Springboard

AI Innovation in Quality Control:

Harnessing Deep Learning and Convolutional Neural Networks to detect Defective Metal Parts in Manufacturing environment

3rd Data Science Intensive Capstone Project

September 9th, 2024

By Javier Jorge Pérez Ontiveros

Springboard

Index

Problem Identification *3*

Introduction 3

Context 3

Objectives 3

Product Application 4

"Defective" Product vs "OK" Product 4

EDA and Data wrangling *5*

EDA and Data Wrangling Learnings 8

Next Steps defined after EDA and Data Wrangling: 8

Preprocessing, Training and Model *9*

Training and Testing Split 10

First CNN Model Creation 10

Second Model 12

Hyperparameter tunning 14

Results and Final Evaluation of Second Model 15

Visualization of how the Model "sees" my defects 16

Transfer Learning *17*

Keras VGG16 model re-trained for Industrial Metal Casting Pictures 17

Model Architecture 18

Transfer Molding Results 18

Model Selection *19*

Final Conclusions *19*

Problem Identification

Introduction

In the manufacturing industry, quality control is a key process that affects a company's reputation and reliability. Traditional methods of quality inspection, especially in metal casting, often depend on manual visual checks. These checks can be slow, tiring, and prone to mistakes because they rely on humans. With the rise of Artificial Intelligence (AI) and Deep Learning, particularly Convolutional Neural Networks (CNNs), there is a great opportunity to automate and improve the accuracy of these inspections. This project looks at how AI can be used to improve the quality control process in metal parts manufacturing, with a focus on submersible pump impellers.

Context

The main problem in manufacturing metal casting products, like submersible pump impellers, is that the current manual inspection process is inefficient. It takes a lot of time, requires a lot of work, and can be inconsistent because of human errors. Defects like blow holes, pinholes, burrs, and shrinkage can cause significant financial losses if not detected correctly and on time. The goal of this project is to create and use a Deep Learning model with CNNs to automate the defect detection process, making it more accurate and efficient.

Objectives

1. **Model Development:** Develop a reliable CNN model that can accurately classify metal casting products as "Defective" or "Non-Defective" using a dataset of 7,348 gray-scale images of submersible pump impellers.

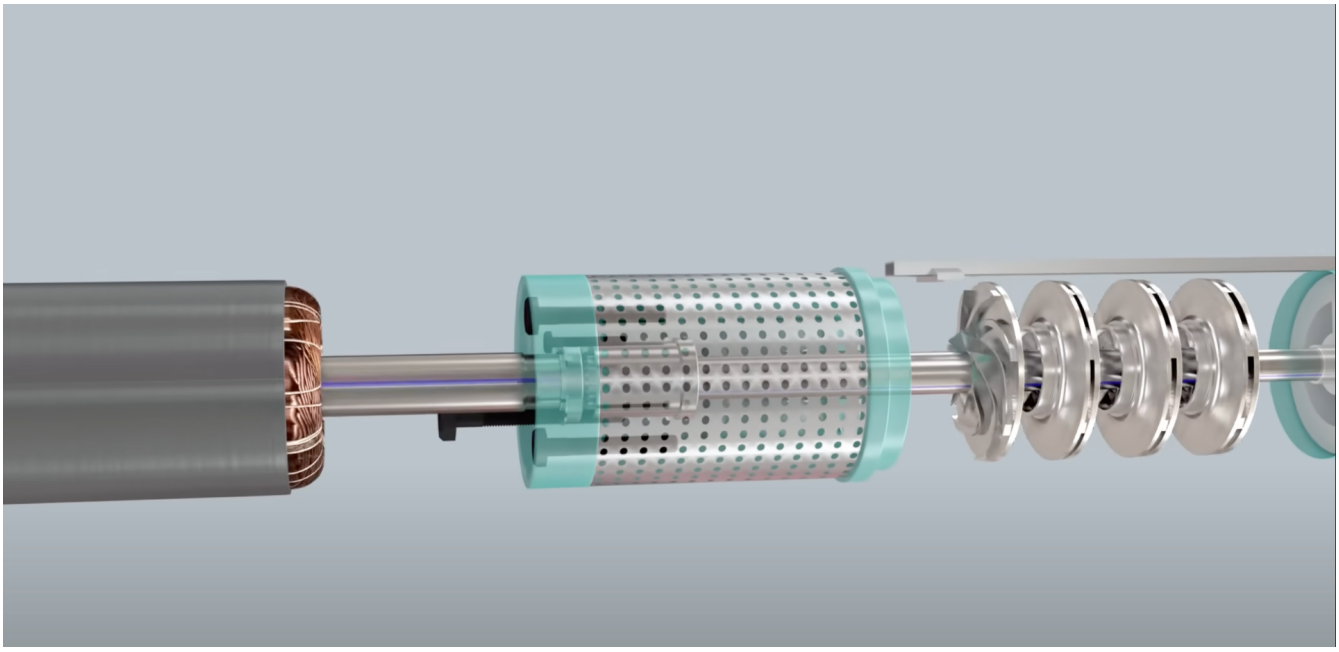
2. **Accuracy Improvement:** Achieve a minimum classification accuracy of 95% in identifying defective and non-defective metal casting products.

3. **Time Efficiency:** Reduce the time needed for quality inspections by at least 50%, which will help increase production efficiency.

4. **Cost Reduction:** Show measurable cost savings by reducing the financial impact of undetected defects and minimizing the need for rework or rejected orders

5. **Scalability and Transfer Learning:** Explore the use of transfer learning to improve model performance and make sure that the solution can be scaled to other similar use cases in the manufacturing industry.

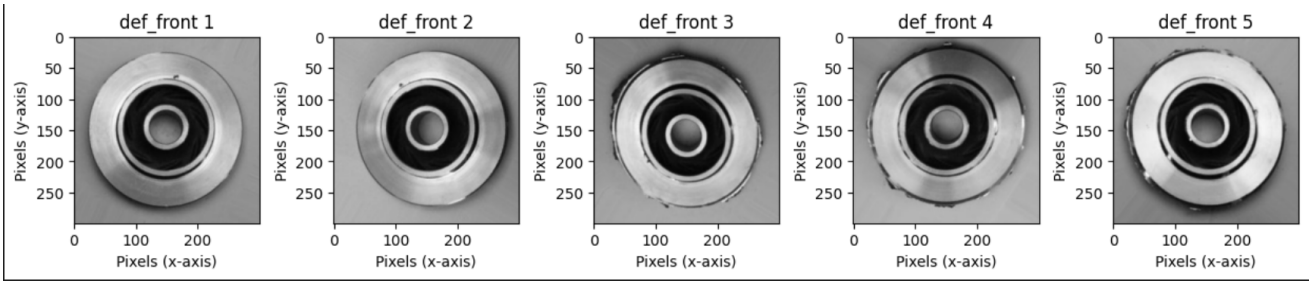
Product Application



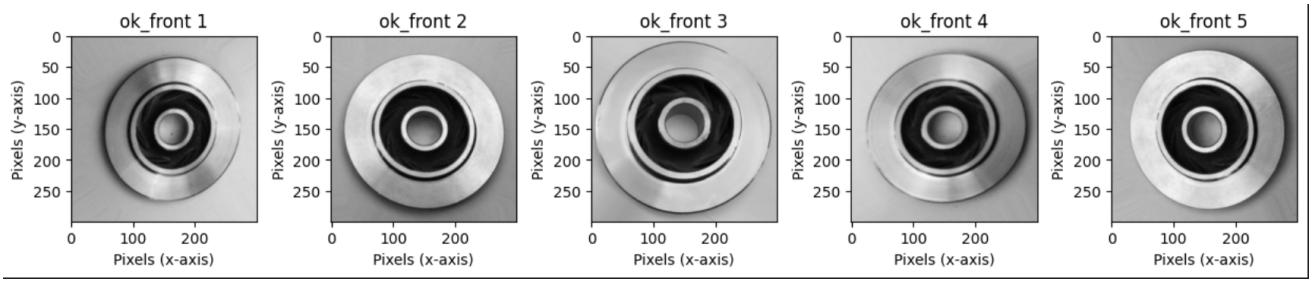
“Defective” Product vs “OK” Product

During this study good parts will be referred as "OK", while defective Parts will be marked as "DEF" (Defective).

These 5 pictures show different metallic parts with defects:



While these 5 pictures don’t have any defect:



EDA and Data wrangling

During this Phase, we conducted an exploratory data analysis (EDA) and various data wrangling processes on a dataset of pump images categorized as defective (`def_front`) and non-defective (`ok_front`) following the next steps

- **Image Visualization**
- **Image Counting by Category: Testing vs Training, OK vs Defective.**
- **Image Augmentation to balance the categories.**
- **Image Size Analysis**
- **Pixel Combination and Histogram Analysis**
- **Entropy Analysis**

The first challenge was to count the number of “Defective” and “OK” images in both folders: Training & Testing.

```
print(f"Training set: {train_def} defective, {train_ok} OK")
print(f"Test set: {test_def} defective, {test_ok} OK")
```

```
Training set: 3758 defective, 2875 OK
Test set: 453 defective, 262 OK
```

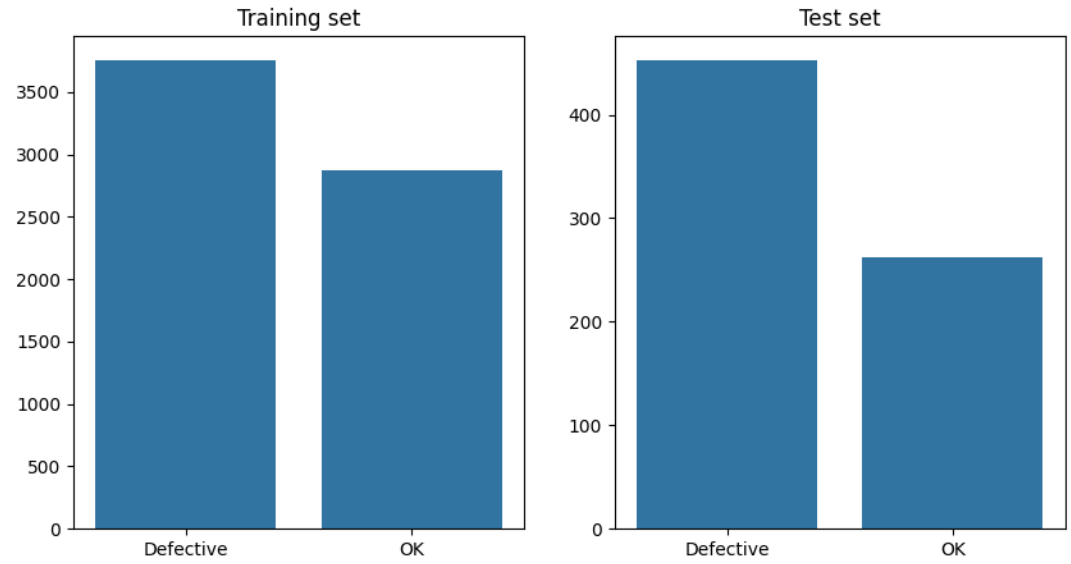
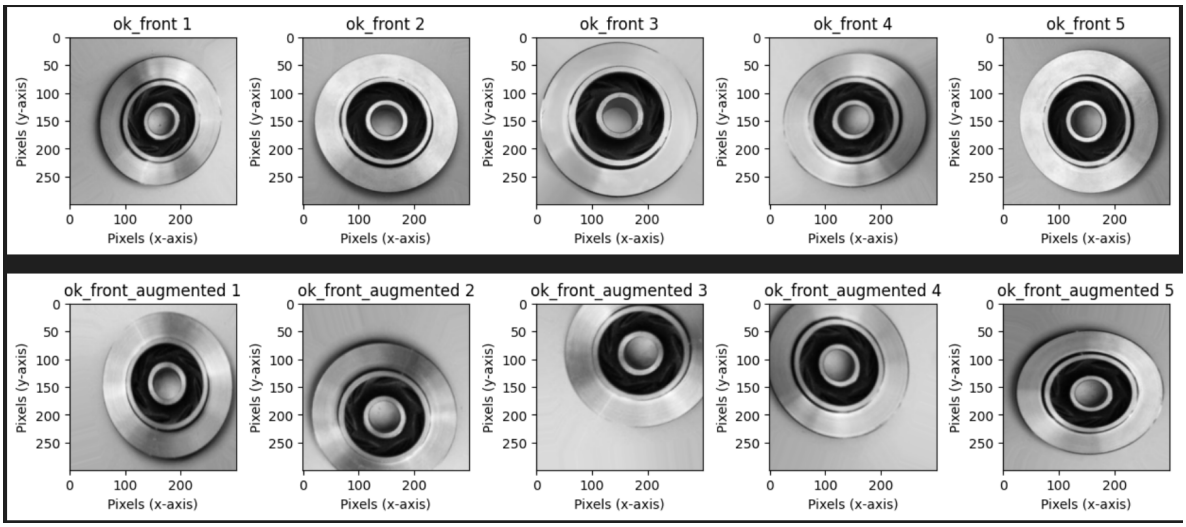


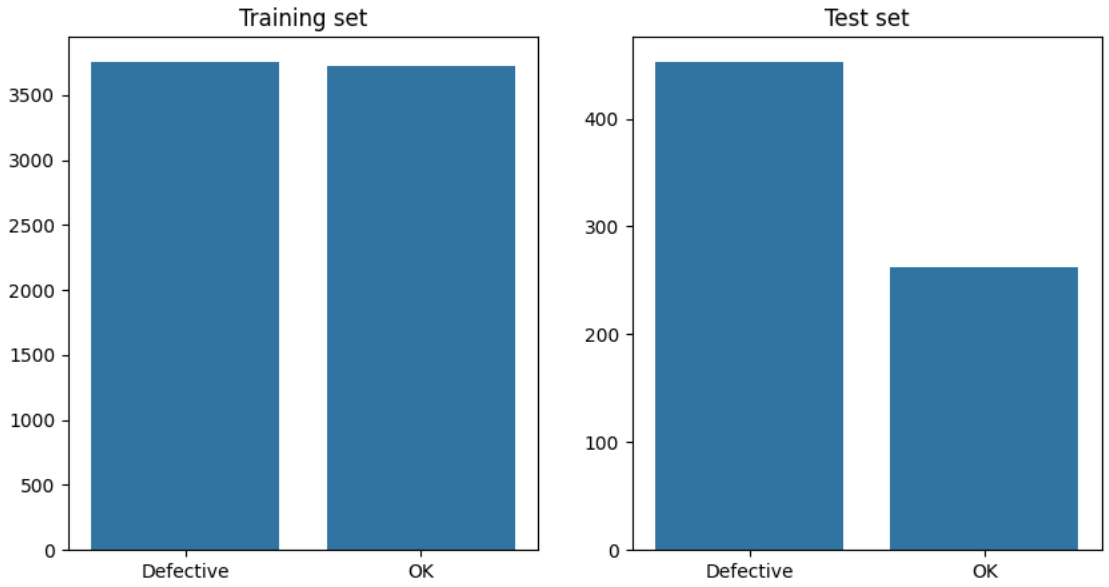
Image Size Analysis:

```
Image shape: (300, 300, 3)
Image shape: (300, 300, 3)
```

As the training set was very imbalanced, number of OK Pictures had to be increased with Data Augmentation Techniques: Rotation, Zoom, Flips, Brightness, etc.,



As a result the Training Pictures was then balanced, but the Testing Pictures remained untouched:

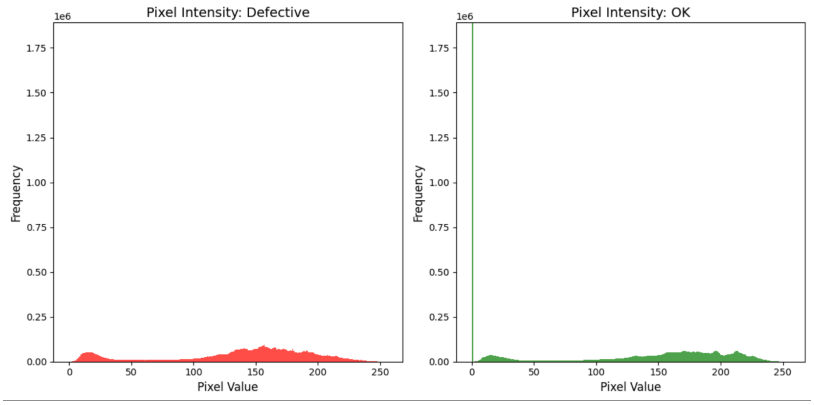


The images of the training set were reviewed to validate the same size (300 x 300)

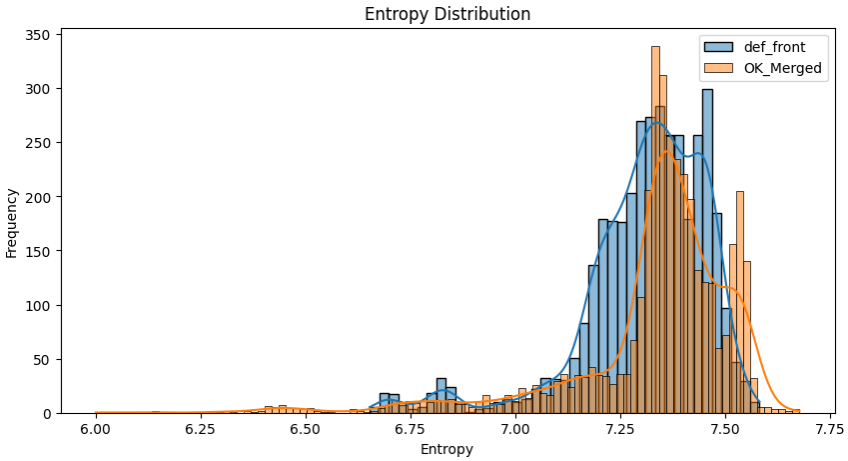
```
Unique image sizes in /def_front: {(300, 300)}
Unique image sizes in /OK_Merged: {(300, 300)}
```

This test confirmed that all the pictures had the same unique Size (300 x 300).

Pixel intensity was tested in both samples, showing a slight difference between the two groups, but the pattern is not easily identified by humans.



Entropy was also measured to identify differences between two categories of pictures by helping quantify the level of uncertainty or “disorder” in the image data.



There are some differences between the two categories:

1. **Shift in Entropy Distribution:**

The Def images show higher average entropy than OK, indicating defective parts tend to have more pixel irregularities, while non-defective parts are more consistent.

2. **Overlap but Distinct Peaks:**

There’s overlap in entropy values, but the peaks differ: Def peaks at 7.25 and OK at 7.35, however classification may be challenging within this overlap.

3. **Standard Deviation Differences:**

Defective parts show a higher standard deviation in entropy, reflecting greater variability caused by irregularities and texture variations from defects.

EDA and Data Wrangling Learnings

1. **Image Visualization**:

- We visually explored a set of images from both categories to understand the general characteristics and differences of defective and non-defective pumps.

2. **Image Distribution by Category**:

- We counted and visualized the distribution of images in both categories across the training and testing sets.

- The dataset appears to be relatively balanced between the two classes, however data augmentation was used to make it more balanced.

3. **Image Size Analysis**:

- We verified the sizes of the images to ensure uniformity within them (300 x300) . This step is crucial for consistent processing in later stages of the pipeline.

4. **Pixel Data Combination and Histogram Analysis**:

- We combined pixel data from a sample of images and generated histograms to visualize the distribution of pixel intensity across both categories.

- This analysis revealed differences in pixel intensity distribution, which could indicate unique characteristics in defective pumps.

5. **Entropy Analysis**:

- We calculated the entropy of the images to measure the amount of information contained within them.

- We found that non-defective images tend to have slightly higher entropy, suggesting they might contain more detail or variability compared to defective images.

Next Steps defined after EDA and Data Wrangling:

1. **Modeling:** Use CNN to train and evaluate classification models.

2. **Advanced Deep Learning:** Consider using more advanced techniques, such as transferred learning to enhance model accuracy.

Preprocessing, Training and Model

During this sprint, 3 steps were further developed: Preprocessing, Training, Model Development. This is a brief summary of the main topics covered in each segment:

1. Preprocessing

- Apply transformations like resizing, normalization, and augmentation to ensure the dataset is well-prepared for training.

- Balance the dataset through data augmentation techniques, ensuring the model learns effectively from both defective and OK parts.

2. Training and Model Development

- Build two to three models to predict whether a part is defective or OK.

- Use convolutional neural networks (CNNs) designed for image classification and experiment with different architectures.

- Fine-tune the hyperparameters of the models to improve accuracy.

3. Evaluation and Model Selection

- Train the models on the dataset while ensuring proper data splitting between training and test sets to avoid overfitting.

- Evaluate each model's performance using accuracy, precision, recall, and F1 score to determine which model works best.

- Choose the best-performing model based on these metrics and apply any necessary adjustments to improve its performance.

Training and Testing Split

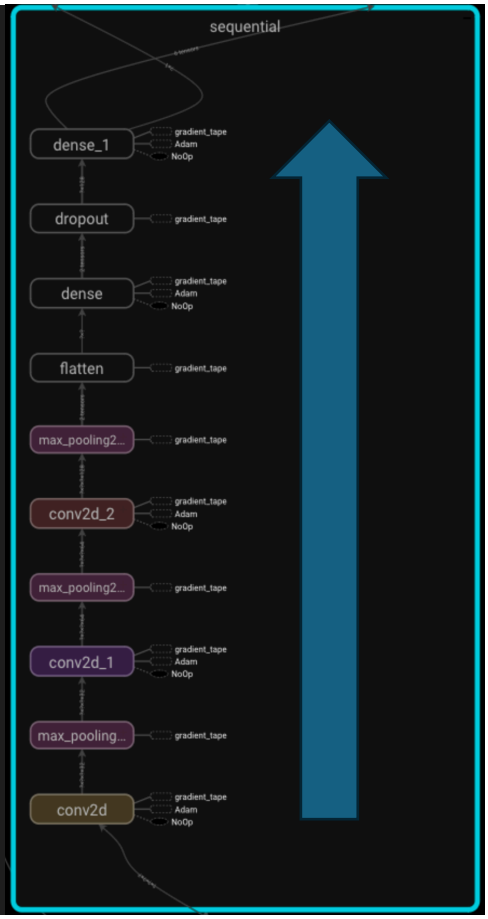
The data was split in Training and Testing as can be observed below:

```
Class indices (mapping of labels): {'def_front': 0, 'ok_front': 1}

Training class distribution:
def_front: 3758 images
ok_front: 3729 images

Test class distribution:
def_front: 453 images
ok_front: 262 images
```

First CNN Model Creation

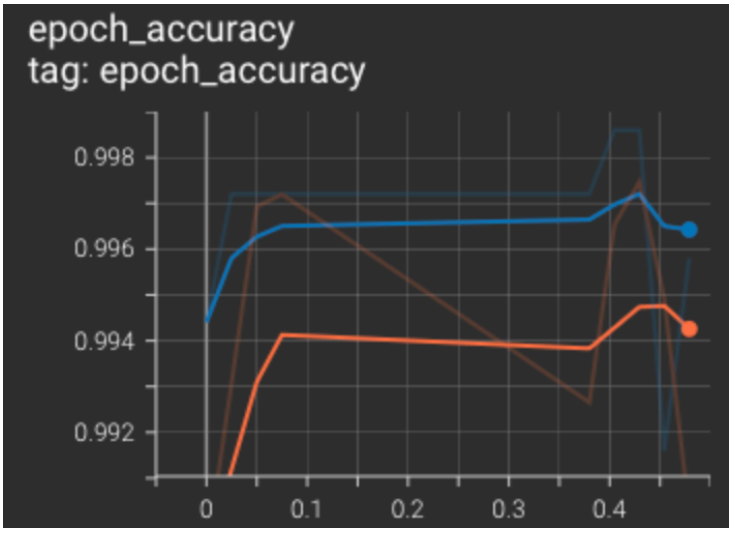
The first model created had a high complexity (Trying to cover as much accuracy as possible) as it can be observed below:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 298, 298, 32)      896

 max_pooling2d (MaxPooling2  (None, 149, 149, 32)      0
 D)

 conv2d_1 (Conv2D)           (None, 147, 147, 64)      18496

 max_pooling2d_1 (MaxPoolin  (None, 73, 73, 64)        0
 g2D)

 conv2d_2 (Conv2D)           (None, 71, 71, 128)       73856

 max_pooling2d_2 (MaxPoolin  (None, 35, 35, 128)       0
 g2D)

 flatten (Flatten)           (None, 156800)            0

 dense (Dense)               (None, 128)               20070528

 dropout (Dropout)           (None, 128)               0

...
Total params: 20163905 (76.92 MB)
Trainable params: 20163905 (76.92 MB)
Non-trainable params: 0 (0.00 Byte)
```

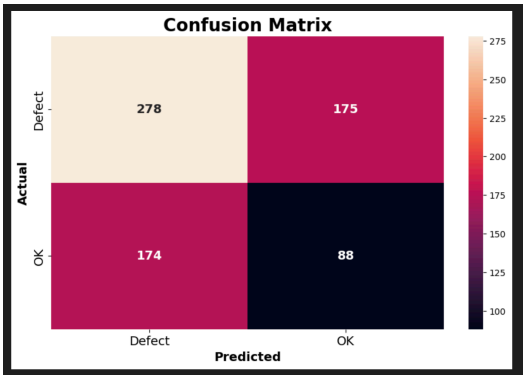


During training, the model seemed to be learning as the Accuracy of Training and Validation Data was having an accuracy above 99%.

```
...
Epoch 19/20
233/233 [==============================] - 88s 378ms/step - loss: 0.0075 - accuracy: 0.9970 - val_loss: 0.0324 - val_accuracy: 0.9929
Epoch 20/20
233/233 [==============================] - 88s 379ms/step - loss: 0.0262 - accuracy: 0.9910 - val_loss: 0.0122 - val_accuracy: 0.9972
```



However, when this model was tested on “unseen” data, the accuracy quickly dropped to 63%:



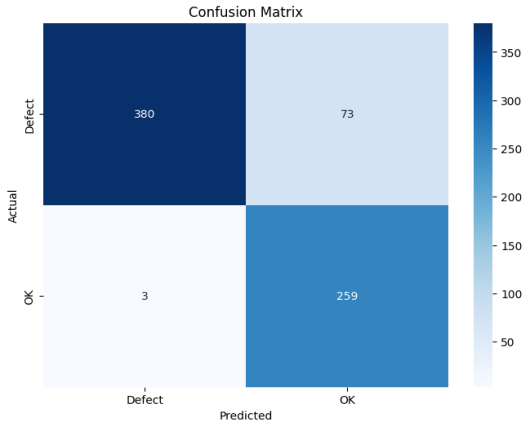

```
23/23 [==============================] - 2s 89ms/step
[[453   0]
 [262   0]]
              precision    recall  f1-score   support

           0       0.63      1.00      0.78       453
           1       0.00      0.00      0.00       262

    accuracy                           0.63       715
   macro avg       0.32      0.50      0.39       715
weighted avg       0.40      0.63      0.49       715
```

Second Model

The Model Architecture was changed to a Simpler model design.

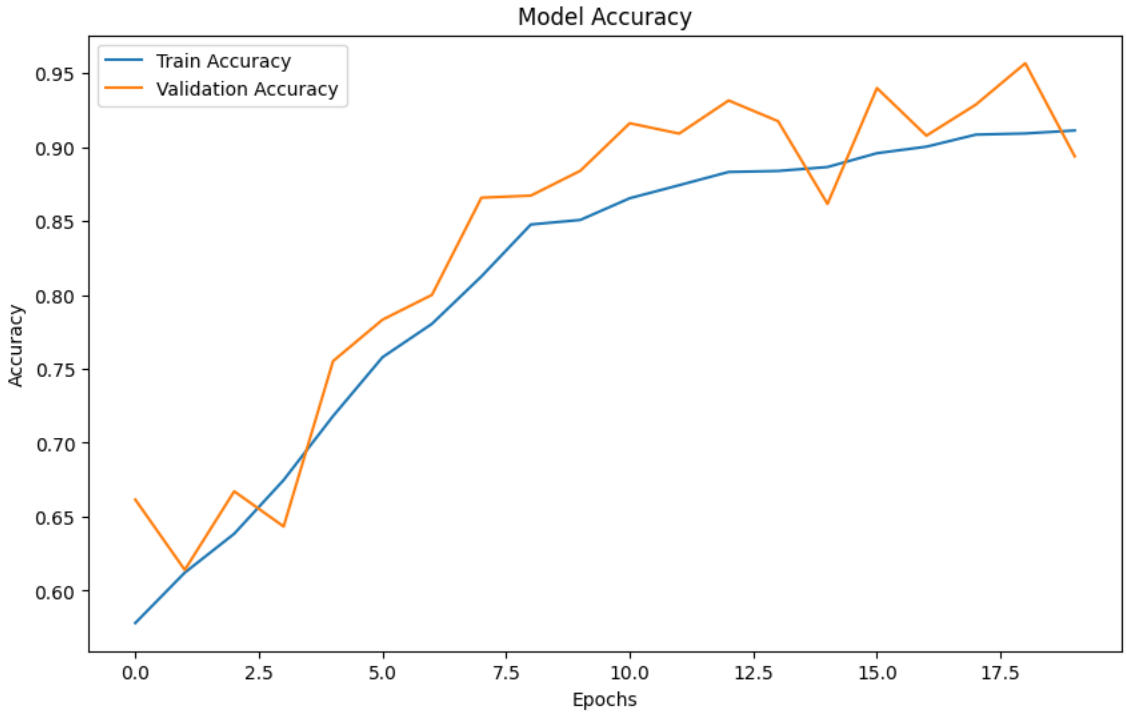
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 150, 150, 16)      2368

max_pooling2d (MaxPooling2D) (None, 75, 75, 16)        0

conv2d_1 (Conv2D)            (None, 75, 75, 32)        4640

max_pooling2d_1 (MaxPooling2D) (None, 37, 37, 32)      0

conv2d_2 (Conv2D)            (None, 37, 37, 64)        18496

max_pooling2d_2 (MaxPooling2D) (None, 18, 18, 64)      0

flatten (Flatten)            (None, 20736)             0

dense (Dense)                (None, 224)               4645088

dropout (Dropout)            (None, 224)               0
```

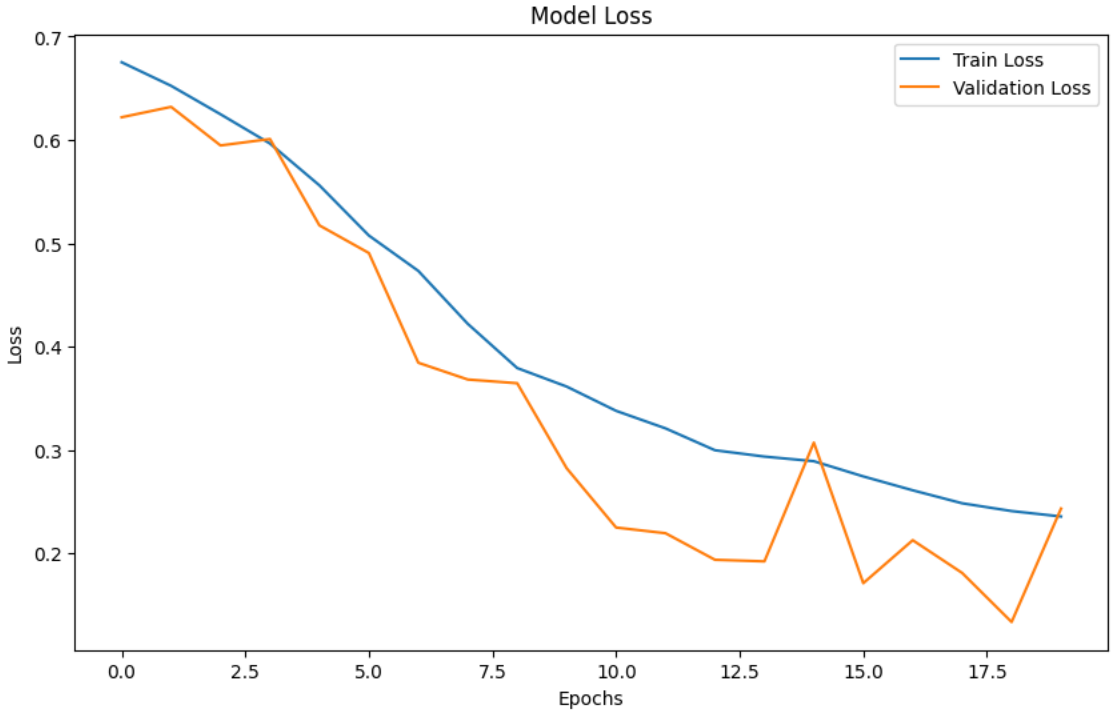
After the first training the Accuracy achieved was up to 89%.



	precision	recall	f1-score	support
Defect	0.99	0.84	0.91	453
OK	0.78	0.99	0.87	262
accuracy			0.89	715
macro avg	0.89	0.91	0.89	715
weighted avg	0.91	0.89	0.90	715

The learning curve after 20 epochs looks like this:





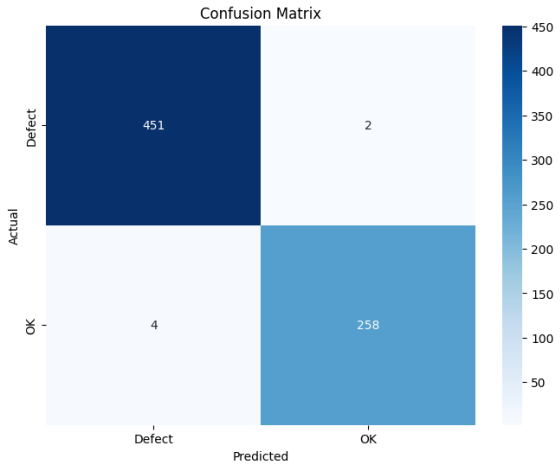
After Epoch #17 the error started to raise, so some Hyperparameter tunning was done afterwards.

Hyperparameter tunning

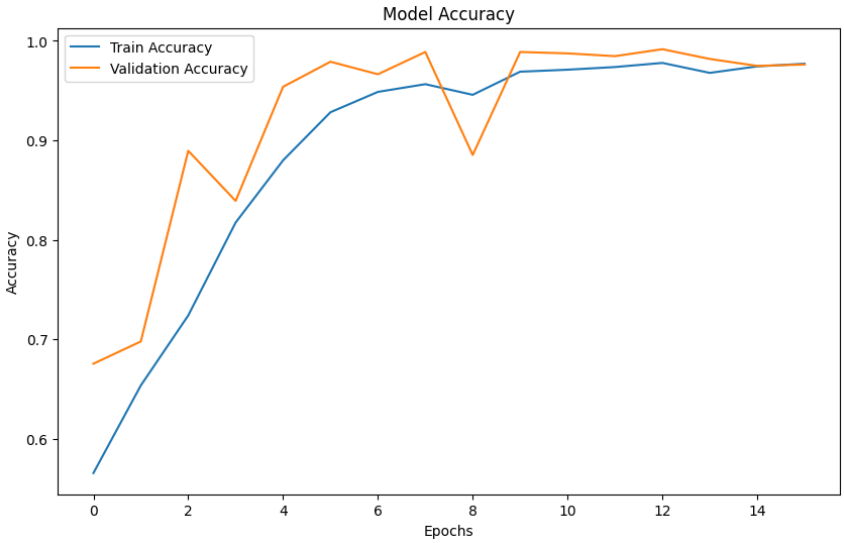
16 out of 20 epochs were compiled (Early stop was activated when the model stopped improving accuracy).

```
Best val_accuracy So Far: 0.988811194896698
Total elapsed time: 01h 33m 03s

The optimal number of filters for the first Conv2D layer is 16,
for the second Conv2D layer is 128,
for the third Conv2D layer is 256.
The optimal number of units in the Dense layer is 512.
The optimal dropout rate is 0.30000000000000004.
The optimal learning rate is 9.878314341240697e-05.
```

With this hyperparameter tunning the accuracy improved > 98%. This is a much better performance as it can be observed in the Confusion matrix:

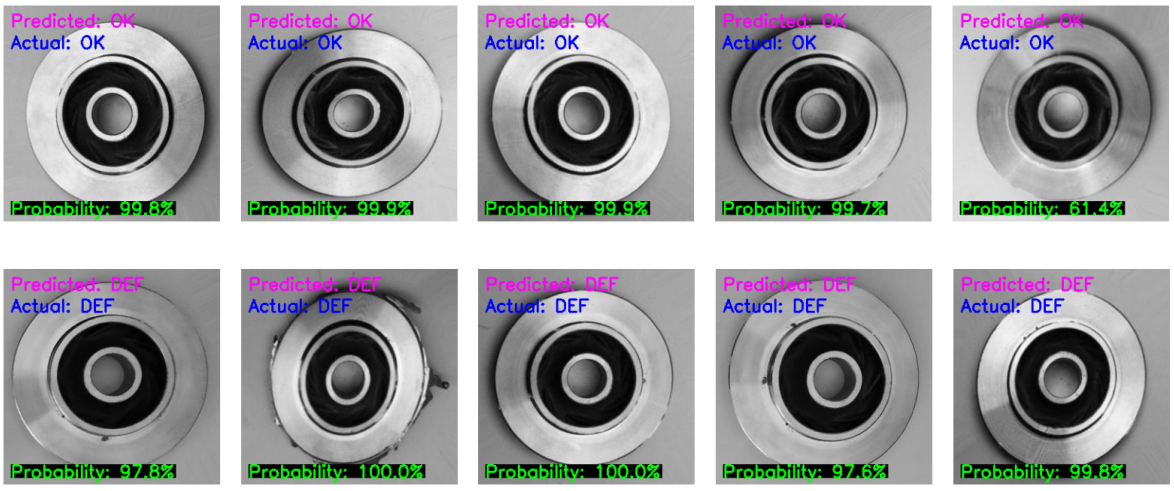


The accuracy of the first 16 Epochs can be observed below:



Results and Final Evaluation of Second Model

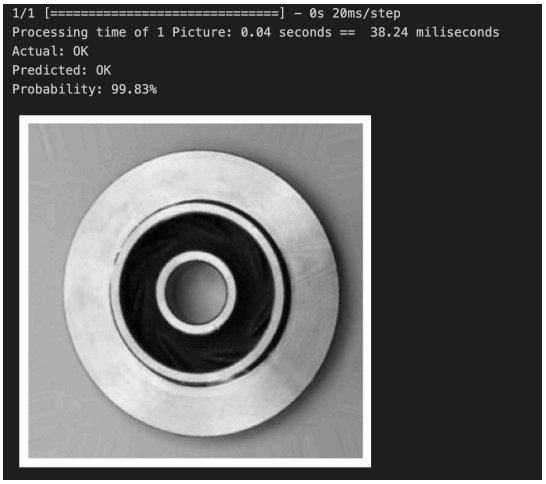
This visual representation of 10 random pictures (5 OK & 5 Defective), the Actual Lable can be observed in Blue and compared with the predicted value in Pink.



The probability calculated by the Model is obtained from the Sigmoid Activation Function at the last layer of the model:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The last step for validating this model was to measure the processing speed.

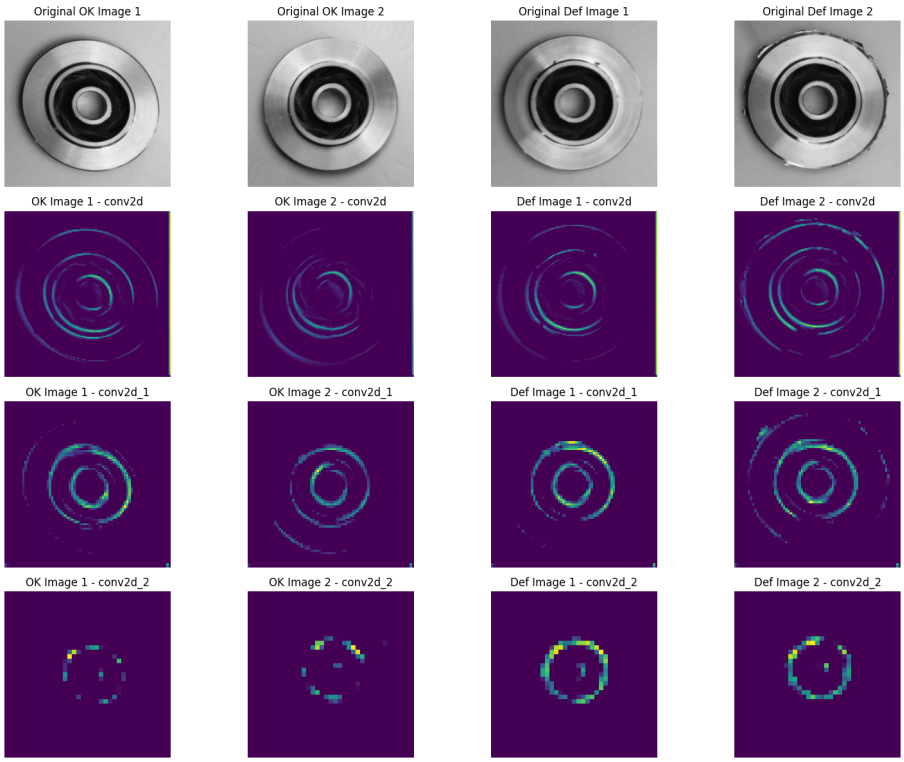


A computer clock was stablished before and after running trials, and the best processing time was 0.04 seconds per piece.

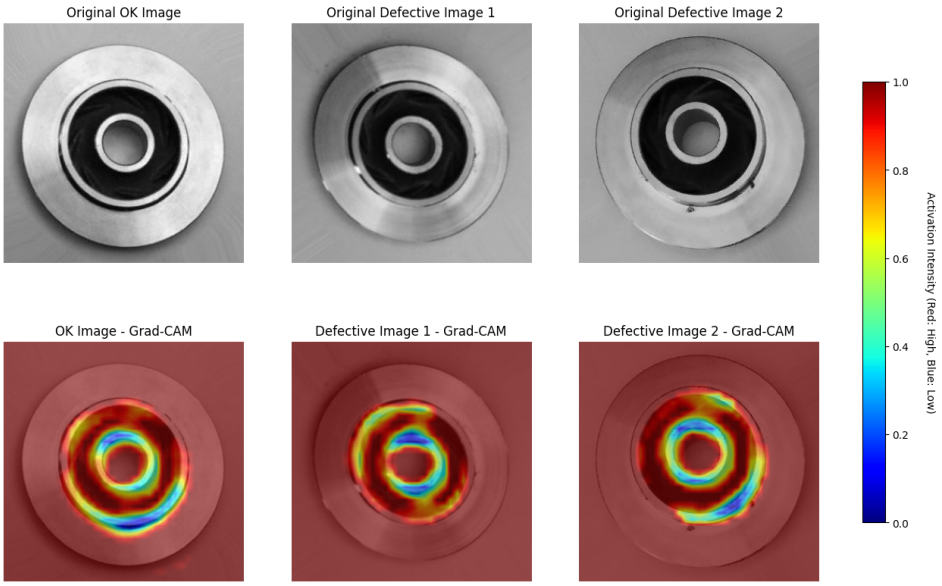
This means that the Neural Network could analize **up to 90,000 pcs/ hr!**

Visualization of how the Model “sees” my defects

Using Keras Model and Image modules, the outcome of the 3 layers of Convolutional Neural Networks was visualized to see how the model “captures” the defects:



Using Grad-Cam combined with Tensor flow, the pixel Activation Intensity was visualized. Red Areas are zones that are highly considered by the model, while blue areas are areas that the computer somehow “ignores”.

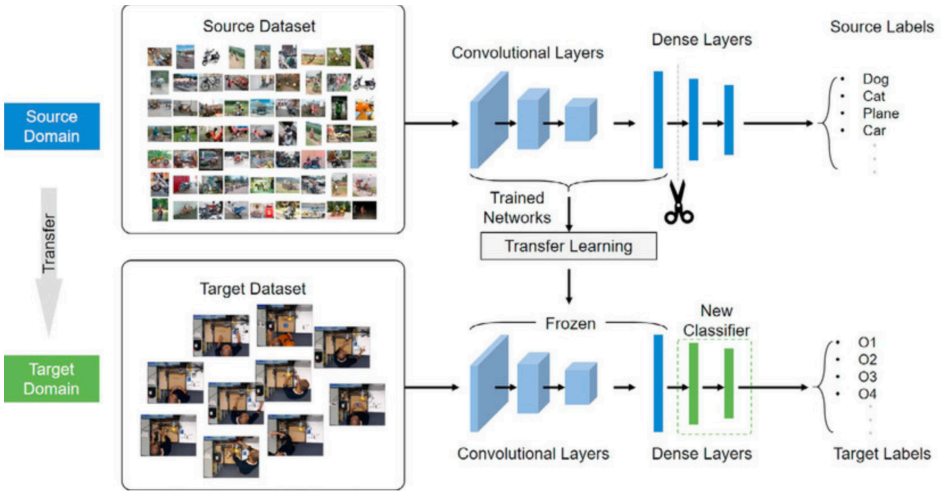


Springboard

Transfer Learning

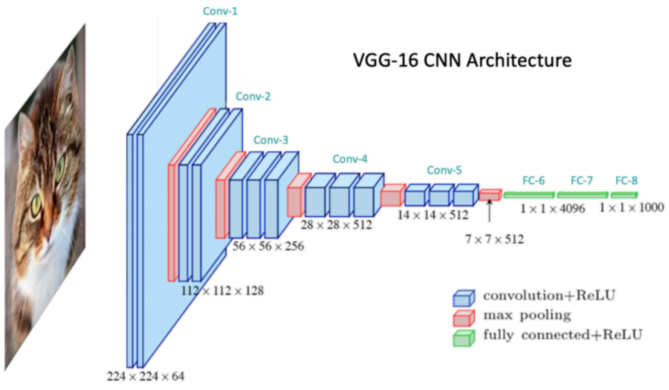
Transfer Learning is a technique in machine learning to take a model trained on a large dataset and adapt it to a new task. Instead of building a model from scratch, a pre-trained model that has already learned to recognize features like edges, textures, and patterns is used. Models like ResNet or VGG have already learned to detect general visual features that are important for recognizing objects in images.

This is a huge advantage because it cuts down on training time and reduces the amount of data needed for a project, especially when resources are limited.



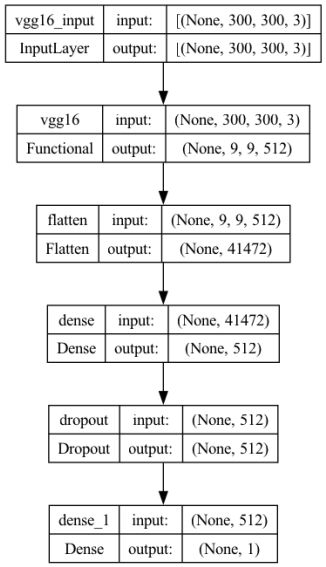
Keras VGG16 model re-trained for Industrial Metal Casting Pictures

VGG16 has been pre-trained on the ImageNet dataset, which consists of over 14 million images across 1,000 categories. It has a total of 16 layers, and in this model, the first 13 convolutional layers are used as a feature extractor. Additionally, some Custom dense and dropout layers were added on top of the model, for the binary selection (Defective or OK). This combination allows the model to focus on features relevant for decision, while preventing overfitting.



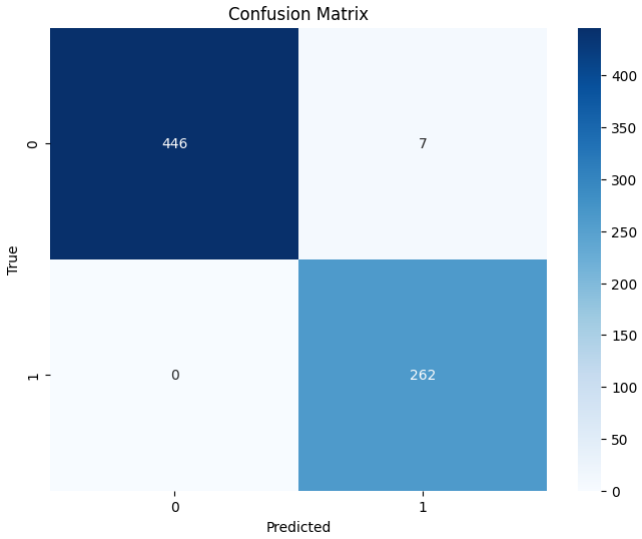
Model Architecture

As it can be observed the first layers are directly "trimmed" from VGG-16 CNN and the last layers are customized to learn the selection of my specific training pictures



Transfer Molding Results

The Transfer Learning model achieved a high accuracy (99%) since the 4th Epoch. Further modification of its layers wasn't necessary.



This confirms that the pre-trained model's lower-level feature extraction was well-suited for the task of identifying defects in metal parts, and only minimal fine-tuning was required.

Model Selection

While building a model from scratch was a good exercise to practice the concepts learned during the Data Science Certification, If I had to start this project again, I would select Transfer Learning method due to its simplicity for implementation.

Final Conclusions

The model trained from scratch delivered an impressive accuracy of 98%, demonstrating that it was able to learn and generalize the features necessary to classify the metal parts effectively.

Building the model from scratch allowed me to fully control its architecture and understand the key parameters needed for the task. By doing it this way, I could learn from my mistakes, because I started designing a very complex model (that overfitted) and ended up using a simpler model. At the end, this model successfully learned meaningful patterns from the data, making it a reliable solution for this type of classification task.

In conclusion, both the custom model trained from scratch and the Transfer Learning model performed exceptionally well, achieving accuracies of 98% and 99%, respectively. The minimal difference between the two demonstrates that the model trained from scratch was highly effective for this specific task. However, by using Transfer Learning, I was able to save significant time and computational resources, as the pre-trained model already possessed a strong understanding of general visual features.