AI Workshop - Lab 2-1: Computer Vision

In this lab, we will use pre-trained models to classify machine part defects. Like yesterday, we will be working with Keras and TensorFlow, but this time we will use the `tf.keras.applications` module to load pre-trained models.

Data Overview

The dataset provided for this lab contains images of an automotive part, the **Fender Apron**, captured under varying conditions such as different angles and scales. The dataset has already been labeled as either **defective** or **healthy**, making it ideal for supervised learning tasks.

- **Total Images**: 250
  - **Healthy Parts**: 139 images
  - **Defective Parts**: 111 images
- **Train/Test Split**:
  - **Training Set**: 90% of the data
  - **Test Set**: 10% of the data (25 randomly selected images)

Key Steps in Lab

1. **Exploration**: We'll start by visualizing the dataset, inspecting some sample images to understand variations and potential challenges.
2. **Preprocessing**: Learn to preprocess images by resizing, normalizing pixel values, and augmenting the dataset to simulate real-world scenarios.
3. **Model Selection**:
   - We'll use the MobileNetV2 architecture, a lightweight and efficient model available in `tf.keras.applications`.
   - The pre-trained model will be fine-tuned to classify images into **defective** and **healthy** categories.
4. **Evaluation**:
   - Evaluate model performance using metrics such as **f1-score**, **precision**, and **recall**.
   - Analyze confusion matrices and visualize predictions for better interpretability.

Goals

By the end of this lab, you will:

- Understand the concept of transfer learning and how to adapt a pre-trained model to solve specific problems.

- Gain hands-on experience with training and deploying a defect detection system.

Now, let's dive into the **dataset preparation** and explore how these images can be preprocessed for model training!

Loading the Dataset

We will start by loading the dataset and visualizing a few sample images to understand the data distribution and characteristics. The dataset is already organized for you into training and testing sets, with separate folders for **defective** and **healthy** parts. Keras makes it easy for us to load images like this using the `image_dataset_from_directory` function.

The following command will download the dataset to your workspace using `wget`, a command-line utility for downloading files from the web. Run the cell below to download the dataset.

In [1]:
```
!wget https://github.com/alexwolson/mdlw_materials/raw/refs/heads/main/data/
```

```
--2024-12-02 12:08:16--  https://github.com/alexwolson/mdlw_materials/raw/re
fs/heads/main/data/parts_dataset.tar.gz
Resolving github.com (github.com)... 140.82.114.3
Connecting to github.com (github.com)|140.82.114.3|:443...
connected.
HTTP request sent, awaiting response...
302 Found
Location: https://raw.githubusercontent.com/alexwolson/mdlw_materials/refs/h
eads/main/data/parts_dataset.tar.gz [following]
--2024-12-02 12:08:16--  https://raw.githubusercontent.com/alexwolson/mdlw_m
aterials/refs/heads/main/data/parts_dataset.tar.gz
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.1
09.133, 185.199.111.133, 185.199.108.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.
109.133|:443... connected.
HTTP request sent, awaiting response...
200 OK
Length: 15614901 (15M) [application/octet-stream]
Saving to: ‘parts_dataset.tar.gz.12’


parts_dataset.tar.g   0%[                    ]       0  --.-KB/s
parts_dataset.tar.g 100%[===================>]  14.89M  --.-KB/s    in 0.1s

2024-12-02 12:08:16 (111 MB/s) - ‘parts_dataset.tar.gz.12’ saved [15614901/1
5614901]
```

Next, we will extract the dataset using the `tar` command. Run the cell below to extract the dataset.

In [2]:
```
!tar -xf parts_dataset.tar.gz
```

In this section, we will load our image dataset into TensorFlow for training and evaluation. TensorFlow provides an easy-to-use utility, `image_dataset_from_directory`, to load and preprocess image data directly from a directory structure.

Steps:

1. **Organizing the Dataset**:
   - The dataset is stored in a directory called `parts_dataset`, with subfolders for `train` and `test`.
   - Each subfolder contains images organized by class (e.g., "defective" and "healthy").
2. **Loading the Data**:
   - We use `image_dataset_from_directory` to load images from the `train` and `test` directories.
   - Images are resized to **224x224 pixels** (a common size for models like MobileNetV2).
   - The dataset is divided into **batches of 32 images** for efficient training.

In [3]:
```
from pathlib import Path

import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from keras.applications import MobileNetV2
from keras.applications.mobilenet_v2 import preprocess_input, decode_predict
from keras.layers import Rescaling, RandomCrop, RandomFlip, RandomRotation,
    GlobalAveragePooling2D, Dense, Input
from keras.models import Model, Sequential
from keras.preprocessing import image, image_dataset_from_directory
```

```
2024-12-02 12:08:17.382293: I tensorflow/core/util/port.cc:153] oneDNN custo
m operations are on. You may see slightly different numerical results due to
floating-point round-off errors from different computation orders. To turn t
hem off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2024-12-02 12:08:17.392012: E external/local_xla/xla/stream_executor/cuda/cu
da_fft.cc:477] Unable to register cuFFT factory: Attempting to register fact
ory for plugin cuFFT when one has already been registered
WARNING: All log messages before absl::InitializeLog() is called are written
to STDERR
E0000 00:00:1733159297.403145   15949 cuda_dnn.cc:8310] Unable to register c
uDNN factory: Attempting to register factory for plugin cuDNN when one has a
lready been registered
E0000 00:00:1733159297.406479   15949 cuda_blas.cc:1418] Unable to register
cuBLAS factory: Attempting to register factory for plugin cuBLAS when one ha
s already been registered
2024-12-02 12:08:17.417986: I tensorflow/core/platform/cpu_feature_guard.cc:
210] This TensorFlow binary is optimized to use available CPU instructions i
n performance-critical operations.
To enable the following instructions: AVX2 AVX512F AVX512_VNNI FMA, in other
operations, rebuild TensorFlow with the appropriate compiler flags.
```

In [4]:
```python
# Define the dataset directory
dataset_dir = Path('parts_dataset')

# Load the dataset
train_dataset = image_dataset_from_directory(
    dataset_dir / 'train',
    image_size=(224, 224),
    batch_size=32,
)

test_dataset = image_dataset_from_directory(
    dataset_dir / 'test',
    image_size=(224, 224),
    batch_size=32,
)
```

```
Found 199 files belonging to 2 classes.
Found 51 files belonging to 2 classes.
```

```
W0000 00:00:1733159298.659927   15949 gpu_device.cc:2344] Cannot dlopen some
GPU libraries. Please make sure the missing libraries mentioned above are in
stalled properly if you would like to use GPU. Follow the guide at https://w
ww.tensorflow.org/install/gpu for how to download and setup the required lib
raries for your platform.
Skipping registering GPU devices...
```

To better understand our dataset, it's helpful to visualize some sample images along with their corresponding labels. This can give us insight into the data quality, variations, and any potential preprocessing needs.

Steps:

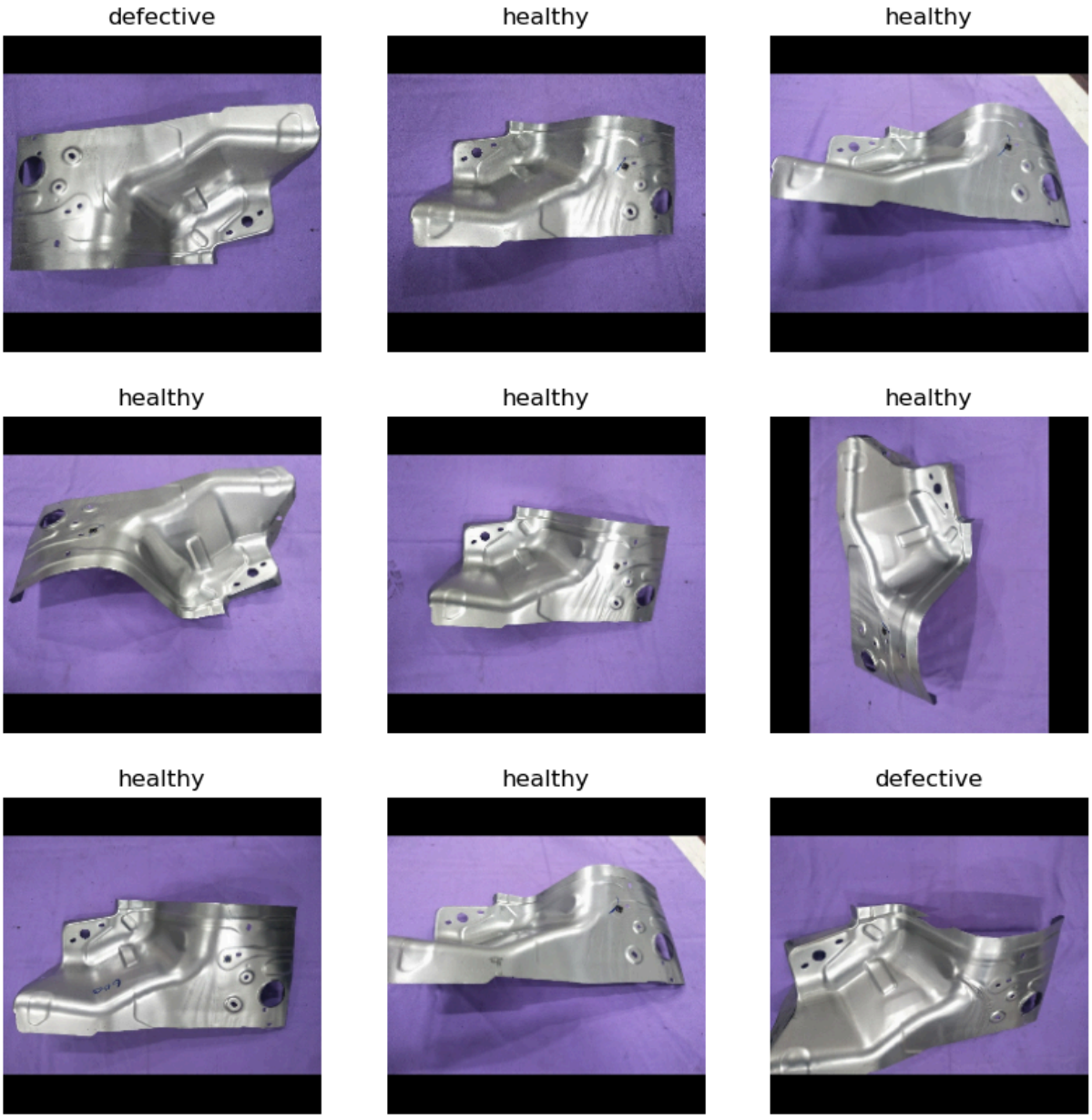
1. **Take a Batch**:

- Use the `.take(1)` method to extract the first batch of images and labels from the `train_dataset`.

2. **Display Images**:
   - Use `matplotlib.pyplot` to create a grid of 3x3 images (9 total).
   - Convert the image tensors to NumPy arrays and ensure the pixel values are properly scaled for display.
3. **Label the Images**:
   - Assign a title of "defective" or "healthy" based on the label value (e.g., `0` for defective and `1` for healthy).

In [5]:
```
# Visualize the first 9 images from the training set
plt.figure(figsize=(10, 10))
for images, labels in train_dataset.take(1):                    # Take
    for i in range(9):                                          # Loop
        ax = plt.subplot(3, 3, i + 1)                           # Crea
        plt.imshow(images[i].numpy().astype("uint8"))           # Disp
        plt.title("defective" if labels[i] == 0 else "healthy") # Add
        plt.axis("off")                                         # Remo
```

```
2024-12-02 12:08:18.909067: I tensorflow/core/framework/local_rendezvous.cc:
405] Local rendezvous is aborting with status: OUT_OF_RANGE: End of sequence
```



Preprocessing the Images

When working with image data, preprocessing is a crucial step to prepare the dataset for training a machine learning model. Preprocessing ensures that the data is in the right format, size, and range for optimal performance. Let's review what has already been done for you and what still needs to be completed.

Preprocessing Already Done:

1. **Resizing**:
   - The original images (4160 × 3120 pixels) were too large for most models.
   - They have been resized to 224 × 224 pixels for training, with padding (black bars) added where necessary to maintain a 1:1 aspect ratio.
2. **Train-Test Split**:

- The dataset has been split into training (90%) and testing (10%) sets, so you can focus on model training and evaluation separately.

Preprocessing to Complete:

Now it's your turn to finish preprocessing. Here's what we'll do next:

1. **Normalization**:
   - Rescale pixel values from their original range ([0, 255]) to the range [0, 1]. This helps models train more effectively by ensuring consistent input scales.
2. **Data Augmentation**:
   - Apply random transformations (e.g., flips, rotations, zooms) to the training images to simulate variations seen in real-world scenarios. This enhances the model's ability to generalize.
3. **Dataset Configuration**:
   - Optimize dataset performance by applying prefetching, caching, and shuffling techniques.

Viewing the Data:

For models, images are represented as matrices of pixel values. Run the code below to inspect the shape of the first image in the training set and view its pixel values (specifically a slice through the middle of the image).

In [6]:
```
for images, labels in train_dataset.take(1):
    print("Image Shape:", images[0].shape)            # Shape of the first i
    print("Pixel Values (Row 100):", images[0][100])  # Pixel values in the
    break
```

```
Image Shape: (224, 224, 3)
Pixel Values (Row 100): tf.Tensor(
[[0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 7.14282990e-01 0.00000000e+00]
 [0.00000000e+00 1.42860413e-01 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 1.42856598e-01 1.71428680e+00]
 [0.00000000e+00 1.00000000e+00 0.00000000e+00]
 [1.85713959e+00 0.00000000e+00 7.14283752e+00]
 [1.17653023e+02 1.01795891e+02 1.56938705e+02]
 [1.24061188e+02 1.02061188e+02 1.66061203e+02]
 [1.20367340e+02 9.53673401e+01 1.60367355e+02]
 [1.17714264e+02 9.57142639e+01 1.59714264e+02]
 [1.25755066e+02 1.03755066e+02 1.67755066e+02]
 [1.20795952e+02 9.87959518e+01 1.62795944e+02]
 [1.15081642e+02 9.30816422e+01 1.57081635e+02]
 [1.20244881e+02 9.82448807e+01 1.61102036e+02]
 [1.20346924e+02 9.83469238e+01 1.60346924e+02]
 [1.19938820e+02 9.79388199e+01 1.59938828e+02]
 [1.18142792e+02 9.61427917e+01 1.58142792e+02]
 [1.17204140e+02 9.52041397e+01 1.57204132e+02]
 [1.17326569e+02 9.53265686e+01 1.57326569e+02]
 [1.16673523e+02 9.46735229e+01 1.56673523e+02]
 [1.16000000e+02 9.40000000e+01 1.56000000e+02]
 [1.12469376e+02 9.04693756e+01 1.52469376e+02]
 [1.12918304e+02 9.09183044e+01 1.52918304e+02]
 [1.11857132e+02 8.98571320e+01 1.51857132e+02]
 [1.10000000e+02 8.80000000e+01 1.50000000e+02]
 [1.11857140e+02 8.98571396e+01 1.51857147e+02]
 [1.12653038e+02 9.06530380e+01 1.52653030e+02]
 [1.14285706e+02 9.22857056e+01 1.54285706e+02]
 [1.11102020e+02 8.91020203e+01 1.51102020e+02]
 [1.11142876e+02 8.91428757e+01 1.51142883e+02]
 [1.12428604e+02 9.04286041e+01 1.52428604e+02]
 [1.13102066e+02 9.11020660e+01 1.53102066e+02]
```

[1.09530602e+02 8.75306015e+01 1.49530609e+02]
[1.09571396e+02 8.75713959e+01 1.49571396e+02]
[1.07591820e+02 8.56734619e+01 1.47428543e+02]
[1.08510185e+02 8.75101852e+01 1.46510178e+02]
[1.03999962e+02 8.48775101e+01 1.41244873e+02]
[9.94285660e+01 8.04285660e+01 1.36285690e+02]
[8.86939011e+01 6.98367538e+01 1.24408195e+02]
[8.55305939e+01 6.75305939e+01 1.19530594e+02]
[8.64897842e+01 6.84897842e+01 1.19632637e+02]
[8.63878021e+01 6.83878021e+01 1.18387802e+02]
[8.59387894e+01 6.79387894e+01 1.17938789e+02]
[8.68571320e+01 6.88571320e+01 1.18857132e+02]
[8.40000000e+01 6.60000000e+01 1.16000000e+02]
[8.38775864e+01 6.58775864e+01 1.15877586e+02]
[7.83469696e+01 6.23469696e+01 1.09346970e+02]
[7.80816116e+01 6.50816116e+01 1.09081612e+02]
[7.87346878e+01 6.32448997e+01 1.07224480e+02]
[7.29387436e+01 5.69387436e+01 1.03857101e+02]
[7.81020432e+01 6.55305710e+01 1.04530647e+02]
[1.70428680e+02 1.64857285e+02 1.78428680e+02]
[2.01306152e+02 2.02020447e+02 1.97020447e+02]
[1.96897934e+02 1.95897934e+02 1.91183640e+02]
[1.95979614e+02 1.94979614e+02 1.90979614e+02]
[1.80020554e+02 1.79020554e+02 1.76163406e+02]
[1.28245132e+02 1.27245132e+02 1.25245132e+02]
[1.29081497e+02 1.27224358e+02 1.27795753e+02]
[1.74571243e+02 1.72571243e+02 1.73571243e+02]
[1.89020401e+02 1.87142838e+02 1.87775513e+02]
[1.97469376e+02 1.96469376e+02 1.92673477e+02]
[1.98244873e+02 1.97244873e+02 1.93244873e+02]
[2.02346878e+02 2.01346878e+02 1.97346878e+02]
[2.04387695e+02 2.03387695e+02 1.99387695e+02]
[2.06020340e+02 2.05020340e+02 2.01020340e+02]
[2.08285660e+02 2.07285660e+02 2.03285660e+02]
[2.11285645e+02 2.10285645e+02 2.06285645e+02]
[2.16183563e+02 2.15183563e+02 2.11183563e+02]
[2.21836655e+02 2.20836655e+02 2.16836655e+02]
[2.25469345e+02 2.25040771e+02 2.22183624e+02]
[2.26408051e+02 2.26408051e+02 2.25836609e+02]
[2.12836655e+02 2.12836655e+02 2.12836655e+02]
[1.60285583e+02 1.61285583e+02 1.63571320e+02]
[1.32510178e+02 1.33510178e+02 1.37510178e+02]
[1.28653015e+02 1.29571381e+02 1.34326492e+02]
[1.51979462e+02 1.50469269e+02 1.54448868e+02]
[2.02285797e+02 2.00285797e+02 2.01285797e+02]
[2.49918320e+02 2.49918320e+02 2.48408127e+02]
[2.47775574e+02 2.47775574e+02 2.45775574e+02]
[2.17285660e+02 2.19285660e+02 2.16285660e+02]
[1.90877594e+02 1.93163300e+02 1.92449005e+02]
[1.58816437e+02 1.62816437e+02 1.64387848e+02]
[1.31653091e+02 1.35224533e+02 1.39081650e+02]
[1.19510170e+02 1.22510170e+02 1.27510170e+02]
[1.17795860e+02 1.20795860e+02 1.27224419e+02]
[1.12979591e+02 1.15979591e+02 1.22979591e+02]
[9.98571320e+01 1.02857132e+02 1.09857132e+02]
[9.47346725e+01 9.77346725e+01 1.04734673e+02]

[9.57755127e+01 9.57755127e+01 1.03775513e+02]
[1.00734634e+02 1.00734634e+02 1.08734634e+02]
[1.02326553e+02 1.03326553e+02 1.08326553e+02]
[1.16306091e+02 1.17306091e+02 1.22306091e+02]
[1.19632629e+02 1.20632629e+02 1.24632629e+02]
[1.17571419e+02 1.18571419e+02 1.20571419e+02]
[1.17530609e+02 1.18530609e+02 1.20530609e+02]
[1.26959091e+02 1.27959091e+02 1.29959091e+02]
[1.32346878e+02 1.33346878e+02 1.35346878e+02]
[1.39081604e+02 1.40653015e+02 1.40938782e+02]
[1.44428452e+02 1.46428452e+02 1.45428452e+02]
[1.59816422e+02 1.61816422e+02 1.59102127e+02]
[2.01285660e+02 2.03285660e+02 2.00285660e+02]
[2.26224289e+02 2.28224289e+02 2.25224289e+02]
[2.35510040e+02 2.37510040e+02 2.32510040e+02]
[2.34285538e+02 2.36285538e+02 2.31285538e+02]
[2.16877533e+02 2.18877533e+02 2.13877533e+02]
[1.72551361e+02 1.74551361e+02 1.71551361e+02]
[1.57877594e+02 1.59877594e+02 1.56877594e+02]
[1.63000000e+02 1.65000000e+02 1.62000000e+02]
[1.75408295e+02 1.77408295e+02 1.74408295e+02]
[1.95306213e+02 1.96734802e+02 1.94020508e+02]
[2.03163361e+02 2.03591919e+02 2.00306244e+02]
[1.98591980e+02 1.99591980e+02 1.94591980e+02]
[1.91020493e+02 1.92734756e+02 1.89163284e+02]
[1.85714340e+02 1.90285782e+02 1.86428635e+02]
[1.84285736e+02 1.89285736e+02 1.85285736e+02]
[1.77979568e+02 1.81979568e+02 1.80979568e+02]
[1.71510284e+02 1.75510284e+02 1.74510284e+02]
[1.60938889e+02 1.64938889e+02 1.63938889e+02]
[1.49204163e+02 1.53204163e+02 1.52204163e+02]
[1.51265182e+02 1.53265182e+02 1.50265182e+02]
[1.53591812e+02 1.55591812e+02 1.52591812e+02]
[1.53428528e+02 1.55428528e+02 1.52428528e+02]
[1.51816299e+02 1.53816299e+02 1.50816299e+02]
[1.53918396e+02 1.55918396e+02 1.52918396e+02]
[1.50428604e+02 1.52428604e+02 1.49428604e+02]
[1.58856934e+02 1.60856934e+02 1.57856934e+02]
[2.21041275e+02 2.23041275e+02 2.20041275e+02]
[2.19285645e+02 2.21285645e+02 2.16571350e+02]
[1.77428604e+02 1.79428604e+02 1.76428604e+02]
[1.60428726e+02 1.64428726e+02 1.65428726e+02]
[1.54469604e+02 1.58469604e+02 1.57469604e+02]
[1.55224747e+02 1.59224747e+02 1.59081863e+02]
[1.71428604e+02 1.68836838e+02 1.80632538e+02]
[1.22755608e+02 1.07184288e+02 1.57183990e+02]
[1.21081657e+02 9.92449188e+01 1.62530655e+02]
[1.32857132e+02 1.07857132e+02 1.75857132e+02]
[1.37469391e+02 1.12469391e+02 1.80469391e+02]
[1.35061188e+02 1.09775482e+02 1.78632599e+02]
[1.34857117e+02 1.08857117e+02 1.79857117e+02]
[1.38428452e+02 1.12428452e+02 1.83428452e+02]
[1.38265320e+02 1.12265312e+02 1.83265320e+02]
[1.35591888e+02 1.09591888e+02 1.82306122e+02]
[1.32571472e+02 1.06571472e+02 1.77571472e+02]
[1.37285660e+02 1.11285660e+02 1.82285660e+02]

```
[1.33836777e+02 1.07836777e+02 1.80836777e+02]
[1.36020355e+02 1.10020348e+02 1.83020355e+02]
[1.36408203e+02 1.10408195e+02 1.81408203e+02]
[1.33285797e+02 1.07285789e+02 1.78285797e+02]
[1.35530548e+02 1.09530548e+02 1.80530548e+02]
[1.35857132e+02 1.09857132e+02 1.80857132e+02]
[1.32877579e+02 1.09877571e+02 1.77877579e+02]
[1.34081635e+02 1.10510223e+02 1.78510223e+02]
[1.35836777e+02 1.10836777e+02 1.78836777e+02]
[1.30816422e+02 1.06959236e+02 1.74959244e+02]
[1.32897797e+02 1.09897804e+02 1.77897797e+02]
[1.35550964e+02 1.12550964e+02 1.80550964e+02]
[1.34000000e+02 1.12000000e+02 1.77000000e+02]
[1.29836777e+02 1.07836784e+02 1.72836777e+02]
[1.28571472e+02 1.05571472e+02 1.73571472e+02]
[1.32571442e+02 1.09571442e+02 1.77571442e+02]
[1.29857147e+02 1.06857147e+02 1.74857147e+02]
[1.33183609e+02 1.10183617e+02 1.78183609e+02]
[1.33183716e+02 1.10183708e+02 1.78183716e+02]
[1.33142868e+02 1.10142868e+02 1.80142868e+02]
[1.31551056e+02 1.08551056e+02 1.78551056e+02]
[1.33081635e+02 1.10081627e+02 1.80081635e+02]
[1.32857162e+02 1.09857162e+02 1.79857162e+02]
[1.35469360e+02 1.12469353e+02 1.82469360e+02]
[1.34877457e+02 1.11877457e+02 1.80448929e+02]
[1.33898071e+02 1.10898071e+02 1.78898071e+02]
[1.37714264e+02 1.14714264e+02 1.82714264e+02]
[1.35938766e+02 1.13938766e+02 1.78938766e+02]
[1.37816391e+02 1.16387802e+02 1.80102097e+02]
[1.28285782e+02 1.10714249e+02 1.67571640e+02]
[1.28576660e+00 0.00000000e+00 5.14306641e+00]
[0.00000000e+00 7.14263916e-01 0.00000000e+00]
[0.00000000e+00 1.00000000e+00 0.00000000e+00]
[0.00000000e+00 0.00000000e+00 0.00000000e+00]
[0.00000000e+00 1.00000000e+00 0.00000000e+00]
[0.00000000e+00 7.14294434e-01 0.00000000e+00]
[0.00000000e+00 0.00000000e+00 0.00000000e+00]
[0.00000000e+00 0.00000000e+00 0.00000000e+00]
[0.00000000e+00 0.00000000e+00 0.00000000e+00]
[0.00000000e+00 0.00000000e+00 0.00000000e+00]
[0.00000000e+00 0.00000000e+00 0.00000000e+00]
[0.00000000e+00 0.00000000e+00 0.00000000e+00]
[0.00000000e+00 0.00000000e+00 0.00000000e+00]
[0.00000000e+00 0.00000000e+00 0.00000000e+00]
[0.00000000e+00 0.00000000e+00 0.00000000e+00]
[0.00000000e+00 0.00000000e+00 0.00000000e+00]
[0.00000000e+00 0.00000000e+00 0.00000000e+00]
[0.00000000e+00 0.00000000e+00 0.00000000e+00]
[0.00000000e+00 0.00000000e+00 0.00000000e+00]
[0.00000000e+00 0.00000000e+00 0.00000000e+00]
[0.00000000e+00 0.00000000e+00 0.00000000e+00]
[0.00000000e+00 0.00000000e+00 0.00000000e+00]
[0.00000000e+00 0.00000000e+00 0.00000000e+00]
[0.00000000e+00 0.00000000e+00 0.00000000e+00]
[0.00000000e+00 0.00000000e+00 0.00000000e+00]
[0.00000000e+00 0.00000000e+00 0.00000000e+00]
```

```
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00]], shape=(224, 3), dtype=floa
t32)
```

As you can see from the output above, the pixel values of the images are currently in the range [0, 255]. Each pixel has three values (red, green, and blue), corresponding to the **RGB format** of the image. This format allows the model to process color images, with each color channel represented as a separate value.

Normalizing Pixel Values

To prepare the images for training, we need to normalize the pixel values to the range [0, 1]. Normalization ensures that:

1. The model trains more effectively by reducing the variability in input values.
2. The input values are on a consistent scale, which helps gradient-based optimization algorithms converge faster.

We will use Keras's `Rescaling` layer to handle the normalization process. The `Rescaling` layer applies a simple transformation:

$$\text{Normalized Value} = \text{Original Value} \times \frac{1}{255}$$

Steps:

1. Create a normalization layer using `Rescaling(1./255)`, which divides all pixel values by 255.
2. Apply the normalization layer to the training dataset using the `map()` function, which processes each image in the dataset.

Run the following code to normalize the dataset:

In [7]:
```
# Create a normalization layer
normalization_layer = Rescaling(1./255)

# Apply normalization to the training dataset
normalized_train_dataset = train_dataset.map(lambda x, y: (normalization_lay
```

In [8]:
```
for images, labels in normalized_train_dataset.take(1):
    print("Image Shape:", images[0].shape)                        # Shape of
    print("Normalized Pixel Values (Row 100):", images[0][100])  # Pixel val
    break
```

```
Image Shape: (224, 224, 3)
Normalized Pixel Values (Row 100): tf.Tensor(
[[0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.         0.        ]
 [0.         0.00392157 0.        ]
 [0.         0.00392157 0.        ]
 [0.0072829  0.         0.02016799]
 [0.5450982  0.47451    0.7103642 ]
 [0.61840725 0.5237294  0.78143245]
 [0.5784713  0.49219683 0.74317724]
 [0.50644326 0.42016873 0.6750707 ]
 [0.54757875 0.45962358 0.7150858 ]
 [0.55054027 0.45250106 0.7135655 ]
 [0.54733914 0.45434183 0.7159666 ]
 [0.542617   0.45634255 0.7112445 ]
 [0.53869545 0.45242104 0.70732296]
 [0.54701877 0.4573829  0.72236884]
 [0.543417   0.45322087 0.71988755]
 [0.54997987 0.45978376 0.72645044]
 [0.5645459  0.47434983 0.7410165 ]
 [0.56318533 0.47298926 0.739656  ]
 [0.5510206  0.46082458 0.72749126]
 [0.56750685 0.47731078 0.7439774 ]
 [0.5519008  0.465066   0.7216487 ]
 [0.55854326 0.47226876 0.7232492 ]
 [0.5171669  0.43481392 0.69867945]
 [0.51468575 0.430252   0.6936373 ]
 [0.52172875 0.43545422 0.6903562 ]
 [0.5161266  0.43433386 0.6785916 ]
 [0.5010005  0.4483393  0.61704695]
 [0.46434572 0.44713888 0.5186075 ]
 [0.56246483 0.55854326 0.5781511 ]
 [0.47899237 0.48291394 0.49075708]
```

```
[0.47162858 0.47555014 0.49123642]
[0.46634668 0.48203295 0.49379766]
[0.47122857 0.48691484 0.49867955]
[0.46234515 0.47803143 0.48979616]
[0.46442583 0.4801121  0.4918768 ]
[0.47619066 0.49187693 0.5036416 ]
[0.48275316 0.49787924 0.51076436]
[0.48331332 0.49731886 0.51580626]
[0.48227304 0.5018809  0.51756716]
[0.47370964 0.49331748 0.50900376]
[0.5234893  0.540296   0.55302113]
[0.68987507 0.70556134 0.7104434 ]
[0.918208   0.92605114 0.9221296 ]
[0.98239315 0.98239315 0.97455   ]
[0.98687464 0.9935973  0.9829531 ]
[0.992637   0.9990396  0.99655855]
[0.92308897 0.9309321  0.92701054]
[0.7541417  0.76982796 0.77374953]
[0.63321316 0.63713473 0.64497787]
[0.5568625  0.56078404 0.5686272 ]
[0.58487403 0.5888756  0.59647864]
[0.6693876  0.6772307  0.67330915]
[0.8466595  0.8511412  0.845539  ]
[0.9663066  0.9685475  0.9539817 ]
[0.98175293 0.9856745  0.96606666]
[0.98951566 0.99343723 0.9738294 ]
[0.9831933  0.9831933  0.97535014]
[0.9791918  0.9791918  0.97134864]
[0.90708274 0.9149259  0.90316117]
[0.80896354 0.8168067  0.80504197]
[0.78455395 0.7923971  0.7806324 ]
[0.7767908  0.78463393 0.7728692 ]
[0.7684674  0.7684674  0.7606243 ]
[0.7512607  0.7512607  0.74341756]
[0.7315728  0.7315728  0.72372967]
[0.7293317  0.7293317  0.72148854]
[0.72517025 0.72517025 0.7173271 ]
[0.6479393  0.6479393  0.6400962 ]
[0.5941577  0.5941577  0.58631456]
[0.60320127 0.60320127 0.59535813]
[0.62465    0.62465    0.61680686]
[0.63889545 0.63889545 0.6310523 ]
[0.64257675 0.64257675 0.6347336 ]
[0.6620249  0.65810335 0.6424171 ]
[0.6643456  0.66042405 0.6447378 ]
[0.66330546 0.6593839  0.6436976 ]
[0.62849176 0.6245702  0.6088839 ]
[0.58431375 0.5803922  0.5647059 ]
[0.6336132  0.62969166 0.6140054 ]
[0.66642636 0.6625048  0.6468185 ]
[0.69139636 0.690836   0.6717885 ]
[0.7026009  0.70652246 0.68523395]
[0.6740299  0.67795146 0.6583436 ]
[0.65842366 0.66234523 0.6427374 ]
[0.65770304 0.6616246  0.64201677]
[0.6384955  0.64633864 0.63457394]
```

```
[0.6235299  0.63137305 0.61960834]
[0.61720604 0.6250492  0.61328447]
[0.614326   0.61824757 0.5986397 ]
[0.5855141  0.5855141  0.577671  ]
[0.5884754  0.5884754  0.58063227]
[0.5719887  0.5719887  0.56414557]
[0.5347739  0.5347739  0.52693075]
[0.49827945 0.49827945 0.49267715]
[0.47306895 0.47306895 0.47306895]
[0.4651466  0.46962824 0.4673874 ]
[0.46386588 0.471709   0.46778744]
[0.46306574 0.47090888 0.4669873 ]
[0.53221315 0.5400563  0.5361347 ]
[0.5382954  0.5461385  0.54221696]
[0.53565437 0.5434975  0.53957593]
[0.536935   0.5447781  0.54085654]
[0.53893626 0.5467794  0.5428578 ]
[0.56270516 0.5705483  0.5666267 ]
[0.56358576 0.5714289  0.5675073 ]
[0.5551823  0.5630254  0.5512607 ]
[0.5535816  0.56142473 0.54966   ]
[0.59119695 0.5990401  0.5872754 ]
[0.60536295 0.6132061  0.6014414 ]
[0.50156295 0.50940615 0.4976414 ]
[0.39423785 0.40768307 0.40152043]
[0.38199311 0.3976794  0.39375782]
[0.3731096  0.38879588 0.39271745]
[0.39431807 0.41000435 0.41392592]
[0.3999203  0.4156066  0.4273713 ]
[0.37847182 0.3941581  0.4059228 ]
[0.36350563 0.3791919  0.39031634]
[0.41976625 0.43545252 0.43377203]
[0.67907196 0.6902765  0.6639459 ]
[0.72997296 0.74173766 0.70644355]
[0.734296   0.7455805  0.7190899 ]
[0.53405535 0.5442994  0.528693  ]
[0.47547078 0.49115705 0.4905967 ]
[0.5036421  0.51932836 0.5232499 ]
[0.53541404 0.55670244 0.5519007 ]
[0.5202887  0.5438181  0.535975  ]
[0.52268946 0.5282919  0.5344542 ]
[0.4871561  0.45226237 0.5690286 ]
[0.34029636 0.27755126 0.4641058 ]
[0.36502594 0.30228084 0.4899558 ]
[0.35998413 0.29139677 0.512605  ]
[0.41272396 0.33525306 0.56374425]
[0.44337732 0.36494595 0.6040016 ]
[0.4364147  0.35014015 0.6011206 ]
[0.48235315 0.39607865 0.64761925]
[0.49996018 0.41032442 0.6663468 ]
[0.5139656  0.4159264  0.6747499 ]
[0.49811918 0.40456164 0.6678668 ]
[0.4953177  0.40792266 0.6661862 ]
[0.48507455 0.39880002 0.653702  ]
[0.48963585 0.39159662 0.6582633 ]
[0.52172863 0.4236894  0.6903561 ]
```

```
[0.5094039  0.41136464 0.6780313 ]
[0.49979967 0.40960357 0.6762703 ]
[0.49691853 0.40672246 0.67338914]
[0.49219647 0.40592197 0.66082394]
[0.5058025  0.41952798 0.67442995]
[0.49972004 0.41344553 0.6683475 ]
[0.5094036  0.42312905 0.678031  ]
[0.49651864 0.4102441  0.6651461 ]
[0.5000402  0.41376564 0.6686676 ]
[0.48539472 0.39912018 0.6540222 ]
[0.5115642  0.42528972 0.6801917 ]
[0.4980393  0.4117648  0.66666675]
[0.5210084  0.43473387 0.6896359 ]
[0.4960384  0.41368544 0.6764306 ]
[0.5010006  0.41472608 0.669628  ]
[0.51356524 0.4272907  0.6821927 ]
[0.5102039  0.42392942 0.6788314 ]
[0.5247696  0.42673033 0.6855539 ]
[0.5293317  0.43129244 0.690116  ]
[0.5243697  0.42633048 0.685154  ]
[0.53309315 0.43505394 0.6949979 ]
[0.53293294 0.4348937  0.7015604 ]
[0.5378154  0.43977615 0.70644283]
[0.52885187 0.43081266 0.69747937]
[0.5341334  0.43609422 0.6971588 ]
[0.54317707 0.44513783 0.70396143]
[0.5187676  0.42072842 0.67955196]
[0.51644677 0.4184076  0.67723113]
[0.5294121  0.4431376  0.69803953]
[0.51620656 0.43777484 0.6725097 ]
[0.00504222 0.         0.02353037]
[0.         0.         0.        ]
[0.         0.00336124 0.        ]
[0.         0.00392157 0.        ]
[0.         0.00336136 0.        ]
[0.         0.00280115 0.        ]
[0.         0.         0.        ]
[0.         0.         0.        ]
[0.         0.         0.        ]
[0.         0.         0.        ]
[0.         0.         0.        ]
[0.         0.         0.        ]
[0.         0.         0.        ]
[0.         0.         0.        ]
[0.         0.         0.        ]
[0.         0.         0.        ]
[0.         0.         0.        ]
[0.         0.         0.        ]
[0.         0.         0.        ]
[0.         0.         0.        ]
[0.         0.         0.        ]
[0.         0.         0.        ]
[0.         0.         0.        ]
[0.         0.         0.        ]
```

```
[0.         0.         0.        ]
[0.         0.         0.        ]], shape=(224, 3), dtype=float32)
```

Data Augmentation: Enhancing Generalization

With the pixel values normalized, the next step in preprocessing is **data augmentation**, which helps improve the model's ability to generalize to unseen data. By applying random transformations to the training images, we can:

- Simulate variations found in real-world scenarios.
- Reduce overfitting by artificially increasing the diversity of the training dataset.

Data augmentation helps the model learn **invariance** to changes that don't matter for the task at hand. For instance:

- **Flipping** or **rotating** an image should not affect whether the model identifies a defect.
- Variations in **brightness**, **contrast**, or **zoom** should not change the model's prediction.

By applying these random transformations, we teach the model to focus on meaningful patterns in the data, rather than being misled by irrelevant differences. This improves robustness and generalization, especially when the training dataset is small.

Augmentation in Keras

Keras provides convenient **image augmentation layers** that can be added directly to the model architecture or applied as a preprocessing step. These layers make it simple to experiment with different strategies.

The augmentation layers we'll use include:

- **RandomCrop**: Crops the image to a specified size.
- **RandomFlip**: Flips the image horizontally or vertically.
- **RandomRotation**: Rotates the image by a random angle within a specified range.
- **RandomZoom**: Zooms in or out of the image.
- **RandomContrast**: Adjusts the contrast of the image randomly.

Visualizing Augmented Images

To better understand each augmentation, we will visualize the effects of individual transformations. For each augmentation layer, we'll show:

1. The **original image**.
2. The **augmented image** after applying the transformation.

In [9]:

```
# Define individual augmentation layers
augmentation_layers = {
    "RandomCrop": RandomCrop(200, 200),  # Crop image to 200x200
    "RandomFlip": RandomFlip("horizontal_and_vertical"),  # Flip horizontall
    "RandomRotation": RandomRotation(0.2),  # Rotate by up to 20%
    "RandomZoom": RandomZoom(0.2, 0.2),  # Zoom in or out
    "RandomContrast": RandomContrast(0.2),  # Adjust contrast
}

# Select one image to demonstrate augmentations
for images, labels in normalized_train_dataset.take(1):
    original_image = images[0].numpy().astype("float32")
    break

# Plot original image and augmented versions
plt.figure(figsize=(15, 15))
for i, (name, layer) in enumerate(augmentation_layers.items()):
    augmented_image = layer(tf.expand_dims(original_image, 0))  # Apply augm
    augmented_image = augmented_image[0].numpy()  # Remove batch dimension
    augmented_image = tf.clip_by_value(augmented_image, 0.0, 1.0).numpy()  #

    # Display original image
    ax = plt.subplot(len(augmentation_layers), 2, i * 2 + 1)
    plt.imshow(original_image)
    plt.title("Original")
    plt.axis("off")

    # Display augmented image
    ax = plt.subplot(len(augmentation_layers), 2, i * 2 + 2)
    plt.imshow(augmented_image)
    plt.title(name)
    plt.axis("off")

plt.tight_layout()
plt.show()
```

Original

RandomCrop

Original

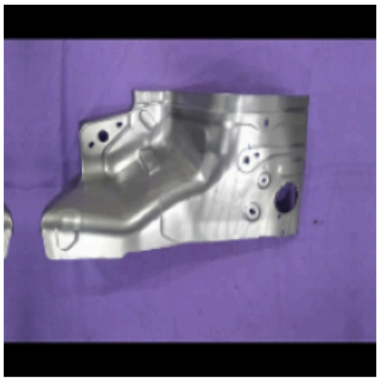
RandomFlip

Original

RandomRotation

Original

RandomZoom



Original

RandomContrast

Model Selection: MobileNetV2 Pre-trained on ImageNet

When training a model for a specific task, it's often easier to start with an **existing model** that has already been trained on a large, general-purpose dataset. These models are called **pre-trained models**, and they save a lot of time and computational resources because they have already learned to recognize patterns and features from a wide range of images.

In this lab, we're using **MobileNetV2**, a pre-trained model that was trained on a dataset called **ImageNet**. ImageNet is a collection of over 1 million images grouped into 1,000 categories, such as dogs, cats, elephants, and more. The MobileNetV2 model has learned to identify these categories very well.

Before we adapt MobileNetV2 for our defect detection task, let's see how it works on its original training data by:

1. Loading the pre-trained MobileNetV2 model.
2. Testing it on a few sample images from ImageNet-like categories.
3. Viewing its predictions to understand what it has already learned.

What is MobileNetV2 Doing?

MobileNetV2 is like a complex decision-making process:

1. **Lower Layers**: Detect simple features, like edges or textures.
2. **Middle Layers**: Combine these simple features into more complex ones, like shapes or objects.
3. **Top Layers**: Decide on the most likely category (e.g., "dog," "cat") using what it has learned during training.

Right now, the MobileNetV2 model is trained to identify objects in ImageNet categories, but we'll see how it performs by testing it on a few example images.

Step 1: Load the Pre-trained MobileNetV2 Model

The `MobileNetV2` model is available in `tf.keras.applications`, and we can load it with just one line of code:

In [10]:
```
# Load MobileNetV2 with the original classification head
imagenet_model = MobileNetV2(weights='imagenet', include_top=True)
```

Step 2: Load and Preprocess Sample Images

To use MobileNetV2, the input images must be prepared in a specific way:

1. Resize the images to 224 × 224 pixels (the size MobileNetV2 expects).
2. Normalize the pixel values to match how the model was trained.

We'll use some sample images of objects like a dog, a cat, and an elephant to test the model:

In [11]:
```
urls = {
    "cat":"https://upload.wikimedia.org/wikipedia/commons/thumb/b/b6/Felis_c
    "dog":"https://upload.wikimedia.org/wikipedia/commons/9/99/Brooks_Chase_
    "elephant":"https://upload.wikimedia.org/wikipedia/commons/thumb/3/37/Af
}

# Load and preprocess the sample images
sample_images = {}
for name, url in urls.items():
    img_path = tf.keras.utils.get_file(f"{name}.jpg", url)
    img = image.load_img(img_path, target_size=(224, 224))
    img_array = image.img_to_array(img)
    img_array = preprocess_input(img_array)
    sample_images[name] = img_array

# Display the sample images
plt.figure(figsize=(15, 5))
for i, (name, img_array) in enumerate(sample_images.items()):
    ax = plt.subplot(1, 3, i + 1)
    plt.imshow((img_array+1)/2) # Rescale pixel values to [0, 1] - in Mobile
    plt.title(name)
    plt.axis("off")
```

Step 3: Use MobileNetV2 to Make Predictions

Now let's use the model to predict what it sees in the images. The predictions will be probabilities for each of the 1,000 categories, and we'll decode these to show the most likely categories:

In [12]:
```
# Make predictions on the sample images
predictions = {}
for name, img_array in sample_images.items():
    predictions[name] = decode_predictions(imagenet_model.predict(np.expand_

# Display the predictions
for name, preds in predictions.items():
    print(f"Predictions for {name}:")
    for pred in preds[0]:
        print(f"  {pred[1]}: {pred[2]*100:.0f}%")
    print()
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 575ms/step
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 586ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step
Predictions for cat:
  lynx: 34%
  Egyptian_cat: 6%
  tabby: 4%

Predictions for dog:
  wire-haired_fox_terrier: 15%
  basenji: 13%
  toy_terrier: 8%

Predictions for elephant:
  African_elephant: 58%
  Indian_elephant: 15%
  tusker: 11%
```

Why Is This Helpful?

You might be wondering: if MobileNetV2 is trained to recognize objects like dogs, cats, and elephants, how does this help us distinguish between defective and healthy auto parts? The key lies in understanding **feature extraction** and how pre-trained models like MobileNetV2 learn to recognize patterns.

What MobileNetV2 Has Learned

1. **Low-Level Features**: The early layers of MobileNetV2 detect basic patterns like edges, corners, and textures. These features are universal—they're useful for

understanding any image, whether it's a dog or an auto part.
2. **High-Level Features**: The deeper layers combine these patterns into more complex shapes, like wheels or fur. These features are specific to the ImageNet dataset and won't directly help with our defect detection task.

By reusing the **low-level feature extraction** layers and replacing the high-level layers with new ones tailored to our dataset, we save time and computational effort. Instead of training from scratch, we only need to teach the model how to use these general features to distinguish defective parts from healthy ones.

Visualizing the Receptive Fields of Early Layers

To see why these low-level features are useful, let's visualize what the **early layers** of MobileNetV2 focus on when processing an image. These layers form the building blocks of the model's understanding of visual data.

Step 1: Load the Base Model

We'll load the MobileNetV2 model but stop at an early layer to analyze the features it extracts:

In [13]:
```
# Load MobileNetV2
base_model = MobileNetV2(weights='imagenet', include_top=False)
layer_name = 'block_1_expand'  # Choose an early layer
feature_extractor = Model(inputs=base_model.input, outputs=base_model.get_la
```

```
/tmp/ipykernel_15949/3992106967.py:2: UserWarning: `input_shape` is undefine
d or non-square, or `rows` is not in [96, 128, 160, 192, 224]. Weights for i
nput shape (224, 224) will be loaded as the default.
  base_model = MobileNetV2(weights='imagenet', include_top=False)
```

Step 2: Visualize Receptive Fields

Let's pass an image through the feature extractor and visualize the activation maps. These maps show which parts of the image the model focuses on at different positions.

In [14]:
```
img_array = sample_images["cat"]  # Choose an image to visualize

# Get the activation maps
activations = feature_extractor.predict(np.expand_dims(img_array, axis=0))

# Visualize the activation maps
plt.figure(figsize=(15, 5))
for i in range(5):
    ax = plt.subplot(1, 5, i + 1)
    plt.imshow(activations[0, :, :, i], cmap='viridis')
```

```
    plt.axis("off")
    plt.title(f"Activation Map {i}")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 49ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step
```

What Do We See?

The activation maps highlight edges, corners, and textures in the image. These low-level patterns are essential for understanding any visual data, including auto parts. Even though the model was trained on ImageNet, these universal features apply to other tasks, like identifying defects in auto parts.

Connecting This to Auto Parts

When distinguishing defective auto parts, the model doesn't need to know what the part is—it just needs to identify patterns that differentiate a healthy part from a defective one. The low-level features learned by MobileNetV2:

1. Help detect scratches, cracks, or texture anomalies.
2. Provide a foundation for recognizing visual inconsistencies.

By keeping these early layers and fine-tuning the top layers, we can adapt MobileNetV2 to focus on defect detection without starting from scratch.

This reuse of pre-trained features is what makes transfer learning so powerful, saving time and making the model both efficient and effective.

Model Selection

With this intro in mind, we are now ready to set up the model for defect detection. We will use the MobileNetV2 architecture as the base model and fine-tune it for our specific task. The model will be trained to classify images into two categories: **defective** and **healthy** parts.

Step 1: Load the Pre-trained Model

We begin by loading MobileNetV2 with pre-trained weights. Since our task is different from the original ImageNet classification, we'll:

1. Remove the top layers of MobileNetV2 (responsible for ImageNet-specific classifications).
2. Keep the base layers, which extract general visual features.

In [15]:
```
# Load MobileNetV2 without the top classification layers
base_model = MobileNetV2(
    input_shape=(224, 224, 3),
    include_top=False,  # Exclude ImageNet-specific top layers
    weights='imagenet'  # Use pre-trained weights
)

# Freeze layers
base_model.trainable = False

# We are also going to pre-process our images to be between [-1, 1] which is
def preprocess(image, label):
    image = preprocess_input(image)
    return image, label

train_dataset = train_dataset.map(preprocess)
test_dataset = test_dataset.map(preprocess)
```

Why freeze the base layers? We want to retain the general features learned from ImageNet without altering them. Instead, we'll train new layers on top to focus on our specific task.

Step 2: Add Task-Specific Layers

To adapt MobileNetV2 to defect detection, we'll add new layers:

- **Global Average Pooling**: Reduces the feature maps from the base model into a single vector, summarizing important information.
- **Fully Connected Layer**: Adds additional learnable parameters for our task.
- **Output Layer**: Produces a single probability (0 = healthy, 1 = defective) using a **sigmoid** activation.

In [16]:
```
# Build the complete model
model = Sequential([
    Input(shape=(224, 224, 3)),       # Input shape matches the pre-trained
    base_model,                       # Use the frozen MobileNetV2 base
    GlobalAveragePooling2D(),         # Reduce spatial dimensions to a singl
    Dense(128, activation='relu'),    # Learnable dense layer
    Dense(64, activation='relu'),     # Additional dense layer
    Dense(1, activation='sigmoid')    # Output layer for binary classificati
])
```

```
# Display the model structure
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Par
mobilenetv2_1.00_224 (Functional)	(None, 7, 7, 1280)	2,257
global_average_pooling2d_1 (GlobalAveragePooling2D)	(None, 1280)	
dense (Dense)	(None, 128)	163
dense_1 (Dense)	(None, 64)	8
dense_2 (Dense)	(None, 1)	

Total params: 2,430,273 (9.27 MB)
Trainable params: 172,289 (673.00 KB)
Non-trainable params: 2,257,984 (8.61 MB)

Notice that unlike yesterday, we now have a distinction between **trainable** and **non-trainable** parameters. This means that we have instructed Keras not to update the weights of the MobileNetV2 base model during training. Instead, only the new layers we added will be trained to classify defective and healthy parts.

Step 4: Compile the Model

Next, we compile the model to define:

- Loss Function: binary_crossentropy for binary classification. Binary Cross-Entropy calculates the error based on *how confident* the model is in its predictions. In principle, the model should only ever be *very confident* about a good prediction, or failing that, *not very confident* at all. It's calculated as follows:

$$\text{Binary Cross-Entropy} = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(p_i) + (1 - y_i) \log(1 - p_i)$$

- Optimizer: For now we'll stick with Stochastic Gradient Descent (SGD), but you can experiment with other optimizers like Adam or RMSprop at the end of the lab.
- Metrics: Accuracy, to monitor performance.

In [17]:
```
model.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['accuracy']
)
```

Step 5: Train the Model

Finally, we train the model on the dataset for 15 epochs and validate its performance. We will also optimize data loading using Keras' `prefetch` method, which loads data in the background while the model is training.

In [18]:
```
# Optimize data loading for performance
AUTOTUNE = tf.data.AUTOTUNE
train_dataset = train_dataset.prefetch(buffer_size=AUTOTUNE)
val_dataset = test_dataset.prefetch(buffer_size=AUTOTUNE)

# Train the model
history = model.fit(
    train_dataset,
    epochs=15,
    validation_data=val_dataset
)
```

```
Epoch 1/15
1/7 ━━━━━━━━━━━━━━━━━━━━ 19s 3s/step - accuracy: 0.4062 - loss: 0.8198
2/7 ━━━━━━━━━━━━━━━━━━━━ 2s 590ms/step - accuracy: 0.4688 - loss: 0.7800
3/7 ━━━━━━━━━━━━━━━━━━━━ 2s 588ms/step - accuracy: 0.4861 - loss: 0.7854
4/7 ━━━━━━━━━━━━━━━━━━━━ 1s 587ms/step - accuracy: 0.4993 - loss: 0.7791
5/7 ━━━━━━━━━━━━━━━━━━━━ 1s 588ms/step - accuracy: 0.5132 - loss: 0.7700
6/7 ━━━━━━━━━━━━━━━━━━━━ 0s 588ms/step - accuracy: 0.5232 - loss: 0.7631
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 508ms/step - accuracy: 0.5303 - loss: 0.7579
7/7 ━━━━━━━━━━━━━━━━━━━━ 8s 788ms/step - accuracy: 0.5356 - loss: 0.7540 - v
al_accuracy: 0.5686 - val_loss: 0.6593
Epoch 2/15
1/7 ━━━━━━━━━━━━━━━━━━━━ 4s 699ms/step - accuracy: 0.5312 - loss: 0.6734
2/7 ━━━━━━━━━━━━━━━━━━━━ 2s 586ms/step - accuracy: 0.5938 - loss: 0.6476
3/7 ━━━━━━━━━━━━━━━━━━━━ 2s 587ms/step - accuracy: 0.6181 - loss: 0.6346
4/7 ━━━━━━━━━━━━━━━━━━━━ 1s 587ms/step - accuracy: 0.6296 - loss: 0.6281
5/7 ━━━━━━━━━━━━━━━━━━━━ 1s 588ms/step - accuracy: 0.6361 - loss: 0.6239
6/7 ━━━━━━━━━━━━━━━━━━━━ 0s 588ms/step - accuracy: 0.6430 - loss: 0.6196
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 506ms/step - accuracy: 0.6480 - loss: 0.6160
7/7 ━━━━━━━━━━━━━━━━━━━━ 5s 668ms/step - accuracy: 0.6518 - loss: 0.6133 - v
al_accuracy: 0.6471 - val_loss: 0.6062
Epoch 3/15
1/7 ━━━━━━━━━━━━━━━━━━━━ 4s 697ms/step - accuracy: 0.7812 - loss: 0.5379
```

2/7 **2s** 582ms/step - accuracy: 0.8203 - loss: 0.4873
3/7 **2s** 584ms/step - accuracy: 0.8142 - loss: 0.4919
4/7 **1s** 585ms/step - accuracy: 0.8118 - loss: 0.4931
5/7 **1s** 587ms/step - accuracy: 0.8120 - loss: 0.4916
6/7 **0s** 587ms/step - accuracy: 0.8129 - loss: 0.4883
7/7 **0s** 505ms/step - accuracy: 0.8138 - loss: 0.4854
7/7 **5s** 665ms/step - accuracy: 0.8145 - loss: 0.4832 - val_accuracy: 0.6863 - val_loss: 0.5749
Epoch 4/15
1/7 **4s** 696ms/step - accuracy: 0.7500 - loss: 0.4225
2/7 **2s** 588ms/step - accuracy: 0.7734 - loss: 0.4037
3/7 **2s** 586ms/step - accuracy: 0.7899 - loss: 0.3940
4/7 **1s** 586ms/step - accuracy: 0.7995 - loss: 0.3927
5/7 **1s** 586ms/step - accuracy: 0.8058 - loss: 0.3904
6/7 **0s** 587ms/step - accuracy: 0.8113 - loss: 0.3882
7/7 **0s** 505ms/step - accuracy: 0.8146 - loss: 0.3873
7/7 **5s** 666ms/step - accuracy: 0.8170 - loss: 0.3867 - val_accuracy: 0.7059 - val_loss: 0.5392
Epoch 5/15
1/7 **4s** 692ms/step - accuracy: 0.9688 - loss: 0.2380
2/7 **2s** 584ms/step - accuracy: 0.9375 - loss: 0.2559
3/7 **2s** 585ms/step - accuracy: 0.9201 - loss: 0.2761
4/7 **1s** 586ms/step - accuracy: 0.9147 - loss: 0.2843
5/7 **1s** 587ms/step - accuracy: 0.9105 - loss: 0.2892
6/7 **0s** 587ms/step - accuracy: 0.9081 - loss: 0.2925
7/7 **0s** 506ms/step - accuracy: 0.9068 - loss: 0.2951
7/7 **5s** 666ms/step - accuracy: 0.9059 - loss: 0.2970 - val_accuracy: 0.7451 - val_loss: 0.5060
Epoch 6/15
1/7 **4s** 697ms/step - accuracy: 1.0000 - loss: 0.2405
2/7 **2s** 587ms/step - accuracy: 0.9844 - loss: 0.2484
3/7 **2s** 585ms/step - accuracy: 0.9653 - loss: 0.2593

```
4/7 ━━━━━━━━━━━━━━━━━━━━ 1s 586ms/step - accuracy: 0.9525 - loss: 0.2675
5/7 ━━━━━━━━━━━━━━━━━━━━ 1s 586ms/step - accuracy: 0.9445 - loss: 0.2693
6/7 ━━━━━━━━━━━━━━━━━━━━ 0s 586ms/step - accuracy: 0.9407 - loss: 0.2676
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 505ms/step - accuracy: 0.9384 - loss: 0.2662
7/7 ━━━━━━━━━━━━━━━━━━━━ 5s 666ms/step - accuracy: 0.9367 - loss: 0.2652 - v
al_accuracy: 0.7451 - val_loss: 0.5133
Epoch 7/15
1/7 ━━━━━━━━━━━━━━━━━━━━ 4s 697ms/step - accuracy: 0.8750 - loss: 0.2524
2/7 ━━━━━━━━━━━━━━━━━━━━ 2s 583ms/step - accuracy: 0.8984 - loss: 0.2325
3/7 ━━━━━━━━━━━━━━━━━━━━ 2s 584ms/step - accuracy: 0.8941 - loss: 0.2331
4/7 ━━━━━━━━━━━━━━━━━━━━ 1s 584ms/step - accuracy: 0.8952 - loss: 0.2316
5/7 ━━━━━━━━━━━━━━━━━━━━ 1s 584ms/step - accuracy: 0.8974 - loss: 0.2307
6/7 ━━━━━━━━━━━━━━━━━━━━ 0s 584ms/step - accuracy: 0.8997 - loss: 0.2313
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 502ms/step - accuracy: 0.9011 - loss: 0.2317
7/7 ━━━━━━━━━━━━━━━━━━━━ 5s 662ms/step - accuracy: 0.9022 - loss: 0.2320 - v
al_accuracy: 0.7843 - val_loss: 0.4680
Epoch 8/15
1/7 ━━━━━━━━━━━━━━━━━━━━ 4s 690ms/step - accuracy: 0.9375 - loss: 0.2163
2/7 ━━━━━━━━━━━━━━━━━━━━ 2s 581ms/step - accuracy: 0.9375 - loss: 0.2154
3/7 ━━━━━━━━━━━━━━━━━━━━ 2s 581ms/step - accuracy: 0.9340 - loss: 0.2130
4/7 ━━━━━━━━━━━━━━━━━━━━ 1s 580ms/step - accuracy: 0.9349 - loss: 0.2095
5/7 ━━━━━━━━━━━━━━━━━━━━ 1s 580ms/step - accuracy: 0.9367 - loss: 0.2039
6/7 ━━━━━━━━━━━━━━━━━━━━ 0s 580ms/step - accuracy: 0.9394 - loss: 0.1981
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 500ms/step - accuracy: 0.9409 - loss: 0.1940
7/7 ━━━━━━━━━━━━━━━━━━━━ 5s 659ms/step - accuracy: 0.9420 - loss: 0.1908 - v
al_accuracy: 0.7843 - val_loss: 0.4893
Epoch 9/15
1/7 ━━━━━━━━━━━━━━━━━━━━ 4s 688ms/step - accuracy: 0.9688 - loss: 0.1415
2/7 ━━━━━━━━━━━━━━━━━━━━ 2s 579ms/step - accuracy: 0.9688 - loss: 0.1491
3/7 ━━━━━━━━━━━━━━━━━━━━ 2s 577ms/step - accuracy: 0.9688 - loss: 0.1507
4/7 ━━━━━━━━━━━━━━━━━━━━ 1s 577ms/step - accuracy: 0.9668 - loss: 0.1541
5/7 ━━━━━━━━━━━━━━━━━━━━ 1s 579ms/step - accuracy: 0.9634 - loss: 0.1557
```

```
6/7 ━━━━━━━━━━━━━━━━━━ 0s 580ms/step - accuracy: 0.9626 - loss: 0.1552
7/7 ━━━━━━━━━━━━━━━━━━ 0s 499ms/step - accuracy: 0.9622 - loss: 0.1546
7/7 ━━━━━━━━━━━━━━━━━━ 5s 659ms/step - accuracy: 0.9619 - loss: 0.1540 - v
al_accuracy: 0.8039 - val_loss: 0.5834
Epoch 10/15
1/7 ━━━━━━━━━━━━━━━━━━ 4s 689ms/step - accuracy: 0.9688 - loss: 0.1283
2/7 ━━━━━━━━━━━━━━━━━━ 2s 580ms/step - accuracy: 0.9609 - loss: 0.1431
3/7 ━━━━━━━━━━━━━━━━━━ 2s 580ms/step - accuracy: 0.9601 - loss: 0.1451
4/7 ━━━━━━━━━━━━━━━━━━ 1s 581ms/step - accuracy: 0.9564 - loss: 0.1494
5/7 ━━━━━━━━━━━━━━━━━━ 1s 585ms/step - accuracy: 0.9564 - loss: 0.1485
6/7 ━━━━━━━━━━━━━━━━━━ 0s 584ms/step - accuracy: 0.9567 - loss: 0.1461
7/7 ━━━━━━━━━━━━━━━━━━ 0s 503ms/step - accuracy: 0.9571 - loss: 0.1439
7/7 ━━━━━━━━━━━━━━━━━━ 5s 662ms/step - accuracy: 0.9575 - loss: 0.1422 - v
al_accuracy: 0.7843 - val_loss: 0.4929
Epoch 11/15
1/7 ━━━━━━━━━━━━━━━━━━ 4s 690ms/step - accuracy: 0.9688 - loss: 0.0876
2/7 ━━━━━━━━━━━━━━━━━━ 2s 579ms/step - accuracy: 0.9531 - loss: 0.1072
3/7 ━━━━━━━━━━━━━━━━━━ 2s 579ms/step - accuracy: 0.9479 - loss: 0.1129
4/7 ━━━━━━━━━━━━━━━━━━ 1s 580ms/step - accuracy: 0.9453 - loss: 0.1176
5/7 ━━━━━━━━━━━━━━━━━━ 1s 580ms/step - accuracy: 0.9463 - loss: 0.1193
6/7 ━━━━━━━━━━━━━━━━━━ 0s 581ms/step - accuracy: 0.9474 - loss: 0.1214
7/7 ━━━━━━━━━━━━━━━━━━ 0s 500ms/step - accuracy: 0.9484 - loss: 0.1229
7/7 ━━━━━━━━━━━━━━━━━━ 5s 659ms/step - accuracy: 0.9492 - loss: 0.1240 - v
al_accuracy: 0.8039 - val_loss: 0.4593
Epoch 12/15
1/7 ━━━━━━━━━━━━━━━━━━ 4s 690ms/step - accuracy: 0.9688 - loss: 0.1113
2/7 ━━━━━━━━━━━━━━━━━━ 2s 577ms/step - accuracy: 0.9766 - loss: 0.0978
3/7 ━━━━━━━━━━━━━━━━━━ 2s 579ms/step - accuracy: 0.9740 - loss: 0.1072
4/7 ━━━━━━━━━━━━━━━━━━ 1s 580ms/step - accuracy: 0.9707 - loss: 0.1135
5/7 ━━━━━━━━━━━━━━━━━━ 1s 580ms/step - accuracy: 0.9703 - loss: 0.1149
6/7 ━━━━━━━━━━━━━━━━━━ 0s 581ms/step - accuracy: 0.9709 - loss: 0.1146
7/7 ━━━━━━━━━━━━━━━━━━ 0s 501ms/step - accuracy: 0.9715 - loss: 0.1141
```

```
7/7 ━━━━━━━━━━━━━━━━━━━━ 5s 661ms/step - accuracy: 0.9719 - loss: 0.1138 - v
al_accuracy: 0.8627 - val_loss: 0.5625
Epoch 13/15
1/7 ━━━━━━━━━━━━━━━━━━━━ 4s 693ms/step - accuracy: 1.0000 - loss: 0.0918
2/7 ━━━━━━━━━━━━━━━━━━━━ 2s 583ms/step - accuracy: 0.9922 - loss: 0.1055
3/7 ━━━━━━━━━━━━━━━━━━━━ 2s 582ms/step - accuracy: 0.9809 - loss: 0.1134
4/7 ━━━━━━━━━━━━━━━━━━━━ 1s 580ms/step - accuracy: 0.9779 - loss: 0.1131
5/7 ━━━━━━━━━━━━━━━━━━━━ 1s 580ms/step - accuracy: 0.9773 - loss: 0.1108
6/7 ━━━━━━━━━━━━━━━━━━━━ 0s 580ms/step - accuracy: 0.9767 - loss: 0.1114
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 500ms/step - accuracy: 0.9765 - loss: 0.1114
7/7 ━━━━━━━━━━━━━━━━━━━━ 5s 659ms/step - accuracy: 0.9763 - loss: 0.1114 - v
al_accuracy: 0.8039 - val_loss: 0.4712
Epoch 14/15
1/7 ━━━━━━━━━━━━━━━━━━━━ 4s 693ms/step - accuracy: 0.9062 - loss: 0.1157
2/7 ━━━━━━━━━━━━━━━━━━━━ 2s 580ms/step - accuracy: 0.9297 - loss: 0.0986
3/7 ━━━━━━━━━━━━━━━━━━━━ 2s 579ms/step - accuracy: 0.9427 - loss: 0.0893
4/7 ━━━━━━━━━━━━━━━━━━━━ 1s 579ms/step - accuracy: 0.9512 - loss: 0.0827
5/7 ━━━━━━━━━━━━━━━━━━━━ 1s 579ms/step - accuracy: 0.9572 - loss: 0.0793
6/7 ━━━━━━━━━━━━━━━━━━━━ 0s 579ms/step - accuracy: 0.9609 - loss: 0.0781
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 499ms/step - accuracy: 0.9636 - loss: 0.0769
7/7 ━━━━━━━━━━━━━━━━━━━━ 5s 658ms/step - accuracy: 0.9656 - loss: 0.0760 - v
al_accuracy: 0.8627 - val_loss: 0.4696
Epoch 15/15
1/7 ━━━━━━━━━━━━━━━━━━━━ 4s 689ms/step - accuracy: 1.0000 - loss: 0.0731
2/7 ━━━━━━━━━━━━━━━━━━━━ 2s 580ms/step - accuracy: 1.0000 - loss: 0.0646
3/7 ━━━━━━━━━━━━━━━━━━━━ 2s 579ms/step - accuracy: 1.0000 - loss: 0.0618
4/7 ━━━━━━━━━━━━━━━━━━━━ 1s 578ms/step - accuracy: 1.0000 - loss: 0.0594
5/7 ━━━━━━━━━━━━━━━━━━━━ 1s 579ms/step - accuracy: 0.9987 - loss: 0.0588
6/7 ━━━━━━━━━━━━━━━━━━━━ 0s 580ms/step - accuracy: 0.9981 - loss: 0.0577
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 499ms/step - accuracy: 0.9976 - loss: 0.0569
7/7 ━━━━━━━━━━━━━━━━━━━━ 5s 658ms/step - accuracy: 0.9973 - loss: 0.0563 - v
al_accuracy: 0.8039 - val_loss: 0.4382
```

In [19]:
```
import matplotlib.pyplot as plt

# Plot the validation accuracy
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Validation Loss Over Epochs')
plt.legend()
plt.show()
```

Evaluating Model Performance: Detailed Breakdown

Now that the model is trained, let's take a closer look at its performance on the **testing set**. While accuracy gives us a general idea of how well the model is doing, it's often helpful to use other metrics to better understand its behavior. We'll compute:

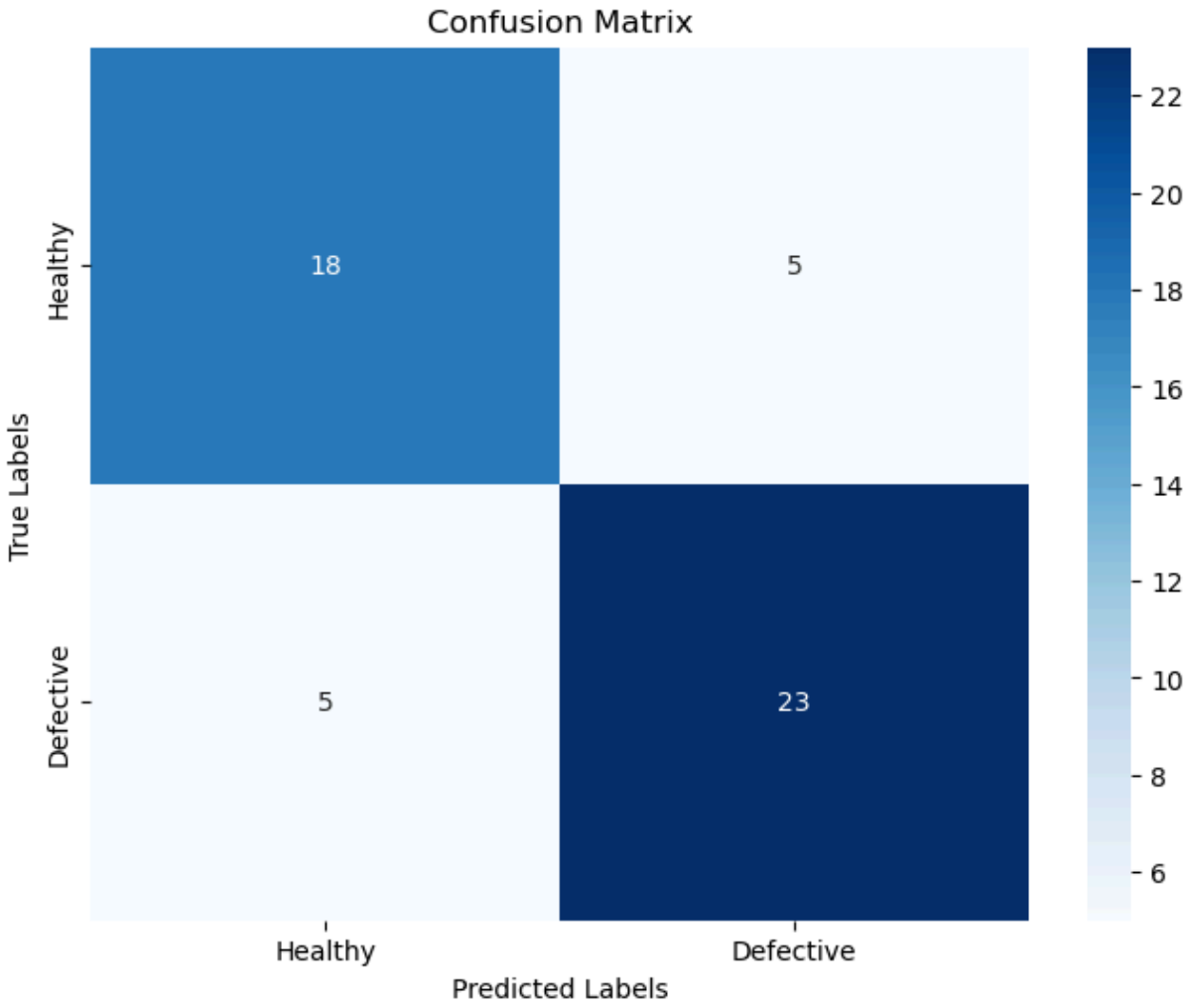
1. **Confusion Matrix**: A summary of true positives, true negatives, false positives, and false negatives.
2. **Precision and Recall**: To understand how well the model identifies defective parts.
3. **F1-Score**: A harmonic mean of precision and recall, giving a single metric that balances the two.

Let's also visualize the confusion matrix to make the results more interpretable.

In [20]:

```
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

# Step 1: Predict on the testing set
test_images, test_labels = [], []
for images, labels in test_dataset:
    test_images.append(images)
    test_labels.append(labels)
test_images = np.concatenate(test_images, axis=0)
test_labels = np.concatenate(test_labels, axis=0)

predictions = model.predict(test_images)
predicted_labels = (predictions > 0.5).astype(int).flatten()  # Convert prob

# Step 2: Compute the confusion matrix
conf_matrix = confusion_matrix(test_labels, predicted_labels)

# Step 3: Display the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['He
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show()

# Step 4: Generate the classification report
report = classification_report(test_labels, predicted_labels, target_names=[
print("Classification Report:")
print(report)
```

```
2024-12-02 12:09:36.292065: I tensorflow/core/framework/local_rendezvous.cc:
405] Local rendezvous is aborting with status: OUT_OF_RANGE: End of sequence
1/2 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 875ms/step
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
2/2 ━━━━━━━━━━━━━━━━━━━━ 2s 886ms/step
```




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

     Healthy       0.78      0.78      0.78        23
   Defective       0.82      0.82      0.82        28

    accuracy                           0.80        51
   macro avg       0.80      0.80      0.80        51
weighted avg       0.80      0.80      0.80        51
```

Analyzing Misclassified and Low-Confidence Samples

To better understand the model's performance, let's inspect:

1. **Misclassified Samples**: Images that the model got wrong. This helps us identify patterns in its mistakes.
2. **Low-Confidence Correct Predictions**: Images that were classified correctly but with a low confidence score. These can reveal edge cases where the model is uncertain.

We will visualize these samples alongside their predicted labels and confidence scores to understand where the model struggles and why.

In [21]:
```python
import numpy as np
import matplotlib.pyplot as plt

# Step 1: Predict on the testing set
predictions = model.predict(test_images).flatten()
predicted_labels = (predictions > 0.5).astype(int)
confidence_scores = np.where(predictions < 0.5, 1 - predictions, predictions

# Step 2: Identify misclassified samples
misclassified_indices = np.where(predicted_labels != test_labels)[0]

# Step 3: Identify low-confidence correct samples
low_confidence_correct_indices = np.where(
    (predicted_labels == test_labels) & (confidence_scores < 0.75)
)[0]

# Step 4: Define a function to visualize samples
def visualize_samples(indices, title, num_samples=9):
    plt.figure(figsize=(12, 12))
    for i, idx in enumerate(indices[:num_samples]):
        plt.subplot(3, 3, i + 1)
        plt.imshow((test_images[idx] + 1)/2)
        plt.axis("off")
        plt.title(
            f"True: {'Defective' if test_labels[idx] else 'Healthy'}\n"
            f"Pred: {'Defective' if predicted_labels[idx] else 'Healthy'}\n"
            f"Confidence: {confidence_scores[idx]:.2f}"
        )
    plt.suptitle(title, fontsize=16)
    plt.tight_layout()
    plt.show()

# Step 5: Visualize misclassified samples
print(f"Total misclassified samples: {len(misclassified_indices)}")
visualize_samples(misclassified_indices, "Misclassified Samples")

# Step 6: Visualize low-confidence correct samples
print(f"Total low-confidence correct samples: {len(low_confidence_correct_ir
visualize_samples(low_confidence_correct_indices, "Low-Confidence Correct Sa
```

```
1/2 ━━━━━━━━━━━━━━━━━━━━ 0s 602ms/step
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 321ms/step
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
2/2 ━━━━━━━━━━━━━━━━━━━━ 1s 331ms/step
Total misclassified samples: 10
```

Misclassified Samples

True: Defective
Pred: Healthy
Confidence: 0.54

True: Healthy
Pred: Defective
Confidence: 0.77

True: Defective
Pred: Healthy
Confidence: 0.87

True: Defective
Pred: Healthy
Confidence: 0.93

True: Defective
Pred: Healthy
Confidence: 0.69

True: Healthy
Pred: Defective
Confidence: 0.65

True: Defective
Pred: Healthy
Confidence: 0.99

True: Healthy
Pred: Defective
Confidence: 0.69

True: Healthy
Pred: Defective
Confidence: 0.61

Total low-confidence correct samples: 2

Comparing Activation Maps: Pre-trained vs. Fine-tuned Model

Now that our model has been fine-tuned, let's revisit the **activation maps** from the `block_1_expand` layer. This time, we'll compare how the pre-trained MobileNetV2 and the fine-tuned model process:

1. A **cat image** (used earlier to observe pre-trained activations).
2. An image from our **defect detection dataset**.

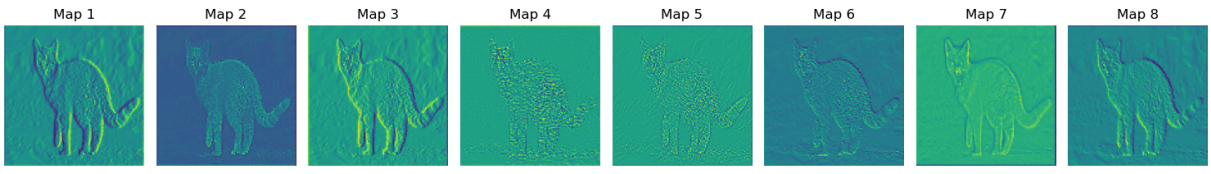
This comparison helps us understand how the model's feature extraction has adapted to the new task.

In [22]:
```
# Step 1: Define the fine-tuned feature extractor using the base model
fine_tuned_feature_extractor = Model(
    inputs=base_model.input,  # Use the base model's input
    outputs=base_model.get_layer('block_1_expand').output  # Extract the out
)

# Step 2: Preprocess the cat image
cat_array = sample_images["cat"]
cat_array = np.expand_dims(cat_array, axis=0)

# Step 3: Select an image from the defect detection dataset
for defect_images, _ in test_dataset.take(1):  # Grab a batch
    defect_image = defect_images[0].numpy()  # Take the first image
    break
defect_array = np.expand_dims(defect_image, axis=0)

# Step 4: Generate activation maps
cat_activations = fine_tuned_feature_extractor.predict(cat_array)
```

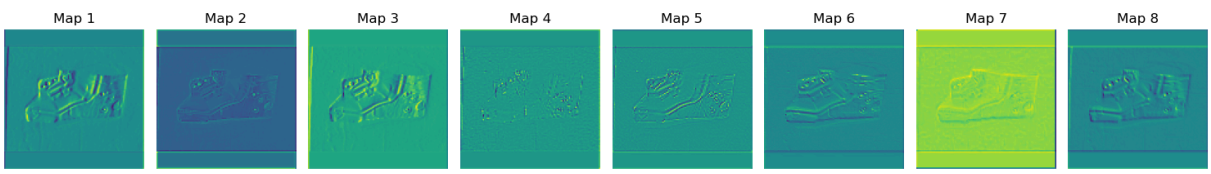
```python
defect_activations = fine_tuned_feature_extractor.predict(defect_array)

# Step 5: Visualize activation maps
def visualize_activation_maps(activations, title, num_maps=8):
    plt.figure(figsize=(15, 8))

    # Display activation maps
    for i in range(num_maps):
        plt.subplot(2, num_maps, i + 1)
        plt.imshow(activations[0, :, :, i], cmap='viridis')
        plt.axis('off')
        plt.title(f"Map {i + 1}")

    plt.suptitle(title, fontsize=16)
    plt.tight_layout()
    plt.show()

# Visualize activations for the cat image
visualize_activation_maps(cat_activations, "Fine-Tuned Activations: Cat Imag

# Visualize activations for the defect detection image
visualize_activation_maps(defect_activations, "Fine-Tuned Activations: Defec
```

```
WARNING:tensorflow:5 out of the last 9 calls to <function TensorFlowTrainer.
make_predict_function.<locals>.one_step_on_data_distributed at 0x7e84d79bbd8
0> triggered tf.function retracing. Tracing is expensive and the excessive n
umber of tracings could be due to (1) creating @tf.function repeatedly in a
loop, (2) passing tensors with different shapes, (3) passing Python objects
instead of tensors. For (1), please define your @tf.function outside of the
loop. For (2), @tf.function has reduce_retracing=True option that can avoid
unnecessary retracing. For (3), please refer to https://www.tensorflow.org/g
uide/function#controlling_retracing and https://www.tensorflow.org/api_docs/
python/tf/function for  more details.
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 49ms/step
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step
```

Fine-Tuned Activations: Cat Image



Fine-Tuned Activations: Defect Detection Image



Real-World Application: Defect Detection Alert System

In a real-world scenario, a trained model can be deployed as part of an automated quality control pipeline. For example, in a manufacturing environment, the model can evaluate images of parts and provide alerts if defects are detected with high confidence.

Here, we'll implement a simple **detector function** that:

1. Accepts an input image and a confidence threshold.
2. Predicts whether the image is defective.
3. Returns an "alert" if the defect probability exceeds the threshold.

In [23]:
```
def detect_defect_from_dataset(image_index, dataset, model, threshold=0.25):
    """
    Detects defects in an image from a dataset using the trained model.

    Parameters:
    - image_index (int): Index of the image in the dataset to evaluate.
    - dataset (tf.data.Dataset): The dataset containing the images to evalua
    - model (keras.Model): The trained defect detection model.
    - threshold (float): Confidence threshold for triggering an alert.

    Returns:
    - str: Result message with defect probability and alert status.
    """
    # Extract the specified image and its label
    dataset_images, dataset_labels = [], []
    for images, labels in dataset:
        dataset_images.append(images)
        dataset_labels.append(labels)
    dataset_images = np.concatenate(dataset_images, axis=0)
    dataset_labels = np.concatenate(dataset_labels, axis=0)

    img = dataset_images[image_index]
    true_label = dataset_labels[image_index]

    # Preprocess the image (add batch dimension)
    img_array = np.expand_dims(img, axis=0)

    # Step 1: Make a prediction
    prediction = model.predict(img_array)[0][0]  # Get defect probability (s

    # Step 2: Determine result based on the threshold
    if prediction < threshold:
        result = f"ALERT: Defect detected with probability {prediction:.2f}
    else:
        result = f"OK: No defect detected (Probability: {prediction:.2f}, Th

    return result, img, true_label, prediction

# Example usage
```

```
image_index = 2  # Index of the image to evaluate
threshold = 0.25  # Define a confidence threshold
result, img, true_label, prediction = detect_defect_from_dataset(image_index

# Display the result
print(result)

# Visualize the image
plt.imshow((img + 1)/2)
plt.axis("off")
plt.title(f"True Label: {'Defective' if true_label else 'Healthy'}\nPredicti
plt.show()
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step
OK: No defect detected (Probability: 0.69, Threshold: 0.25)
```

Conclusion

In this lab, we took a deep dive into applying **transfer learning** and **computer vision** techniques to detect defects in automotive parts. By building on the foundational skills introduced earlier in the workshop, we explored advanced workflows for image classification and model adaptation.

Here's a summary of what we covered:

1. **Dataset Exploration and Preprocessing**:

- Loaded and visualized the **Fender Apron** dataset to understand the characteristics of defective and healthy parts.
- Performed preprocessing steps including resizing, normalization, and data augmentation to prepare the dataset for training.

2. **Understanding Pre-trained Models**:
   - Explored how MobileNetV2, pre-trained on ImageNet, extracts features and how these features can be adapted to a new task.
   - Visualized activation maps from early layers to see what the model focuses on before and after fine-tuning.
3. **Model Training and Fine-tuning**:
   - Used the MobileNetV2 architecture as the base model and fine-tuned it for binary classification (defective vs. healthy).
   - Optimized the training process with data augmentation and prefetching for performance improvements.
4. **Evaluation and Insights**:
   - Analyzed the model's performance using metrics like precision, recall, F1-score, and confusion matrices.
   - Inspected misclassified and low-confidence samples to identify patterns and potential edge cases.
5. **Real-world Applications**:
   - Implemented a simple defect detection alert system that could be used in automated quality control pipelines.
   - Demonstrated how a trained model can make predictions on test images and trigger alerts based on confidence thresholds.

Key Takeaways

- **Transfer learning** leverages pre-trained models to save time and computational resources, making it an effective approach for specialized tasks like defect detection.
- Visualizing activation maps and misclassified samples provides deeper insights into model behavior and areas for improvement.
- The combination of data augmentation, efficient preprocessing, and fine-tuning enables the creation of robust and generalizable models.

As we move forward in this workshop, you are encouraged to:

- Experiment with different architectures and data augmentation strategies.
- Investigate ways to improve the model’s performance on edge cases or ambiguous samples.

- Consider deployment strategies for integrating defect detection models into real-world systems.

Bonus Exercise: Trying a Different Pre-trained Model

To deepen your understanding of transfer learning and model fine-tuning, try swapping out MobileNetV2 with a different pre-trained model. This exercise will help you explore the flexibility of pre-trained models and understand the trade-offs between different architectures.

Step 1: Choose a New Pre-trained Model

Keras provides several pre-trained models in the `tf.keras.applications` module. Some popular choices include:

- **ResNet50**: A deeper architecture designed to learn complex patterns, known for its residual connections.
- **InceptionV3**: A model that uses a creative "Inception" block to capture features at multiple scales.
- **EfficientNetB0**: A highly optimized model that balances accuracy and computational efficiency.

Choose one of these models based on your goals:

- If you want higher accuracy and don't mind a longer training time, try **ResNet50** or **InceptionV3**.
- If you need efficiency and speed, go with **EfficientNetB0**.

Step 2: Replace MobileNetV2 with Your Chosen Model

1. Import your chosen model from `tf.keras.applications`.
2. Replace MobileNetV2 in the model setup:
   - Ensure you set `include_top=False` to exclude the original classification head.
   - Specify the input shape `(224, 224, 3)` (or modify it to match your dataset).
3. Freeze the base layers of the new model so you can fine-tune only the top layers.

Step 3: Adjust Preprocessing

Each pre-trained model has specific preprocessing requirements:

- Use the corresponding `preprocess_input` function from `tf.keras.applications.<model>` to normalize your dataset correctly.
- For example:

- ResNet50 and InceptionV3 require inputs scaled to [-1, 1].
- EfficientNetB0 requires inputs scaled to [0, 1].

Update your preprocessing pipeline to match the model's expectations.

Step 4: Add New Task-Specific Layers

Attach new layers to adapt the model to your defect detection task:

1. Use a **GlobalAveragePooling2D** layer to reduce spatial dimensions.
2. Add one or more **Dense** layers for additional learning.
3. Use a **Dense(1, activation='sigmoid')** layer as the output for binary classification.

Step 5: Compile and Train

1. Compile the model using the same loss function (`binary_crossentropy`) and optimizer (`Adam`).
2. Train the model on the same dataset for 15 epochs.
3. Monitor the training and validation accuracy to compare performance with MobileNetV2.

Step 6: Evaluate and Compare Results

After training, evaluate the model on the testing set using:

1. Accuracy, precision, recall, and F1-score.
2. Confusion matrices and misclassified samples.

Compare the results with those from MobileNetV2:

- Did the new model perform better or worse?
- Was the training time noticeably different?
- Were there any patterns in the errors or low-confidence predictions?

In [24]:
```
from keras.applications import ResNet50
from keras.models import Sequential
from keras.layers import GlobalAveragePooling2D, Dense, Input
from keras.optimizers import Adam
from keras.applications.resnet50 import preprocess_input
import tensorflow as tf

# Step 1: Load the ResNet50 model without the top layers
base_model = ResNet50(
    input_shape=(224, 224, 3),
    include_top=False,  # Exclude the classification head
    weights='imagenet'  # Use pre-trained weights from ImageNet
)
```

```
# Freeze the base model layers
base_model.trainable = False

# Step 2: Preprocess the dataset for ResNet50
def preprocess(image, label):
    image = preprocess_input(image)  # Rescale to match ResNet50's requireme
    return image, label

train_dataset_resnet = train_dataset.map(preprocess)
test_dataset_resnet = test_dataset.map(preprocess)

# Step 3: Build the new model with task-specific layers
model_resnet = Sequential([
    Input(shape=(224, 224, 3)),       # Input shape
    base_model,                       # Pre-trained ResNet50 base
    GlobalAveragePooling2D(),         # Reduce spatial dimensions
    Dense(128, activation='relu'),    # Fully connected layer
    Dense(1, activation='sigmoid')    # Output layer for binary classificati
])

# Step 4: Compile the model
model_resnet.compile(
    optimizer=Adam(learning_rate=0.001),  # Use Adam optimizer
    loss='binary_crossentropy',           # Binary cross-entropy loss
    metrics=['accuracy']                  # Monitor accuracy
)

# Step 5: Train the model
AUTOTUNE = tf.data.AUTOTUNE
train_dataset_resnet = train_dataset_resnet.prefetch(buffer_size=AUTOTUNE)
test_dataset_resnet = test_dataset_resnet.prefetch(buffer_size=AUTOTUNE)

history_resnet = model_resnet.fit(
    train_dataset_resnet,
    epochs=15,
    validation_data=test_dataset_resnet
)

# Step 6: Evaluate the model
import matplotlib.pyplot as plt

# Plot the validation accuracy
plt.plot(history_resnet.history['val_accuracy'], label='Validation Accuracy'
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Validation Accuracy Over Epochs (ResNet50)')
plt.legend()
plt.show()

# Evaluate on the test set
test_loss, test_accuracy = model_resnet.evaluate(test_dataset_resnet)
print(f"Test Accuracy: {test_accuracy:.2f}")
```

```
Epoch 1/15
1/7 ━━━━━━━━━━━━━━━━━━━━ 28s 5s/step - accuracy: 0.4062 - loss: 1.0071
```

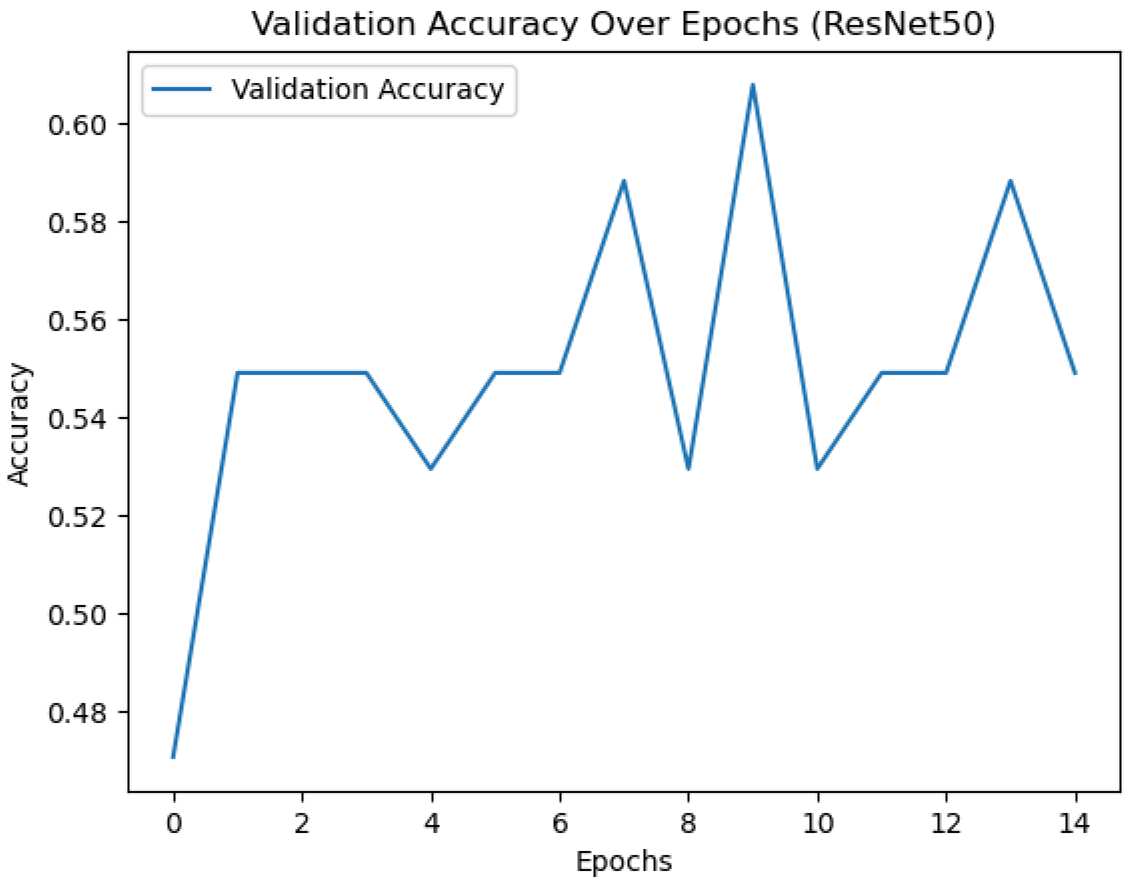
```
2/7 ━━━━━━━━━━━━━━━━━━━━ 6s 1s/step - accuracy: 0.4453 - loss: 0.9285
3/7 ━━━━━━━━━━━━━━━━━━━━ 5s 1s/step - accuracy: 0.4601 - loss: 0.9042
4/7 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - accuracy: 0.4779 - loss: 0.8817
5/7 ━━━━━━━━━━━━━━━━━━━━ 2s 1s/step - accuracy: 0.4860 - loss: 0.8710
6/7 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step - accuracy: 0.4918 - loss: 0.8605
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - accuracy: 0.4962 - loss: 0.8523
7/7 ━━━━━━━━━━━━━━━━━━━━ 15s 2s/step - accuracy: 0.4995 - loss: 0.8462 - val
_accuracy: 0.4706 - val_loss: 0.7009
Epoch 2/15
1/7 ━━━━━━━━━━━━━━━━━━━━ 8s 1s/step - accuracy: 0.5938 - loss: 0.6707
2/7 ━━━━━━━━━━━━━━━━━━━━ 6s 1s/step - accuracy: 0.5391 - loss: 0.6986
3/7 ━━━━━━━━━━━━━━━━━━━━ 5s 1s/step - accuracy: 0.5017 - loss: 0.7190
4/7 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - accuracy: 0.4935 - loss: 0.7243
5/7 ━━━━━━━━━━━━━━━━━━━━ 2s 1s/step - accuracy: 0.4923 - loss: 0.7253
6/7 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step - accuracy: 0.4970 - loss: 0.7239
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - accuracy: 0.5007 - loss: 0.7226
7/7 ━━━━━━━━━━━━━━━━━━━━ 10s 1s/step - accuracy: 0.5034 - loss: 0.7216 - val
_accuracy: 0.5490 - val_loss: 0.7248
Epoch 3/15
1/7 ━━━━━━━━━━━━━━━━━━━━ 8s 1s/step - accuracy: 0.5312 - loss: 0.7389
2/7 ━━━━━━━━━━━━━━━━━━━━ 6s 1s/step - accuracy: 0.5234 - loss: 0.7468
3/7 ━━━━━━━━━━━━━━━━━━━━ 5s 1s/step - accuracy: 0.5399 - loss: 0.7329
4/7 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - accuracy: 0.5436 - loss: 0.7288
5/7 ━━━━━━━━━━━━━━━━━━━━ 2s 1s/step - accuracy: 0.5436 - loss: 0.7264
6/7 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step - accuracy: 0.5459 - loss: 0.7229
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - accuracy: 0.5469 - loss: 0.7204
7/7 ━━━━━━━━━━━━━━━━━━━━ 10s 1s/step - accuracy: 0.5476 - loss: 0.7186 - val
_accuracy: 0.5490 - val_loss: 0.6908
Epoch 4/15
1/7 ━━━━━━━━━━━━━━━━━━━━ 8s 1s/step - accuracy: 0.5938 - loss: 0.6744
2/7 ━━━━━━━━━━━━━━━━━━━━ 6s 1s/step - accuracy: 0.5781 - loss: 0.6776
3/7 ━━━━━━━━━━━━━━━━━━━━ 5s 1s/step - accuracy: 0.5660 - loss: 0.6780
```

```
4/7 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - accuracy: 0.5612 - loss: 0.6788
5/7 ━━━━━━━━━━━━━━━━━━━━ 2s 1s/step - accuracy: 0.5615 - loss: 0.6788
6/7 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step - accuracy: 0.5590 - loss: 0.6788
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - accuracy: 0.5589 - loss: 0.6784
7/7 ━━━━━━━━━━━━━━━━━━━━ 10s 1s/step - accuracy: 0.5587 - loss: 0.6782 - val
_accuracy: 0.5490 - val_loss: 0.6936
Epoch 5/15
1/7 ━━━━━━━━━━━━━━━━━━━━ 8s 1s/step - accuracy: 0.6250 - loss: 0.6431
2/7 ━━━━━━━━━━━━━━━━━━━━ 6s 1s/step - accuracy: 0.6250 - loss: 0.6465
3/7 ━━━━━━━━━━━━━━━━━━━━ 5s 1s/step - accuracy: 0.6215 - loss: 0.6511
4/7 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - accuracy: 0.6126 - loss: 0.6644
5/7 ━━━━━━━━━━━━━━━━━━━━ 2s 1s/step - accuracy: 0.6051 - loss: 0.6746
6/7 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step - accuracy: 0.5971 - loss: 0.6833
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - accuracy: 0.5915 - loss: 0.6892
7/7 ━━━━━━━━━━━━━━━━━━━━ 10s 1s/step - accuracy: 0.5873 - loss: 0.6936 - val
_accuracy: 0.5294 - val_loss: 0.6796
Epoch 6/15
1/7 ━━━━━━━━━━━━━━━━━━━━ 8s 1s/step - accuracy: 0.5938 - loss: 0.6789
2/7 ━━━━━━━━━━━━━━━━━━━━ 6s 1s/step - accuracy: 0.6016 - loss: 0.6719
3/7 ━━━━━━━━━━━━━━━━━━━━ 5s 1s/step - accuracy: 0.5747 - loss: 0.6810
4/7 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - accuracy: 0.5482 - loss: 0.6920
5/7 ━━━━━━━━━━━━━━━━━━━━ 2s 1s/step - accuracy: 0.5348 - loss: 0.6954
6/7 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step - accuracy: 0.5290 - loss: 0.6962
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - accuracy: 0.5238 - loss: 0.6965
7/7 ━━━━━━━━━━━━━━━━━━━━ 10s 1s/step - accuracy: 0.5199 - loss: 0.6967 - val
_accuracy: 0.5490 - val_loss: 0.6801
Epoch 7/15
1/7 ━━━━━━━━━━━━━━━━━━━━ 8s 1s/step - accuracy: 0.6250 - loss: 0.6509
2/7 ━━━━━━━━━━━━━━━━━━━━ 6s 1s/step - accuracy: 0.6562 - loss: 0.6298
3/7 ━━━━━━━━━━━━━━━━━━━━ 5s 1s/step - accuracy: 0.6389 - loss: 0.6445
4/7 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - accuracy: 0.6237 - loss: 0.6593
5/7 ━━━━━━━━━━━━━━━━━━━━ 2s 1s/step - accuracy: 0.6127 - loss: 0.6711
```

```
6/7 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step - accuracy: 0.6061 - loss: 0.6768
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - accuracy: 0.6006 - loss: 0.6809
7/7 ━━━━━━━━━━━━━━━━━━━━ 10s 1s/step - accuracy: 0.5965 - loss: 0.6839 - val
_accuracy: 0.5490 - val_loss: 0.6799
Epoch 8/15
1/7 ━━━━━━━━━━━━━━━━━━━━ 8s 1s/step - accuracy: 0.5938 - loss: 0.6550
2/7 ━━━━━━━━━━━━━━━━━━━━ 6s 1s/step - accuracy: 0.6328 - loss: 0.6533
3/7 ━━━━━━━━━━━━━━━━━━━━ 5s 1s/step - accuracy: 0.6128 - loss: 0.6576
4/7 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - accuracy: 0.5983 - loss: 0.6600
5/7 ━━━━━━━━━━━━━━━━━━━━ 2s 1s/step - accuracy: 0.5811 - loss: 0.6645
6/7 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step - accuracy: 0.5685 - loss: 0.6672
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - accuracy: 0.5605 - loss: 0.6688
7/7 ━━━━━━━━━━━━━━━━━━━━ 10s 1s/step - accuracy: 0.5545 - loss: 0.6701 - val
_accuracy: 0.5882 - val_loss: 0.6843
Epoch 9/15
1/7 ━━━━━━━━━━━━━━━━━━━━ 8s 1s/step - accuracy: 0.5625 - loss: 0.6714
2/7 ━━━━━━━━━━━━━━━━━━━━ 6s 1s/step - accuracy: 0.6250 - loss: 0.6608
3/7 ━━━━━━━━━━━━━━━━━━━━ 5s 1s/step - accuracy: 0.6458 - loss: 0.6533
4/7 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - accuracy: 0.6582 - loss: 0.6472
5/7 ━━━━━━━━━━━━━━━━━━━━ 2s 1s/step - accuracy: 0.6616 - loss: 0.6449
6/7 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step - accuracy: 0.6598 - loss: 0.6445
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - accuracy: 0.6582 - loss: 0.6441
7/7 ━━━━━━━━━━━━━━━━━━━━ 10s 1s/step - accuracy: 0.6569 - loss: 0.6439 - val
_accuracy: 0.5294 - val_loss: 0.6927
Epoch 10/15
1/7 ━━━━━━━━━━━━━━━━━━━━ 8s 1s/step - accuracy: 0.6250 - loss: 0.6140
2/7 ━━━━━━━━━━━━━━━━━━━━ 6s 1s/step - accuracy: 0.6406 - loss: 0.6182
3/7 ━━━━━━━━━━━━━━━━━━━━ 5s 1s/step - accuracy: 0.6250 - loss: 0.6252
4/7 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - accuracy: 0.6270 - loss: 0.6268
5/7 ━━━━━━━━━━━━━━━━━━━━ 2s 1s/step - accuracy: 0.6328 - loss: 0.6285
6/7 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step - accuracy: 0.6359 - loss: 0.6301
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - accuracy: 0.6391 - loss: 0.6312
```

```
7/7 ━━━━━━━━━━━━━━━━━━━━ 10s 1s/step - accuracy: 0.6415 - loss: 0.6319 - val
_accuracy: 0.6078 - val_loss: 0.6838
Epoch 11/15
1/7 ━━━━━━━━━━━━━━━━━━━━ 8s 1s/step - accuracy: 0.6875 - loss: 0.6297
2/7 ━━━━━━━━━━━━━━━━━━━━ 6s 1s/step - accuracy: 0.7109 - loss: 0.6274
3/7 ━━━━━━━━━━━━━━━━━━━━ 5s 1s/step - accuracy: 0.7205 - loss: 0.6249
4/7 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - accuracy: 0.7142 - loss: 0.6252
5/7 ━━━━━━━━━━━━━━━━━━━━ 2s 1s/step - accuracy: 0.7051 - loss: 0.6266
6/7 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step - accuracy: 0.6952 - loss: 0.6286
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - accuracy: 0.6864 - loss: 0.6308
7/7 ━━━━━━━━━━━━━━━━━━━━ 10s 1s/step - accuracy: 0.6797 - loss: 0.6325 - val
_accuracy: 0.5294 - val_loss: 0.6756
Epoch 12/15
1/7 ━━━━━━━━━━━━━━━━━━━━ 8s 1s/step - accuracy: 0.6562 - loss: 0.6446
2/7 ━━━━━━━━━━━━━━━━━━━━ 6s 1s/step - accuracy: 0.6172 - loss: 0.6503
3/7 ━━━━━━━━━━━━━━━━━━━━ 5s 1s/step - accuracy: 0.6024 - loss: 0.6496
4/7 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - accuracy: 0.5924 - loss: 0.6501
5/7 ━━━━━━━━━━━━━━━━━━━━ 2s 1s/step - accuracy: 0.5740 - loss: 0.6559
6/7 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step - accuracy: 0.5642 - loss: 0.6587
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - accuracy: 0.5569 - loss: 0.6607
7/7 ━━━━━━━━━━━━━━━━━━━━ 10s 1s/step - accuracy: 0.5513 - loss: 0.6623 - val
_accuracy: 0.5490 - val_loss: 0.6774
Epoch 13/15
1/7 ━━━━━━━━━━━━━━━━━━━━ 8s 1s/step - accuracy: 0.6250 - loss: 0.6381
2/7 ━━━━━━━━━━━━━━━━━━━━ 6s 1s/step - accuracy: 0.6562 - loss: 0.6164
3/7 ━━━━━━━━━━━━━━━━━━━━ 5s 1s/step - accuracy: 0.6667 - loss: 0.6063
4/7 ━━━━━━━━━━━━━━━━━━━━ 3s 1s/step - accuracy: 0.6602 - loss: 0.6122
5/7 ━━━━━━━━━━━━━━━━━━━━ 2s 1s/step - accuracy: 0.6531 - loss: 0.6204
6/7 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step - accuracy: 0.6467 - loss: 0.6269
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 1s/step - accuracy: 0.6397 - loss: 0.6334
7/7 ━━━━━━━━━━━━━━━━━━━━ 10s 1s/step - accuracy: 0.6345 - loss: 0.6382 - val
_accuracy: 0.5490 - val_loss: 0.6763
Epoch 14/15
1/7 ━━━━━━━━━━━━━━━━━━━━ 8s 1s/step - accuracy: 0.6875 - loss: 0.6148
```

2/7 **6s** 1s/step - accuracy: 0.6641 - loss: 0.6183
3/7 **5s** 1s/step - accuracy: 0.6267 - loss: 0.6328
4/7 **3s** 1s/step - accuracy: 0.6068 - loss: 0.6423
5/7 **2s** 1s/step - accuracy: 0.5829 - loss: 0.6615
6/7 **1s** 1s/step - accuracy: 0.5665 - loss: 0.6733
7/7 **0s** 1s/step - accuracy: 0.5545 - loss: 0.6816
7/7 **10s** 1s/step - accuracy: 0.5455 - loss: 0.6878 - val_accuracy: 0.5882 - val_loss: 0.6836
Epoch 15/15
1/7 **8s** 1s/step - accuracy: 0.7500 - loss: 0.6029
2/7 **6s** 1s/step - accuracy: 0.6953 - loss: 0.6092
3/7 **5s** 1s/step - accuracy: 0.6580 - loss: 0.6200
4/7 **3s** 1s/step - accuracy: 0.6439 - loss: 0.6231
5/7 **2s** 1s/step - accuracy: 0.6389 - loss: 0.6224
6/7 **1s** 1s/step - accuracy: 0.6339 - loss: 0.6234
7/7 **0s** 1s/step - accuracy: 0.6295 - loss: 0.6248
7/7 **10s** 1s/step - accuracy: 0.6262 - loss: 0.6258 - val_accuracy: 0.5490 - val_loss: 0.7111




```
1/2 ━━━━━━━━━━ 1s 1s/step - accuracy: 0.5312 - loss: 0.7480
2/2 ━━━━━━━━━━ 0s 739ms/step - accuracy: 0.5401 - loss: 0.7295
2/2 ━━━━━━━━━━ 2s 758ms/step - accuracy: 0.5431 - loss: 0.7234
Test Accuracy: 0.55
```

In [ ]: