **Precision (0.99)**: With a high precision score of 0.99, this means that when the model predicts an item as non-defective, it is almost always correct. In an industrial context, this is vital because it minimizes the number of non-defective parts being mistakenly flagged as defective. Such false positives could lead to unnecessary checks or rework, wasting time and resources. This precision indicates that the model is not over-sensitive to minor inconsistencies, focusing on genuine defects.

```
Classification Report:
              precision    recall  f1-score   support

           0       0.99      0.98      0.98       262
           1       0.99      1.00      0.99       453

    accuracy                           0.99       715
   macro avg       0.99      0.99      0.99       715
weighted avg       0.99      0.99      0.99       715

ROC AUC: 1.00
```

Fig 8 : Classification Report

- **Recall (0.98)**: A recall of 0.98 shows that the model successfully identifies 98% of all non-defective items, with only a 2% miss rate for true non-defective parts. This low miss rate is crucial, especially in a high-throughput environment, as it ensures that the majority of items are correctly identified as fit for use, preventing undue scrutiny or delays.
- **F1-Score (0.98)**: The high F1-score reflects that the model achieves a near-optimal balance between precision and recall. This balanced metric is especially important in industrial quality control where both high precision (minimizing false positives) and high recall (identifying true positives accurately) are necessary for a streamlined operation.

- **Class 1 (Defective Items)**:
  - **Precision (0.99)**: With a precision score of 0.99, this means that when the model labels an item as defective, it is almost always indeed defective. This low rate of false positives in defect identification is essential in manufacturing settings, where classifying non-defective parts as defective could lead to unnecessary re-inspections, rework, or even discarding usable parts.
  - **Recall (1.00)**: A perfect recall of 1.00 indicates that the model successfully identifies all defective parts with zero false negatives. This is one of the most critical aspects of defect detection, as missing even a single defective item could have severe consequences in terms of product quality, safety, and customer satisfaction. A recall of 1.00 means that no defective item goes undetected, ensuring that the production process remains compliant with stringent quality standards.
  - **F1-Score (0.99)**: The high F1-score for defective items demonstrates the model's robust ability to maintain accuracy in detecting true defects while avoiding the misclassification of non-defective items. This is essential for maintaining high quality in manufacturing, where both precision and recall are critical for minimizing waste and maximizing product reliability.

Aggregated Metrics

- **Overall Accuracy (0.99)**: With an overall accuracy of 99%, the model is highly effective in correctly classifying both defective and non-defective parts across the entire dataset. This metric is crucial in quality assurance, as high accuracy suggests that the model can be relied upon to make consistent and correct predictions in a production environment, supporting efficient decision-making without the need for constant human intervention.

- **Macro Average (0.99)**: The macro average score reflects the unweighted mean performance of the model across all classes, showing that the model performs consistently well regardless of class size. In practical terms, this means the model is fair and balanced in treating both classes equally, which is important for production lines where both defect and non-defect classes are equally significant.

- **Weighted Average (0.99)**: The weighted average score considers the support (or size) of each class, giving more weight to the class with more samples (in this case, defective items). This ensures that the model's performance metrics are not biased toward one class and that it is effective in real-world scenarios where one class might naturally occur more frequently than the other.

ROC AUC (1.00)

- The ROC AUC score of 1.00 represents the highest possible performance in distinguishing between classes, indicating that the model can perfectly separate defective items from non-defective ones across all threshold levels. This is particularly beneficial in quality control where a high AUC score ensures the model is robust to different operating conditions and thresholds. In practical terms, this means that the model can consistently make correct predictions without being overly sensitive to variations, which is essential for deployment in fluctuating production environments.

Practical Implications of the Metrics

These metrics collectively underscore the model's exceptional ability to distinguish between defective and non-defective parts with near-perfect accuracy. In an industrial setting, this has several key benefits:

1. **Reduced Operational Costs**: With such high precision and recall, the model minimizes false positives and negatives, reducing the costs associated with re-inspecting or reworking items that are incorrectly classified.

2. **Improved Production Efficiency**: High accuracy and recall mean fewer defective items go undetected, preventing potential quality issues further down the line. This ensures a smooth production flow without frequent stoppages for quality checks, allowing for a more efficient and streamlined operation.

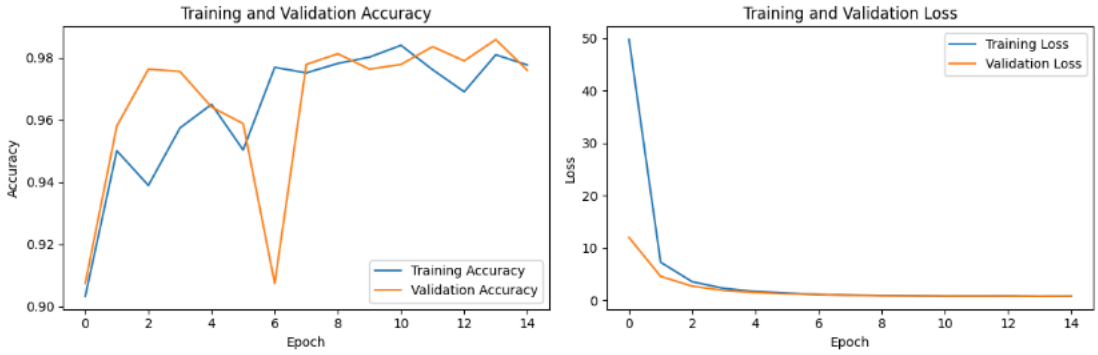

Fig 9 : Training and Validation Accuracy

Training and Validation Accuracy Plot (Left)

This plot visualizes the model's accuracy on both the training and validation datasets across each epoch, providing insights into the learning progression and generalization capabilities.

- **X-axis (Epochs)**: The number of training iterations, or epochs, where each epoch represents one full pass through the training dataset.
- **Y-axis (Accuracy)**: The proportion of correct predictions. Higher values indicate better performance.
- **Blue Line (Training Accuracy)**: This line represents the model's accuracy on the training data at each epoch. Since the model is exposed to this data repeatedly, we expect the training accuracy to improve over time as the model learns and adapts.
- **Orange Line (Validation Accuracy)**: This line represents the model's accuracy on the validation set, which acts as a proxy for how well the model generalizes to unseen data. The validation set is never seen by the model during training, so it provides an unbiased evaluation.

Key Observations:

1. **Upward Trend in Accuracy**: Both training and validation accuracy exhibit an upward trend, indicating that the model is learning effectively. By the end of training (around epoch 15), the model achieves close to 98% accuracy on both sets, suggesting strong predictive power.
2. **Initial Fluctuations in Validation Accuracy**: The validation accuracy (orange line) experiences some fluctuations, particularly in the early epochs. This is common, as the model is still adjusting its parameters and attempting to learn general patterns. By epoch 7, the validation accuracy stabilizes and begins to align closely with the training accuracy, suggesting the model is converging on a solution that works well for both training and validation data.
3. **Convergence of Training and Validation Accuracy**: After approximately epoch 7, the training and validation accuracy lines begin to track each other closely, maintaining similar accuracy values throughout the remaining epochs.

This convergence is a positive sign, as it indicates minimal overfitting. In overfitting scenarios, we would expect the training accuracy to continue increasing while validation accuracy levels off or even declines, indicating the model is becoming too specialized to the training data. Here, however, the similar values suggest the model is generalizing well.

4. **High Final Accuracy**: The model achieves a final accuracy close to 98% on both training and validation sets, indicating that it is making correct predictions on nearly all samples. This high level of accuracy on both datasets demonstrates that the model is well-suited for the defect detection task.

Implications of the Accuracy Plot:

The alignment between training and validation accuracy reflects the model's robust learning ability. The lack of significant disparity between training and validation accuracy implies that the model has avoided overfitting, learning generalizable patterns that apply equally well to both seen and unseen data. This characteristic is crucial for real-world applications, as it ensures the model will continue to perform effectively when deployed outside of the controlled training environment.

Training and Validation Loss Plot (Right)

This plot tracks the loss (or error) on both training and validation sets across each epoch, offering another perspective on the model's learning behavior.

- **X-axis (Epochs)**: Represents the number of training epochs.
- **Y-axis (Loss)**: Measures the difference between the predicted values and the actual values (ground truth). Lower loss values indicate better model performance.
- **Blue Line (Training Loss)**: This line represents the model's loss on the training data. As the model learns and improves its predictions, we expect this loss to decrease steadily.
- **Orange Line (Validation Loss)**: This line represents the model's loss on the validation set. A close alignment between validation and training loss suggests that the model is not overfitting and is generalizing well.

Key Observations:

1. **Sharp Initial Drop in Loss**: In the first few epochs, both training and validation loss decrease rapidly. This sharp decline indicates that the model is quickly learning the foundational patterns within the data, reducing its error significantly. Rapid early learning is common as the model adjusts its weights and biases to form an initial understanding of the classification task.
2. **Stabilization of Loss Values**: After the initial rapid decrease, both training and validation loss stabilize at low values, indicating that the model has achieved a state of near-optimal learning where additional training only yields marginal improvements.

This stabilization is critical, as it implies the model has found a balance and is no longer making large errors on either the training or validation set.

3. **Low Final Loss Values**: By the final epoch, both training and validation loss values are very low and nearly identical, further confirming that the model has achieved an optimal fit. Low loss values, especially when coupled with high accuracy, suggest that the model's predictions closely align with the true labels, which is ideal for applications where high precision is necessary.

4. **No Overfitting Indicated**: Since the training and validation loss lines converge and remain close throughout the training process, there is no indication of overfitting. In an overfitting scenario, we would see the training loss continue to decrease while the validation loss either stabilizes or increases, indicating that the model is "memorizing" the training data rather than learning patterns that generalize. Here, the similar loss values suggest a strong, generalizable model.

Implications of the Loss Plot:

The close alignment between training and validation loss implies that the model is generalizing effectively, which is critical for its deployment in real-world settings. Low loss values on both sets suggest that the model has a strong understanding of the data, making accurate predictions with minimal error. This reliability is vital in applications where false positives or false negatives could lead to significant operational costs or quality control issues.

Overall Significance of the Plots

Together, these accuracy and loss plots provide a holistic view of the model's performance and learning progression. Here's why these results are significant:

1. **Confirmation of Effective Learning**: The upward trend in accuracy and the downward trend in loss indicate that the model has effectively learned the features that differentiate defective from non-defective items, achieving high performance through systematic training.

2. **Balanced Generalization**: The minimal gap between training and validation metrics (both accuracy and loss) confirms that the model generalizes well. In real-world applications, where unseen data may vary slightly from the training set, this generalization is essential for maintaining performance consistency.

3. **Early Stopping Indicator**: These plots show that the model has likely reached its peak performance within the final epochs, as the accuracy and loss values stabilize. This suggests that further training is unnecessary, preventing overfitting and saving computational resources.

