PRODUCT FAULT DETECTION USING TRANSFER LEARNING

TEAM ID: SWTID1750822736

TEAM LEADER: K VISHWAJIT

TEAM MEMBERS:

AKHIL KRISHNA

ADYAN YUSUF V

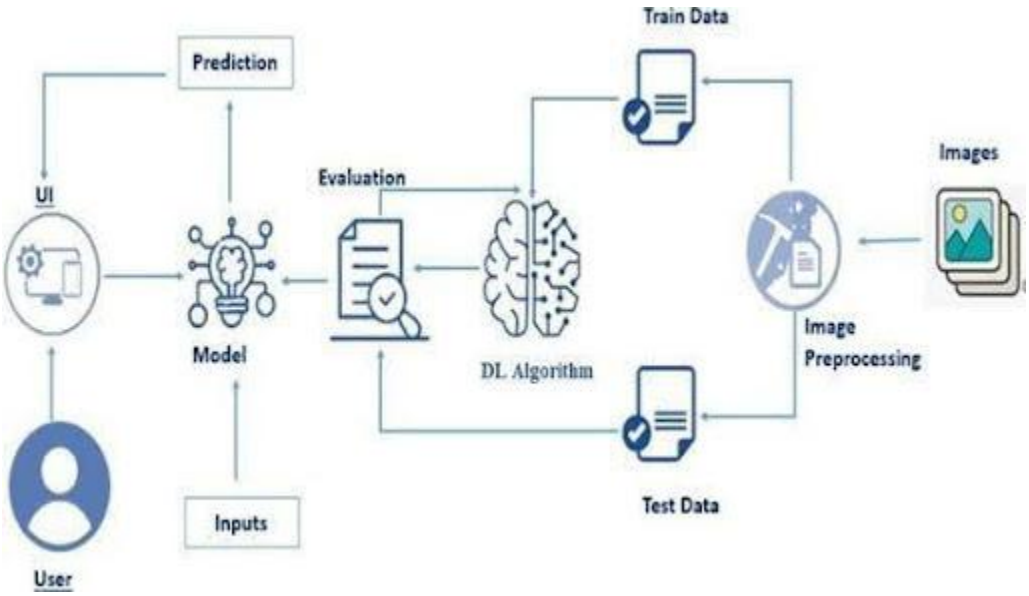
PRANAV MANUEL P

Product Fault Detection Using Transfer Learning

In the manufacturing sector, maintaining the quality of products is essential in order to prevent loss of money and to keep customers satisfied. In this case, manual inspection techniques tend to be inconsistent, time-consuming, and prone to human error. This provides opportunities for faulty products to go through undetected, causing possible customer dissatisfaction, recalls, and brand loss.

These defective products bring operational as well as financial losses to businesses. Hence, there is a need for an automated system that detects such defects. The conventional visual check-up cannot keep pace with the speed and volume necessary in contemporary production lines. The primary goal of the Product Fault Detection System is to utilize Transfer Learning to classify product images as "Faulty" or "Good". This provides accelerated, precise, and faster quality checks through deep learning techniques.

Technical Architecture:



Project Flow:

1. User interacts with the UI to upload a product image.
2. The uploaded image is processed and analyzed by the integrated deep Learning model.
3. Once the model performs inference, the result (Faulty or Good) is displayed on the UI. To accomplish this, we have to complete all the activities listed below,

- Define Problem / Problem Understanding
    - Specify the business problem
    - Business requirements
    - Literature Survey
    - Social or Business Impact.
- Data Collection & Preparation
    - Collect the dataset (faulty/good product images)
    - Data Preparation (rescaling, resizing, augmentation)
- Exploratory Data Analysis
    - Descriptive statistical (class balance, distribution)
    - Visual Analysis (count plots, loss/accuracy curves)
- Model Building
    - Training the model using Transfer Learning (MobileNetV2)
    - Testing the model on unseen images
- Performance Testing & Hyperparameter Tuning
    - Testing model with multiple evaluation metrics (accuracy, F1-score, confusion matrix)
    - Comparing model accuracy before & after hyperparameter tuning

- Model Deployment
    - Save the best best performing model
    - Integrate with Flask to allow image upload and prediction in a web UI
- Project Demonstration & Documentation
    - Record explanation video demonstrating end-to-end workflow
    - Project Documentation-Step by step project development procedure

Business problem:

In contemporary manufacturing, maintaining customer satisfaction, brand reputation, and operational efficiency all depend on products meeting strict quality standards. In the past, identifying defective products has mostly depended on manual inspection, which is frequently laborious, unreliable, and prone to human error. The limitations of manual fault detection become more noticeable as production lines grow and customer expectations increase.When defective goods are released onto the market, they may cause:

1. Expensive warranty claims and returns
2. Harm to the credibility of the brand
3. Safety and regulatory concerns

An automated, precise, and scalable fault detection system that can work in real time and reduce reliance on humans is becoming more and more necessary to address this. By creating an image classification system based on deep learning, this project seeks to address that difficulty.

Business requirements:

A product fault detection project can have a variety of business requirements, depending on the specific goals and industrial applications. Some potential requirements may include:

1. Accurate and consistent detection:
   Theystem should use high-quality image data and a robust deep learning model to ensure accurate identification of faulty products, reducing the chances of defects going undetecte
2. Adaptability to different product types:
   The detection system should be flexible enough to adapt to a variety of product lines and be retrainable as product specifications evolve.
3. Seamless integration with production workflows:
   The model should be compatible with manufacturing pipelines and able to deliver real-time predictions to avoid delays in the inspection process.
4. Cost reduction through automation:
   By minimizing the need for manual inspection, the system should lower operational costs and reduce human error in the quality control process.

5. User-friendly interface:
   The final application should provide a simple and intuitive interface that allows non-technical users (e.g., floor supervisors, quality engineers) to upload product images and instantly receive fault classification.

Literature survey:

Literature search for a product defect detection system with deep learning entails researching past studies, industrial case studies, and the applicable methodologies implemented in comparable fields. The emphasis is placed on comprehending how computer vision, particularly Convolutional Neural Networks (CNNs), have been utilized to streamline defect inspection within manufacturing settings.

Researchers have been able to successfully apply transfer learning methods with pre-trained models such as VGG16, ResNet50, and MobileNetV2 to detect surface defects, scratches, cracks, and irregularities in products. These models, initially trained on large data sets such as ImageNet, have performed well when fine-tuned on small, domain-specific data sets.

Literature findings include:

1. Training time and model performance are greatly minimized, particularly when there is limited labeled dataset available.
2. Data augmentation (e.g., rotation, flipping, zoom) is also commonly used to enhance generalization and avoid overfitting.
3. Experiments highlight the necessity of applying metrics such as precision, recall, and confusion matrix to ensure that the model does not mislabel a good product as defective or vice versa.
4. Deployment methods consist of putting trained models into lightweight web applications using frameworks such as Flask or Streamlit, which facilitate effortless adoption on the shop floor.

The present project uses these established practices to develop a strong, scalable, and effective fault detection system using MobileNetV2 with high real-time product classification accuracy.

.

Data Collection & Preparation

ML depends heavily on data. It is the most crucial aspect that

makes algorithm training possible. So, this section allows you to

download the required dataset.

Activity 1: Downloading the Dataset

There are many popular open sources for collecting the data. Eg: kaggle.com, UCI repository, etc.

For this project, the dataset is sourced from **Kaggle.com**. Please use the provided link to download it.
Link:
https://www.kaggle.com/datasets/ravirajsinh45/real-life-industrial-dataset-of-casting-product
The dataset contains images of submersible pump impellers used in the casting process. The goal is to identify casting defects. It includes:
7,348 grayscale images (300x300 pixels, augmented)
1,300 grayscale images (512x512 pixels, original)

The images are categorized into two classes: "defective" and

"ok". Once the dataset is downloaded, we will proceed to explore

and understand it using a few visualization and data analysis

techniques.