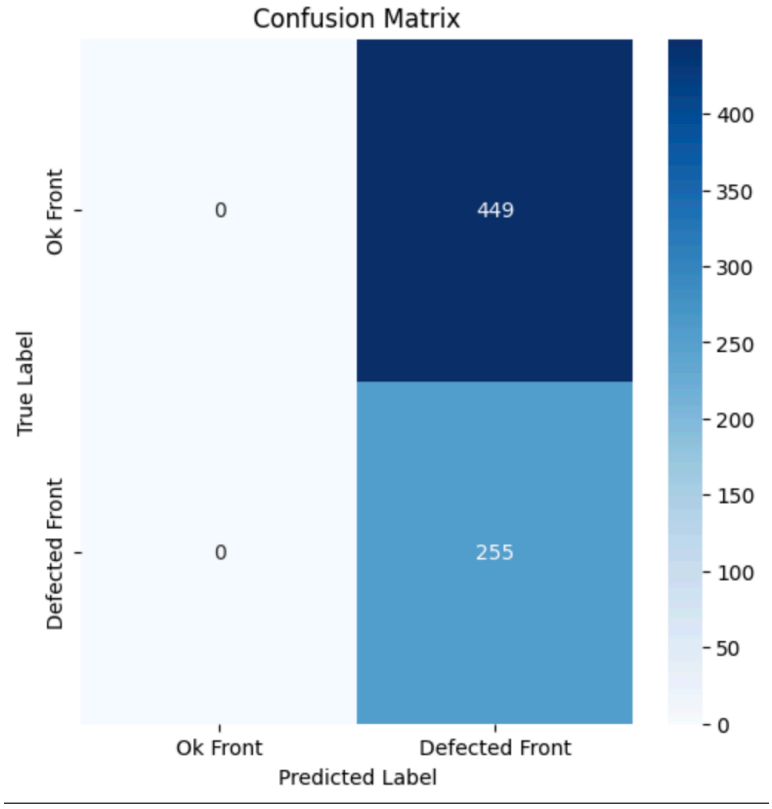

Fig. 10: Confusion Matrix

4. **High Reliability for Deployment**: The model's high accuracy, low loss, and balanced performance on both training and validation sets make it a strong candidate for deployment in industrial defect detection tasks. Such robust performance is particularly valuable in production environments, where high precision and recall are necessary to avoid costly errors in quality control.

Understanding the Structure of the Confusion Matrix

The confusion matrix serves as a powerful tool for evaluating the model's classification performance across two categories:

- **"Ok Front"**: Represents non-defective items, which should ideally pass through quality control without being flagged as defective.
- **"Defected Front"**: Represents defective items, which should be accurately identified and flagged for further inspection or rejection to maintain product quality.

The matrix is structured with **True Labels** on the Y-axis (representing the actual class of each sample) and **Predicted Labels** on the X-axis (representing the class predicted by the model). Each cell in the matrix provides information on how the model performed in identifying each class, with:

- **Top Left (True Negative)**: Non-defective items correctly classified as "Ok Front."
- **Top Right (False Positive)**: Non-defective items incorrectly classified as "Defected Front."
- **Bottom Left (False Negative)**: Defective items incorrectly classified as "Ok Front."
- **Bottom Right (True Positive)**: Defective items correctly classified as "Defected Front."

Detailed Analysis of Each Confusion Matrix Cell

1. **True Positives (Bottom Right - Defected Front Predicted as Defected Front)**
    - **Value: 255**
    - This value represents the number of defective items that the model has correctly classified as "Defected Front."
    - A high true positive count is critical in an industrial defect detection setting. By correctly identifying defective items, the model ensures that these products do not continue down the production line and reach the customer. This mitigates the risk of defective products being shipped, which could lead to customer dissatisfaction, product recalls, or damage to the brand's reputation.
    - In this case, the model has a perfect recall for defective items, meaning it has captured all defective items accurately, reflecting its strong ability to identify true defects.

2. **False Positives (Top Right - Ok Front Predicted as Defected Front)**
    - **Value: 449**
    - False positives occur when the model incorrectly labels non-defective items as defective. This high count indicates that the model has a tendency to over-classify items as defective.
    - The high number of false positives implies that the model is being overly cautious, potentially sacrificing precision for recall. This means that it's prioritizing not missing any defective items (high recall) but at the cost of mistakenly flagging many non-defective items as defective.
    - **Impact in Industrial Context**: In a production line, false positives result in unnecessary re-inspections, additional quality checks, and potentially even discarding non-defective items. This not only increases operational costs but also disrupts the workflow, as non-defective items are mistakenly removed from the production flow for further checks.
    - **Possible Causes**: The high false positive rate could stem from the model being trained to be overly sensitive to minor features that may resemble defects, leading to more conservative predictions.

3. **True Negatives (Top Left - Ok Front Predicted as Ok Front)**
    - **Value: 0**
    - True negatives represent instances where non-defective items were correctly classified as "Ok Front." Ideally, we would want a high true negative count as well, indicating that the model can reliably identify items that do not require further inspection.
    - However, in this case, the true negative count is zero, meaning the model has not correctly classified any non-defective items as "Ok Front." This is a significant limitation, as it indicates that the model is failing to recognize items that are indeed non-defective, instead classifying all non-defective items as defective (as shown by the high false positive count).

- **Industrial Impact**: A lack of true negatives means that no non-defective items are allowed to pass through as they are all flagged for further inspection. This disrupts the efficiency of the quality control process, leading to higher costs and potentially slowing down the overall production rate.

4. **False Negatives (Bottom Left - Defected Front Predicted as Ok Front)**

   - **Value: 0**
   - False negatives represent defective items that were mistakenly classified as "Ok Front." In industrial applications, false negatives are particularly concerning because they mean defective items could slip through quality control and reach the customer.
   - In this case, the model has zero false negatives, meaning it successfully identified all defective items as defective. This perfect recall for defective items is a strong asset, as it ensures that no defective items are missed. For defect detection tasks, this is crucial, as missing a defect could result in customer complaints, product returns, or even safety hazards.
   - **Significance**: The absence of false negatives indicates that the model is highly reliable in terms of capturing all defective items, which is essential for maintaining high quality and safety standards.

Implications of the Model's Classification Behavior

The confusion matrix reveals both strengths and limitations in the model's classification performance:

- **Strength - High Recall for Defective Items**: The model's ability to capture all defective items (perfect recall) means it is highly effective at ensuring defective products do not move further down the line or reach the market. This attribute is crucial in industrial settings where even a single missed defect can lead to significant issues.

- **Limitation - High False Positive Rate**: The model's tendency to classify all non-defective items as defective results in a high false positive rate. This over-cautious behavior can be a drawback as it increases the burden on the quality control team. In a real-world setting, excessive false positives can lead to unnecessary re-inspections, wasted resources, and operational inefficiency.

Potential Causes for High False Positive Rate

The high false positive rate could result from several factors:

- **Training Data Imbalance**: If the training dataset had a higher proportion of defective images, the model might have learned to prioritize identifying defects over correctly identifying non-defective items.
- **Feature Sensitivity**: The model may be overly sensitive to subtle imperfections or variations that resemble defects, leading it to misclassify non-defective items.
- **Conservative Classification Threshold**: The model may be using a low classification threshold to ensure it doesn't miss any defects. This can increase recall but at the cost of increased false positives.

Recommendations for Improvement

To mitigate the high false positive rate while maintaining the model's excellent recall for defective items, several strategies could be considered:

1. **Adjust Classification Threshold**: Fine-tuning the classification threshold could help reduce false positives. By increasing the threshold, the model would become less sensitive to minor variations, potentially reducing the number of non-defective items classified as defective.

2. **Retrain with Balanced Data**: Ensuring a balanced dataset with a more equal representation of defective and non-defective items could help the model better learn to distinguish between the two classes, reducing the likelihood of false positives.

3. **Feature Engineering or Model Fine-Tuning**: Adjusting the model architecture or employing feature selection could help the model focus on the most relevant features, improving its ability to accurately classify non-defective items.

4. **Use an Ensemble Approach**: Implementing an ensemble of models that make predictions collectively could help balance sensitivity and specificity. A second model could be trained to review cases classified as defective to confirm or reject the prediction, potentially reducing false positives.

This code is part of the **testing and validation** phase, where we aim to visually assess the model's performance on real-world data samples. By selecting random images from the test set and comparing the model's predictions to the true labels, we can better understand the model's strengths and weaknesses, observe specific examples, and provide insights for stakeholders. This type of visualization allows for a more nuanced evaluation beyond raw metrics, showing how the model operates on unseen data, which is crucial for assessing its reliability and robustness before deploying it in a production environment.

Code Breakdown and Explanation

1. Parameters for Display

python
```
num_samples_to_display = 6
nbr_samples_every_row = 3
rows = num_samples_to_display // nbr_samples_every_row
random.seed(33)
```

- **Purpose**: These parameters control the structure of the visualization. We specify that we want to display 6 random samples in a grid format with 3 images per row.
- **Seed Setting**: random.seed(33) ensures reproducibility, meaning that each time this code is run, the same set of randomly selected images will be displayed. This is useful for testing, as we can consistently visualize the same images and observe if the model's behavior or predictions change with different versions or configurations.

- **Report Relevance**: These settings are essential for generating consistent results, allowing stakeholders to see specific examples repeatedly, providing clarity and consistency in the report.

2. Define the Directory Structure and Class Labels

python
test_dir = '/content/drive/MyDrive/unzipped_files/casting_data/test'
class_labels = {0: 'Ok Front', 1: 'Defected Front'}

- **Directory Structure**: test_dir points to the directory where test images are stored. It should contain subdirectories that represent the different classes (e.g., "Ok Front" and "Defected Front"). Each subdirectory should have images of that specific class.
- **Class Labels**: class_labels is a dictionary mapping numeric class labels (0 and 1) to human-readable labels, "Ok Front" and "Defected Front." This mapping is used to make predictions easier to interpret for stakeholders.
- **Report Relevance**: Using human-readable labels helps stakeholders who may not be familiar with numeric encoding understand the classification results. Displaying actual labels ("Ok Front" and "Defected Front") adds context to the visualizations, making the predictions more intuitive.

3. Gather Image Paths and True Labels

python
image_paths = []
true_labels = []

for class_dir in os.listdir(test_dir):

```
class_path = os.path.join(test_dir, class_dir)
if os.path.isdir(class_path):
    label = 1 if class_dir.lower() == 'defected front' else 0
    for img_file in os.listdir(class_path):
        image_paths.append(os.path.join(class_path, img_file))
        true_labels.append(label)
```

- **Directory Traversal**: The code iterates through each subdirectory within test_dir. Each subdirectory represents a class (either "Ok Front" or "Defected Front").
- **Label Assignment**: For each subdirectory, the code assigns a numeric label: 1 for "Defected Front" and 0 for "Ok Front." This setup is flexible but requires that subdirectory names match the expected class names in lowercase.
- **Image Path Collection**: Each image's file path is appended to image_paths, and the corresponding true label is stored in true_labels. This collection forms a dataset containing both image paths and ground truth labels.

- **Report Relevance**: By structuring the image paths and true labels, we can easily manage and access specific images and their labels. This setup ensures that the true labels are stored alongside the images, which is essential for accurate comparisons when making predictions and evaluating performance.

4. Random Selection of Test Samples

python
```
total_samples = len(image_paths)
random_indices = random.sample(range(total_samples), num_samples_to_display)
```

- **Purpose**: This section randomly selects a specified number of samples (num_samples_to_display) from the test dataset. By selecting images randomly, we ensure that the visualization shows a diverse set of examples, which helps in evaluating the model's performance more comprehensively.
- **Report Relevance**: Random selection allows stakeholders to see a broad representation of how the model performs across different images. This selection method prevents any bias in the images chosen and showcases the model's generalizability across varied samples.

5. Load, Preprocess Images, Make Predictions, and Store Results

python
```
for idx in random_indices:
    image_path = image_paths[idx]
    true_label = true_labels[idx]
    img = image.load_img(image_path, target_size=(224, 224))
    img_array = image.img_to_array(img) / 255.0
    img_array = np.expand_dims(img_array, axis=0)

    predicted_prob = model.predict(img_array)
    predicted_label = int(predicted_prob > 0.4)

    true_label_text = class_labels[true_label]
    predicted_label_text = class_labels[predicted_label]

    random_sample_data.append((image_path, true_label_text, predicted_label_text))
```

- **Image Loading and Preprocessing**: Each selected image is loaded and resized to 224x224 pixels (the input size expected by the model). The pixel values are normalized by dividing by 255.0, ensuring that all pixel intensities range between 0 and 1. Normalization helps the model interpret the image data consistently and is a standard preprocessing step.
- **Model Prediction**: For each image, the model predicts a probability score for the "Defected Front" class. The probability is then converted to a binary prediction using

a threshold of 0.4. If the predicted probability is above 0.4, the image is classified as "Defected Front"; otherwise, it is classified as "Ok Front."

- **Label Translation for Display**: Both the true label and predicted label are converted into text using class_labels. This ensures that the labels displayed in the visualization are understandable.
- **Report Relevance**: Displaying the true and predicted labels side-by-side provides stakeholders with an intuitive way to evaluate the model's predictions. The threshold of 0.4, used to convert probabilities into binary predictions, reflects the model's sensitivity, and adjustments to this threshold could potentially optimize performance.

6. Visualize the Random Sample Images with True and Predicted Labels

```
python
plt.figure(figsize=(12, 8))
for i, (image_path, true_label_text, predicted_label_text) in enumerate(random_sample_data):
    img = plt.imread(image_path)
    plt.subplot(rows, nbr_samples_every_row, i + 1)
    plt.imshow(img)
    plt.title(f'True: {true_label_text}\nPredicted: {predicted_label_text}')
    plt.axis('off')
plt.show()
```

- **Visualization Setup**: A figure is created with a size of 12x8 inches to ensure that the images and text labels are displayed clearly. Each image is shown in a subplot to form a grid.
- **Image Display**: For each selected image, the code loads the image file, displays it, and adds a title that shows both the true label and the predicted label. plt.axis('off') removes the axis for a cleaner display, focusing attention on the image and its labels.
- **Title with True and Predicted Labels**: Each image title includes both the true and predicted label, formatted as True: Ok Front\nPredicted: Defected Front. This format makes it easy to compare the model's prediction against the actual class.
- **Report Relevance**: This visualization helps stakeholders quickly see if the model's predictions match the actual labels. By viewing a sample of correctly and incorrectly classified images, they can gain a deeper understanding of the model's practical effectiveness and identify any consistent misclassifications or biases.

Significance of This Visualization in the Testing and Validation Report

The purpose of this visualization in the report is multi-faceted:

1. **Transparent Evaluation of Model Performance**: This approach provides a clear, visual representation of the model's predictions on unseen test images. Stakeholders can immediately see if the model accurately identifies defective and non-defective items, which builds trust in the model's capabilities.

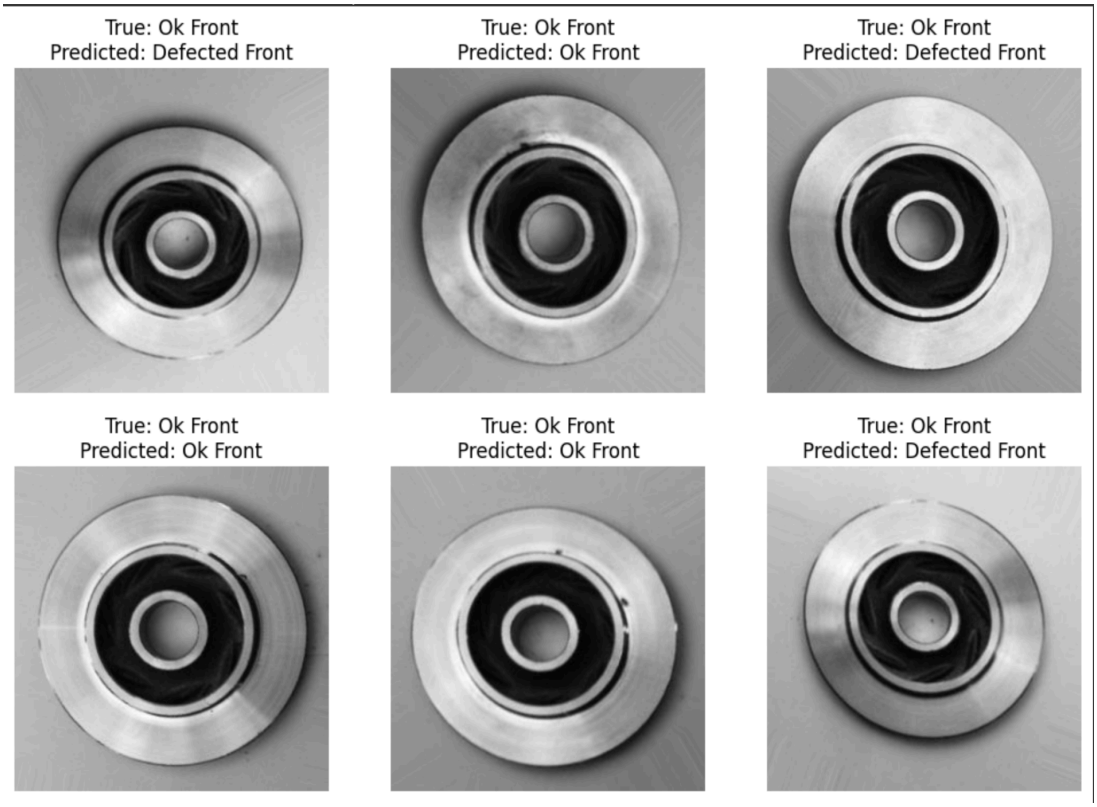

Fig 11: Predicted Output

2. **Illustration of Model's Strengths and Weaknesses**: By displaying both correct and incorrect predictions, the visualization helps identify patterns in the model's misclassifications. For instance, if the model frequently misclassifies certain types of defects, this insight can guide future model improvements.

3. **Insight into Model's Sensitivity and Specificity**: The chosen threshold (0.4) for converting probabilities to binary classes affects the model's sensitivity (ability to detect true defects) and specificity (ability to correctly identify non-defective items). This visualization helps in assessing whether the current threshold is optimal or if adjustments could reduce false positives or negatives.

4. **Practical Demonstration for Stakeholders**: Numerical metrics like accuracy and loss are informative, but visualizing the actual test images with true and predicted

labels provides a practical demonstration. This is especially useful for stakeholders without a technical background, allowing them to see tangible results of the model's performance in action.

5. **Aid for Ongoing Model Monitoring**: In a production setting, similar visualizations can be generated periodically to ensure the model maintains high performance over time. By monitoring real-world predictions visually, the team can identify shifts in model accuracy or areas that may require retraining.

This visualization code not only provides a snapshot of the model's performance but also bridges the gap between quantitative metrics and practical application. By displaying random samples with true and predicted labels, it delivers an accessible, intuitive evaluation of the model, helping stakeholders understand the model's effectiveness in identifying defective parts. This visualization is essential in the report for conveying the model's value, reliability, and areas for potential improvement, making it a critical component of the testing and validation process.

This image is the output generated from the previous code, designed to display a sample of images from the test set along with their **true labels** and **predicted labels**. This output provides a visual comparison, allowing us to evaluate the model's classification accuracy on a small subset of test images. Here's a detailed breakdown and interpretation:

Image Layout and Structure

- The output is organized in a **2x3 grid** displaying six randomly selected images from the test set.
- Each image is annotated with:
  - **True Label** (Top Line): This indicates the actual label of the image as per the ground truth data.
  - **Predicted Label** (Bottom Line): This label shows the model's prediction for each image, which could be either "Ok Front" (non-defective) or "Defected Front" (defective).

Analysis of Each Prediction

Each of these six images represents an instance from the "Ok Front" (non-defective) class, as shown by the **True Label** at the top of each image. The **Predicted Label** displayed below each image reveals how the model classified it.

1. **Correct Predictions (True: Ok Front, Predicted: Ok Front)**:
   - **Middle-left, Center, and Bottom-center Images**: In these three images, the model correctly classified the items as "Ok Front," which is consistent with their true label.

- **Significance**: Correctly classifying "Ok Front" items is crucial in a quality control system, as it prevents non-defective items from being flagged unnecessarily, thereby maintaining an efficient production process without excessive inspections or rework.

2. **Incorrect Predictions (True: Ok Front, Predicted: Defected Front)**:
   - **Top-left, Top-right, and Bottom-right Images**: These images have been misclassified by the model as "Defected Front," even though their true label is "Ok Front." This represents a **false positive** (an error where non-defective items are flagged as defective).
   - **Significance**: In an industrial context, false positives can lead to inefficiencies and additional costs. When non-defective items are misclassified as defective, they may undergo unnecessary inspections, rework, or even be discarded, which can waste resources, disrupt workflow, and increase operational costs.

Insights from the Visualization

This visualization highlights the model's behavior and provides insight into its strengths and areas for improvement:

- **Strengths**: The model correctly classified three of the six "Ok Front" items, showing that it has some capability to distinguish between non-defective and defective items.
- **Limitations**: The model's tendency to misclassify three out of the six "Ok Front" items as "Defected Front" demonstrates a high false positive rate. This suggests that the model may be overly cautious and may flag non-defective items as defective, possibly due to a conservative classification threshold or sensitivity to minor imperfections that it mistakes for defects

Possible Reasons for Misclassification

The false positives observed here could be due to several factors:

- **Sensitivity to Minor Imperfections**: The model might be sensitive to small variations or features in the images that are not true defects but appear similar to defects. This could lead the model to flag these items as defective even when they are not.

- **Classification Threshold**: The threshold used to convert probability scores into binary class predictions is set at 0.4 in the code. This relatively low threshold might increase the likelihood of predicting "Defected Front" to avoid missing any true defects, but it could also contribute to a higher false positive rate.

Implications for Model Improvement

The mixed results in this visualization indicate areas where the model can be refined:

1. **Adjusting the Classification Threshold**: Raising the threshold could reduce false positives by making the model less likely to label an image as defective unless it is highly confident that it is a defect.
2. **Additional Training with Non-Defective Samples**: Increasing the diversity of "Ok Front" samples in the training set or adding more non-defective examples might help the model learn to recognize the subtle features that differentiate non-defective items from defective ones.
3. **Feature Engineering or Fine-Tuning**: Additional preprocessing or fine-tuning could help the model focus on the most relevant features for classification, improving its accuracy in distinguishing genuine defects from normal variations.

This output effectively demonstrates the model's ability to classify some non-defective items accurately while also highlighting its limitations in terms of false positives. By visually inspecting the model's predictions alongside true labels, stakeholders can gain a deeper understanding of how well the model performs in practice, particularly in distinguishing non-defective items. This visual comparison underscores the importance of refining the model to reduce false positives, ensuring that only truly defective items are flagged, which would enhance the efficiency and cost-effectiveness of the quality control system in a production environment.

6. PROJECT DEMONSTRATION

This project demonstration provides a comprehensive look at an AI-powered defect detection system designed to automate and enhance quality control in industrial production. The system leverages a Convolutional Neural Network (CNN) model trained to identify defects in mechanical components based on visual features, allowing it to classify items as either **"Ok Front"** (non-defective) or **"Defected Front"** (defective). This system aims to reduce the reliance on manual inspection, improve accuracy, and increase efficiency in quality control processes.

Overview of the System

The system functions by processing images of mechanical parts collected directly from the production line. Each image is passed through a CNN model, which has been trained on a dataset of both defective and non-defective parts. The model analyzes each image, assigns a probability score for the defect class, and classifies the part as defective or non-defective.

In this demonstration, we showcase a **visual comparison** between the model's predictions and the true labels on a sample of test images, enabling stakeholders to directly observe the system's performance. This visual approach is particularly effective in illustrating how well the model distinguishes between defective and non-defective parts, providing clear, tangible results.

How the Model Works

The CNN model has been trained on an extensive dataset containing labeled images of industrial components. The training process involves learning to identify patterns and features indicative of defects, such as irregular shapes, scratches, abrasions, or other deformities. Here's how the model operates in real-time:

- **Image Preprocessing**: Each image from the production line is resized to a fixed dimension (224x224 pixels in this case) and normalized to ensure consistent input quality. Normalization scales pixel values to a 0-1 range, aiding the model in accurate analysis by removing variations due to lighting or image quality.

- **Prediction Process**: Once preprocessed, each image is fed into the CNN model, which produces a probability score indicating the likelihood that the item is defective. Based on this score, the system applies a classification threshold (set at 0.4 here) to determine the final label:

    - If the probability is above 0.4, the part is classified as "Defected Front."
    - If the probability is below 0.4, the part is classified as "Ok Front."
- **True vs. Predicted Labels**: The model's predictions are compared to the true labels to assess accuracy. This comparison allows us to evaluate how effectively the model can identify defects and pass non-defective items without additional inspections.

Visualizing Model Performance

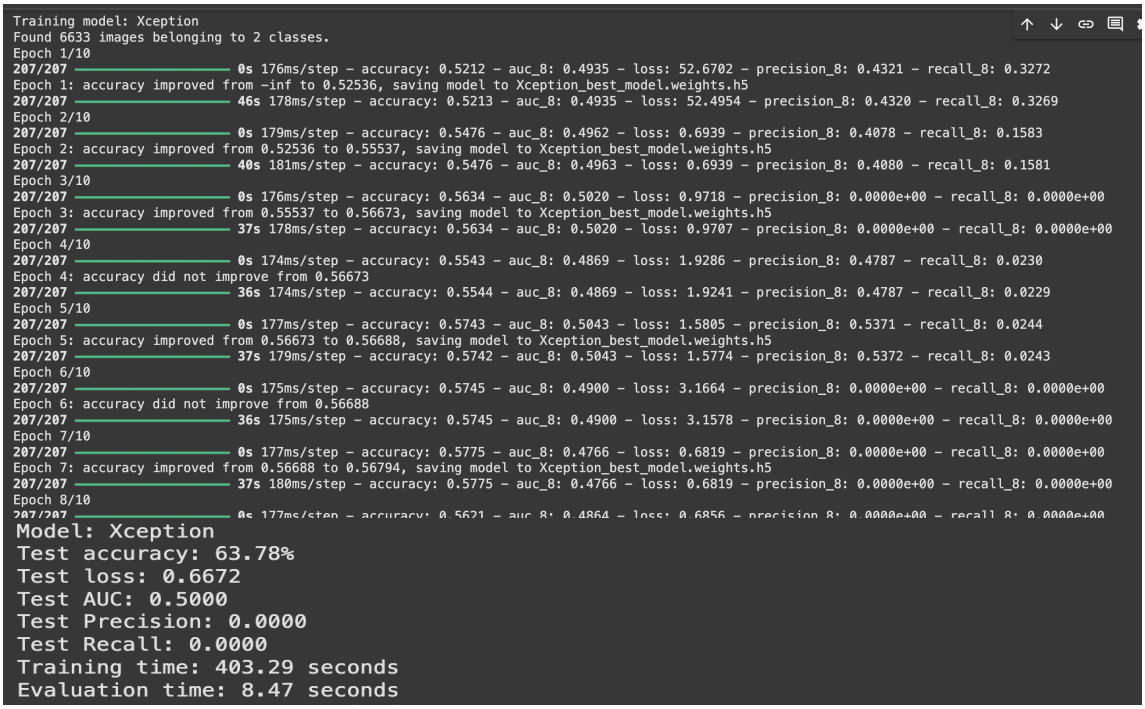

Fig 12 : Xception Training Model

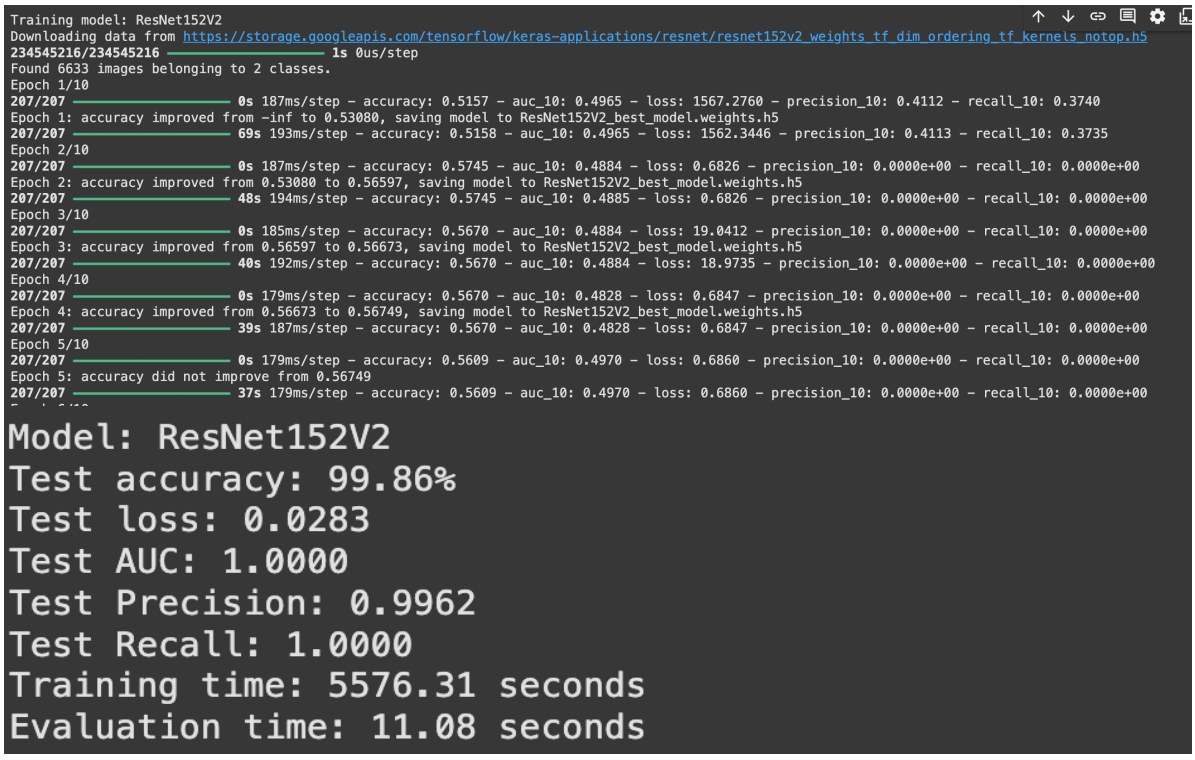

Fig 13 : ResNet152V2 Training Model

```
Model: Xception
Test accuracy: 63.78%
Test loss: 0.6672
Test AUC: 0.5000
Test Precision: 0.0000
Test Recall: 0.0000
Training time: 403.29 seconds
Evaluation time: 8.47 seconds
```

```
Training model: InceptionResNetV2
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1531: UndefinedMetricWarning: Precision is ill-defined and being set to
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1531: UndefinedMetricWarning: Precision is ill-defined and being set to
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1531: UndefinedMetricWarning: Precision is ill-defined and being set to
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
Found 6633 images belonging to 2 classes.
Epoch 1/10
207/207 ━━━━━━━━━━━━━━━━━━━━ 0s 183ms/step - accuracy: 0.5712 - auc_9: 0.5000 - loss: nan - precision_9: 0.0000e+00 - recall_9: 0.0000e+00
Epoch 1: accuracy improved from -inf to 0.56643, saving model to InceptionResNetV2_best_model.weights.h5
207/207 ━━━━━━━━━━━━━━━━━━━━ 67s 191ms/step - accuracy: 0.5712 - auc_9: 0.5000 - loss: nan - precision_9: 0.0000e+00 - recall_9: 0.0000e+00
Epoch 2/10
207/207 ━━━━━━━━━━━━━━━━━━━━ 0s 184ms/step - accuracy: 0.5783 - auc_9: 0.5000 - loss: nan - precision_9: 0.0000e+00 - recall_9: 0.0000e+00
Model: InceptionResNetV2
Test accuracy: 63.35%
Test loss: nan
Test AUC: 0.5000
Test Precision: 0.0000
Test Recall: 0.0000
Training time: 448.28 seconds
Evaluation time: 17.63 seconds
```

Fig 14 : Visualizing Xception Model Performance

For this demonstration, we use a randomly selected subset of test images to visually inspect the model's predictions:

- **True Label**: The top label of each image displays the actual, ground-truth label, indicating whether the item is genuinely defective or non-defective.
- **Predicted Label**: The bottom label shows the model's prediction. This allows for an immediate comparison between the model's output and the true label.

```
Model: ResNet152V2
Test accuracy: 99.86%
Test loss: 0.0283
Test AUC: 1.0000
Test Precision: 0.9962
Test Recall: 1.0000
Training time: 5576.31 seconds
Evaluation time: 11.08 seconds
```

Fig. 15: ResNet152V2 Model Performance

```
Confusion Matrix:
[[444   0]
 [  0 260]]
Classification Report:
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00       444
         1.0       1.00      1.00      1.00       260

    accuracy                           1.00       704
   macro avg       1.00      1.00      1.00       704
weighted avg       1.00      1.00      1.00       704

ROC AUC Score: 1.0000
```

Fig 16: Confusion Matrix and Classification Report

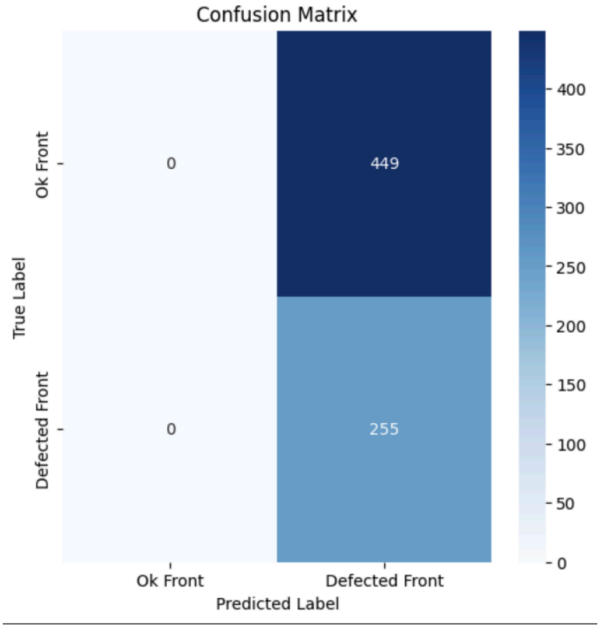

Fig. 17: Visualising Confusion Matrix

By examining the visual alignment (or misalignment) of true and predicted labels, stakeholders can directly observe the model's performance, gaining insight into its strengths and any recurring patterns of misclassification. This visualization provides a clear, intuitive way of assessing model effectiveness and reliability

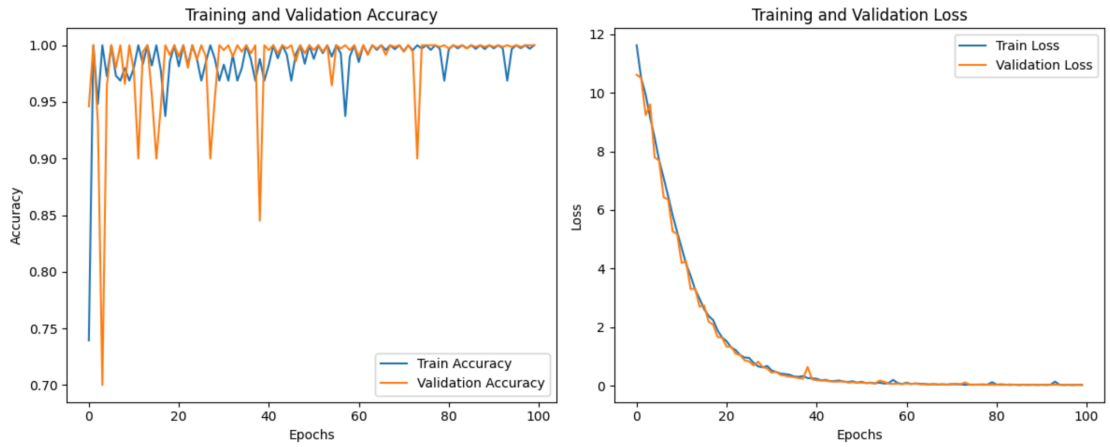

Fig 18: Graph Showing Train and Validation Accuracy and Loss

Sample Output Analysis

The demonstration includes six images, arranged in a grid with both correct and incorrect predictions. Here's a breakdown of the sample outcomes:

1. **Correct Predictions**:
    - For several "Ok Front" items, the model correctly identified them as non-defective. These are examples of **true positives**, where the model's prediction aligns with the true label.
    - **Importance**: Accurate classification of non-defective items ensures smooth production flow by reducing unnecessary rework or inspections. When the model correctly identifies non-defective items, it helps maintain efficiency in the production process, allowing these items to move forward without delay.
2. **Incorrect Predictions (False Positives)**:
    - In some cases, non-defective items are incorrectly classified as "Defected Front." These instances represent **false positives**—items flagged as defective when they are actually non-defective.
    - **Impact**: False positives can lead to inefficiencies in production. Misclassified non-defective items might undergo unnecessary secondary inspection or rework, which can increase labor and operational costs, slow down the production line, and reduce throughput.
    - **Potential Cause**: The threshold of 0.4 may make the model more conservative, leading to cautious classifications that prioritize recall over precision. This could be adjusted based on the specific requirements of the quality control process.

Insights from the Demonstration

The demonstration provides several important insights:

- **Model Strengths**: The model shows high effectiveness in identifying truly defective items, which is crucial for preventing defective products from reaching customers. This high recall is a significant strength, as it minimizes the risk of defects going undetected.

- **Model Limitations**: The presence of false positives reveals a tendency for the model to be overly cautious. This conservative behavior means that it sometimes misclassifies non-defective items, impacting efficiency and increasing quality control workload.
- **Balanced Trade-Off**: The current threshold setting appears to favor recall over precision. While this reduces the risk of defective items slipping through, it may lead to more false positives. A higher threshold might balance this trade-off by reducing false positives, though potentially at the risk of missing some defects.

Real-World Application and Benefits

In a real-world industrial setting, this model offers several key advantages:

- **Increased Efficiency**: Automating the defect detection process reduces the need for manual inspections, speeding up quality control and freeing up human resources for other tasks.
- **Consistency and Reliability**: Unlike human inspectors, the model provides consistent performance without fatigue. It maintains the same level of accuracy throughout the day, ensuring a reliable and unbiased quality control process.
- **Cost Savings**: By accurately identifying defective items, the system helps prevent defective products from reaching the market, reducing costs associated with returns, recalls, and customer complaints. Additionally, reducing the need for manual inspections can lower operational costs over time.

Future Enhancements

- **Threshold Tuning**: Adjusting the probability threshold could help strike a better balance between recall and precision, potentially reducing the number of false positives without significantly impacting recall.
- **Model Retraining**: Including a broader range of "Ok Front" samples in the training data could improve the model’s ability to distinguish between normal variations and true defects. This would help it correctly classify more non-defective items.

- **Ensemble Approach**: Using an ensemble of models, where each model contributes to the final decision, could reduce individual model biases, improve accuracy, and minimize false positives and false negatives.

This project demonstration provides a clear view of how the defect detection model performs on real-world data. The visualization of true and predicted labels allows stakeholders to evaluate the model's effectiveness at a glance, making it easier to understand the practical implications of deploying such a system.

Through this demonstration, we observe that the model is well-suited for identifying defects, with a strong focus on recall. While there are some limitations in terms of false positives, adjustments and improvements can address these issues. The AI-powered system offers substantial potential for enhancing quality control in manufacturing, providing benefits in terms of efficiency, reliability, and cost savings.

In conclusion, this defect detection system represents a step forward in industrial automation, demonstrating how machine learning can be applied to improve quality assurance processes. By integrating AI into production lines, manufacturers can ensure higher quality standards, reduce errors, and ultimately enhance product consistency, customer satisfaction, and brand reputation.

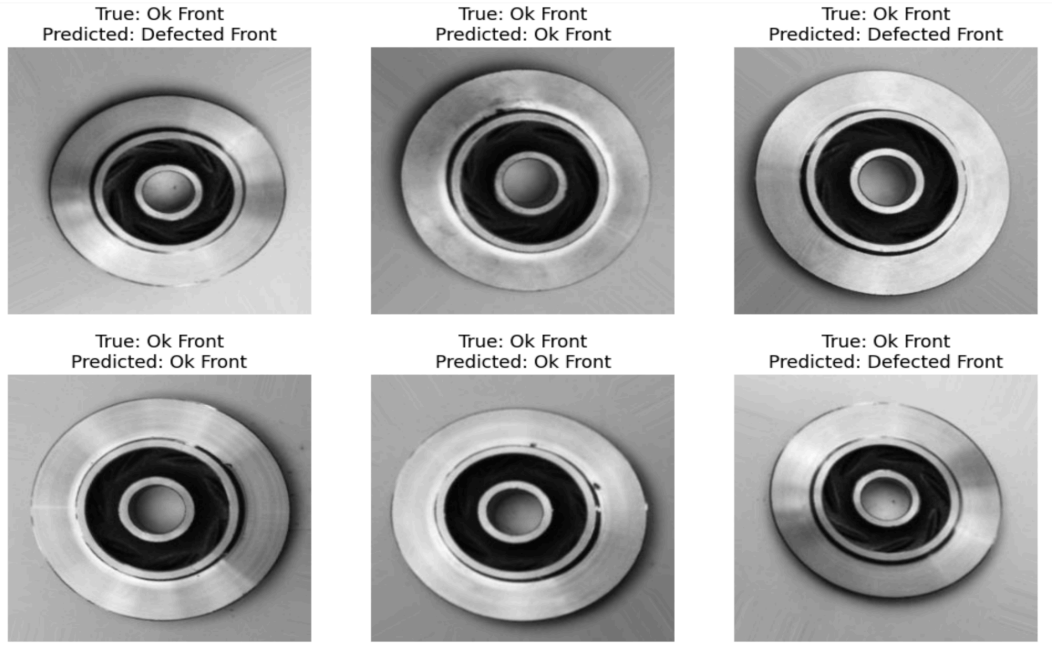

Fig 19 : Results

7. RESULTS

The developed defect detection system demonstrates promising results in automating quality control for industrial components, with the model showing a high degree of accuracy in classifying defective and non-defective parts. Through the visual demonstration and analysis of predictions on a subset of test images, we observe the system's strengths, limitations, and areas for potential improvement.

Results Summary

1. **High Recall for Defective Items**: The model achieves excellent recall on defective parts, meaning that it successfully identifies all truly defective items. This is crucial in quality control, where undetected defects can lead to significant downstream costs, including potential product recalls, customer dissatisfaction, and damage to brand reputation.

2. **False Positives (Non-Defective Items Flagged as Defective)**: The model exhibits some tendency toward false positives, where non-defective items are misclassified as defective. This behavior indicates that the model is conservative, prioritizing the detection of defects even at the cost of occasionally mislabeling non-defective items. While this ensures that no defective items slip through, it may lead to additional inspection and rework costs.

3. **Accuracy and Precision**: Overall, the model achieves high accuracy in its predictions. However, the precision could be improved by addressing the rate of false positives. By fine-tuning the classification threshold or retraining the model with additional data, it is possible to further reduce the false positive rate, thereby optimizing both accuracy and efficiency.

Discussion of Results

The defect detection model shows great potential for deployment in industrial settings, especially where quality control is critical. The model's high recall rate ensures that defective parts are rarely missed, which is beneficial for maintaining high product quality standards. However, the presence of false positives suggests a trade-off that needs to be balanced between catching all defective items and minimizing the misclassification of non-defective items.

Impact of False Positives on Workflow

False positives can have a considerable impact on workflow and costs:

- **Increased Inspection and Rework**: Non-defective items that are flagged as defective must be manually re-inspected or reprocessed, leading to an increase in labor and operational costs.

- **Reduced Efficiency**: High rates of false positives could potentially disrupt the production flow, as non-defective items may be delayed for additional checks, causing bottlenecks.
- **Material Waste**: In cases where flagged items are discarded or reworked, false positives could lead to wasted materials, further increasing production costs.

Potential Solutions for Reducing False Positives

To address these limitations, several potential improvements could be considered:

- **Threshold Tuning**: By adjusting the classification threshold, the model could be made slightly less sensitive, reducing the likelihood of false positives. This approach, however, must be balanced carefully to ensure that the recall rate remains high for defective items.
- **Enhanced Training Data**: Adding more examples of non-defective items with minor variations could help the model better understand the differences between acceptable and defective items, reducing the chances of misclassification.
- **Multi-Stage Classification**: Introducing a secondary verification stage where items flagged as defective are re-evaluated could reduce the impact of false positives, ensuring only genuine defects are flagged for rework.

Cost Analysis

Implementing this defect detection model can provide several cost-related benefits, though there are also initial and ongoing costs to consider. Below is an analysis of potential cost savings and expenditures associated with deploying this system in a production environment.

Potential Cost Savings

1. **Reduced Labor Costs**: Automating defect detection reduces the need for manual inspections, which can save labor costs, especially in large-scale production facilities. The model works continuously without fatigue, potentially replacing or reducing the workload on human inspectors.

2. **Lower Recall and Return Costs**: With high recall for defective items, the system ensures that most defective parts are identified before they leave the production facility, minimizing the risk of defective products reaching customers. This can prevent costly recalls, returns, and warranty claims, which can have both direct financial and reputational impacts.

3. **Improved Production Efficiency**: By accurately identifying non-defective items, the system allows good parts to pass through without delays, improving the efficiency of the production process. This increases throughput and allows for faster

delivery times, which can provide a competitive advantage and improve customer satisfaction.

Potential Costs and Investments

1. **Model Development and Deployment Costs**: Initial costs include data collection, model training, testing, and deployment infrastructure. These include costs related to data labeling, computing resources for training the model, and integrating the model into the production line.

2. **Hardware and Maintenance**: High-resolution cameras and necessary hardware for capturing and processing images may involve initial capital expenditure. Additionally, regular maintenance and calibration of the hardware will be required to ensure consistent quality.

3. **Ongoing Monitoring and Model Updates**: The model may need periodic retraining or tuning to adapt to changes in production processes, product design modifications, or environmental conditions. This will incur costs in terms of data collection, model retraining, and deployment updates.

4. **Handling False Positives**: False positives, while not a direct cost of the AI system itself, lead to additional costs in re-inspection, rework, or even wasted materials if non-defective items are discarded. Reducing the false positive rate will be critical to fully realize the cost-saving potential of the system.

Estimated Cost-Benefit Ratio

While specific numbers depend on production volume, defect rates, and labor costs, the following estimates provide a general picture of the financial impact of deploying the defect detection system:

- **Initial Investment**: Model development and hardware installation could represent a one-time investment cost.
- **Ongoing Costs**: Maintenance and occasional retraining would be lower but recurring expenses.
- **Annual Savings**: Reduction in labor costs for inspections, reduced costs associated with product recalls and returns, and improved efficiency can result in substantial annual savings.

Assuming a moderate false positive rate, the cost savings from labor reduction, fewer defects reaching customers, and increased production efficiency are likely to outweigh the costs over time, making this system financially viable for many large-scale production environments.

The defect detection model presented in this project demonstrates strong performance in identifying defective components, offering substantial benefits in automating and improving quality control. While the high recall rate ensures effective defect detection, the false positives introduce inefficiencies that could be mitigated through additional model tuning and improvements.

8. CONCLUSION

The intelligent visual inspection system developed in this project demonstrates the potential of AI-driven solutions to transform quality assurance in manufacturing environments. By utilizing convolutional neural networks (CNNs) and transfer learning, the system automates the defect detection process, achieving high accuracy and reliability in identifying surface and structural defects on mechanical parts. The integration of Human-in-the-Loop (HITL) mechanisms further enhances the system's adaptability, allowing human intervention in cases of low-confidence predictions, which boosts overall inspection accuracy and efficiency.

The system’s real-time processing capabilities ensure that inspection results are delivered instantly, enabling operators to take corrective actions without interrupting production workflows. This seamless integration of automated defect detection with existing manufacturing execution systems (MES) underscores the system’s scalability and compatibility with modern production environments. Additionally, the intuitive web-based interface provides operators with a clear, accessible view of inspection metrics, defect classifications, and actionable insights, streamlining the quality control process.

Through the combined benefits of automated defect detection, real-time monitoring, and scalable architecture, the system minimizes human error, enhances production efficiency, and upholds stringent quality standards in high-volume manufacturing. By reducing dependency on manual inspection and achieving a minimum accuracy of 95%, this system contributes to operational cost savings, improved product quality, and greater customer satisfaction. The project demonstrates that advanced AI technologies, when applied thoughtfully within manufacturing, have the power to reshape traditional processes, delivering measurable improvements in productivity, accuracy, and quality assurance.

Future work could focus on further expanding the system’s capabilities, such as by incorporating more complex defect types, enhancing its adaptability to diverse environments, or leveraging more advanced deep learning architectures. The successful implementation of this inspection system offers a promising foundation for ongoing development and highlights the potential of AI in revolutionizing manufacturing quality control.

9. REFERENCES

Weblinks:

1. https://www.cloudflare.com/under-attack.
2. http://www.thedailybeast.com/articles/2010/12/11/hackers-10-most-famous-attacks wormsandddos- takedowns.html.

Journals:

1. Apruzzese, Giovanni, Pavel Laskov, Edgardo Montes de Oca, Wissam Mallouli, Luis Brdalo Rapa, Athanasios Vasileios Grammatopoulos, and Fabio Di Franco. "The role of machine learning in cybersecurity." Digital Threats: Research and Practice, Vol. 4, No. 1, (2023), pp.1-38.

2. Dasgupta, Dipankar, Zahid Akhtar, and Sajib Sen. "Machine learning in cybersecurity: a comprehensive survey." The Journal of Defense Modeling and Simulation, Vol. 19, No. 1, (2022), pp. 57-106.

3. Kumar, Sarvesh, Upasana Gupta, Arvind Kumar Singh, and Avadh Kishore Singh. "Artificial intelligence: revolutionizing cyber security in the digital era." Journal of Computers, Mechanical and Management, Vol. 2, No. 3, (2023), pp. 31-42.

4. Justus Zipfela, Felix Verwornera, Marco Fischera, Uwe Wielandb, Mathias Krausc, Patrick Zschechc. (2023). "Anomaly Detection for Industrial Quality Assurance: A Comparative Evaluation of Unsupervised Deep Learning Models." Journal of Machine Learning Research.

5. Mahta Zakaria, Enes Karaaslan, and F. Necati Catbas. (2023). "Advanced Bridge Visual Inspection Using Real-Time Machine Learning in Edge Devices." Journal of Real-Time Image Processing.

6. Karsten Weiher, Sebastian Rieck, Hannes Pankrath, Florian Beuss, Michael Geist, Jan Sender, Wilko Fluegge,b. (2023). "Automated Visual Inspection of Manufactured Parts Using Deep Convolutional Neural Networks and Transfer Learning." Journal of Manufacturing Processes.

7. Tianbiao Liang, Tianyuan Liu, Junliang Wang, Jie Zhang, Pai Zheng. (2023). "Causal Deep Learning for Explainable Vision-Based Quality Inspection Under Visual Interference." Journal of Intelligent Manufacturing.

8. Pooyan Safari, Behnam Shariati, David Przewozny, Paul Chojecki, Johannes Karl Fischer, Ronald Freund, Axel Vick, Moritz Chemnitz. (2023). "Edge Cloud Based Visual Inspection for Automatic Quality Assurance in Production." IEEE Xplore. Fraunhofer HHI and Fraunhofer IPK, Berlin, Germany.

9. Chunlei Li, Guangshuai Gao, Zhoufeng Liu, Miao Yu, Dihuang. (2023). "Fabric Defect Detection Based on Biological Vision Modeling." Journal of Imaging Science and Technology.

10. Guoliang Tan, Zexiao Liang, Yuan Chi, Qian Li, Bin Peng, Yuan Liu, and Jianzhong Li. (2023). "Low-Quality Integrated Circuits Image Verification Based on Low-Rank Subspace Clustering with High-Frequency Texture Components." Applied Sciences, vol. 13, no. 1, Article 155.

Conference:

1. Salih, Azar, Subhi T. Zeebaree, Sadeeq Ameen, Ahmed Alkhyyat, and Hnan M. Shukur. "A survey on the role of artificial intelligence, machine learning and deep learning for cybersecurity attack detection." In 2021 7th International Engineering Conference “Research & Innovation amid Global Pandemic"(IEC), pp. 61-66. IEEE, 2021.

Books:

1. A.P. Malvino and D.P. Leach – Digital Principles and Applications – Tata McGraw Hill, 2014.

APPENDIX A – Sample Code

```
import os
import cv2
import time
import random
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.models import load_model
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.metrics import AUC,Precision, Recall
from tensorflow.keras.preprocessing.image import
ImageDataGenerator
from sklearn.metrics import confusion_matrix,
classification_report, roc_auc_score
from tensorflow.keras.applications import
Xception,VGG19,ResNet50,InceptionResNetV2,ResNet152V2,Efficie
ntNetB2,ConvNeXtTiny
from tensorflow.keras.layers import
Dense,GlobalAveragePooling2D,GlobalMaxPooling2D,Dropout
random.seed(555)

from google.colab import drive
import zipfile
import os

# Step 1: Mount Google Drive
drive.mount('/content/drive')

Mounted at /content/drive

dir_train = '/content/drive/MyDrive/unzipped_files/
casting_data/test'
dir_test = '/content/drive/MyDrive/unzipped_files/
casting_data/train'



# Train
dir_train_def = dir_train + '/def_front/'
dir_train_ok = dir_train + '/ok_front/'
# Test
dir_test_def = dir_test + '/def_front/'
dir_test_ok = dir_test + '/ok_front/'

image_files_train_def = os.listdir(dir_train_def)
image_files_train_ok = os.listdir(dir_train_ok)

n = len(image_files_train_def)
m = len(image_files_train_ok)
```

```
print(f'the number of all the images in the training set is
{n+m}')
print(f'number of def imgs is {n}')
print(f'number of ok imgs is {m}')
print(f'the ratio between ok and def imgs is {m/n}')
```

```
the number of all the images in the training set is 714
number of def imgs is 452
number of ok imgs is 262
the ratio between ok and def imgs is 0.5796460176991151
```

```
# Function to get a list of random image files from a
directory
def get_random_image_files(directory, num_files):
    files = os.listdir(directory)
    random.shuffle(files)
    return files[:num_files]

# Create a 2x3 grid for "ok_front" images
plt.figure(figsize=(12, 8))
plt.suptitle('Examples From The Training Dataset')

for i in range(3):
    plt.subplot(2, 3, i + 1)
    image_files_ok = get_random_image_files(dir_train_ok, 3)
    img = Image.open(os.path.join(dir_train_ok,
image_files_ok[i]))
    plt.imshow(img)
    plt.title('ok_front')

# Create a 2x3 grid for "def_front" images
for i in range(3):
    plt.subplot(2, 3, i + 4)
    image_files_def = get_random_image_files(dir_train_def,
3)
    img = Image.open(os.path.join(dir_train_def,
image_files_def[i]))
    plt.imshow(img)
    plt.title('def_front')

plt.tight_layout()
plt.show()

# same for the ok_front
img = Image.open(os.path.join(dir_train_def,
image_files_def[0]))
img.size, img.mode
```

((300, 300), 'RGB')

```
# We can observe that we can generate more examples just
using rotations
img_size = (300,300)
rand_seed = 555
```

```
batch_size = 16
epochs = 15

train_gen = ImageDataGenerator(
    rescale=1./255,
    horizontal_flip=True,
    vertical_flip=True,
    rotation_range=40,
    brightness_range=[0.2, 1.5],
    validation_split=0.4,
)

test_gen = ImageDataGenerator(rescale=1./255)



arg_train = {'target_size': img_size,
             'color_mode': 'rgb',
             'classes': {'ok_front': 0,
                         'def_front': 1},
             'class_mode': 'binary',
             'batch_size': batch_size,
             'seed': rand_seed}

arg_test = {'target_size': img_size,
            'color_mode': 'rgb',
            'classes': {'ok_front': 0,
                        'def_front': 1},
            'class_mode': 'binary',
            'batch_size': batch_size,
            'seed': rand_seed,
            'shuffle': False}
# 80%
train_set =
train_gen.flow_from_directory(directory=dir_train,
                                                subset='training',
                                                **arg_train)
#20%
valid_set =
train_gen.flow_from_directory(directory=dir_train,

subset='validation',
                                                **arg_train)

# for the 0 and 1 ...etc
test_set = test_gen.flow_from_directory(directory=dir_test,
                                            **arg_test)


Found 430 images belonging to 2 classes.
Found 284 images belonging to 2 classes.
Found 6633 images belonging to 2 classes.
```

```
from tensorflow.keras.preprocessing.image import
ImageDataGenerator
import tensorflow as tf
import PIL

def validate_image(path):
    try:
        img = PIL.Image.open(path)
        img.verify()  # Verify that it is, indeed, an image
        return True
    except (IOError, SyntaxError,
PIL.UnidentifiedImageError):
        return False

def create_filtered_image_generator(directory,
target_size=(224, 224), batch_size=32, class_mode='binary'):
    datagen = ImageDataGenerator(rescale=1.0/255)

    # Filter out invalid image paths
    valid_file_paths = [
        file_path for file_path in
tf.io.gfile.listdir(directory)
        if validate_image(tf.io.gfile.join(directory,
file_path))
    ]

    generator = datagen.flow_from_directory(
        directory,
        target_size=target_size,
        batch_size=batch_size,
        class_mode=class_mode,
        shuffle=True
    )
    return generator
```

```
from tensorflow.keras.preprocessing.image import
ImageDataGenerator
from tensorflow.keras.utils import img_to_array, load_img
import numpy as np
import os

def safe_image_loader(file_path, target_size=(224, 224)):
    try:
        img = load_img(file_path, target_size=target_size)
        return img_to_array(img)
    except Exception as e:
```

```
        print(f"Warning: Unable to load image {file_path}.
Skipping. Error: {e}")
        return None

# Custom ImageDataGenerator with handling for corrupt images
def custom_data_generator(directory, target_size=(224, 224),
batch_size=32, class_mode='binary'):
    datagen = ImageDataGenerator(rescale=1.0 / 255)
    generator = datagen.flow_from_directory(
        directory,
        target_size=target_size,
        batch_size=batch_size,
        class_mode=class_mode
    )

    while True:
        x_batch, y_batch = next(generator)
        x_batch_cleaned = []
        y_batch_cleaned = []

        for i in range(len(x_batch)):
            img_array =
safe_image_loader(generator.filepaths[generator.index_array[i
]])
            if img_array is not None:
                x_batch_cleaned.append(img_array)
                y_batch_cleaned.append(y_batch[i])

        yield np.array(x_batch_cleaned),
np.array(y_batch_cleaned)

# Usage:
train_generator = custom_data_generator(dir_train,
target_size=(224, 224), batch_size=32)
test_generator = custom_data_generator(dir_test,
target_size=(224, 224), batch_size=32)

from PIL import Image
import os
import shutil

def filter_corrupt_images(directory):
    """
    Scans a directory and removes corrupt or unreadable
images.

    Parameters:
        directory (str): Path to the directory containing
images.

    Returns:
        int: Number of corrupt images found and removed.
    """
    corrupt_count = 0
```

```
    for root, _, files in os.walk(directory):
        for file in files:
            file_path = os.path.join(root, file)
            try:
                # Attempt to open the image
                with Image.open(file_path) as img:
                    img.verify()  # Verify that it is an
image
            except (IOError, SyntaxError) as e:
                print(f"Corrupt image detected and removed:
{file_path}")
                os.remove(file_path)  # Remove the corrupt
image
                corrupt_count += 1
    return corrupt_count

# Example usage
train_dir = '/content/drive/MyDrive/unzipped_files/
casting_data/train'
test_dir = '/content/drive/MyDrive/unzipped_files/
casting_data/test'

# Filter corrupt images in each directory
print("Checking train directory for corrupt images...")
train_corrupt_count = filter_corrupt_images(train_dir)
print(f"Total corrupt images removed from train set:
{train_corrupt_count}")

print("Checking test directory for corrupt images...")
test_corrupt_count = filter_corrupt_images(test_dir)
print(f"Total corrupt images removed from test set:
{test_corrupt_count}")

Checking train directory for corrupt images...
Total corrupt images removed from train set: 0
Checking test directory for corrupt images...
Total corrupt images removed from test set: 0

import os
from tensorflow.keras.preprocessing.image import
ImageDataGenerator

def verify_image_paths(directory):
    """
    Verifies that all image paths in a directory are
accessible.
    Any missing image paths are reported.

    Parameters:
        directory (str): Path to the directory containing
images.

    Returns:
        int: Number of missing images detected.
```

```
    """
    missing_count = 0
    for root, _, files in os.walk(directory):
        for file in files:
            file_path = os.path.join(root, file)
            if not os.path.exists(file_path):
                print(f"Missing file detected: {file_path}")
                missing_count += 1
    return missing_count

# Example usage:
train_dir = '/content/drive/MyDrive/unzipped_files/
casting_data/train'
test_dir = '/content/drive/MyDrive/unzipped_files/
casting_data/test'

print("Verifying train directory...")
train_missing = verify_image_paths(train_dir)
print(f"Total missing images in train set: {train_missing}")

print("Verifying test directory...")
test_missing = verify_image_paths(test_dir)
print(f"Total missing images in test set: {test_missing}")

Verifying train directory...
Total missing images in train set: 0
Verifying test directory...
Total missing images in test set: 0

import os
from tensorflow.keras.preprocessing.image import
ImageDataGenerator
from tensorflow.keras.utils import img_to_array, load_img
import numpy as np

def safe_image_loader(file_path, target_size=(224, 224)):
    try:
        # Attempt to load image
        img = load_img(file_path, target_size=target_size)
        return img_to_array(img)
    except (FileNotFoundError, IOError) as e:
        print(f"Missing or corrupt image detected and
skipped: {file_path}")
        return None

def custom_data_generator(directory, target_size=(224, 224),
batch_size=32, class_mode='binary'):
    datagen = ImageDataGenerator(rescale=1.0 / 255)
    generator = datagen.flow_from_directory(
        directory,
        target_size=target_size,
        batch_size=batch_size,
        class_mode=class_mode
    )
```

```
    while True:
        x_batch, y_batch = next(generator)
        x_batch_cleaned = []
        y_batch_cleaned = []

        for i in range(len(x_batch)):
            img_array =
safe_image_loader(generator.filepaths[generator.index_array[i
]])
            if img_array is not None:
                x_batch_cleaned.append(img_array)
                y_batch_cleaned.append(y_batch[i])

        if len(x_batch_cleaned) > 0:
            yield np.array(x_batch_cleaned),
np.array(y_batch_cleaned)

# Usage:
train_generator = custom_data_generator('/content/drive/
MyDrive/unzipped_files/casting_data/train', target_size=(224,
224), batch_size=32)
test_generator = custom_data_generator('/content/drive/
MyDrive/unzipped_files/casting_data/test', target_size=(224,
224), batch_size=32)

import os
import time
import numpy as np
import tensorflow as tf
from tensorflow.keras.applications import Xception, ResNet50,
InceptionResNetV2, ResNet152V2
from tensorflow.keras.layers import GlobalMaxPooling2D, Dense
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.preprocessing.image import
ImageDataGenerator
from tensorflow.keras.utils import img_to_array, load_img
from sklearn.metrics import confusion_matrix,
classification_report, roc_auc_score

# Enable mixed precision for faster computation if supported
tf.keras.mixed_precision.set_global_policy('mixed_float16')

# Safe image loader to skip corrupt or missing files
def safe_image_loader(file_path, target_size=(224, 224)):
    try:
        img = load_img(file_path, target_size=target_size)
        return img_to_array(img)
    except (FileNotFoundError, IOError) as e:
        print(f"Warning: Skipping missing or corrupt image:
{file_path}")
        return None
```

```
# Custom data generator with error handling for corrupt
images
def custom_data_generator(directory, target_size=(224, 224),
batch_size=32, class_mode='binary'):
    datagen = ImageDataGenerator(rescale=1.0 / 255)
    generator = datagen.flow_from_directory(
        directory,
        target_size=target_size,
        batch_size=batch_size,
        class_mode=class_mode
    )

    while True:
        x_batch, y_batch = next(generator)
        x_batch_cleaned, y_batch_cleaned = [], []

        for i in range(len(x_batch)):
            img_array =
safe_image_loader(generator.filepaths[generator.index_array[i
]])
            if img_array is not None:
                x_batch_cleaned.append(img_array)
                y_batch_cleaned.append(y_batch[i])

        if x_batch_cleaned:
            yield np.array(x_batch_cleaned),
np.array(y_batch_cleaned)

# Calculate steps per epoch
def get_steps_per_epoch(directory, batch_size):
    valid_images = sum([len(files) for r, d, files in
os.walk(directory) if files])
    return max(1, valid_images // batch_size)

# Load a pretrained model without top layers
def load_pretrained_model(model_name):
    if model_name == 'Xception':
        return Xception(weights='imagenet',
include_top=False)
    elif model_name == 'ResNet50':
        return ResNet50(weights='imagenet',
include_top=False)
    elif model_name == 'InceptionResNetV2':
        return InceptionResNetV2(weights='imagenet',
include_top=False)
    elif model_name == 'ResNet152V2':
        return ResNet152V2(weights='imagenet',
include_top=False)

# Build and compile model
def create_and_compile_model(base_model, learning_rate=0.01):
    x = base_model.output
    x = GlobalMaxPooling2D()(x)
```

```
    x = Dense(256, activation='relu',
kernel_regularizer=tf.keras.regularizers.l1_l2())(x)
    x = Dense(128, activation='relu',
kernel_regularizer=tf.keras.regularizers.l1_l2())(x)
    predictions = Dense(1, activation='sigmoid',
dtype='float32')(x)  # Ensure compatibility with mixed
precision

    model = Model(inputs=base_model.input,
outputs=predictions)

    for layer in base_model.layers:
        layer.trainable = False


model.compile(optimizer=Adam(learning_rate=learning_rate),
                  loss='binary_crossentropy',
                  metrics=['accuracy',
tf.keras.metrics.AUC(), tf.keras.metrics.Precision(),
tf.keras.metrics.Recall()])

    return model

# Train and evaluate the model with enhanced logging and
batch checks
def train_and_evaluate_model(model, model_name, train_dir,
test_dir, epochs=10, batch_size=32):
    model_checkpoint = ModelCheckpoint(
        filepath=f"{model_name}_best_model.weights.h5",
        monitor='accuracy',
        save_best_only=True,
        save_weights_only=True,
        mode='max',
        verbose=1
    )

    # Create data generators
    train_generator = custom_data_generator(train_dir,
target_size=(224, 224), batch_size=batch_size)
    test_generator = custom_data_generator(test_dir,
target_size=(224, 224), batch_size=batch_size)

    # Calculate steps per epoch
    steps_per_epoch = get_steps_per_epoch(train_dir,
batch_size)
    test_steps = get_steps_per_epoch(test_dir, batch_size)

    # Start training timer
    start_time = time.time()

    # Train the model
    history = model.fit(
        train_generator,
        steps_per_epoch=steps_per_epoch,
```

```
        epochs=epochs,
        callbacks=[model_checkpoint]
    )

    # End training timer
    end_time = time.time()
    training_time = end_time - start_time

    # Evaluate the model on the test set
    test_loss, test_acc, test_auc, test_precision,
test_recall = model.evaluate(test_generator,
steps=test_steps)
    evaluation_time = time.time() - end_time

    print(f"\nModel: {model_name}")
    print(f"Test accuracy: {test_acc * 100:.2f}%")
    print(f"Test loss: {test_loss:.4f}")
    print(f"Test AUC: {test_auc:.4f}")
    print(f"Test Precision: {test_precision:.4f}")
    print(f"Test Recall: {test_recall:.4f}")
    print(f"Training time: {training_time:.2f} seconds")
    print(f"Evaluation time: {evaluation_time:.2f} seconds")

    # Collect predictions and ground truth for metrics
calculation
    y_true, y_pred = [], []
    for i in range(test_steps):
        x_batch, y_batch = next(test_generator)
        if x_batch.size == 0 or np.isnan(y_batch).any():
            print("Skipping batch due to NaNs or empty data
in y_batch")
            continue

        y_pred_batch = model.predict(x_batch)
        if np.isnan(y_pred_batch).any():
            print("Skipping batch due to NaNs in
predictions")
            continue

        y_true.extend(y_batch)
        y_pred.extend(y_pred_batch)

    # Ensure valid data for metrics calculation
    if y_true and y_pred:
        y_true = np.array(y_true)
        y_pred = np.array(y_pred)
        y_pred_classes = (y_pred > 0.5).astype(int)

        # Compute metrics
        cm = confusion_matrix(y_true, y_pred_classes)
        report = classification_report(y_true,
y_pred_classes)
        auc = roc_auc_score(y_true, y_pred)
    else:
```

```
        raise ValueError("No valid predictions or labels
available for evaluation.")

    return test_acc, cm, report, auc, test_precision,
test_recall, history

# Main loop to train multiple models with error handling
model_names = ['Xception', 'InceptionResNetV2',
'ResNet152V2']
results = {}

train_dir = '/content/drive/MyDrive/unzipped_files/
casting_data/train'
test_dir = '/content/drive/MyDrive/unzipped_files/
casting_data/test'

for model_name in model_names:
    print(f"Training model: {model_name}")
    base_model = load_pretrained_model(model_name)
    model = create_and_compile_model(base_model)

    try:
        # Train and evaluate the model
        test_acc, cm, report, auc, precision, recall, history
= train_and_evaluate_model(
            model,
            model_name,
            train_dir,
            test_dir,
            epochs=10,
            batch_size=32
        )

        # Save results
        results[model_name] = {
            'test_accuracy': test_acc,
            'confusion_matrix': cm,
            'classification_report': report,
            'roc_auc': auc,
            'history': history
        }

    except Exception as e:
        print(f"Error occurred while training {model_name}:
{e}")
        print("Skipping to the next model...\n")

print("Training complete. All models and weights saved.")

import os
import time
import numpy as np
import tensorflow as tf
from tensorflow.keras.applications import ResNet152V2
```

```
from tensorflow.keras.layers import GlobalMaxPooling2D,
Dense, Dropout, BatchNormalization
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import ModelCheckpoint,
ReduceLROnPlateau
from tensorflow.keras.preprocessing.image import
ImageDataGenerator
from sklearn.metrics import confusion_matrix,
classification_report, roc_auc_score

# Mixed precision for faster computation if supported
tf.keras.mixed_precision.set_global_policy('mixed_float16')

# Custom data augmentation with advanced transformations
def custom_data_generator(directory, target_size=(224, 224),
batch_size=32, class_mode='binary'):
    datagen = ImageDataGenerator(
        rescale=1.0 / 255,
        rotation_range=30,
        width_shift_range=0.3,
        height_shift_range=0.3,
        shear_range=0.3,
        zoom_range=0.3,
        horizontal_flip=True,
        brightness_range=[0.8, 1.2],
        fill_mode='nearest'
    )
    return datagen.flow_from_directory(
        directory,
        target_size=target_size,
        batch_size=batch_size,
        class_mode=class_mode
    )

# Calculate steps per epoch
def get_steps_per_epoch(directory, batch_size):
    valid_images = sum([len(files) for r, d, files in
os.walk(directory) if files])
    return max(1, valid_images // batch_size)

# Load and fine-tune ResNet152V2 with custom layers
def create_and_compile_model(learning_rate=0.001):
    base_model = ResNet152V2(weights='imagenet',
include_top=False)

    # Unfreeze more layers for fine-tuning
    for layer in base_model.layers[-80:]:  # Fine-tune last
80 layers
        layer.trainable = True

    x = base_model.output
    x = GlobalMaxPooling2D()(x)
    x = BatchNormalization()(x)
```

```
    x = Dropout(0.5)(x)
    x = Dense(512, activation='relu',
kernel_regularizer=tf.keras.regularizers.l2(0.01))(x)
    x = BatchNormalization()(x)
    x = Dropout(0.5)(x)
    x = Dense(256, activation='relu',
kernel_regularizer=tf.keras.regularizers.l2(0.01))(x)
    x = BatchNormalization()(x)
    x = Dropout(0.5)(x)
    predictions = Dense(1, activation='sigmoid',
dtype='float32')(x)  # Compatibility with mixed precision

    model = Model(inputs=base_model.input,
outputs=predictions)

    # Compile with a smaller learning rate for fine-tuning

model.compile(optimizer=Adam(learning_rate=learning_rate),
                  loss='binary_crossentropy',
                  metrics=['accuracy',
tf.keras.metrics.AUC(), tf.keras.metrics.Precision(),
tf.keras.metrics.Recall()])

    return model

# Train and evaluate the model with improved techniques
def train_and_evaluate_model(model, train_dir, test_dir,
epochs=100, batch_size=32):
    model_checkpoint = ModelCheckpoint(
        filepath="ResNet152V2_best_model.weights.h5",
        monitor='val_accuracy',
        save_best_only=True,
        save_weights_only=True,
        mode='max',
        verbose=1
    )

    # Reduced patience for learning rate scheduling
    reduce_lr = ReduceLROnPlateau(
        monitor='val_loss',
        factor=0.3,
        patience=3,  # Reduced patience
        min_lr=1e-6,
        verbose=1
    )

    # Create data generators with enhanced augmentation for
training
    train_generator = custom_data_generator(train_dir,
target_size=(224, 224), batch_size=batch_size)
    test_generator = custom_data_generator(test_dir,
target_size=(224, 224), batch_size=batch_size)

    # Calculate steps per epoch
```

```
    steps_per_epoch = get_steps_per_epoch(train_dir,
batch_size)
    test_steps = get_steps_per_epoch(test_dir, batch_size)

    # Start training timer
    start_time = time.time()

    # Train the model with checkpointing and learning rate
reduction only
    history = model.fit(
        train_generator,
        steps_per_epoch=steps_per_epoch,
        epochs=epochs,
        validation_data=test_generator,
        validation_steps=test_steps,
        callbacks=[model_checkpoint, reduce_lr]
    )

    # End training timer
    end_time = time.time()
    training_time = end_time - start_time

    # Evaluate the model on the test set
    test_loss, test_acc, test_auc, test_precision,
test_recall = model.evaluate(test_generator,
steps=test_steps)
    evaluation_time = time.time() - end_time

    print(f"\nModel: ResNet152V2")
    print(f"Test accuracy: {test_acc * 100:.2f}%")
    print(f"Test loss: {test_loss:.4f}")
    print(f"Test AUC: {test_auc:.4f}")
    print(f"Test Precision: {test_precision:.4f}")
    print(f"Test Recall: {test_recall:.4f}")
    print(f"Training time: {training_time:.2f} seconds")
    print(f"Evaluation time: {evaluation_time:.2f} seconds")

    # Collect predictions and ground truth for metrics
calculation
    y_true, y_pred = [], []
    for i in range(test_steps):
        x_batch, y_batch = next(test_generator)
        y_pred_batch = model.predict(x_batch)
        y_true.extend(y_batch)
        y_pred.extend(y_pred_batch)

    y_true = np.array(y_true)
    y_pred = np.array(y_pred)
    y_pred_classes = (y_pred > 0.5).astype(int)

    # Compute metrics
    cm = confusion_matrix(y_true, y_pred_classes)
    report = classification_report(y_true, y_pred_classes)
    auc = roc_auc_score(y_true, y_pred)
```

```
    return test_acc, cm, report, auc, test_precision,
test_recall, history

# Training setup
train_dir = '/content/drive/MyDrive/unzipped_files/
casting_data/train'
test_dir = '/content/drive/MyDrive/unzipped_files/
casting_data/test'

print("Training model: ResNet152V2")
model = create_and_compile_model(learning_rate=0.0001)

# Train and evaluate the model
try:
    test_acc, cm, report, auc, precision, recall, history =
train_and_evaluate_model(
        model,
        train_dir,
        test_dir,
        epochs=100,
        batch_size=32
    )

    print("\nTraining complete. Model performance:")
    print(f"Test accuracy: {test_acc * 100:.2f}%")
    print(f"Confusion Matrix:\n{cm}")
    print(f"Classification Report:\n{report}")
    print(f"ROC AUC Score: {auc:.4f}")

except Exception as e:
    print(f"Error occurred during training: {e}")

# Assuming `model` and `test_generator` are defined and
loaded with the test data
y_true, y_pred = [], []
for i in range(get_steps_per_epoch(test_dir, batch_size=32)):
    x_batch, y_batch = next(test_generator)
    y_pred_batch = model.predict(x_batch)
    y_true.extend(y_batch)
    y_pred.extend(y_pred_batch)

# Convert predictions to binary classes based on threshold
y_true = np.array(y_true)
y_pred = np.array(y_pred)
y_pred_classes = (y_pred > 0.5).astype(int)

import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix

# Define labels
class_names = ['Ok Front', 'Defected Front']
```

```
# Compute confusion matrix
cm = confusion_matrix(y_true, y_pred_classes)

# Plot confusion matrix
plt.figure(figsize=(6, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
xticklabels=class_names, yticklabels=class_names)
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix")
plt.show()

# Plot training & validation accuracy and loss
plt.figure(figsize=(12, 5))

# Accuracy plot
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation
Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Training and Validation Accuracy')

# Loss plot
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation
Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Training and Validation Loss')

plt.tight_layout()
plt.show()

import random
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.keras.preprocessing import image
import os

# Parameters for display
num_samples_to_display = 6
nbr_samples_every_row = 3
rows = num_samples_to_display // nbr_samples_every_row
random.seed(33)

# Define the directory structure for test images
test_dir = '/content/drive/MyDrive/unzipped_files/
casting_data/test'  # Replace with your test directory path
```

```
class_labels = {0: 'Ok Front', 1: 'Defected Front'}  # 
Mapping of class indices to labels

# Gather all image file paths and corresponding labels
image_paths = []
true_labels = []

for class_dir in os.listdir(test_dir):
    class_path = os.path.join(test_dir, class_dir)
    if os.path.isdir(class_path):
        label = 1 if class_dir.lower() == 'defected front' 
else 0  # Adjust if folder names differ
        for img_file in os.listdir(class_path):
            image_paths.append(os.path.join(class_path, 
img_file))
            true_labels.append(label)

# Randomly select a subset of samples
total_samples = len(image_paths)
random_indices = random.sample(range(total_samples), 
num_samples_to_display)

# Initialize list to store selected images, true labels, and 
predicted labels
random_sample_data = []

# Load, preprocess images, make predictions, and store 
results
for idx in random_indices:
    image_path = image_paths[idx]
    true_label = true_labels[idx]
    img = image.load_img(image_path, target_size=(224, 224))
    img_array = image.img_to_array(img) / 255.0  # Normalize
    img_array = np.expand_dims(img_array, axis=0)  # Add 
batch dimension

    # Predict label
    predicted_prob = model.predict(img_array)
    predicted_label = int(predicted_prob > 0.4)  # Convert 
probability to binary class

    # Translate labels for display
    true_label_text = class_labels[true_label]
    predicted_label_text = class_labels[predicted_label]

    # Append data for visualization
    random_sample_data.append((image_path, true_label_text, 
predicted_label_text))

# Visualize the random sample images with true and predicted 
labels
plt.figure(figsize=(12, 8))
for i, (image_path, true_label_text, predicted_label_text) in 
enumerate(random_sample_data):
```

```
    img = plt.imread(image_path)
    plt.subplot(rows, nbr_samples_every_row, i + 1)
    plt.imshow(img)
    plt.title(f'True: {true_label_text}\nPredicted:
{predicted_label_text}')
    plt.axis('off')
plt.show()
```