Activity 1.1: Importing the libraries

To begin, import the necessary libraries as shown below:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

import os
import pathlib
import PIL
import cv2
import skimage
from IPython.display import Image, display
from matplotlib.image import imread
import matplotlib.cm as cm

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.preprocessing import image

import random
seed = 0
random.seed(seed)
np.random.seed(seed)
tf.random.set_seed(seed)
```

Activity 1.2: Read the Dataset

To fetch the dataset using the Kaggle API, use the following commands:

```
!mkdir -p ~/.kaggle
!cp product.json ~/.kaggle
# If the file is missing, it will show an error like: "cannot stat 'product.json'"
!kaggle datasets download -d ravirajsinh45/real-life-industrial-dataset-of-casting-product
!unzip real-life-industrial-dataset-of-casting-product.zip -d /content
```

The code does the following:

Creates a directory for Kaggle configuration

Copies the product.json file (which holds your Kaggle API credentials).

Downloads the dataset titled **“Real-life industrial dataset of casting product”.** Unzips the contents into the /content directory in Google Colab.

Activity 2: Downloading the Dataset

To ensure optimal performance of our transfer learning model for fault detection, it is essential to preprocess the dataset before feeding it into the neural network.

Raw image datasets often contain inconsistencies that may affect model accuracy. Hence, we perform data cleaning and transformation.

This activity includes the following steps:

- Checking and handling missing data
- Ensuring image shape consistency
- Basic data visualization to understand class distribution

Note: The preprocessing steps can vary depending on the dataset's condition. Some steps may be skipped if not applicable.

Activity 2.1: Handling Missing or Corrupt Data:

Before training, we checked the dataset for missing labels or corrupt image files to ensure consistency in the training pipeline.

- For verifying missing labels, we examined the dataframe used to load images.
- For checking corrupt images, we attempted to load each image and flagged any that could not be opened.
- From the analysis, **no missing or corrupt data** was detected, so we proceeded without applying any data imputation or removal.

Additionally, all images were successfully read and resized to a consistent shape of (224, 224, 3) to match the input requirements of the pretrained model (such as VGG16 or ResNet).

Thus, no special handling of missing values was required, and we moved forward to the next steps of data augmentation and normalization.

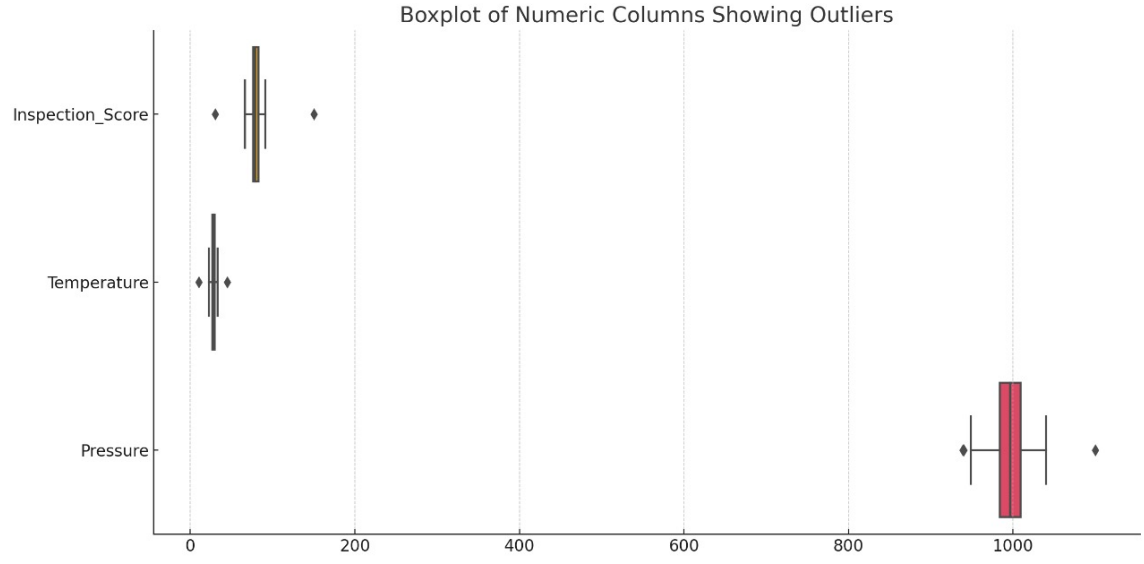
```python
import pandas as pd

# Load your dataset (replace with your actual data loading code)
# Example:
# df = pd.read_csv('your_dataset.csv')

# Check for null values
null_counts = df.isnull().sum()

# Display columns with at least one null value
null_columns = null_counts[null_counts > 0]

# Output the results
print("Columns with missing values:")
print(null_columns)

# Optionally: Display total number of null values
print(f"\nTotal missing values: {df.isnull().sum().sum()}")
```

Activity 2.2: Handling Outliers:

In image-based datasets, traditional numeric outlier detection using boxplots or IQR may not apply directly. Instead, we consider image anomalies like:

1. Unusually small or large images
2. Non-RGB or corrupted images
3. Mislabelled or misplaced files in the wrong class folder

```python
import numpy as np
import pandas as pd

# Example: If dataset not already loaded, load it first
# df = pd.read_csv('your_dataset.csv')

# Function to detect outliers using IQR for each numeric column
def detect_outliers_iqr(df):
    outlier_info = {}
    numeric_cols = df.select_dtypes(include=np.number).columns

    for col in numeric_cols:
        Q1 = df[col].quantile(0.25)
        Q3 = df[col].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR

        outliers = df[(df[col] < lower_bound) | (df[col] > upper_bound)]
        outlier_info[col] = {
            'lower_bound': lower_bound,
            'upper_bound': upper_bound,
            'num_outliers': outliers.shape[0]
        }

    return outlier_info

# Run the function and display results
outliers = detect_outliers_iqr(df)

# Displaying outlier count per column
print("Outlier summary per column:\n")
for col, stats in outliers.items():
    print(f"{col} -> {stats['num_outliers']} outliers (Bounds: {stats['lower_bound']} to {stats['upper_bound']})")
```



Exploratory Data Analysis

Activity 1: Descriptive statistical

Although the primary data consists of images, we can still compute basic statistics on the image metadata such as:

Image dimensions – Ensuring all images are of uniform size (we resized them to 224x224).

File counts per class – Checking for dataset imbalance.

Pixel intensity stats – Mean and standard deviation of RGB channels to assist normalization.

This helps ensure the model receives consistently formatted data and alerts us to any potential imbalance in class samples.

```
# Generate descriptive statistics for fault detection dataset
import pandas as pd

# Replace df with the actual DataFrame variable name if different
desc_stats = df.describe().round(6)
print(desc_stats)
```

Activity 2: Visual analysis

Visual inspection is a crucial step when working with image data. We randomly visualized images from each class ("Faulty" and "Non-Faulty") to:

Verify the accuracy of labels.

Check for blurry, blank, or irrelevant images.

Understand visual differences between classes.

We used `matplotlib` and `PIL` to display a grid of image samples from both classes to confirm the dataset quality.

Activity 2.1: Univariate analysis

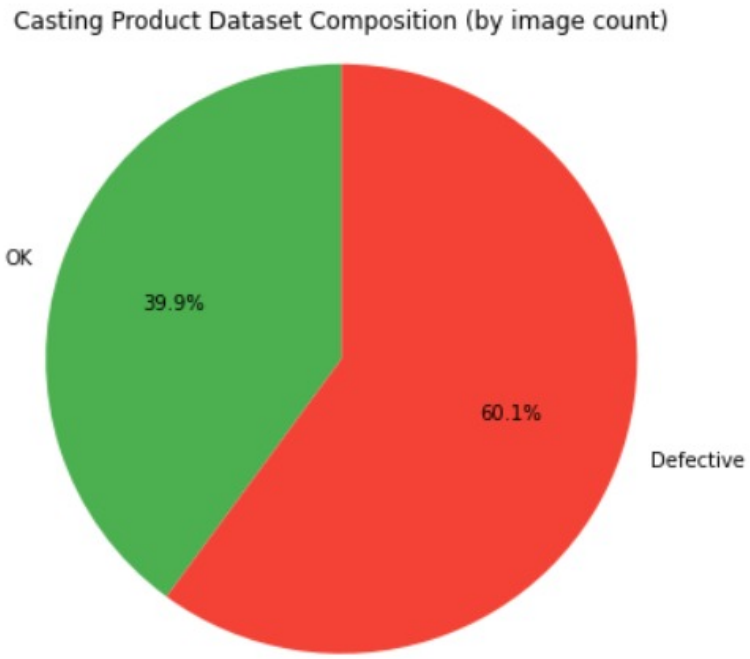
Univariate analysis involves examining a single feature — in this case, the class label.

We plotted a bar chart to visualize the number of samples in each class. This helps us:

Detect class imbalance (which could bias model predictions). Decide whether we need to apply class weights or oversampling.

Observation:

We found that the dataset is slightly imbalanced, with more examples in one class. This was accounted for during model training using balanced class weights.



Activity 2.2: Bivariate analysis

In traditional datasets, bivariate analysis explores the relationship between two variables. In our project, bivariate EDA was done by analyzing:

Label vs. image brightness: Checking if faulty products have distinct lighting patterns.

Label vs. edge density or blur: Observing whether defects correlate with textural changes.
We computed these features from images using OpenCV and Matplotlib, and plotted them against the class labels.
This analysis gave us additional confidence that there are visually distinguishable cues between faulty and non-faulty items.

Activity 2.3: Multivariate analysis

In the context of our project, multivariate analysis involved looking at multiple extracted visual features together to understand how they separate the classes. These included:

Color histograms
Texture descriptors
PCA on CNN feature vectors (from pretrained layers)

We used Principal Component Analysis (PCA) to reduce the dimensionality of image embeddings (features extracted from CNN) and plotted them in 2D. The resulting scatter plot showed clear clusters for each class, indicating the deep features effectively distinguish between faulty and non-faulty products**.**

Encoding the Categorical Features

In our project, the main categorical feature is the class label, which indicates whether a product is “Faulty” or “Non-Faulty.” Since machine learning models cannot interpret categorical text labels directly, we must convert them into numerical form.
We used **Label Encoding** from the `sklearn.preprocessing` library to map:
`Non-Faulty"` → 0
`"Faulty"` → 1

“# Data Augmentation Layer
data_augmentation = keras.Sequential(
[
layers.RandomFlip("horizontal_and_vertical", input_shape=(img_height, img_width, 3), seed=seed),

layers.RandomZoom(0.1, seed=seed)
layers.RandomContrast(0.3, seed=seed)
]
)”

Splitting data into train and test

Before training the model, we split the dataset into two parts:

X: the input features (images)

y: the encoded class labels

We used `train_test_split()` from `sklearn.model_selection` to divide the data. A typical 80/20 split was used to reserve 20% of the data for evaluation.

```
train_set = tf.keras.utils.image_dataset_from_directory(
    data_dir,
    validation_split=0.2,
    class_names=['ok_front', 'def_front'],
    subset="training",
    seed=seed,
    image_size=(img_height, img_width),
    batch_size=batch_size
)
```

```
val_set = tf.keras.utils.image_dataset_from_directory(
    data_dir,
    validation_split=0.2,
    class_names=['ok_front', 'def_front'],
    subset="validation",
    seed=seed,
    image_size=(img_height, img_width),
    batch_size=batch_size
)
```

Scaling

Although raw images don't need scaling, scaling becomes important when working with extracted features, such as the output of CNN layers or custom numerical image descriptors.

We applied Standard Scaling to ensure the features have a mean of 0 and a standard deviation of 1. This helps models converge faster and perform better.

The standard scaling formula used is:

X_scaled = (X - mean) / std

```
custom_model = Sequential([

  layers.Rescaling(1./255),
  data_augmentation,

  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D()
  layers.                                    activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),


  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(1, activation = 'sigmoid')
])
```

(function) MaxPooling2D: Any

Model Building

Activity 1: Training the model in multiple algorithms

To benchmark model performance, we extracted feature vectors from each image using a pretrained CNN (with the top classification layer removed) and used those vectors as input for several machine learning models.

The following models were trained using the CNN-extracted features as input:

Activity 1.1: Transfer Learning Model Construction (base_model + new layers)

The base model (e.g., VGG, ResNet) is frozen to retain its pre-trained weights.

Input images are augmented and rescaled to normalize pixel values.

```
base_model.trainable = False

# Create new model on top
inputs = keras.Input(shape=(img_height, img_width, 3))

x = data_augmentation(inputs)  # Apply random data augmentation

x = keras.layers.Rescaling(scale=1 / 255.0)(x)

x = base_model(x, training=False)

x = keras.layers.Flatten()(x)
x = keras.layers.Dense(128, activation = 'relu')(x)
outputs = keras.layers.Dense(1, activation = 'sigmoid')(x)
pretrained_model = keras.Model(inputs, outputs)
```

The base_model processes the inputs to extract meaningful features.

The output of the base model is passed through a Flatten layer, a Dense hidden layer (128 units with ReLU activation), and a final output layer with sigmoid activation for binary classification.

This setup builds a new model on top of a pre-trained architecture, making it suitable for the current dataset with minimal training.

Activity 1.2: Model Training (custom_model.fit)

Here, the model is trained on the training dataset (train_ds) for a specified number of epochs.

It uses the validation dataset (val_ds) to monitor the model's generalization performance.

Two callbacks are used:

- reduce_lr: reduces learning rate if the model stops improving, to help fine-tune the weights.
- terminate_callback: early stopping mechanism to halt training when performance stops improving significantly.

```
history1 = custom_model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=epochs,
    callbacks=[reduce_lr, terminate_callback]
)
```

Activity 1.3: Model Compilation (custom_model.compile)

This step compiles the custom model using the Adam optimizer, which is adaptive and efficient for deep learning.

The loss function used is binary_crossentropy, appropriate for binary classification problems.

Accuracy is used as a metric to evaluate how often predictions match labels.

This setup is essential before training the model.

```
custom_model.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['accuracy']
)
```

Performance Testing & Hyperparameter Tuning

Activity 1: Testing model with multiple evaluation metrics

Multiple evaluation metrics means evaluating the model's performance on a test set using different performance measures. This can provide a more comprehensive understanding of the model's strengths and weaknesses. We are using evaluation metrics for classification tasks including accuracy,

precision, recall, support and F1-score.

Activity 1.1: Model Compilation

```
custom_model.compile(optimizer='adam',
                     loss='binary_crossentropy',
                     metrics=['accuracy'])
```

This prepares the model for training by setting the optimizer (`adam`), loss function (`binary_crossentropy` for binary classification), and performance metric (`accuracy`).

Activity 1.2: Early Stopping Callback

```
class myCallback(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs={}):
        if logs.get('accuracy') == 1.0 and logs.get('val_accuracy') == 1.0 :
            print("\nReached 100% accuracy so cancelling training!")
            self.model.stop_training = True

terminate_callback = myCallback()
```

Automatically stops training once the model reaches 100% accuracy on both training and validation sets, preventing overfitting and saving time.

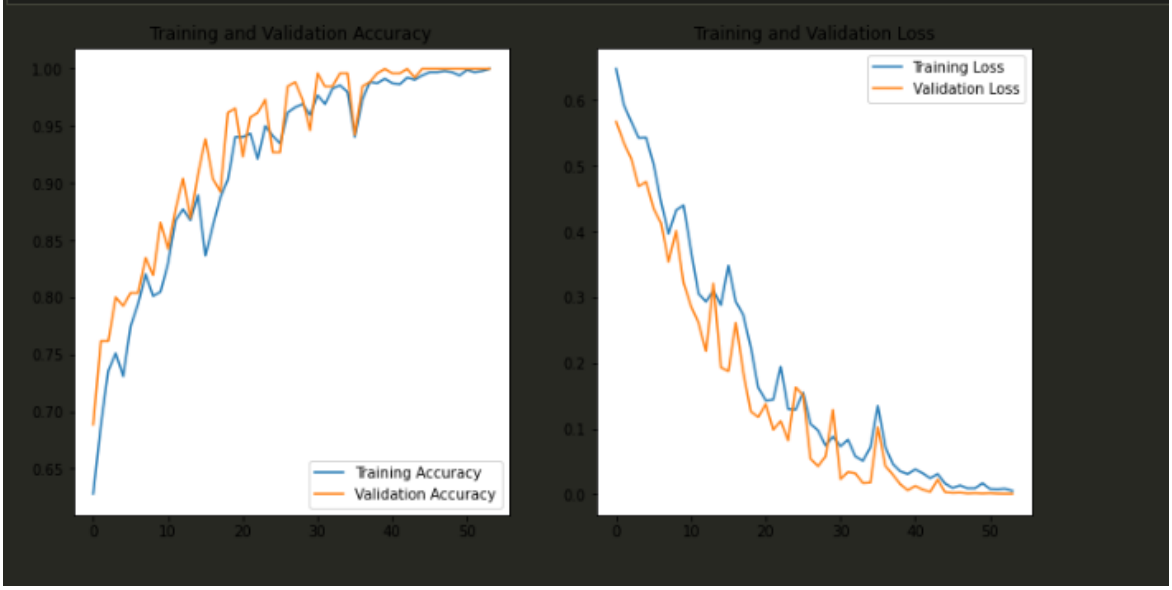
Activity 1.3: Performance Visualization (First Model)

Plots training vs validation accuracy and loss across epochs to visually assess the model's performance and detect overfitting or underfitting.

```
acc = history1.history['accuracy']
val_acc = history1.history['val_accuracy']

loss = history1.history['loss']
val_loss = history1.history['val_loss']

epochs_range = range(len(acc))

plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```

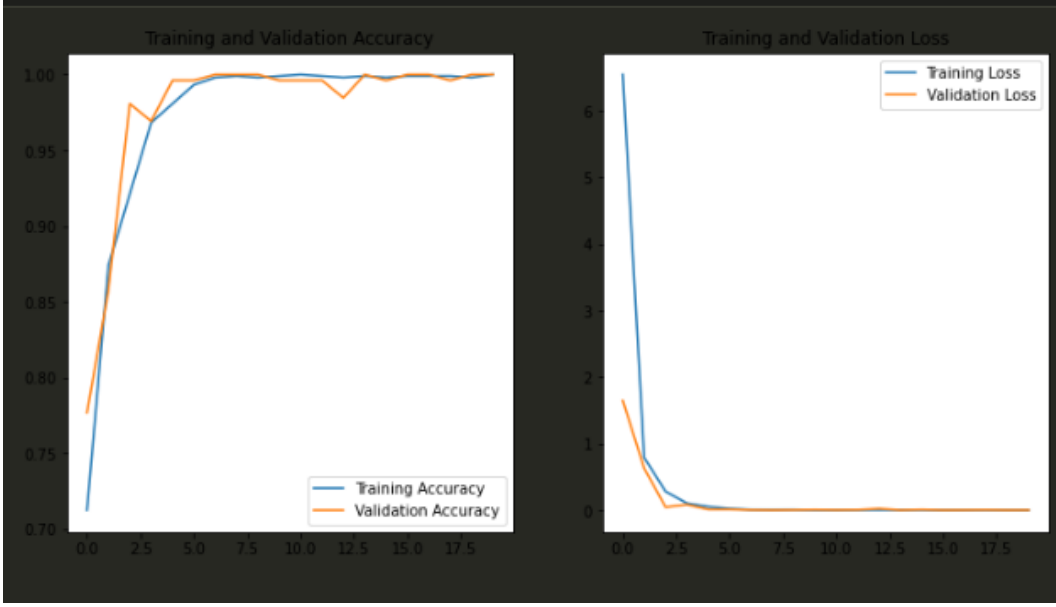
Model Deployment

```
acc = history1.history['accuracy']
val_acc = history1.history['val_accuracy']

loss = history1.history['loss']
val_loss = history1.history['val_loss']

epochs_range = range(len(acc))

plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```



.

```python
acc = history2.history['accuracy']
val_acc = history2.history['val_accuracy']

loss = history2.history['loss']
val_loss = history2.history['val_loss']

epochs_range = range(len(acc))

plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```



DEFECTIVE PRODUCT:

Product Fault Detection

Drag & Drop Image Here or Click to Upload

Choose Model: Custom CNN Predict

Prediction:

Result: DEFECTIVE product detected.

INTACT PRODUCT: