QATAR UNIVERSITY

COLLEGE OF ENGINEERING

CORROSION DETECTION USING TRANSFER LEARNING-BASED MODELING FOR IMAGE CLASSIFICATION

BY

AHMAD HASAN BADER AQEL

A Thesis Submitted to

the Faculty of the College of Engineering

in Partial Fulfillment of the Requirements for the Degree of

Masters of Science in Mechanical Engineering

June 2019

COMMITTEE PAGE

The members of the Committee approve the Thesis of
Ahmad Aqel defended on 16/05/2019

Prof. Abdelmagid S. Hammuda
Thesis/Dissertation Supervisor

Dr. John-John Cabibihan
Program Coordinator

Dr. Hossam Kishawy
Examiner

Dr. Farayi Musharavati
Examiner

Approved:

Abdel Magid Hamouda, Dean, College of Engineering

ABSTRACT

AQEL, AHMAD H., Masters: June 2019, Masters of Science in Mechanical Engineering

Title: Training Image-based Neutral Network Models for the Detection of Corrosion Using Transfer Learning

Supervisor of Thesis: Prof. Abelmagid S. Hammuda.

This study uses image classification-based transfer learning to train models on the task of corrosion-detection on metallic surfaces. This is done by photographing images of samples of aluminium, iron and steel before and after corrosion to create visually differentiated datasets. With the exception of Model 1 which was trained and tested with a split of the original training set, the models were trained and tested on a newly prepared set to measure their accuracies fairly and realistically.

Model 1 was used to evaluate hypermeters, achieving an accuracy of 96.5%. Model 2, categorizing all images into corroded and uncorroded, scored an accuracy of 97.67%. Model 3, categorizing images into corroded, uncorroded and pitted, scored an accuracy of 95.67%. Model 4, trained to separate images into uncorroded aluminium, corroded aluminium, uncorroded steel/iron and corroded steel/iron, performed relatively poorly at 80%, but revealed that the majority of mislabeling is the result of combining the two materials in the sample model. Models 5 and 6 were trained on steel alone and aluminium alone, respectively. Model 5 scored an accuracy of 99.38%, while Model 6 scored a perfect 100%.

DEDICATION

To the mentors who have enlightened my path,

To the brothers-in arms who have shared with me this march,

And to those who my heart forever holds dearest,

I dedicate my work to you.

ACKNOWLEDGMENTS

To begin, I would like to kindly and wholeheartedly thank my mentor and supervisor, Prof. Abdelmagid, for his continued support and guidance throughout my journey as a Masters Student. I follow that with a kind thank you to all the faculty and staff members of the Department of Mechanical and Industrial Engineering in Qatar University. In particular I would like to thank Prof. Elsadig Mahdi for his enlightened guidance that helped shape this study into the shape in which I present it in here in this thesis.

I would also like to send the warmest of regards and sincerest of thanks to Dr. Elsadig El-Tai for introducing me to the world of corrosion, helping me realize the significance of my work, and guiding me on how to carry out this research. I would also like to kindly thank one of the dearest people to my heart in the university, Eng. Yehia Darwish, for his ever-present guidance and help every step of the way. Last but not least, I would like to send a very warm thank you to Mr. Sunith Padinjarayil for his continued support, assistance, and bearing with me as I learned more and more about corrosion and its workings.

TABLE OF CONTENTS

DEDICATION....................iv

ACKNOWLEDGMENTS....................v

LIST OF TABLES....................x

LIST OF FIGURES....................xi

Chapter 1: The Introduction....................1

1.1 Problem Statement....................1

1.2 Objective....................2

1.3 Thesis Overview....................3

1.3.1 Chapter 1: The Introduction....................3

1.3.2 Chapter 2: Literature Review....................3

1.3.3 Chapter 3: The Methodology....................3

1.3.4 Chapter 4: Model Training....................4

1.3.5 Chapter 5: Discussion....................4

1.3.6 Chapter 6: Conclusion & Future Recommendation....................4

Chapter 2: Literature Review....................5

2.1 Corrosion....................5

2.2 Robotic Inspection & Computer Vision....................7

2.3 Deep Learning & Transfer Learning....................13

2.4 Relevant Work..15

Chapter 3: The Medthodology..16

3.1 Sample Preparation 16

3.2 Image Acquisition 17

3.3 Model Training...18

3.4 Evaluative Metrics..23

Chapter 4: Model Training...24

4.1 Model 1: The Pilot Run..24

4.1.1 Objective...24

4.1.2 Dataset 24

4.1.3 Categorization..24

4.1.4 Results & Analysis 25

4.2 Model 2: Inception 30

4.2.1 Objective...30

4.2.2 Dataset 30

4.2.3 Categorization..30

4.2.4 Results & Analysis 31

4.3 Model 3: Pitted vs Corroded 33

4.3.1 Objective...33

4.3.2 Dataset 33

4.3.3 Categorization.....33

4.3.4 Results & Analysis.....34

4.4 Model 4: Four Classes.....38

4.4.1 Objective.....38

4.4.2 Dataset.....38

4.4.3 Categorization:.....38

4.4.4 Results & Analysis.....39

4.5 Model 5: Model of Steel.....41

4.5.1 Objective.....41

4.5.2 Datasets.....41

4.5.3 Categorization.....41

4.5.4 Results & Analysis.....42

4.6 Model 6: The Aluminium Model.....44

4.6.1 Objective.....44

4.6.2 Datasets.....44

4.6.3 Categorization.....44

4.6.4 Results & Analysis.....45

Chapter 5: The Discussion.....46

5.1 The Model’s Learning.....46

5.2 Result Evaluation.....54

5.2.1 Model: The Pilot Run 54

5.2.2 Model 2: Inception..55

5.2.3 Model 3: Pitted v Corroded 55

5.2.4 Model 4: Four Classes 56

5.2.5 Model 5: Model of Steel 56

5.2.6 Model 6: The Aluminium Model...57

5.3 The Consensus...58

Chapter 6: Conclusion & Future Recommendation.....................................59

6.1 Study Summary...59

6.2 Further Testing & Implementation...62

6.3 Recommendations 63

6.3.1 New Models...63

6.3.2 Different Architecture..63

6.3.3 The Many Forms of Corrosion 63

6.3.4 Staged Model 64

References..65

Appendix A 69

LIST OF TABLES

Table 1: Model 1 Set Split 25

Table 2: Computational Results of Model 1 25

Table 3: Model 2 Set Split 31

Table 4: Model 3 Set Split 34

Table 5: Model 4 Set Split 39

Table 6: Mislabeling Crossover Matrix 40

Table 7: Model 5 Set Split 42

Table 8: Model 6 Set Split 44

Table 9: Statitiscal Measures of Binary Models 58

Table 10: Models Summary 60

LIST OF FIGURES

Figure 1: The components of robotized inspection system design [15] 7

Figure 2: Inspection system algorithm [16]...8

Figure 3: In-pipe inspection robot classification [17]...9

Figure 4: 3D design of mobility system [17]..10

Figure 5: Smart-Spider design and prototype [18] 10

Figure 6: Visual inspection flow [22] 11

Figure 7: Classical programming versus machine learning [29] 13

Figure 8: Convolutional neural network layer [30] 14

Figure 9: Corrosion-classification model output [32]...15

Figure 10: Content of online repository...19

Figure 11: Dataset folder example...20

Figure 12: Evaluative output of model 22

Figure 13: Image 5425, mislabeled by all runs...26

Figure 14: (a) Image 6094 (b) Image 6096...27

Figure 15: Avg. computing time vs learning rate plot...28

Figure 16: Image 6247...35

Figure 17: Images (a) 6571 (b) 6572 (c) 6577 (d) 6598 37

Figure 18: Image 6536, the only mislabeling instance of Model 5 42

Figure 19: Sample set of uncorroded aluminium..48

Figure 20: Sample set of uncorroded steel..51

Figure 21: Sample set of corroded steel..52

Figure 22: Sample set of corroded aluminium..53

Figure 23: Three stages of steel (no corrosion, mid-corrosion, late corrosion)...........64

CHAPTER 1: THE INTRODUCTION

1.1 Problem Statement

Despite substantial research and preventive measures, material corrosion remains a critical industrial problem that has taken its toll on human life[1] and the economy, resulting in an expenditure of over $2.5 billion dollars annually on the global scale.[2] A significant method of detecting corrosion is visual inspection, which, when done by humans is costly, time-consuming and exposes said humans to potential life-threatening situations.[3] The cost and physical limitations of human inspection in general has given a rise to the presence of robotic inspection systems, which tend to be more cost-efficient and can be faster in their processing and inspection.[4]

While a majority of inspection and vision-based systems have been reliant on task-specific commands with limited accuracy, the rise of deep learning and convolutional neural networks (CNNs) have allowed for higher accuracies and more efficient modelling for image recognition applications.[5] While deep learning requires very large datasets to be trained effectively, existing modules can be re-trained on a smaller dataset for a similar task to achieve comparable accuracies. This process is dubbed as transfer learning.[6] This raises the question of how well can a transfer learning image-recognition model perform on a corrosion-detection task.

1.2 Objective

The objective of this study is to use Google’s TensorFlow for Poets Image Classification module to re-train the pre-trained Inception V3 architecture on surface images of corroded and uncorroded samples of aluminium, mild steel and iron, creating models that are capable of binary and multiple classifications between corroded and uncorroded element. This will, in turn, train neural network models that are capable of detection corrosion in surface images, a desirable complimentary element to the growing presence of automated inspection robotics systems. A solid representation of said objectives is the following:

- Use transfer learning to train models on images of a lab-prepared set of samples with the target of classifying images based on their surface condition, whether it may be corroded or not.
- Test the created models to fairly analyze their potential in performing the given task of detecting corrosion on surface images.
- Analyze the results of the testing and the output of the study, before ending with a conclusion and appropriate recommendation based on said results.

1.3 Thesis Overview

1.3.1 Chapter 1: The Introduction

The inaugural chapter has introduced the problem statement which has motivated this study, before moving on to declaring its objective. The final part of the first chapter is the current one, giving an overview of the chapters of the thesis and their content.

1.3.2 Chapter 2: Literature Review

This chapter introduces a brief background on the three elements of the study: corrosion, robotic inspection and particularly vision-based inspection, and then deep learning and neutral networks, before moving to a brief survey of work that is directly relevant to the study.

1.3.3 Chapter 3: The Methodology

The chapter details the methodology steps used to create the datasets, train the models and evaluate the results. It goes into the experimental preparation of the samples, the acquisition of the images that were used in the training and test sets. The chapter follows through with a detailed recounting of how the TensorFlow for Poets/Inception V3 module was used in the training of the model. Chapter 3 ends with a small demonstration of the evaluative measures utilized in the study.

1.3.4 Chapter 4: Model Training

The fourth chapter details the objectives of each model trained, the sets used for the training, the form of categorization utilized among the images of the datasets, and the results of the testing of the trained model. This is usually accompanied with an account of what the model might have mislabeled and a brief analysis of said results.

1.3.5 Chapter 5: Discussion

This chapter begins with a discussion of the learning of the model, and the significance behind the images utilized in the dataset. It then moves on to an analysis of the output of each model from Chapter 4, before arriving to a consensus on what the output of the study entails.

1.3.6 Chapter 6: Conclusion & Future Recommendation

The final chapter of the thesis summarizes the entire body of work, before entailing the necessity of further testing and potential implementation of the models created in this study. The final chapter concludes with a section on some future recommendations for alternatives in training and modelling.

CHAPTER 2: LITERATURE REVIEW

2.1 Corrosion

For the purposes of this body of work, Corrosion, an often widely used term for a multitude of physical phenomena with a wide array of definitions, will be defined as per Joseph R. Davis' Corrosion: Understanding the Basics; Corrosion is a chemical, or possibly electrochemical, reaction between a material, usually a metallic one, and its immediate environment, which results in a physical deterioration of the material and its properties. The scientific reasoning behind this occurrence is mainly due to a lack of stability of corrodible elements; metals, for an instance, have an inherent tendency to oxidize in order to revert back to a state where they resemble the minerals that they were first extracted from – it is from that particular behaviorism that corrosion has been referred to as metallurgy in reverse.[7] The classification of the various forms of corrosion is said to be dependent on three primary factors:

- The nature of the corrosive element, whether it is wet or dry.
- The mechanism of the corrosion, whether it is electrochemical or purely chemical in nature.
- The visual appearance of the corroded metal, whether it is uniformly occurring throughout the object or localized in certain areas.

The detection and classification of the form of corroded metal is commonly based on a visual inspection[3], whether through magnification or by the naked eye. Common forms of aqueous-based corrosion include:

- Uniform Corrosion[8]: most commonly occurring type, results from presence of material in corrosive medium.
- Pitting Corrosion[9]: similar in causative nature to uniform corrosion but differs in the presence of "pits" in the corroded surface.
- Crevice Corrosion[10]: physically similar to uniform, occurring specifically in crevices and similar structural forms.
- Galvanic Corrosion[11]: corrosion occurring as a result of the contact between two different metals.

Preventive measures against corrosion include applying cathodic protection[12], the use of coatings[13] and inhibitors[14]. The diagnosis of corrosion includes visual inspection of both the macro- and microscopic variety, chemical analysis, mechanical testing, and non-destructive testing techniques.[7] The importance of preventing corrosion failure comes in its risk to both the environment and human life, as a number of corrosion-related catastrophes have led to the loss of human life.[1] The economic impact of corrosion has also been humongous, evaluated by NACE at $2.5 trillion annually, equating to roughly 3.4% of the global GDP.[2] This led to the topic of corrosion, its inspection, prediction and prevention to being a highly researchable one, with several journals being completely dedicated to this area of research.

2.2 Robotic Inspection & Computer Vision

The utility of inspection robots for industrial purposes has seen a growth since the mid-1990's perpetuated by the development of new sensors operating on better performance and lower costs. This was perpetuated back in 1994 through a study conducted by Lund University where the objective was to build on the existing rise of industrial robots by developing automated inspection and quality control through in line with said rise. Figure 1 provides an overview of the components considered in the design of a robotized inspection system.[15]

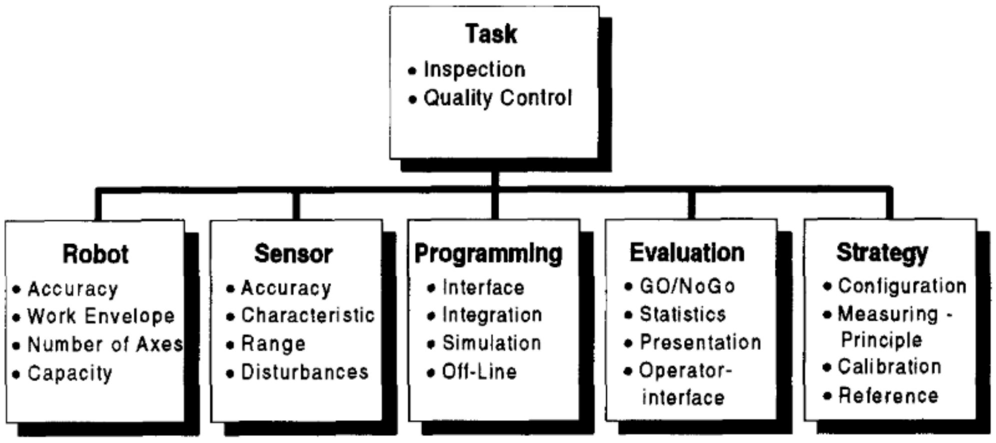


Figure 1: The components of robotized inspection system design [15]

A primary area where inspection robots has seen heavy research output is pipeline monitoring, with robots being used to inspect pipeline conditions using a combination of sensors and visual inspection. An early research into this field was conducted by a team of Egyptian researchers back in 2004, citing the lack of consistency in human evaluators and the reduction of inspection costs coupled with the improvement in inspection quality as the primary motives behind the shift toward utilizing automated inspection for detecting gas pipelines welding defects. The output is an image processing-back inspection system capable of expertly identifying the main major welding defects associated with gas pipelines welded by shielded metal arc welding. Figure 2 is a black diagram demonstrating the algorithm which was utilized in the inspection system.[16]

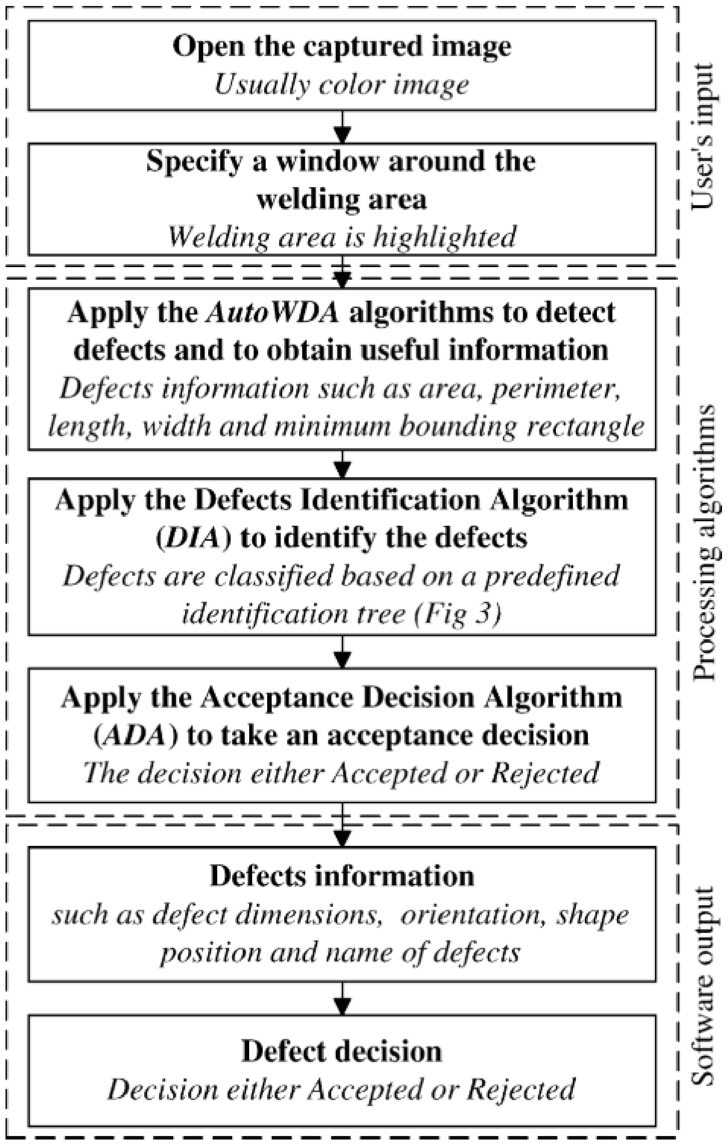

Open the captured image
Usually color image
Specify a window around the welding area
Welding area is highlighted
User's input
Apply the AutoWDA algorithms to detect defects and to obtain useful information
Defects information such as area, perimeter, length, width and minimum bounding rectangle
Apply the Defects Identification Algorithm (DIA) to identify the defects
Defects are classified based on a predefined identification tree (Fig 3)
Apply the Acceptance Decision Algorithm (ADA) to take an acceptance decision
The decision either Accepted or Rejected
Processing algorithms
Defects information
such as defect dimensions, orientation, shape position and name of defects
Defect decision
Decision either Accepted or Rejected
Software output

Figure 2: Inspection system algorithm [16]

Another paper published in the 12th Global Congress on Manufacturing and Management back in 2014 emphasized a lack of human accessibility and the dangers and hazards of toxic chemicals associated with pipelines as the driving force behind the increasing shift toward automated robotics inspection. The study proposed a design specifically for use inside the pipeline, flowing along and inspecting the inner walls for the pipeline. A robot of this form required an emphasis on maneuverability and mobility. Figure 3 provides a breakdown of the varying types of In-Pipe Inspection Robots considered for this study, with the proposed design by the paper displayed in Figure 4.[17]

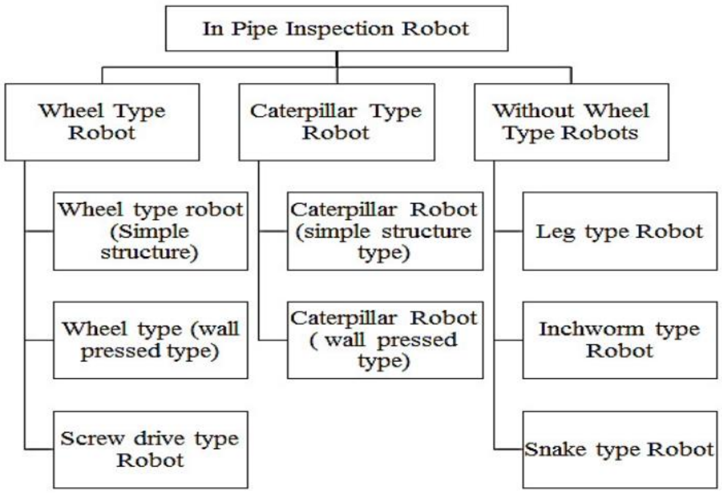

In Pipe Inspection Robot
Wheel Type Robot
Caterpillar Type Robot
Without Wheel Type Robots
Wheel type robot (Simple structure)
Wheel type (wall pressed type)
Screw drive type Robot
Caterpillar Robot (simple structure type)
Caterpillar Robot (wall pressed type)
Leg type Robot
Inchworm type Robot
Snake type Robot

Figure 3: In-pipe inspection robot classification [17]

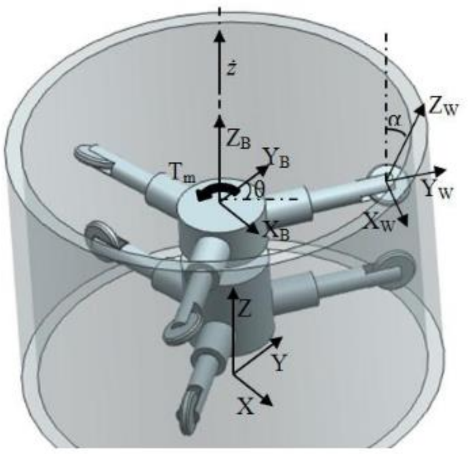


Figure 4: 3D design of mobility system [17]

Another advancement in the realm of In-Line Inspection Robots came in a 2018 paper which proposed a design titled as Smart-Spider; an autonomous robot that utilizes a set of "legs" with wheels to maneuver through the pipeline, collecting data as it moves along. The structure and prototype of the Smart-Spider are seen in Figure 5.[18] Other similar recent studies have involved the development of pipeline robotic systems of various designs and mobility mechanisms.[19,20]

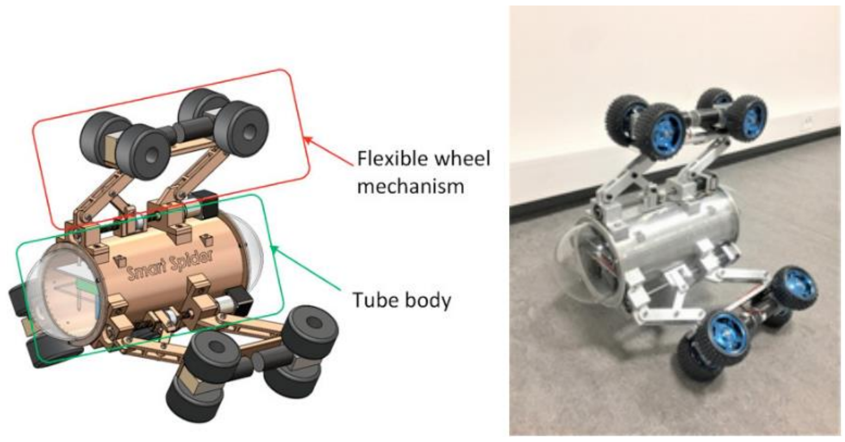


Figure 5: Smart-Spider design and prototype [18]

The oil and gas industry in general is one that sees a persistent use of autonomous inspection robots ranging from the aforementioned in-pipe inspection robots, tank inspection robots and unmanned air vehicles.[4] Inspection robots, or sometimes referred to as Inspection Robotic Systems (IRS), have had industrial applications beyond the oil and gas industry, mainly concerning their utility in inspecting large-scale constructs like plants, bridges or roads where human inspection could expose individuals to unfavorable conditions of operation. [21]

The full capacity in inspection robots is in their combined ability with visual-based inspection. The capability of robots as the basis of an automated visual inspection (AVI) system holds the promise of enhancing the automation capability of production systems through detecting faults. Figure 6 provides a block diagram description of a traditional machine vision system utilized with inspection robots.[22]

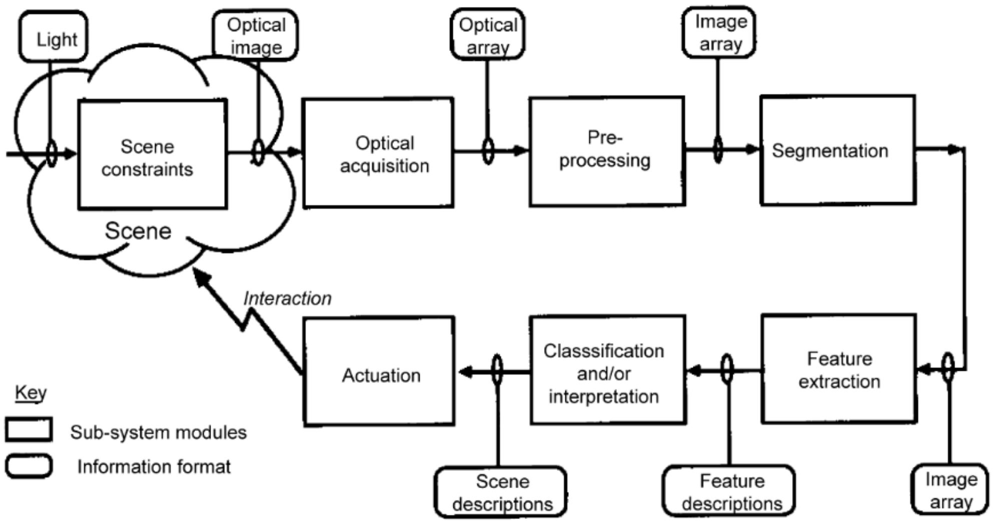


Figure 6: Visual inspection flow [22]

Vision-based robotic inspection systems have also been proposed as a monitoring system for bridges, replacing manual inspection by using image processing to evaluate the cracking present in the bridge's structures.[23] Vision-based inspection was used in bearing defect inspection, achieving an F-measure of 98%.[24] Similar models were created for grading dates, achieving an accuracy of 80% [25] and two models were created for crack-detection in eggs, achieving accuracies of 94% [26] and 100% [27] Automatic detection of cracks using images from pavements is another studied utility of vision-based inspection systems.[28]

2.3 Deep Learning & Transfer Learning

Machine learning, in its supervised capacity, is based on providing a dataset of "labeled" examples. This allows the statistical training of a model that can be used for predictive or classification tasks by allowing it to "learn" from the labeled dataset. Figure 7 demonstrates a clear schematic for how machine learning differs from the more classical programming methods utilized in the majority of image processing research. Deep learning, a variant of the machine learning, uses the same principle, but differs in its method of training. While machine learning relies on statistical tools such as regression and nearest neighbor, deep learning focuses on utilizing neural networks of increasingly meaningful layers over large datasets to create highly accurate models, has seen a significant and wide-reaching growth, from healthcare to finance to image recognition.[29]

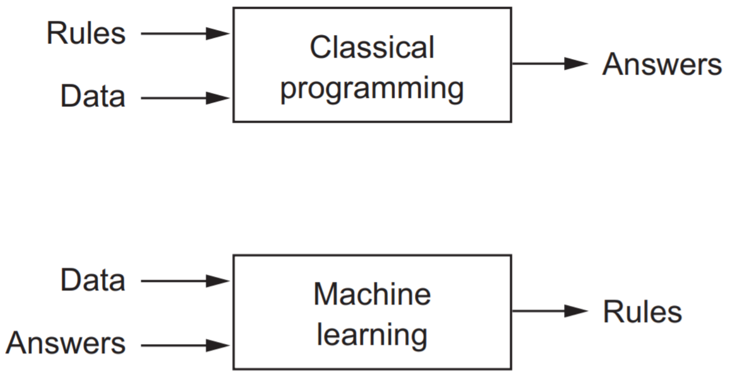


Figure 7: Classical programming versus machine learning [29]

Image recognition tasks have primarily utilized the convolutional neural network (CNN) variant of neural networks as the primary tool for training, finding utility in a variety of image detection applications such as self-driving cars and facial recognition. The process of convolutional network training involves feature extraction, where each neuron takes synaptic inputs from a local receptive field that is present in the previous layer of the network. Feature mapping is then used to constrain the individual neurons to share the same set of synaptic weights. In Figure 8, a basic layout of a convolutional layer is displayed.[30]

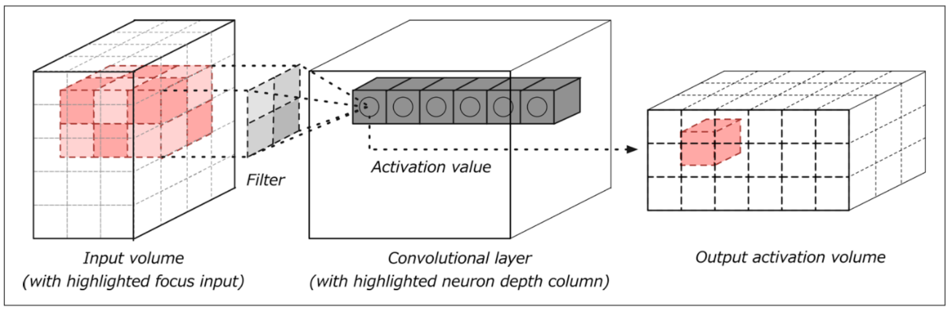


Figure 8: Convolutional neural network layer [30]

The training of a CNN is designed to be utilized for large sets of data, which limited its capability for smaller datasets. This was remedied by the emergence of transfer learning, the process of re-training an already trained CNN model on a new smaller dataset for a similar task, after research has shown its promising capability for visual recognition and image classification[6] and thus became the basis for this study on visual recognition of corrosion.

2.4 Relevant Work

While the scope of this study has never been attempted in the given capacity, several studies have been conducted on the utility of CNNs in general in corrosion detection. The closest to the study in scope was an evaluative study that was carried out on several different CNN architectures on images rated at pixel values of up to 128x128, achieving a maximum F-score of roughly 0.98 in their best-performing model. [31]

A similar studied analyzed the utility of CNNs for detecting damage types in images, achieving scores as impressive as 98% including corrosion.[32] A common theme with the last two studies is attempting to divide images into pixels and then producing images where corroded pixels are highlighted, rather than simply categorize the images. A demonstration of an output of the former study can be seen in Figure 9.

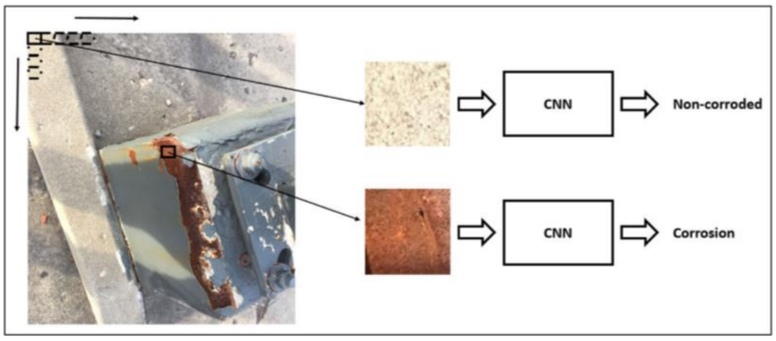


Figure 9: Corrosion-classification model output [32]

A study attempted to compared traditional computer vision programming with CNNs based on the ability to categorize images based on the presence of corrosion, where the former stalled at accuracies of roughly 69% and the latter achieved accuracies as high as 88%. [33] This leaves room for the evaluation of how transfer learning techniques may compare on this same task.

CHAPTER 3: THE MEDTHODOLOGY

3.1 Sample Preparation

Sample preparation, like several aspects of the methodology, involves an essence of randomness and unplanned variability in the preparation of the samples. This is to allow increased variety in the samples created, allowing for a minimal approach to the obtainment of an inclusive and easy-to-train-and-generalize dataset.

First, sheets of aluminium, mild steel and iron of varying thickness were cut into samples of roughly 50x50mm to make the samples easier to submerge in acid in the next step of the process. Most of the aluminium and iron samples were relatively thin, at a thickness of roughly less than 1mm, while steel samples where thicker, at approximately 5mm thickness. All choices thus far, including which materials to use, was largely based on what was available during the time of the study.

The "uncorroded" images are photographed. This is how the uncorroded parts of all datasets are created. The sample is then taken and submerged for varying times in hydrochloric acid (for aluminium and steel) and nitric acid (for iron). Nitric acid was used in a concentration range of 20-40%, while hydrochloric acid baths were prepared in ranges of 20-35%. After the samples are left in their respective acid baths for the desired period, they are taken out, wiped and left to sit and dry. This is how all the samples used in the preparation of the datasets used in the study were created.

3.2 Image Acquisition

All images used for the creation of the datasets used in this study were captured using the 12-megapixel rear camera of a personal Apple's iPhone 8, which possesses built-in optical image stabilization and a 5x digital camera zoom, which is handy in the creation of zoomed images on the surfaces of the samples.

The photographing of the samples utilized a variety of angles, zooming level and orientation of the sample. This is to ensure a sufficient variety in the datasets created which is critical to training an accurate model. Blurry images were also allowed as part of the set. A common element used in all images was that they were only images of the surface of the sample, never including any exterior background, as to be in-line with the objective of the study.

Another important element to vary was the lighting condition when capturing the images. The following conditions were utilized throughout the photographing of the samples:

- Normal indoor with lighting.
- Normal indoor during the day without lighting.
- Dark indoors with camera flash.
- Outdoors during sunny weather.
- Outdoors during cloudy weather.

A total of 1005 images were captured and used as part of the training model, while 300 images were captured for the test set.

3.3 Model Training

Operating on the principle of transfer learning, the models for this study are built using Google's TensorFlow for Poets. TensorFlow for Poets utilized models, or more commonly referred to as modules, that are pre-trained on datasets. The module used is that of the image feature extraction module with the Inception V3 architecture being trained on the ImageNet dataset, which includes over a million images.

The model divides images into squared pixels, at a dimension of 224x224 for this study where each pixel represents a given value acting as the feature value on which the network is trained, and starts training by building "bottlenecks", layers created right before the final output layer in the neural network, aimed to center the learning of the old model to fit the examples of the new model. A bottleneck is created for every single image in the training set.

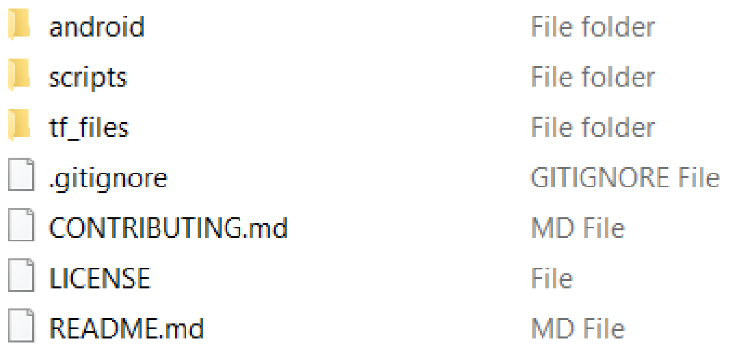
After that, the actual training of the top layer of the neural network proceeds. The way this system works is through training steps: In every step, ten images are chosen randomly to be fed to the final layer and get a labeling; this labeling is compared to the actual labeling in order to update the weights of the final layer of the network. This process is carried out repeatedly, showing a drastic improve in the model's self-tested accuracy, reaching levels of 95-100% for the models trained in this study.[34]

The commands actual repository of TensorFlow for Poets used in this study are ones that were uploaded on the GitHub of one Ritesh Kumar Maurya[35], who included ample instructions on how to build image classification models using this method, primarily as a part of his YouTube course on the matter. The following is a thorough explanation of how to build a model using the TensorFlow for Poets method.

After installing an appropriate version of Python 3 on a Windows System, the TensorFlow library is installed, which was developed by Google to be a go-to for data scientists and machine learning engineers in their model building and testing. The library is installed via the Command Prompt using the command:

pip install tensorflow

After having had installed the library, the next step is to download the repository, or source code, which was obtained from Maurya's GitHub. Figure 10 shows the content of the downloaded repository. According to Maurya, the android folder is for creating the Mobile and Lite version of the models, which is not investigated in this study. The scripts folder is the critical element, harboring the code for the training of the model (with detailed code in Appendix A), also including the critical label_image.py, which is the function used to determine the labeling of an image during testing. The tf_files folder is an originally empty folder that is later used to store all the bottlenecks and the model created. It is also where the model's builder chose to store the training dataset, normally labeled for this study as simply "dataset".



android	File folder
scripts	File folder
tf_files	File folder
.gitignore	GITIGNORE File
CONTRIBUTING.md	MD File
LICENSE	File
README.md	MD File

Figure 10: Content of online repository

Before training the model, it is essential to arrange the dataset required, with different categories placed in separate folders named after the category of the dataset. For example, for one of the runs of Model 1, the dataset folder can be seen in Figure 11.

corroded	3/25/2019 1:54 PM	File folder
uncorroded	3/25/2019 1:54 PM	File folder

Figure 11: Dataset folder example

The contents of the originally obtained TensorFlow for Poets repository is copied into the folder of the model to be trained. Using the command prompt, one must first go to the directory of the model’s folder before the training can be initiated. The training process described earlier can then be started using the following code:

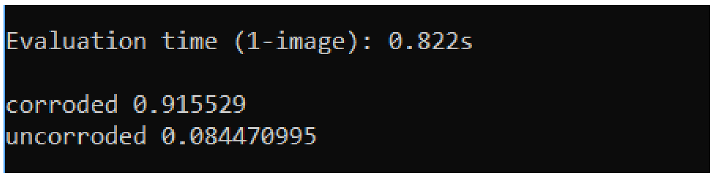
```
1        python -m scripts.retrain \
2        --bottleneck_dir=tf_files/bottlenecks \
3        --model_dir=tf_files/models \
4        --summaries_dir=tf_files/training_summaries \
5        --output_graph=tf_files/retrained_graph.pb \
6        --how_many_training_steps=4000 \
7        --output_labels=tf_files/retrained_labels.txt \
8        --architecture=inception_v3 \
9        --image_dir=tf_files/dataset
```

First line indicates that this code to is to be carried out by python, before calling the retrain function. The second to fifth lines, along with the seventh, create the necessary files of the trained model and stores them in the tf_files folder. These are updated throughout the training and into the newly trained model. Line six inputs the number of learning steps, which is 4000 by default. Line eight inputs the architecture, which is the defaulted Inception V3 architecture for this study. The final lines choose the directory of the folder with the training set. These commands are entered as a single line in the Command Prompt. The model training would start, and take approximately an hour, depending on the number of learning steps and the size of the training set.

After the model is trained, the images can be tested using this command:

```
python -m scripts.label_image
--graph=tf_files/retrained_graph.pb
--image=[Your Image HERE]
```

In this code, the first line used python to call the label_image.py function, with the graph used for labeling is the one trained and stored in the tf_files folder in the training process. The last line is where one can input the directory of their tested image, from within the directory of the model's folder. The output once an image is tested, as seen in Figure 12, is the computing time along with a certainty split, giving an evaluative measure of how certain the model is that this image is corroded (91.55%) or uncorroded (8.45%).



```
Evaluation time (1-image): 0.822s

corroded 0.915529
uncorroded 0.084470995
```

Figure 12: Evaluative output of model

3.4 Evaluative Metrics

In the process of analyzing the models, certain metrics will be taken into consideration. For all models, an accuracy will be computed using the following equation:

$$Model\ Accuracy = \frac{Test\ Set - Mislabeled\ Images}{Test\ Set} x100\%$$

Additional statistical tools will be used in discussing the performance of binary models (corroded vs uncorroded) in Chapter 5. These are the recall, precision and F-measure. They will be computed using the following formulae:

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

$$F - measure = 2\ x\ \frac{Precision\ x\ Recall}{Precision + Recall}$$

where for the purposes of this study:

True Positive = corroded images correctly labeled as such.

False Positive = uncorroded images falsely labeled as corroded.

False Negative = corroded images falsely labeled as uncorroded.

CHAPTER 4: MODEL TRAINING

4.1 Model 1: The Pilot Run

4.1.1 Objective

Model 1 is trained as a test run to. The model is also intended to test the variation of classification accuracy, trained model data size and average classification time with different learning rates.

4.1.2 Dataset

This model was trained before the preparation of the test set. Instead, a sample of 200 images from the entire training set of 1005 images were chosen as the test set, with the remainder of the 805 used to train the model.

4.1.3 Categorization

Model 1 was trained to classify images on two categories: Uncorroded, which includes images of uncorroded surfaces from steel, iron and aluminium, and Corroded, which includes images of corroded surfaces from steel, iron and aluminium. Across the training and test sets, the two categories were split as represented in Table 1.

Table 1: Model 1 Set Split

	Uncorroded	**Corroded**
Training Set	280	525
Test Set	60	140

4.1.4 Results & Analysis

This model was run at four different learning rates: 1000, 2000, 4000 & 8000. The accuracy of the model, the average computing time and data size of the four runs are documented in Table 2.

Table 2: Computational Results of Model 1

	Test Set Accuracy (%)	**Avg. Computing Time (s)**	**Model Data Size (MB)**
1000	96.00%	0.8375	479
2000	96.50%	0.8539	485
4000	96.50%	1.0421	497
8000	96.50%	0.8094	522

Dissecting the model’s results, the test set accuracy seemed to be the least variant across all runs. A total of 7 images were miscategorized by all four models. Figure 13 displays Image 5425, which is an example of one image that was mislabeled by all models as corroded, despite being a part of the Uncorroded test set.

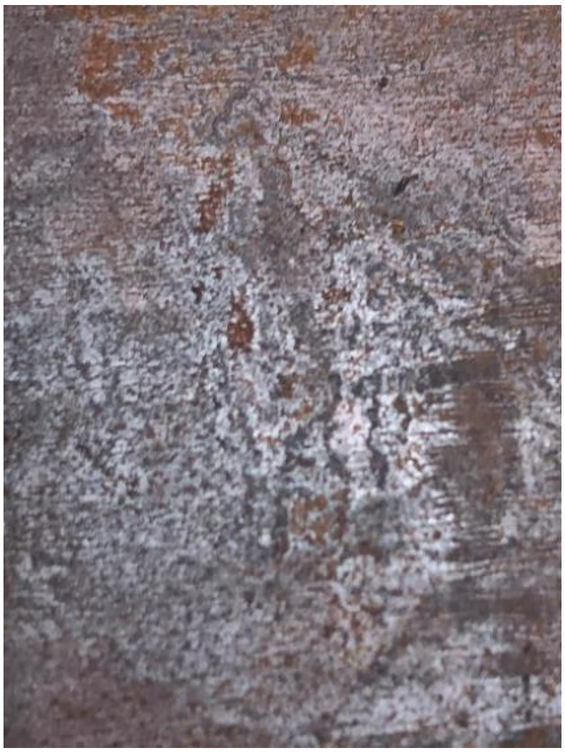


Figure 13: Image 5425, mislabeled by all runs.

This raises the significant question of whether the model is inherently flawed in labeling 5425 as corroded, when it clearly is a visual representation that is similar to corroded samples. While it is natural for any neural network to mislabel examples in the dataset, and a score of over 96% for all model runs demonstrates a solid statistical significance ($\alpha < 0.05$), it remains a recurring theme in this study that a trained model has mislabeled images that even an expert human eye could have mislabeled. The issue of confusing images in the datasets, and the possibility of mislabeling is discussed further in the next chapter but seemed pertinent to make note of at this stage.

The image that was mislabeled at the 1000 run but labeled correctly otherwise is Image 6094, displayed in Figure 14a. In truth, all models yielded a certainty of roughly 50-50 when attempting to classify this image, with the 1000 run leaning slightly toward mislabeling it as corroded despite being part of the Uncorroded test set, and the rest of the runs labeling it appropriately. Figure 14b displays Image 6096, of the same surface as 6094, but was somehow correctly labeled as uncorroded by all models, raising the question of what the model may or may not be learning through its new-found re-trained neural connections.

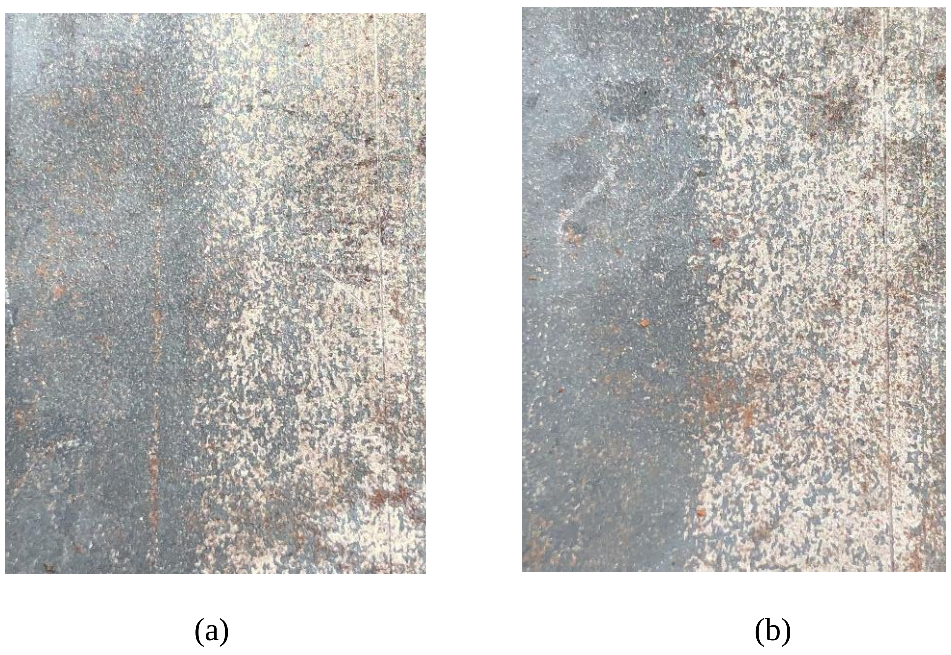


Figure 14: (a) Image 6094 (b) Image 6096.

It is also observed through testing of the model that all correct labeling was done at higher certainty on models built through a larger learning rate. This seems to lean the odds toward the 8000-rate to be the optimal, since accuracy and certainty are both the most vital aspects of the model.

Another vital aspect that was considered was the average computing time, which was sampled from the time taken to classify a given set of images. The variation of average computing time with learning rate is displayed in the graph in Figure 15.

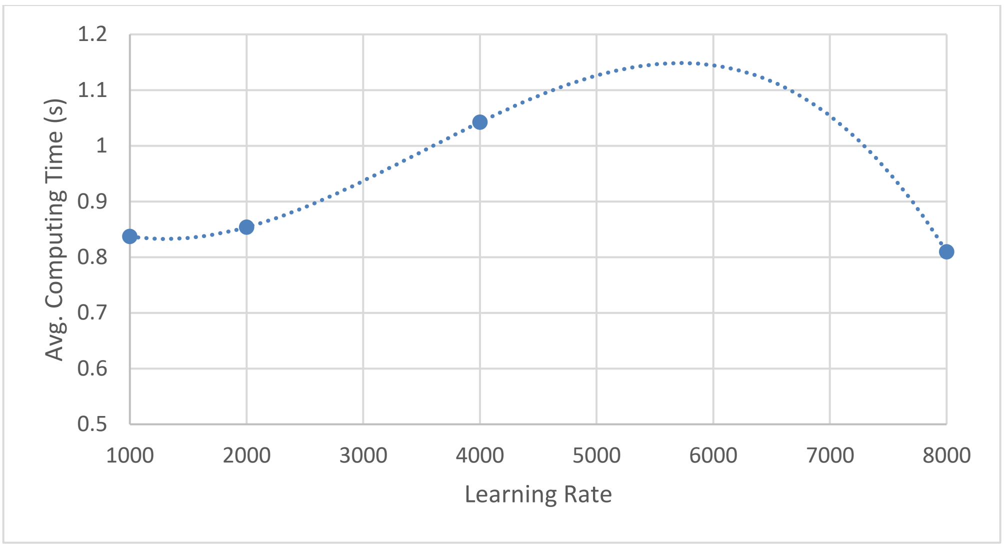


Figure 15: Avg. computing time vs learning rate plot

This particular metric does again show a high favorability towards the 8000 run, which seemed to yield the fastest computing time on average. This metric does need to be taken with a grain-of-salt nonetheless, as speed can vary with the processing speed of the computer, which in itself can vary for a number of reasons. It is nonetheless probable that the observed faster computing time is due to the higher certainty that was observed for images tested in the 8000 run, as the higher number of iterations of training could have yielded stronger connections and a smoother classification process.

The last metric in consideration is that of data size, which seems to show only small variations across all runs, with the 8000 run being only 8.98% larger than the 1000 run in terms of model data size. Since the only drawback to the 8000 run is it need the longest training time, and since training time is not an issue since a model needs to be trained only once and can be pre-trained before deploying onto an inspection system, it can be concluded that a learning rate of 8000 is optimal and will be utilized for training the remaining models.

While Model 1 achieved an impressive accuracy of 96.5%, it should be noted that the same samples were used to generate both the training and test sets of this model, giving rise to the likelihood that an image or pattern is repeated in both sets, which normally would give an unfair advantage to the model and inflate its accuracy. This is why a test set was created through the preparation of new samples and generating never-seen before images, which are used to test the remaining models, resulting in more credible accuracy findings.

4.2 Model 2: Inception

4.2.1 Objective

This model is set to be the first model to be trained by the entirety of the available training set, before testing it via the never-seen-before test set, resulting in a more credible analysis of the model, and a higher significance of the accuracy obtained through it.

4.2.2 Dataset

The entire training set of 1005 images was used for the training of this model, before testing it on the complete test set of 300 images to obtain the model's accuracy.

4.2.3 Categorization

Model 2 followed suit to Model 1, categorizing the images into two categories: Uncorroded, which includes images of uncorroded surfaces from steel, iron and aluminium, and Corroded, which includes images of corroded surfaces from steel, iron and aluminium. The image datasets were split as represented in Table 3.

Table 3: Model 2 Set Split

	Uncorroded	**Corroded**
Training Set	340	665
Test Set	100	200

4.2.4 Results & Analysis

This model achieved an astounding accuracy of 97.67%, correctly labelling 293 of the 300 images in the dataset. This achieved a significant statistical accuracy ($\alpha<0.05$). Of the 7 miscategorized images, 5 were corroded images mislabeled as uncorroded, while 2 were uncorroded images mislabeled as corroded.

It was observed that all mislabeled images were aluminium rather than steel, which does raise the question of what lead to the error in categorization; whether it was due to appearing visually similar to steel or was the model unable to even distinguish between uncorroded and corroded aluminium in those five mislabeled instances. Regardless, it should be noted that a total only these five were mislabeled, while 95 other images of corroded aluminium surfaces were correctly labeled as corroded, and all 100 images of corroded steel were correctly labeled, likely due to the higher observability of the visual properties of corroded steel.

Of the 100 images in the uncorroded test set, two were mislabeled as corroded, both from steel samples. This can be attributed to a significant factor that is likely to be persistent throughout the study: the steel samples used in the study clearly had visible traces of rusting on its surface, or other forms of surface damage that could have resulted in the mislabeling.

Regardless of these instances of mislabeling, an accuracy of over 97% is thoroughly impressive and likely be highly functional if applied in the real world. This model was originally the primary objective of this study, before the curiosity of the limitation and powers of this form of modeling lead to the continued exploration that is present in the remaining four models.

4.3 Model 3: Pitted vs Corroded

4.3.1 Objective

This model tests a multiple classification model for the purposes of this study. This will be through further classification of corroded image datasets into two categories based on whether the corrosion is of an aluminium sample or a steel/iron sample, resulting in a total of three categories. This is beneficial in exploring a model's capability in identifying the different forms of corrosion.

4.3.2 Dataset

The entirety of the training set of 1005 images was used for the training, before testing it on the complete test set of 300 images to obtain the model's accuracy.

4.3.3 Categorization

The model uses a new categorization compared to the previous two models: the first category, Uncorroded, is the same, including images of uncorroded surfaces from steel, iron and aluminium. On the other hand, we have a new second and third categories, the second being the category Corroded, which now is exclusive to images corroded surfaces of corroded surfaces from iron and steel only. The images of corroded surfaces from aluminium samples form a new category: Pitted. Table 4 demonstrates the split of the image datasets across the three categories.

Table 4: Model 3 Set Split

	Uncorroded	Corroded	Pitted
Training Set	340	447	218
Test Set	100	100	100

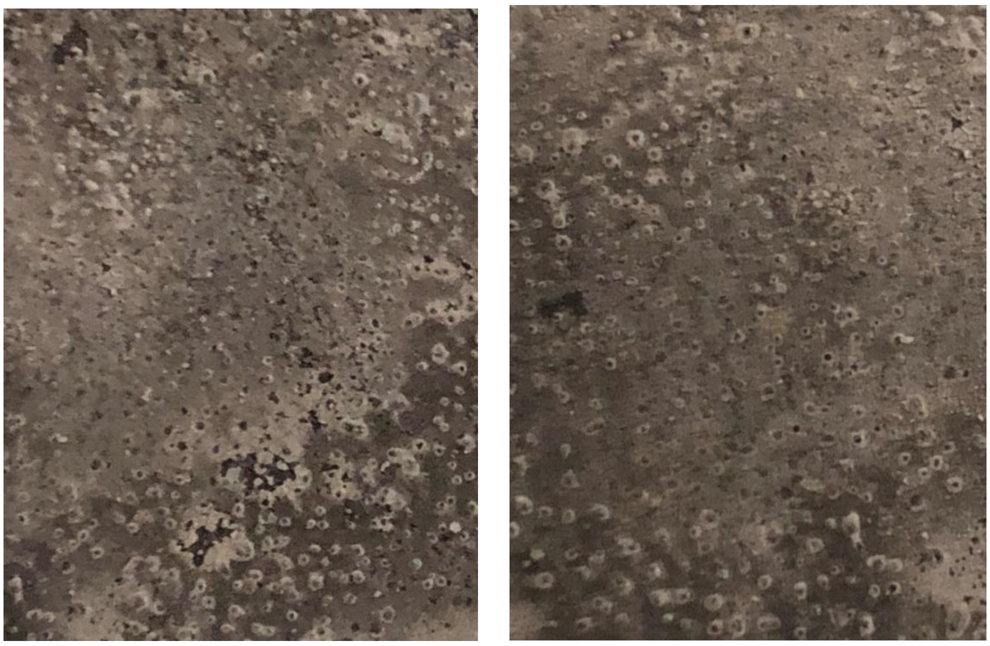
4.3.4 Results & Analysis

This model achieved an accuracy of 95.67%. While still in the same realm of statistical significance ($\alpha<0.05$) as the previous model, the addition of the Pitted category proved to be a drawback for this particular model. All 100 images from the corroded dataset were labeled correctly, again likely attributing to the higher observability of corrosion in steel samples. Similarly, the uncorroded test set only mislabeled a single imaged, Image 6247 (see Figure 16), from the 100 images included in the test set. This is the same image that was mislabeled by Model 2, likely due to the same persisting reason of the questionable surface condition of the steel samples used before corrosion.

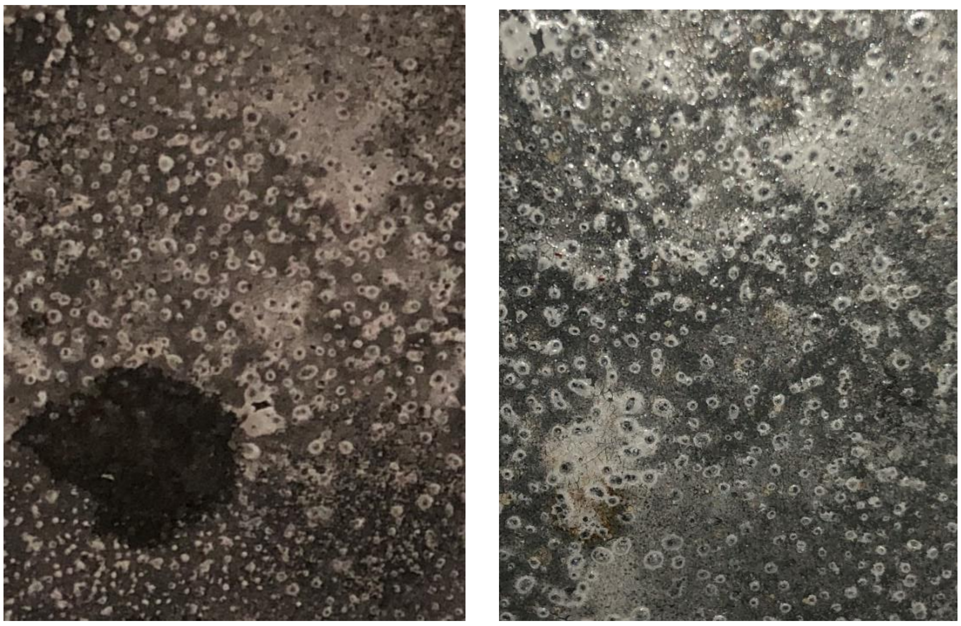
Figure 16: Image 6247

The pitted test set had a grand total of 12 images mislabeled; 4 mislabeled as corroded steel, while 8 were mislabeled as uncorroded all together. The images of corroded aluminium surfaces remain the most problematic and are now contributing further to lowering the accuracy of the model when trained separately in their own category. Images 6571, 6572 and 6577 (Figure 17a-c) could have been confusing to the model and mislabeled as corroded steel due to the lighting conditions they were taken in, while 6598 (Figure 17d) seems to visually resemble steel more so than aluminium, and damaged steel at that, which is likely the reasoning behind the mislabeling. The effect of uncontrolled lighting conditions on the image could be a factor in a fair share of mislabeling instances when utilizing these models for industrial purposes, which should be considered separately for each application.

It can also be observed that four of these mislabeling instances are to the images of corrosion of a completely different material, which is likely why Model 2 had a higher accuracy since this type of mislabeling would not have been apparent. The remaining 8 instances of mislabeling in the Pitted category are still unclear to whether the model is confusing corroded aluminium with corroded steel or simply unable to identify the difference between corroded and uncorroded aluminium. This is explored in Model 4.



(a) (b)



(c) (d)

Figure 17: Images (a) 6571 (b) 6572 (c) 6577 (d) 6598

4.4 Model 4: Four Classes

4.4.1 Objective

The objective is to test a higher level of multiple classification, including a classification between two uncorroded materials, to further evaluate the effectiveness of a corrosion-detection model at multiple classification. It can also provide invaluable insight as to the reason behind the instances of mislabeling in previous models.

4.4.2 Dataset

Similar to the two previous models, the entirety of the training set of 1005 images was used for the training of this model, before testing it on the complete test set of 300 images to obtain the model's accuracy.

4.4.3 Categorization:

This model will have a total of four categories, the highest in this entire study. The categories are:

- Aluminium Corroded (AC), featuring images of corroded surfaces of aluminium.
- Aluminium Uncorroded (AU), featuring images of uncorroded surfaces of aluminium.
- Steel Corroded (SC), featuring images of corroded surfaces of iron and steel.

- Steel Uncorroded (SU), featuring images of uncorroded surfaces of iron and steel.

Table 5 demonstrates how the image datasets were split across the four categories.

Table 5: Model 4 Set Split

	AC	AU	SC	SU
Training Set	218	141	447	119
Test Set	100	40	100	60

4.4.4 Results & Analysis

Model 4 has performed horribly in comparison to earlier models, attaining an accuracy of 80% in the given test set. Of the 300 images in the test set, a grand total of 60 images were mislabel: 36 from the always problematic aluminium corroded test set, 10 from the aluminium uncorroded set, 1 image from the steel corroded set and 13 from the steel uncorroded set.

While an underperforming model, it has still attained its actual purpose: providing invaluable insight to why instances of mislabeling has occurred in the previous models, and how the future models can perform even better.

Looking at Table 6, it can be observed that the largest mislabeling crossover is where aluminium corroded images are mislabeled as uncorroded steel, with the second largest being uncorroded steel mislabeled as uncorroded aluminium. Another problematic occurrence is the mislabeling of uncorroded aluminium as uncorroded steel.

Table 6: Mislabeling Crossover Matrix

	Aluminium Corroded	**Aluminium Uncorroded**	**Steel Corroded**	**Steel Uncorroded**
Aluminium Corroded		0	6	30
Aluminium Uncorroded	0		0	10
Steel Corroded	0	0		1
Steel Uncorroded	0	13	0	

This does provide an important insight into the nature of these mislabeling instances. They are primarily inter-materialistic in nature. As a matter of fact, of the 60 mislabeled images, 59 of them were inter-materialistic, meaning an image of steel/aluminium was categorized into a category representative of the other material over 98% of the time in this model.

While the original aim of the models created for this study to be as general purpose as possible, the results of Model 4 do beg the question as to whether it would yield a superior accuracy if we isolated the materials, creating separate models to identify corrosion in aluminium and steel. These will be Models 5 & 6.

4.5 Model 5: Model of Steel

4.5.1 Objective

The objective behind Model 5 is to evaluate how a trained model will perform if trained and tested on a single type of material, and in this case that type of material steel. Therefore, Model 5 aims to perform a single level of classification between corroded and uncorroded steel. Galvanized iron images are also included as parts of the steel datasets.

4.5.2 Datasets

This model only uses the steel and iron portions of the image datasets, which amounts to 646 images in the training set, and 160 images in the test set.

4.5.3 Categorization

This model follows the suit of the first two models, being categorized on simply two categories: Uncorroded, which includes images of uncorroded surfaces from steel and iron, and Corroded, which includes images of corroded surfaces from steel and iron. Table 7 details how the dataset is split across the two categories.

Table 7: Model 5 Set Split

	Uncorroded	**Corroded**
Training Set	280	525
Test Set	60	100

4.5.4 Results & Analysis

This model demonstrates the superiority of a single-level classification for one material for the application of corrosion-detection through deep learning, achieving an astounding accuracy of 99.38%. It correctly labeled all but one of the 160 images in the dataset. The mislabeled image is Image 6536, which was a corroded image labeled as uncorroded. The image is displayed in Figure 18.

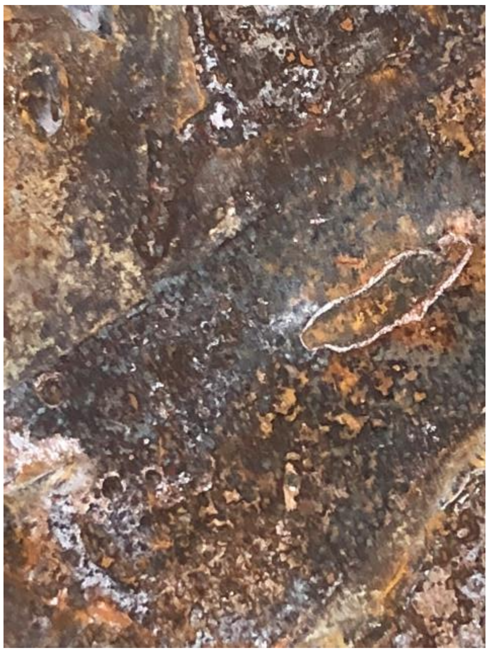


Figure 18: Image 6536, the only mislabeling instance of Model 5

While it does seem questionable that such a high-performing model would get this clearly corroded image wrong, the level of statistically significant accuracy achieved here could allow the forgoing of this mislabeling. It remains a peculiar matter trying to understand which aspect of the image caused the model to mislabel it, but that in itself brings out the highly intriguing topic of what exactly did the model learn.

4.6 Model 6: The Aluminium Model

4.6.1 Objective

Model 6 investigates a model trained solely on aluminium images.

4.6.2 Datasets

This model only uses the aluminium portion of the image datasets, which amounts to 359 images in the training set, and 140 images in the test set.

4.6.3 Categorization

This model classifies aluminium images into two categories: Uncorroded, containing images of uncorroded surfaces from aluminium, and Corroded, containing images of corroded surfaces from aluminium. The dataset split is document in Table 8.

Table 8: Model 6 Set Split

	Uncorroded	**Corroded**
Training Set	141	218
Test Set	40	100

4.6.4 Results & Analysis

Model 6 achieved an accuracy of 100%. It has labeled correctly every single image in its respective test set.

CHAPTER 5: THE DISCUSSION

5.1 The Model's Learning

This section is meant to act as an imposed form of learning for the reader, to understand the process of supervised learning, and specifically dataset selection, by becoming increasingly engaged in the process. While it may be impossible to actually know what a neural network learns, it is in the hands of the model builder to create a representative training dataset that will ideally convey to the network the characteristics that the model builder bears in mind. The following few pages provide a set of examples for images used to train the model. The significance in the selection process and inclusion of certain images in the dataset is presented from the writer's perspective. While this form of writing is unorthodox in the realm of academia, it is pertinent to the value of a deep learning study.

Figure 19 displays a sample of uncorroded aluminium surface images. The first image, Figure 19a was chosen particularly for its peculiar mixed pattern, a resulting element from leaving a recently painted sample face-down on a plastic sheet. The inclusion of an image that is this convoluted in patterning is due to its similarity to patterns visible in corroded surfaces, allowing the model to learn that not all convoluted patterns are corrosion. A model trained on this image being an example of an "uncorroded" surface stands at a better chance of realizing the latest statement.

Figure 19b has a more straightforward purpose in training the model. It is teaching the model that an uncorroded element does not necessarily have the traditional colors of steel, aluminium or iron. The idea in the inclusion of painted surfaces was originally intended to train the model using images of the painted material post-corrosion, but the paint simply ended up dissolving away in the acid bath, leaving the capacity of painted samples to be for training the model on the possibility of colored surfaces.

Figure 19c-d are examples of a more standardized uncorroded image of aluminium, where scratched and slightly damaged surfaces hold value in the training process. It is imperative that the model created is specifically designed for corrosion-detection. Therefore, the presence of other forms of surface damage in the "uncorroded" side of the training reinforces the set goal of the model. Figure 19e holds similar value, with the addition of a different lighting condition: dimly lit room, while Figure 19f was captured outdoors in a sunny condition. As discussed all the way back in Chapter 3, the inclusion of a variety of lighting conditions across all sets helps teach the model to understand differences in lighting can exist within the same model, and are not in fact a determinant factor in labeling the images.

Figures 19g-i represent images of standard uncorroded aluminium samples, taken in a dimly lit room with a camera flash, with surface scratches for added value. These 9 images are representative of the aluminium part of the uncorroded set, which were used to train all the models, including The Aluminium Model, which achieved a 100% accuracy rate.

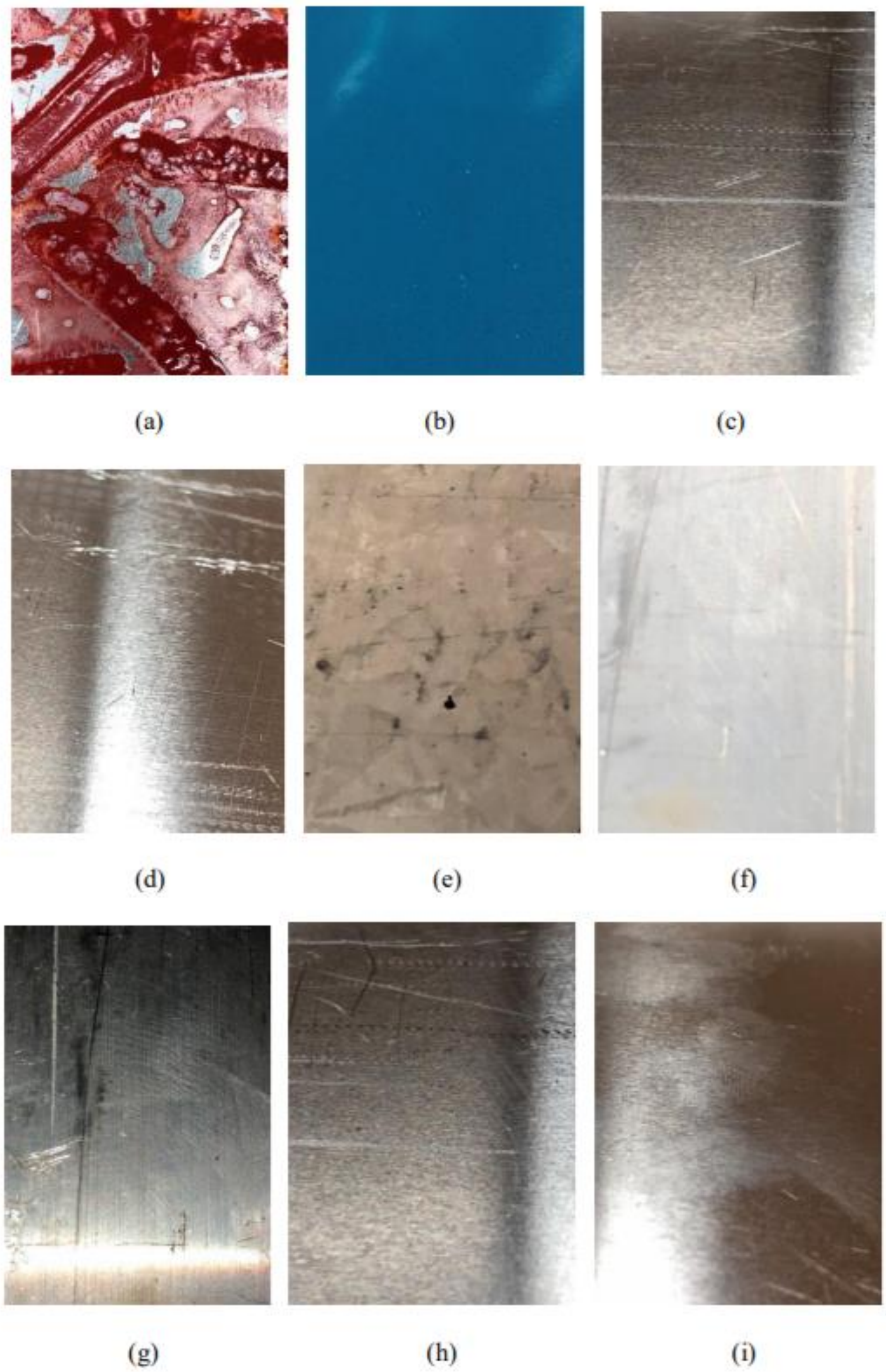


Figure 19: Sample set of uncorroded aluminium

Figure 20 displays a sample of the uncorroded steel/iron training set. The first image, seen in Figure 20a is from a sample created similarly to the one from the first aluminium image, being painted and then left face down, creating the convoluted patterns visible. Figures 20b and 11c are from painted samples of steel and galvanized irons, respectively. The variation in color is essential to widening the understanding of a trained model, but the color options were limited due to the already strong color of the natural surfaces of iron and steel. While not as inclusive and varied as colors used for aluminium, it should be re-noted that the original intent of this study was to simply create an uncorroded vs corroded model, which was achieved in Model 2. Back when these images were captured, aluminium and iron/steel were not planned to act as separate categories.

Figure 20d is an image of a surface of galvanized iron. Originally intended to be a bigger part of this study, possibly warranting its own category, it was eventually included with steel as a single category simply due to the observed similarity in form and shape post-corrosion. Looking at Figure 21, it is unpassable to think that any visual inspection of the images would be able to tell which is steel and which is iron. While removing galvanized iron seemed unnecessary, but so was focusing heavily on it, especially that the number of samples available for the study was limited compared to aluminium and mild steel. It should also be noted that galvanized iron was highly corrodible even in low concentrations of nitric acid, dissolving in a timely manner if left unattended.

Figures 20e and f are more of the standard images representing the uncorroded steel set, with surface scratches and non-corrosion-related damage to further train the model on isolating corrosion as primary element for classifying the images into their respective categories of corroded or uncorroded. Figures 20g and 20h are taken in two different lighting conditions but share the trait of possessing traces of corrosion in the surface, an element that is questionable on its impact on the model and its accuracy. Figure 20i displays a blurry image, its characteristics unclear, but is a form of distortion that was used to train the model on the possibility of a visually ambiguous image.

Figure 21 displays the most visually striking of all categories: the corroded steel set. Glancing at the images included within the figure gives an impression on what visual characteristics do iron and steel-based corrosion possess, and why a model trained on these images can be easily identifiable in comparison to the other sets. The images captured for this set included, as with the rest, a variation in lighting conditions and image quality combined with as wide of a variety of patterns of rusting as the prepared samples for the training set allowed.

The last figure in this section, Figure 22 is a collection of images representing the set on the other end of the spectrum: the problematic corroded aluminium set. The visually striking element that can be seen in particular in Figures 22b, 22c and 22e is the white foaming that is present in conjunction with the corrosion of aluminium. The primary "pitting" effect, which is clearly visible in f, is a less visually striking element, which can be a drawback when training the model. This can be remedied through larger size of datasets, but the performance of Model 6 does indicate the likelihood that the bigger problem is in the similarity between images such as those in Figure 19e and 22e, of corroded aluminium and uncorroded steel, resulted in the accuracy drop in inter-materialistic model, especially when allowing these two sets their own categories, like what happened in Model, resulting in 30 instances of mislabeling, the highest in the study.

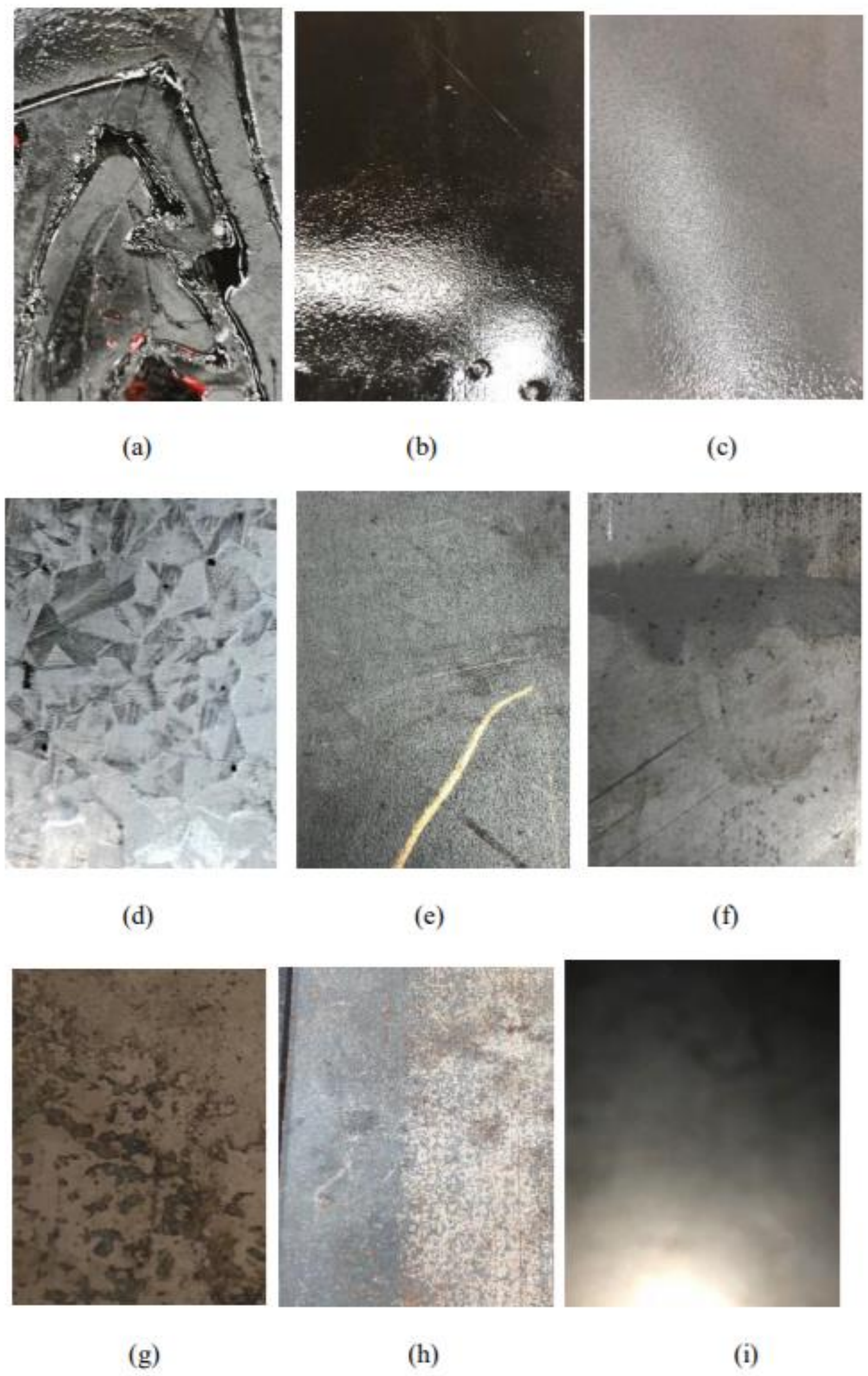


Figure 20: Sample set of uncorroded steel

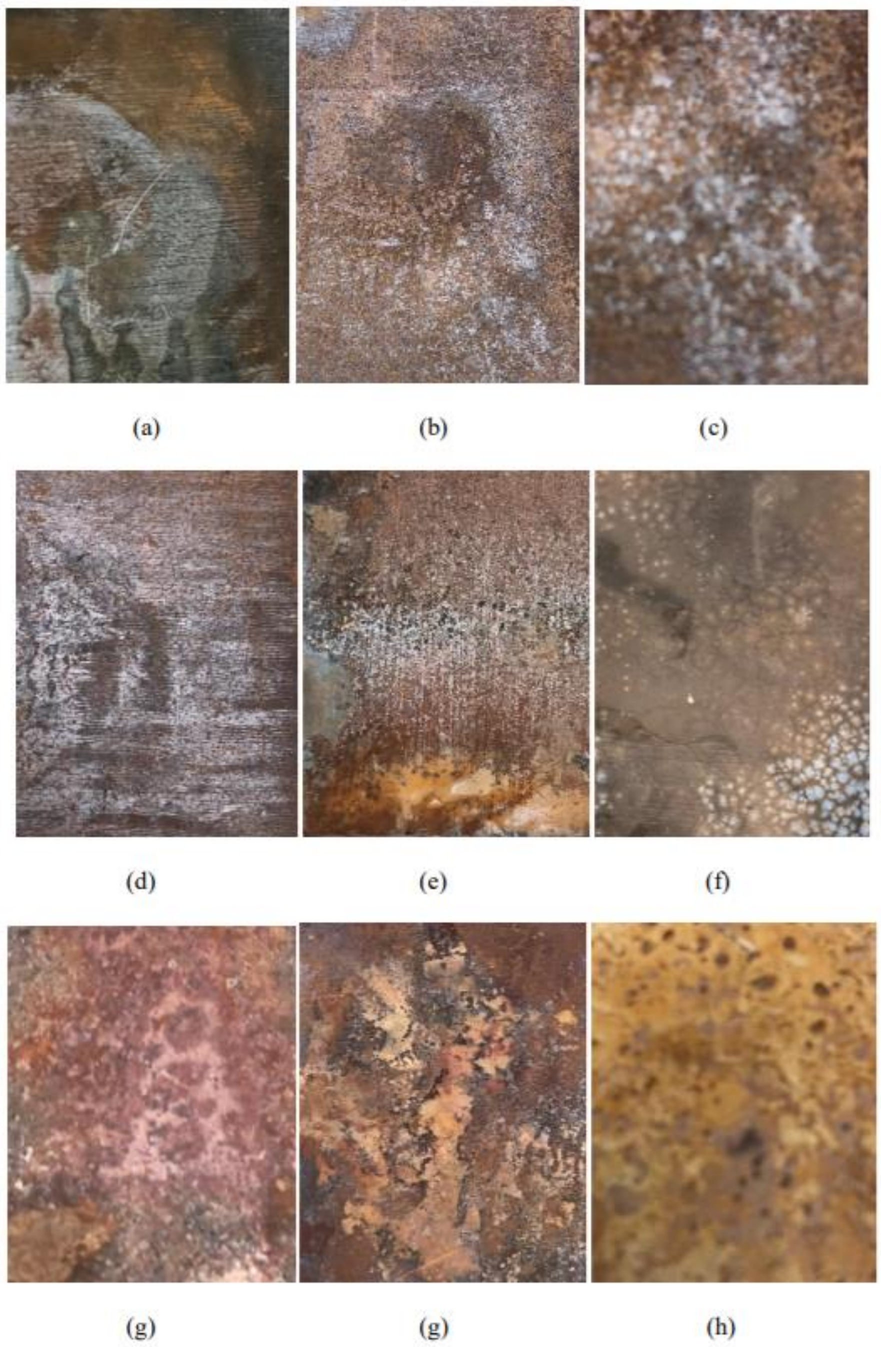


Figure 21: Sample set of corroded steel

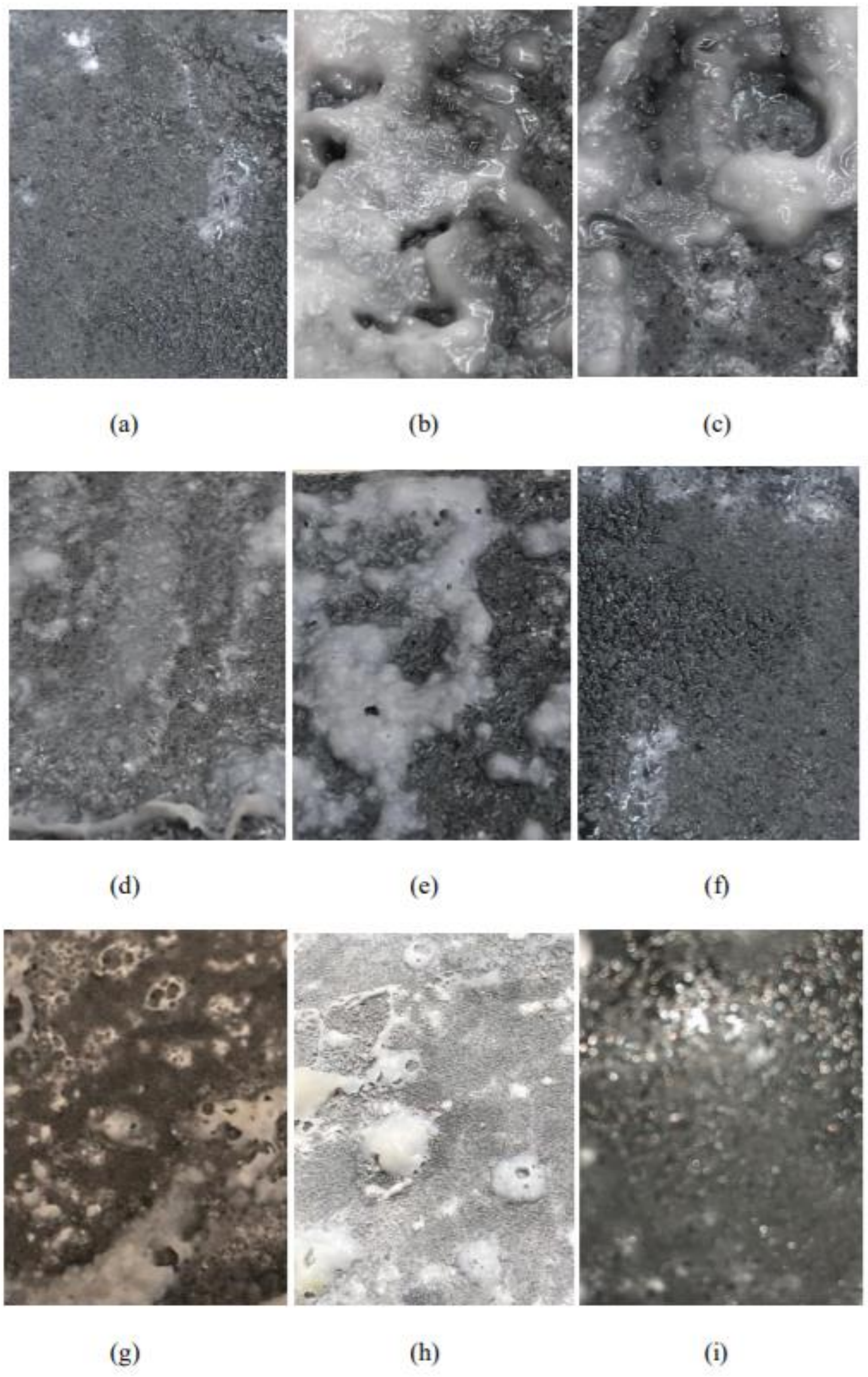


Figure 22: Sample set of corroded aluminium

5.2 Result Evaluation

The process of evaluating the results of the last chapter requires falling back on the significance of each model, detailing a story of discovery that emphasizes the outputs from each model in a serialized fashion that lead to the promising conclusion that is embodied in the high accuracies of Models 5 and 6.

5.2.1 Model: The Pilot Run

Model 1 was intended as an evaluative tool for the number of epochs that should optimally be trained within the available timespan of the study. However, it performed very favorably across all runs, achieving an impressive 96% accuracy at its lowest learning rate and 96.5% for the remainder of the runs. It was used to determine a learning rate of 8000 as the standard for the study, but its true significance was in the promise of high accuracies in corrosion detection. This study used a split of the full training set into a slightly smaller training set and a test set, and while this may not be uncommon in the realm of deep learning, a separate never-seen before test set would always be favorable in testing a model, which is what was created to test the remainder of the models.

5.2.2 Model 2: Inception

Model 2 was meant to be the primary model of the entire study. The idea of a trained model that could filter surface images according to the presence of corrosion, couple with the rise of automated robotry, is sure to hold promise for easier visual inspection of corrodible elements. That was the entire premise on which the study was built, and Model 2 performed at a level that is far above satisfactory, achieving a test set accuracy of 97.67%. This led to Model 2 being a solidifying statement of the success of the study, within reason. As stated in the previous chapter, the study could have simply stopped here, but further investigation held the promise of learning more about the application of deep learning in corrosion detection, and as revealed in the following models, learning how to perfect such model.

5.2.3 Model 3: Pitted v Corroded

Model 3 was taken as the natural next step in this investigative study. Giving a model the capability of not only identifying the presence of corrosion, but also identifying the type of corrosion that is present, would be a tremendous output from the study that is sure to have continued implications in the realms of mechanical maintenance and corrosion-prevention. The Model rose to the task, accurately and correctly categorizing 95.67%. While impressive, the Model revealed an important distinction, pointing to the Pitted set of corroded aluminium surface images as clearly highly problematic, resulting in 12 of the 13 images mislabeled in the test set. Since the models incorporated both aluminium and steel/iron images, the next logical step was to separate these images on both sides of the spectrum: corroded and uncorroded.

5.2.4 Model 4: Four Classes

Model 4 was the result of the segmentation that succeeded Model 3. This Model was the statistical low-point of the entire study, achieving a mere 80% accuracy in the test set, a significant downfall from the +95% models of yore. A grand total of 60 images were mislabeled, 36 of them were from the seemingly ever-problematic corroded aluminium set. Despite this, or more likely as a result of this, this Model became the key to achieving nearly-perfect models. As documented thoroughly in the previous chapter, Model 4 revealed that the majority of all mislabeling instances are inter-materialistic in nature, meaning that the inclusion of both aluminium and steel/iron in the training of a single corrosion-detection model. This, naturally, raised the necessity to test the capability of models that are trained solely on one material type, whether it be aluminium or iron/steel. This led to the training and testing of Models 5 and 6.

5.2.5 Model 5: Model of Steel

Model 5 is a scientific revelation in its unprecedented levels of accuracy for this application, achieving a grand percentage of 99.38% accuracy, only mislabeling a single image out of a total of 160 images in the test set. This largely significant result was only outshined by the next model: The Aluminium Model.

5.2.6 Model 6: The Aluminium Model

Model 6 is likely the most challenging model to discuss, as there is very little to be said about a model that achieved a perfect 100% accuracy on a completely new test set. This along, with Model 5, stand as the grand accomplishment of this study.

5.3 The Consensus

Significant statistical values for recall, precision and F-measure are computed and tabulated in Table 9 for all the binary classification models (Models 1, 2, 5 & 6). For measures with an ideal value of 1, the sheer nearness by which all binary classification models approach the ideal (and even reach it thoroughly for Model 6), it can be clear just how well did the models trained for this study have performed.

Table 9: Statitiscal Measures of Binary Models

	Model 1	**Model 2**	**Model 5**	**Model 6**
Precision	0.98	0.99	1.00	1.00
Recall	0.96	0.98	0.99	1.00
F-Measure	0.97	0.98	0.99	1.00

This study has demonstrated a certain superiority to binary classification for the purposes of corrosion-inspection, have demonstrated a far clearer superiority of single-material model training. Models 5 and 6, with their over-the-top accuracies and evaluative scores, stand as the grand achievement of this study

CHAPTER 6: CONCLUSION & FUTURE RECOMMENDATION

6.1 Study Summary

This study has operated on the premise that a deep learning model trained on a large dataset like the ImageNet dataset, can utilize transfer learning to produce a new model that will be able to label surface images correctly based on the presence or absence of corrosion in said surface. The critical benefit of such model is in the early detection of corroded elements via automated inspection, which in its own right saw a significant rise in utility as detailed back in Chapter 2.

The choice of galvanized iron, mild steel and aluminium came from their availability, ease of corrodibility and the added benefit of resembling a fair share of industrial grade-metals in both their uncorroded and corroded forms. These samples were photographed, corroded, and then photographed once more, creating the datasets required for the training and testing of the models in this study.

A total of six models were created for the study. The first model, Model 1, acted as a test run to evaluate the effect of learning rate on the output of the model, and was used to select a learning rate of 8000 epochs. Model 2 was built to classify images into corroded and uncorroded, while Model 3 classified them into corroded, uncorroded and pitted (for aluminium corrosion). While the previous models performed at an impressive accuracy, Model 4 saw a significant drop in accuracy when classifying images into four categories: aluminium uncorroded, aluminium corroded, steel corroded and steel uncorroded. This, however, proved highly beneficial to understanding the effect of using different types of materials in the same model, as the dominant form of mislabeling Model 4, with 59 of the 60 mislabeling instances being inter-materialistic in nature. This led to the separate of iron/steel and aluminium into two separate models: Model 5 and Model 6, respectively.

Table 10 is a summary of all the critical data that has been uncovered during this study. This was complimented with the addition of the data size and average computing times of the models, which seemed consistently close in values throughout the study.

Table 10: Models Summary

Model	**1**	**2**	**3**	**4**	**5**	**6**
Training Set	805	1005	1005	1005	646	359
Test Set	200	300	300	300	160	140
Data Size (MB)	522	527	536	542	519	511
Avg. Computing Time (s)	0.809	0.789	0.856	0.813	0.793	0.800
F-measure	0.97	0.98			0.99	1.00
Accuracy (%)	96.5	97.67	95.67	80.00	99.38	100

The resulting outcome of this body of work could be summarized into this handful of conclusions:

- As per the results of Models 1 and 2, imaged-based transfer learning proves effective in the detection of corrosion, at positively high accuracies.
- As per Model 4, the inclusion of several materials with similar visual properties can become a drawback for the trained model.
- The conclusion from Model 4 lead to Models 5 and 6, where it was clearly shown that single-material corrosion-detection through deep learning can yield perfect or near-perfect accuracies.

6.2 Further Testing & Implementation

While the models have generally performed very well on the test set, it cannot be ignored that further testing is always critical in verifying the model trained. This is especially important for Models 5 and 6 since they could easily be considered the prime output of this study. An important note when testing is to vary the conditions significantly for the captured test set. While this study involved a training set and test set that were created and captured separately, they were still created in a similar manner to one another, possibly giving the model an easier time in classifying the images. Beyond testing comes the crucial aspect of implementing the models into active robotic visual inspection systems out in the field. This implementation will give access to what full capabilities lie in the models built here in this study and future models built on similar bases and for similar purposes.

6.3 Recommendations

6.3.1 New Models

The tools utilized in this study hold their biggest value not in the models created for this study, but in the potential models that could be created in a similar fashion, either using slight modifications to the tools, or by creating a different dataset to train the model.

6.3.2 Different Architecture

This study has utilized Google's own powerful Inception[36] architecture, and while it has achieved perfect/near-perfect runs on Models 5 and 6, there are still different architectures out there to experiment with for the purposes of this study.

6.3.3 The Many Forms of Corrosion

As stated at the beginning of the study, the objective of the models here will be limited to the uninform aqueous corrosion of iron and steel and the pitted aqueous corrosion of aluminium. However, several other forms of corrosion were mentioned back in Chapter 2, and these forms occur across an entire spectrum of different materials. This creates a large matrix of corrosion forms against materials that could be motive to obtain image datasets and train dozens of models to handle each corrosion form-material combination, especially since this study clearly demonstrated the superiority of models trained specifically for one type of metal and corrosion-form.

6.3.4 Staged Model

In the realm of one type of material aligned with one form of corrosion, a model can be created to differentiate the severity of the corrosion. This would require utilizing a material sample (such as steel) and immersing it into a corrosive environment for different lengths of time, creating a varying array of corrosion severity that, if visually observable, could be used as a dataset to create the model. While this study involved a randomized process for sample creation, one can still observe three levels of corrosion (They can be referred to as no-corrosion, mid-corrosion, late-corrosion) that are visible in the images used from this study’s training set, seen in Figure 23.

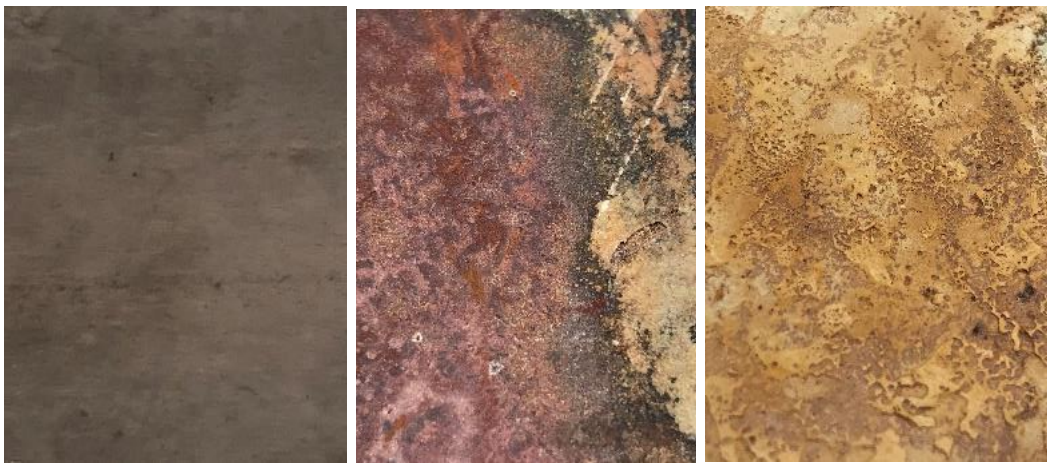


Figure 23: Three stages of steel (no corrosion, mid-corrosion, late corrosion)

REFERENCES

1. Petrovic, Z. Catastrophes caused by corrosion. *Vojnoteh. Glas.* **64**, 1048–1064 (2016).
2. NACE. Economic Impact. *nace.org* (2019). Available at: http://impact.nace.org/economic-impact.aspx. (Accessed: 9th April 2019)
3. Walker, R. Principles and prevention of corrosion. *Mater. Des.* **14**, 207 (1993).
4. Shukla, A. & Karki, H. Application of robotics in onshore oil and gas industry-A review Part i. *Rob. Auton. Syst.* **75**, 490–507 (2016).
5. Nguyen, V. N., Jenssen, R. & Roverso, D. Automatic autonomous vision-based power line inspection: A review of current status and the potential role of deep learning. *Int. J. Electr. Power Energy Syst.* **99**, 107–120 (2018).
6. Donahue, J. *et al.* DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition. (2013).
7. Davis, J. R. *Corrosion: understanding the basics*. *Choice Reviews Online* **37**, (2013).
8. Lazzari, L. Uniform Corrosion. *Eng. Tools Corros.* **298**, 25–38 (2017).
9. Zhang, B. & Ma, X. L. A review - pitting corrosion initiation investigated by TEM. *J. Mater. Sci. Technol.* **35**, 1455–1465 (2019).
10. Popov, B. N. *Pitting and Crevice Corrosion*. *Corrosion Engineering* (2015). doi:10.1016/B978-0-444-62722-3.00007-0
11. Cottis, R. A. *Galvanic corrosion*. *British Corrosion Journal* **25**, (2013).
12. Li, Q., Zeng, D. & An, M. Elevating the photo-generated cathodic protection of corrosion product layers on electrogalvanized steel through nano-electrodeposition. *Chem. Phys. Lett.* **722**, 1–5 (2019).

13. Li, H., Zhang, C., Liu, C. & Huang, M. Improvement in corrosion resistance of CrN coatings. *Surf. Coatings Technol.* **365**, 158–163 (2018).

14. Alrashed, M. M., Jana, S. & Soucek, M. D. Corrosion performance of polyurethane hybrid coatings with encapsulated inhibitor. *Prog. Org. Coatings* **130**, 235–243 (2019).

15. Hedenborn, P. & Bolmsj, G. production economlcs Robotics in automated inspection i Accuracy. **41**, 161–166 (1995).

16. Shafeek, H. I., Gadelmawla, E. S., Abdel-Shafy, A. A. & Elewa, I. M. Automatic inspection of gas pipeline welding defects using an expert vision system. *NDT E Int.* **37**, 301–307 (2004).

17. Nayak, A. & Pradhan, S. K. Design of a new in-pipe inspection robot. *Procedia Eng.* **97**, 2081–2091 (2014).

18. Qu, Y., Durdevic, P. & Yang, Z. Smart-Spider: Autonomous Self-driven In-line Robot for Versatile Pipeline Inspection∗. *IFAC-PapersOnLine* **51**, 251–256 (2018).

19. Vikram Singh Bhadoriya, A., Kumar Gupta, V. & Mukherjee, S. Development of In-pipe Inspection Robot. *Mater. Today Proc.* **5**, 20769–20776 (2018).

20. Sup Yoon, J., Jeong, K. & Lee, B. S. An Inspection Robot of Sewerage Pipes. *IFAC Proc. Vol.* **34**, 51–56 (2017).

21. Rea, P. & Ottaviano, E. Design and development of an Inspection Robotic System for indoor applications. *Robot. Comput. Integr. Manuf.* **49**, 143–151 (2018).

22. Golnabi, H. & Asadpour, A. Design and application of industrial machine vision systems. *Robot. Comput. Integr. Manuf.* **23**, 630–637 (2007).

23. Oh, J. K. *et al.* Bridge inspection robot system with machine vision. *Autom.*

Constr. **18**, 929–941 (2009).

24. Shen, H., Li, S., Gu, D. & Chang, H. Bearing defect inspection based on machine vision. *Meas. J. Int. Meas. Confed.* **45**, 719–733 (2012).

25. Al Ohali, Y. Computer vision based date fruit grading system: Design and implementation. *J. King Saud Univ. - Comput. Inf. Sci.* **23**, 29–36 (2011).

26. Priyadumkol, J., Kittichaikarn, C. & Thainimit, S. Crack detection on unwashed eggs using image processing. *J. Food Eng.* **209**, 76–82 (2017).

27. Li, Y., Dhakal, S. & Peng, Y. A machine vision system for identification of micro-crack in egg shell. *J. Food Eng.* **109**, 127–134 (2012).

28. Zou, Q., Cao, Y., Li, Q., Mao, Q. & Wang, S. CrackTree: Automatic crack detection from pavement images. *Pattern Recognit. Lett.* **33**, 227–238 (2012).

29. Chollet, F. *Deep Learning with Python*. (2018).

30. Patterson, J. & Gibson, A. *Deep Learning A practitioner's approach*. *Clinical Cancer Research* **14**, (2008).

31. Atha, D. J. & Jahanshahi, M. R. Evaluation of deep learning approaches based on convolutional neural networks for corrosion detection. *Struct. Heal. Monit.* **17**, 1110–1128 (2018).

32. Cha, Y. J., Choi, W., Suh, G., Mahmoudkhani, S. & Büyüköztürk, O. Autonomous Structural Visual Inspection Using Region-Based Deep Learning for Detecting Multiple Damage Types. *Comput. Civ. Infrastruct. Eng.* **33**, 731–747 (2018).

33. Petricca, L., Moss, T., Figueroa, G. & Broen, S. Corrosion Detection Using A.I : A Comparison of Standard Computer Vision Techniques and Deep Learning Model. 91–99 (2016). doi:10.5121/csit.2016.60608

34. TensorFlow. How to Retrain an Image Classifier for New Categories. Available

at: https://www.tensorflow.org/hub/tutorials/image_retraining. (Accessed: 1st April 2019)

35. Maurya, R. Ritesh Kumar Maurya’s GitHub Repository for TFP2. Available at: https://github.com/MauryaRitesh/tensorflow-for-poets-2. (Accessed: 20th March 2019)

36. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J. & Wojna, Z. Rethinking the Inception Architecture for Computer Vision. (2015).

APPENDIX A

__init__.py

```
from __future__ import absolute_import

from __future__ import division

from __future__ import print_function
```

count_ops.py

```
from __future__ import absolute_import

from __future__ import division

from __future__ import print_function


import os


import sys

import tensorflow as tf


def load_graph(file_name):

  with open(file_name,'rb') as f:

    content = f.read()

  graph_def = tf.GraphDef()

  graph_def.ParseFromString(content)

  with tf.Graph().as_default() as graph:

    tf.import_graph_def(graph_def, name='')

  return graph


def count_ops(file_name, op_name = None):

  graph = load_graph(file_name)


  if op_name is None:

    return len(graph.get_operations())
  else:
    return sum(1 for op in graph.get_operations()
               if op.name == op_name)


if __name__ == "__main__":
  os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
  print(count_ops(*sys.argv[1:]))
```

evaluate.py

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os


import sys
import argparse


import numpy as np
import PIL.Image as Image
import tensorflow as tf


import scripts.retrain as retrain
from scripts.count_ops import load_graph


def evaluate_graph(graph_file_name):
  with load_graph(graph_file_name).as_default() as graph:
    ground_truth_input = tf.placeholder(
      tf.float32, [None, 5], name='GroundTruthInput')


    image_buffer_input = graph.get_tensor_by_name('input:0')
    final_tensor = graph.get_tensor_by_name('final_result:0')
    accuracy, _ = retrain.add_evaluation_step(final_tensor, ground_truth_input)

    logits = graph.get_tensor_by_name("final_training_ops/Wx_plus_b/add:0")
    xent = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
        labels = ground_truth_input,
        logits = logits))

image_dir = 'tf_files/flower_photos'
testing_percentage = 10
validation_percentage = 10
validation_batch_size = 100
category='testing'

image_lists = retrain.create_image_lists(
    image_dir, testing_percentage,
    validation_percentage)
class_count = len(image_lists.keys())

ground_truths = []
filenames = []

for label_index, label_name in enumerate(image_lists.keys()):
  for image_index, image_name in enumerate(image_lists[label_name][category]):
    image_name = retrain.get_image_path(
        image_lists, label_name, image_index, image_dir, category)
    ground_truth = np.zeros([1, class_count], dtype=np.float32)
    ground_truth[0, label_index] = 1.0
    ground_truths.append(ground_truth)
    filenames.append(image_name)

accuracies = []
xents = []
with tf.Session(graph=graph) as sess:
    for filename, ground_truth in zip(filenames, ground_truths):
        image = Image.open(filename).resize((224,224),Image.ANTIALIAS)
        image = np.array(image, dtype=np.float32)[None,...]
        image = (image-128)/128.0

        feed_dict={
            image_buffer_input: image,
            ground_truth_input: ground_truth}

        eval_accuracy, eval_xent = sess.run([accuracy, xent], feed_dict)

        accuracies.append(eval_accuracy)
        xents.append(eval_xent)


return np.mean(accuracies), np.mean(xents)
```

```
if __name__ == "__main__":
    os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
    accuracy,xent = evaluate_graph(*sys.argv[1:])
    print('Accuracy: %g' % accuracy)
    print('Cross Entropy: %g' % xent)
```

graph_pb2tb.py

```
import os
import sys

import tensorflow as tf

def load_graph(graph_pb_path):
  with open(graph_pb_path,'rb') as f:
    content = f.read()
  graph_def = tf.GraphDef()
  graph_def.ParseFromString(content)
  with tf.Graph().as_default() as graph:
    tf.import_graph_def(graph_def, name='')
  return graph


def graph_to_tensorboard(graph, out_dir):
  with tf.Session():
    train_writer = tf.summary.FileWriter(out_dir)
    train_writer.add_graph(graph)


def main(out_dir, graph_pb_path):
  graph = load_graph(graph_pb_path)
  graph_to_tensorboard(graph, out_dir)


if __name__ == "__main__":
  os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
  main(*sys.argv[1:])
```

label_image.py

```
from __future__ import absolute_import

from __future__ import division

from __future__ import print_function


import argparse

import sys

import time


import numpy as np

import tensorflow as tf


def load_graph(model_file):

  graph = tf.Graph()

  graph_def = tf.GraphDef()


  with open(model_file, "rb") as f:

    graph_def.ParseFromString(f.read())

  with graph.as_default():

    tf.import_graph_def(graph_def)


  return graph


def read_tensor_from_image_file(file_name, input_height=299, input_width=299,
                                input_mean=0, input_std=255):
  input_name = "file_reader"
  output_name = "normalized"
  file_reader = tf.read_file(file_name, input_name)
  if file_name.endswith(".png"):
    image_reader = tf.image.decode_png(file_reader, channels = 3,
                                       name='png_reader')
  elif file_name.endswith(".gif"):
    image_reader = tf.squeeze(tf.image.decode_gif(file_reader,
                                                  name='gif_reader'))
  elif file_name.endswith(".bmp"):
    image_reader = tf.image.decode_bmp(file_reader, name='bmp_reader')
  else:
    image_reader = tf.image.decode_jpeg(file_reader, channels = 3,
                                        name='jpeg_reader')
  float_caster = tf.cast(image_reader, tf.float32)
  dims_expander = tf.expand_dims(float_caster, 0);
  resized = tf.image.resize_bilinear(dims_expander, [input_height, input_width])
  normalized = tf.divide(tf.subtract(resized, [input_mean]), [input_std])
  sess = tf.Session()
  result = sess.run(normalized)

  return result

def load_labels(label_file):
  label = []
  proto_as_ascii_lines = tf.gfile.GFile(label_file).readlines()
  for l in proto_as_ascii_lines:
    label.append(l.rstrip())
  return label

if __name__ == "__main__":
  file_name = "tf_files/flower_photos/daisy/3475870145_685a19116d.jpg"
  model_file = "tf_files/retrained_graph.pb"
  label_file = "tf_files/retrained_labels.txt"
  input_height = 299
  input_width = 299
  input_mean = 0
  input_std = 255
  input_layer = 'Mul'
  output_layer = "final_result"

  parser = argparse.ArgumentParser()
  parser.add_argument("--image", help="image to be processed")
  parser.add_argument("--graph", help="graph/model to be executed")
  parser.add_argument("--labels", help="name of file containing labels")
  parser.add_argument("--input_height", type=int, help="input height")
  parser.add_argument("--input_width", type=int, help="input width")
  parser.add_argument("--input_mean", type=int, help="input mean")
  parser.add_argument("--input_std", type=int, help="input std")
  parser.add_argument("--input_layer", help="name of input layer")
  parser.add_argument("--output_layer", help="name of output layer")
  args = parser.parse_args()


  if args.graph:
    model_file = args.graph
  if args.image:
    file_name = args.image
  if args.labels:
    label_file = args.labels
  if args.input_height:
    input_height = args.input_height
  if args.input_width:
    input_width = args.input_width
  if args.input_mean:
    input_mean = args.input_mean
  if args.input_std:
    input_std = args.input_std
  if args.input_layer:
    input_layer = args.input_layer
  if args.output_layer:
    output_layer = args.output_layer


  graph = load_graph(model_file)
  t = read_tensor_from_image_file(file_name,
                                  input_height=input_height,
                                  input_width=input_width,
                                  input_mean=input_mean,
                                  input_std=input_std)

  input_name = "import/" + input_layer
  output_name = "import/" + output_layer
  input_operation = graph.get_operation_by_name(input_name);
  output_operation = graph.get_operation_by_name(output_name);

  with tf.Session(graph=graph) as sess:
    start = time.time()
    results = sess.run(output_operation.outputs[0],
                      {input_operation.outputs[0]: t})
    end=time.time()
  results = np.squeeze(results)

  top_k = results.argsort()[-5:][::-1]
  labels = load_labels(label_file)

  print('\nEvaluation time (1-image): {:.3f}s\n'.format(end-start))

  for i in top_k:
    print(labels[i], results[i])
```

quantize_graph.py

```
r"""Transforms a float-trained graph into an equivalent quantized version.

An example of command-line usage is:
bazel build tensorflow/tools/quantization:quantize_graph \
&& bazel-bin/tensorflow/tools/quantization/quantize_graph \
--input=tensorflow_inception_graph.pb
--output_node_names="softmax2" --print_nodes --output=/tmp/quantized_graph.pb \
--mode=eightbit --logtostderr

"""

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os
import collections
import re
import numpy as np

from tensorflow.core.framework import attr_value_pb2
from tensorflow.core.framework import graph_pb2
from tensorflow.core.framework import node_def_pb2
from tensorflow.python.client import session
from tensorflow.python.framework import constant_op
from tensorflow.python.framework import dtypes
from tensorflow.python.framework import graph_util
from tensorflow.python.framework import importer
from tensorflow.python.framework import ops
from tensorflow.python.framework import tensor_shape
from tensorflow.python.framework import tensor_util
from tensorflow.python.ops import array_ops
from tensorflow.python.platform import app
from tensorflow.python.platform import flags as flags_lib
from tensorflow.python.platform import gfile


flags = flags_lib
FLAGS = flags.FLAGS


flags.DEFINE_boolean("print_nodes", False, """Lists all nodes in the model.""")
flags.DEFINE_string("input", "", """TensorFlow 'GraphDef' file to load.""")
flags.DEFINE_string("output_node_names", "",
                    """Output node names, comma separated.""")
flags.DEFINE_string("output", "", """File to save the output graph to.""")
flags.DEFINE_integer("bitdepth", 8,
                     """How many bits to quantize the graph to.""")
flags.DEFINE_string("mode", "round",
                    """What transformation to apply (round, quantize,"""
                 """ eightbit, weights, or weights_rounded).""")
flags.DEFINE_string("test_input_dims", "1,224,224,3",
                 """The size of the input tensor to use when testing a"""
                 """ graph loaded from a file.""")
flags.DEFINE_boolean("strip_redundant_quantization", True,
                  """Removes redundant dequantize/quantize pairs.""")
flags.DEFINE_boolean("quantized_input", False,
                  "If true, assume Placeholders are quantized with values "
                  "covering [--quantized_input_min,--quantized_input_max]. "
                  "Only supported when --mode=eightbit")
flags.DEFINE_float("quantized_input_min", 0,
                "The minimum of the actual input range when "
                "--quantized_input")
flags.DEFINE_float("quantized_input_max", 1,
                "The maximum of the actual input range when "
                "--quantized_input")
flags.DEFINE_float(
    "quantized_fallback_min", None,
    "The fallback 'min' value to use for layers which lack min-max "
    "information. Note: this should be considered a coarse tool just good "
    "enough for experimentation purposes, since graphs quantized in this way "
    "would be very inaccurate.")
flags.DEFINE_float(
    "quantized_fallback_max", None,
    "The fallback 'max' value to use for layers which lack min-max "
    "information. Note: this should be considered a coarse tool just good "
    "enough for experimentation purposes, since graphs quantized in this way "
    "would be very inaccurate.")


def print_input_nodes(current_node, nodes_map, indent, already_visited):
  print(" " * indent + current_node.op + ":" + current_node.name)
  already_visited[current_node.name] = True
  for input_node_name in current_node.input:
    if input_node_name in already_visited:
      continue
    input_node = nodes_map[input_node_name]
    print_input_nodes(input_node, nodes_map, indent + 1, already_visited)


def create_node(op, name, inputs):
  new_node = node_def_pb2.NodeDef()
  new_node.op = op
  new_node.name = name
  for input_name in inputs:
    new_node.input.extend([input_name])
  return new_node


def create_constant_node(name, value, dtype, shape=None):
  node = create_node("Const", name, [])

  set_attr_dtype(node, "dtype", dtype)

  set_attr_tensor(node, "value", value, dtype, shape)

  return node




def copy_attr(node, key, attr_value):

  try:

    node.attr[key].CopyFrom(attr_value)

  except KeyError:

    pass




def set_attr_dtype(node, key, value):

  try:

    node.attr[key].CopyFrom(

        attr_value_pb2.AttrValue(type=value.as_datatype_enum))

  except KeyError:

    pass




def set_attr_shape(node, key, value):

  try:

    node.attr[key].CopyFrom(

        attr_value_pb2.AttrValue(shape=tensor_shape.as_shape(value).as_proto()))
  except KeyError:
    pass


def set_attr_tensor(node, key, value, dtype, shape=None):
  try:
    node.attr[key].CopyFrom(
        attr_value_pb2.AttrValue(tensor=tensor_util.make_tensor_proto(
            value, dtype=dtype, shape=shape)))
  except KeyError:
    pass


def set_attr_string(node, key, value):
  try:
    node.attr[key].CopyFrom(attr_value_pb2.AttrValue(s=value))
  except KeyError:
    pass


def set_attr_int_list(node, key, value):
  list_value = attr_value_pb2.AttrValue.ListValue(i=value)
  try:
    node.attr[key].CopyFrom(attr_value_pb2.AttrValue(list=list_value))
  except KeyError:
    pass



def set_attr_bool(node, key, value):
  try:
    node.attr[key].CopyFrom(attr_value_pb2.AttrValue(b=value))
  except KeyError:
    pass



def set_attr_int(node, key, value):
  try:
    node.attr[key].CopyFrom(attr_value_pb2.AttrValue(i=value))
  except KeyError:
    pass



def set_attr_float(node, key, value):
  try:
    node.attr[key].CopyFrom(attr_value_pb2.AttrValue(f=value))
  except KeyError:
    pass



def node_name_from_input(node_name):
  """Strips off ports and other decorations to get the underlying node name."""
  if node_name.startswith("^"):
    node_name = node_name[1:]
  m = re.search(r"(.*):\d+$", node_name)
  if m:
    node_name = m.group(1)
  return node_name


def ensure_tensor_name_has_port(node_name):
  """Makes sure that a tensor name has :0 if no explicit port exists."""
  m = re.search(r"(.*):\d+$", node_name)
  if m:
    name_with_port = node_name
  else:
    name_with_port = node_name + ":0"
  return name_with_port


def unique_node_name_from_input(node_name):
  """Replaces invalid characters in input names to get a unique node name."""
  return node_name.replace(":", "__port__").replace("^", "__hat__")


def quantize_array(arr, num_buckets):
```

```
"""Quantizes a numpy array.

This function maps each scalar in arr to the center of one of num_buckets
buckets. For instance,
quantize_array([0, 0.3, 0.6, 1], 2) => [0.25, 0.25, 0.75, 0.75]

Args:
  arr: The numpy array to quantize.
  num_buckets: The number of buckets to map "var" to.
Returns:
  The quantized numpy array.
Raises:
  ValueError: when num_buckets < 1.
"""
if num_buckets < 1:
  raise ValueError("num_buckets must be >= 1")
arr_max = arr.max()
arr_min = arr.min()
if arr_max == arr_min:
  return arr
bucket_width = (arr_max - arr_min) / num_buckets
# Map scalars to bucket indices. Take special care of max(arr).
bucket_indices = np.floor((arr - arr_min) / bucket_width)
bucket_indices[bucket_indices == num_buckets] = num_buckets - 1
# Map each scalar to the center of a bucket.
```

```
  arr = arr_min + bucket_width * (bucket_indices + 0.5)
  return arr



def quantize_weight_rounded(input_node):
  """Returns a replacement node for input_node containing bucketed floats."""
  input_tensor = input_node.attr["value"].tensor
  tensor_value = tensor_util.MakeNdarray(input_tensor)
  shape = input_tensor.tensor_shape
  # Currently, the parameter FLAGS.bitdepth is used to compute the
  # number of buckets as 1 << FLAGS.bitdepth, meaning the number of
  # buckets can only be a power of 2.
  # This could be fixed by introducing a new parameter, num_buckets,
  # which would allow for more flexibility in chosing the right model
  # size/accuracy tradeoff. But I didn't want to add more parameters
  # to this script than absolutely necessary.
  num_buckets = 1 << FLAGS.bitdepth
  tensor_value_rounded = quantize_array(tensor_value, num_buckets)
  tensor_shape_list = tensor_util.TensorShapeProtoToList(shape)
  return [
      create_constant_node(
          input_node.name,
          tensor_value_rounded,
          dtypes.float32,
          shape=tensor_shape_list)
```

```
 ]



def quantize_weight_eightbit(input_node, quantization_mode):
  """Returns replacement nodes for input_node using the Dequantize op."""
  base_name = input_node.name + "_"
  quint8_const_name = base_name + "quint8_const"
  min_name = base_name + "min"
  max_name = base_name + "max"
  float_tensor = tensor_util.MakeNdarray(input_node.attr["value"].tensor)
  min_value = np.min(float_tensor.flatten())
  max_value = np.max(float_tensor.flatten())
  # Make sure that the range includes zero.
  if min_value > 0.0:
    min_value = 0.0
  # min_value == max_value is a tricky case. It can occur for general
  # tensors, and of course for scalars. The quantized ops cannot deal
  # with this case, so we set max_value to something else.
  # It's a tricky question what is the numerically best solution to
  # deal with this degeneracy.
  # TODO(petewarden): Better use a tolerance than a hard comparison?
  if min_value == max_value:
    if abs(min_value) < 0.000001:
      max_value = min_value + 1.0
    elif min_value > 0:
```

```
    max_value = 2 * min_value
  else:
    max_value = min_value / 2.0

sess = session.Session()
with sess.as_default():
  quantize_op = array_ops.quantize_v2(
      float_tensor,
      min_value,
      max_value,
      dtypes.quint8,
      mode=quantization_mode)
  quint8_tensor = quantize_op[0].eval()
shape = tensor_util.TensorShapeProtoToList(input_node.attr["value"]
                                            .tensor.tensor_shape)
quint8_const_node = create_constant_node(
    quint8_const_name, quint8_tensor, dtypes.quint8, shape=shape)
min_node = create_constant_node(min_name, min_value, dtypes.float32)
max_node = create_constant_node(max_name, max_value, dtypes.float32)
dequantize_node = create_node("Dequantize", input_node.name,
                              [quint8_const_name, min_name, max_name])
set_attr_dtype(dequantize_node, "T", dtypes.quint8)
set_attr_string(dequantize_node, "mode", quantization_mode)
return [quint8_const_node, min_node, max_node, dequantize_node]
```

```
EightbitizeRecursionState = collections.namedtuple(
    "EightbitizeRecursionState",
    ["already_visited", "output_node_stack", "merged_with_fake_quant"])



class GraphRewriter(object):
  """Takes a float graph, and rewrites it in quantized form."""

  def __init__(self,
               input_graph,
               mode,
               quantized_input_range,
               fallback_quantization_range=None):
    """Sets up the class to rewrite a float graph.

    Args:
      input_graph: A float graph to transform.
      mode: A string controlling how quantization is performed -
        round, quantize, eightbit, or weights.
      quantized_input_range: if set, assume the input is
        quantized and represents the range
        [quantized_input_range[0], quantized_input_range[1]]
      fallback_quantization_range: if set, then for nodes where the quantization
        range can't be inferred from the graph, use the range
    [fallback_quantization_range[0], fallback_quantization_range[1]) instead
    of using a RequantizationRange node in the graph.

Raises:
  ValueError: Two nodes with the same name were found in the graph.
"""
self.input_graph = input_graph
self.nodes_map = self.create_nodes_map(input_graph)
self.output_graph = None
self.mode = mode
self.final_node_renames = {}
if quantized_input_range:
  self.input_range = (quantized_input_range[0], quantized_input_range[1])
  if self.input_range[0] >= self.input_range[1]:
    raise ValueError("Invalid quantized_input_range: [%s,%s]" %
                     self.input_range)
  if self.mode != "eightbit":
    raise ValueError(
        "quantized_input_range can only be specified in eightbit mode")
else:
  self.input_range = None

if fallback_quantization_range:
  self.fallback_quantization_range = [
      fallback_quantization_range[0], fallback_quantization_range[1]
  ]
  if (self.fallback_quantization_range[0] >=
      self.fallback_quantization_range[1]):
    raise ValueError("Invalid fallback_quantization_range: [%s,%s]" %
                     self.fallback_quantization_range)
  if self.mode != "eightbit":
    raise ValueError("fallback_quantization_range can only be "
                     "specified in eightbit mode")
 else:
  self.fallback_quantization_range = None

 # Data that is valid only during the recursive call to rewrite the graph.
 self.state = None

def create_nodes_map(self, graph):
 """Builds a mapping of node names to their defs from the graph."""
 nodes_map = {}
 for node in graph.node:
  if node.name not in nodes_map.keys():
    nodes_map[node.name] = node
  else:
    raise ValueError("Duplicate node names detected.")
 return nodes_map

def rewrite(self, output_node_names):
```

```
"""Triggers rewriting of the float graph.

Args:
  output_node_names: A list of names of the nodes that produce the final
    results.

Returns:
  A quantized version of the float graph.
"""
self.output_graph = graph_pb2.GraphDef()
output_nodes = [
    self.nodes_map[output_node_name]
    for output_node_name in output_node_names
]
if self.mode == "round":
  self.already_visited = {}
  for output_node in output_nodes:
    self.round_nodes_recursively(output_node)
elif self.mode == "quantize":
  self.already_visited = {}
  self.already_quantized = {}
  for output_node in output_nodes:
    self.quantize_nodes_recursively(output_node)
elif self.mode == "eightbit":
  self.set_input_graph(graph_util.remove_training_nodes(self.input_graph))
```

```
output_nodes = [
    self.nodes_map[output_node_name]
    for output_node_name in output_node_names
]

self.state = EightbitizeRecursionState(
    already_visited={}, output_node_stack=[], merged_with_fake_quant={})
for output_node in output_nodes:
  self.eightbitize_nodes_recursively(output_node)
self.state = None
if self.input_range:
  self.add_output_graph_node(
      create_constant_node("quantized_input_min_value", self.input_range[
          0], dtypes.float32, []))
  self.add_output_graph_node(
      create_constant_node("quantized_input_max_value", self.input_range[
          1], dtypes.float32, []))
if self.fallback_quantization_range:
  self.add_output_graph_node(
      create_constant_node("fallback_quantization_min_value",
                           self.fallback_quantization_range[0],
                           dtypes.float32, []))
  self.add_output_graph_node(
      create_constant_node("fallback_quantization_max_value",
                           self.fallback_quantization_range[1],
                                    dtypes.float32, []))
      if FLAGS.strip_redundant_quantization:
        self.output_graph = self.remove_redundant_quantization(
            self.output_graph)
        self.remove_dead_nodes(output_node_names)
      self.apply_final_node_renames()
    elif self.mode == "weights":
      self.output_graph = self.quantize_weights(self.input_graph,
                                                b"MIN_COMBINED")
      self.remove_dead_nodes(output_node_names)
    elif self.mode == "weights_rounded":
      self.output_graph = self.quantize_weights(self.input_graph, self.mode)
      self.remove_dead_nodes(output_node_names)
    else:
      print("Bad mode - " + self.mode + ".")
    return self.output_graph

  def round_nodes_recursively(self, current_node):
    """The entry point for simple rounding quantization."""
    if self.already_visited[current_node.name]:
      return
    self.already_visited[current_node.name] = True
    for input_node_name in current_node.input:
      input_node_name = node_name_from_input(input_node_name)
      input_node = self.nodes_map[input_node_name]
      self.round_nodes_recursively(input_node)
    nodes_to_quantize = ["Conv2D", "BiasAdd", "MatMul"]
    if any(current_node.op in s for s in nodes_to_quantize):
      new_node = node_def_pb2.NodeDef()
      new_node.CopyFrom(current_node)
      new_node.name = current_node.name + "_original"
      self.add_output_graph_node(new_node)
      levels = 1 << FLAGS.bitdepth
      constant_name = current_node.name + "_round_depth"
      constant_tensor = constant_op.constant(
          levels, dtype=dtypes.int32, name=constant_name)
      constant_node = constant_tensor.op.node_def
      self.add_output_graph_node(constant_node)
      quantize_node = node_def_pb2.NodeDef()
      quantize_node.op = "RoundToSteps"
      quantize_node.name = current_node.name
      quantize_node.input.extend([current_node.name + "_original"])
      quantize_node.input.extend([constant_node.name])
      self.add_output_graph_node(quantize_node)
    else:
      new_node = node_def_pb2.NodeDef()
      new_node.CopyFrom(current_node)
      self.add_output_graph_node(new_node)


  def quantize_nodes_recursively(self, current_node):
    """The entry point for quantizing nodes to eight bit and back."""
    if self.already_visited[current_node.name]:
      return
    self.already_visited[current_node.name] = True
    for input_node_name in current_node.input:
      input_node_name = node_name_from_input(input_node_name)
      input_node = self.nodes_map[input_node_name]
      self.quantize_nodes_recursively(input_node)
    nodes_to_quantize = ["Conv2D", "BiasAdd", "MatMul"]
    if any(current_node.op in s for s in nodes_to_quantize):
      for input_name in current_node.input:
        input_name = node_name_from_input(input_name)
        input_node = self.nodes_map[input_name]
        self.quantize_node(input_node)
      self.quantize_node(current_node)
    else:
      new_node = node_def_pb2.NodeDef()
      new_node.CopyFrom(current_node)
      self.add_output_graph_node(new_node)

  def quantize_node(self, input_node):
    """Handles quantizing a single node."""
    input_name = input_node.name
    if input_name in self.already_quantized:
      return
```

```
self.already_quantized[input_name] = True

original_input_name = input_name + "_original"

reshape_name = input_name + "_reshape"

reshape_dims_name = input_name + "_reshape_dims"

max_name = input_name + "_max"

min_name = input_name + "_min"

dims_name = input_name + "_dims"

quantize_name = input_name + "_quantize"

dequantize_name = input_name

original_input_node = node_def_pb2.NodeDef()

original_input_node.CopyFrom(input_node)

original_input_node.name = original_input_name

self.add_output_graph_node(original_input_node)

reshape_dims_node = create_constant_node(reshape_dims_name, -1,

                                         dtypes.int32, [1])

self.add_output_graph_node(reshape_dims_node)

reshape_node = create_node("Reshape", reshape_name,

                           [original_input_name, reshape_dims_name])

set_attr_dtype(reshape_node, "T", dtypes.float32)

self.add_output_graph_node(reshape_node)

dims_node = create_constant_node(dims_name, 0, dtypes.int32, [1])

self.add_output_graph_node(dims_node)

max_node = create_node("Max", max_name, [reshape_name, dims_name])

set_attr_dtype(max_node, "T", dtypes.float32)

set_attr_bool(max_node, "keep_dims", False)
```

```
  self.add_output_graph_node(max_node)
  min_node = create_node("Min", min_name, [reshape_name, dims_name])
  set_attr_dtype(min_node, "T", dtypes.float32)
  set_attr_bool(min_node, "keep_dims", False)
  self.add_output_graph_node(min_node)
  quantize_node = create_node("Quantize", quantize_name,
                              [original_input_name, min_name, max_name])
  set_attr_dtype(quantize_node, "T", dtypes.quint8)
  set_attr_string(quantize_node, "mode", b"MIN_FIRST")
  self.add_output_graph_node(quantize_node)
  dequantize_node = create_node("Dequantize", dequantize_name,
                                [quantize_name, min_name, max_name])
  set_attr_dtype(dequantize_node, "T", dtypes.quint8)
  set_attr_string(dequantize_node, "mode", b"MIN_FIRST")
  self.add_output_graph_node(dequantize_node)

def should_merge_with_fake_quant_node(self):
  """Should the current node merge with self.state.output_node_stack[-1]?"""
  if not self.state.output_node_stack:
    return False
  top = self.state.output_node_stack[-1]
  return top[1] == 0 and top[0].op in ["FakeQuantWithMinMaxVars"]

def should_quantize_const(self, node):
  if not self.state.output_node_stack:
    return False
  top = self.state.output_node_stack[-1]
  if not top[2]:
    return False
  dtype = dtypes.as_dtype(node.attr["dtype"].type)
  assert dtype == dtypes.float32, (
      "Failed to quantized constant %s of type %s" % (node.name, dtype))
  return True

def eightbitize_nodes_recursively(self, current_node):
  """The entry point for transforming a graph into full eight bit."""
  if current_node.name in self.state.already_visited:
    if (self.should_merge_with_fake_quant_node() or
        current_node.name in self.state.merged_with_fake_quant):
      raise ValueError("Unsupported graph structure: output of node %s "
                       "is processed by a FakeQuant* node and should have "
                       "no other outputs.", current_node.name)
    return
  self.state.already_visited[current_node.name] = True

  for i, input_node_name in enumerate(current_node.input):
    quantize_input = False
    if current_node.op in ("MatMul", "Conv2D", "BiasAdd", "MaxPool",
                           "AvgPool", "Relu", "Relu6",
                           "BatchNormWithGlobalNormalization"):
```

```
    quantize_input = True
  elif current_node.op == "Concat" and i > 0:
    quantize_input = (
        dtypes.as_dtype(current_node.attr["T"].type) == dtypes.float32)
  elif current_node.op == "Reshape" and i == 0:
    quantize_input = (
        dtypes.as_dtype(current_node.attr["T"].type) == dtypes.float32)

  self.state.output_node_stack.append((current_node, i, quantize_input))

  input_node_name = node_name_from_input(input_node_name)
  input_node = self.nodes_map[input_node_name]
  self.eightbitize_nodes_recursively(input_node)

  self.state.output_node_stack.pop()

if current_node.op == "MatMul":
  self.eightbitize_mat_mul_node(current_node)
elif current_node.op == "Conv2D":
  self.eightbitize_conv_node(current_node)
elif current_node.op == "BiasAdd":
  self.eightbitize_bias_add_node(current_node)
elif current_node.op == "MaxPool" or current_node.op == "AvgPool":
  self.eightbitize_single_input_tensor_node(current_node,
                                            self.add_pool_function)
```

```
elif current_node.op == "Relu" or current_node.op == "Relu6":
  self.eightbitize_single_input_tensor_node(current_node,
                                            self.add_relu_function)
elif (current_node.op == "Concat" and
      dtypes.as_dtype(current_node.attr["T"].type) == dtypes.float32):
  self.eightbitize_concat_node(current_node)
elif current_node.op == "BatchNormWithGlobalNormalization":
  self.eightbitize_batch_norm_node(current_node)
elif (current_node.op == "Reshape" and
      dtypes.as_dtype(current_node.attr["T"].type) == dtypes.float32):
  self.eightbitize_reshape_node(current_node)
elif (self.input_range and
      current_node.op in ("Placeholder", "PlaceholderV2")):
  self.eightbitize_placeholder_node(current_node)
elif current_node.op == "FakeQuantWithMinMaxVars":
  # It will have been merged into the underlying node.
  pass
elif current_node.op == "Const":
  if self.should_quantize_const(current_node):
    for n in quantize_weight_eightbit(current_node, b"MIN_FIRST"):
      self.add_output_graph_node(n)
  else:
    new_node = node_def_pb2.NodeDef()
    new_node.CopyFrom(current_node)
    self.add_output_graph_node(new_node)
    ######################################################################
    # Note: if more cases are added here, you may need to update the op
    # name lists in the loop over children at the start of the function.
    ######################################################################
    else:
      new_node = node_def_pb2.NodeDef()
      new_node.CopyFrom(current_node)
      self.add_output_graph_node(new_node)

    if (self.should_merge_with_fake_quant_node() and
        current_node.name not in self.state.merged_with_fake_quant):
      raise ValueError(
          "FakeQuant* node %s failed to merge with node %s of type %s" %
          (self.state.output_node_stack[-1][0], current_node.name,
           current_node.op))

  def add_eightbit_prologue_nodes(self, original_node):
    """Adds input conversion nodes to handle quantizing the underlying node."""
    namespace_prefix = original_node.name + "_eightbit"
    reshape_dims_name,                 reduction_dims_name                 =
self.add_common_quantization_nodes(
        namespace_prefix)
    input_names = []
    min_max_names = []
  for original_input_name in original_node.input:
   quantize_input_name, min_input_name, max_input_name = (
     self.eightbitize_input_to_node(namespace_prefix, original_input_name,
                                    reshape_dims_name,
                                    reduction_dims_name))
   input_names.append(quantize_input_name)
   min_max_names.append(min_input_name)
   min_max_names.append(max_input_name)
  all_input_names = []
  all_input_names.extend(input_names)
  all_input_names.extend(min_max_names)
  return all_input_names

 def add_common_quantization_nodes(self, namespace_prefix):
  """Builds constant nodes needed for quantization of inputs."""
  reshape_dims_name = namespace_prefix + "_reshape_dims"
  reduction_dims_name = namespace_prefix + "_reduction_dims"

  reshape_dims_node = create_constant_node(reshape_dims_name, -1,
                                           dtypes.int32, [1])
  self.add_output_graph_node(reshape_dims_node)
  reduction_dims_node = create_constant_node(reduction_dims_name, 0,
                                             dtypes.int32, [1])
  self.add_output_graph_node(reduction_dims_node)
  return reshape_dims_name, reduction_dims_name
```

```
def eightbitize_input_to_node(self, namespace_prefix, original_input_name,
                              reshape_dims_name, reduction_dims_name):
  """Takes one float input to an op, and converts it to quantized form."""
  unique_input_name = unique_node_name_from_input(original_input_name)
  reshape_input_name = namespace_prefix + "_reshape_" + unique_input_name
  min_input_name = namespace_prefix + "_min_" + unique_input_name
  max_input_name = namespace_prefix + "_max_" + unique_input_name
  quantize_input_name = namespace_prefix + "_quantize_" + unique_input_name
  reshape_input_node = create_node("Reshape", reshape_input_name,
                                   [original_input_name, reshape_dims_name])
  set_attr_dtype(reshape_input_node, "T", dtypes.float32)
  self.add_output_graph_node(reshape_input_node)
  min_input_node = create_node("Min", min_input_name,
                               [reshape_input_name, reduction_dims_name])
  set_attr_dtype(min_input_node, "T", dtypes.float32)
  set_attr_bool(min_input_node, "keep_dims", False)
  self.add_output_graph_node(min_input_node)
  max_input_node = create_node("Max", max_input_name,
                               [reshape_input_name, reduction_dims_name])
  set_attr_dtype(max_input_node, "T", dtypes.float32)
  set_attr_bool(max_input_node, "keep_dims", False)
  self.add_output_graph_node(max_input_node)
  quantize_input_node = create_node(
      "QuantizeV2", quantize_input_name,
      [original_input_name, min_input_name, max_input_name])
  set_attr_dtype(quantize_input_node, "T", dtypes.quint8)
  set_attr_string(quantize_input_node, "mode", b"MIN_FIRST")
  self.add_output_graph_node(quantize_input_node)
  min_output_name = quantize_input_name + ":1"
  max_output_name = quantize_input_name + ":2"
  return quantize_input_name, min_output_name, max_output_name

def add_quantize_down_nodes(self, original_node, quantized_output_name):
  quantized_outputs = [
      quantized_output_name, quantized_output_name + ":1",
      quantized_output_name + ":2"
  ]
  min_max_inputs = None
  if self.should_merge_with_fake_quant_node():
    # Use the inputs to the FakeQuantWithMinMaxVars node as the inputs to
    # Requantize.
    fake_quant_node = self.state.output_node_stack[-1][0]
    min_max_inputs = [fake_quant_node.input[1], fake_quant_node.input[2]]
    assert original_node.name not in self.state.merged_with_fake_quant
    self.state.merged_with_fake_quant[original_node.name] = True
  elif self.fallback_quantization_range:
    min_max_inputs = [
        "fallback_quantization_min_value:0",
        "fallback_quantization_max_value:0"
    ]
  else:
    # Add a RequantizationRange node for finding the min and max values.
    requant_range_node = create_node(
        "RequantizationRange", original_node.name + "_eightbit_requant_range",
        quantized_outputs)
    set_attr_dtype(requant_range_node, "Tinput", dtypes.qint32)
    self.add_output_graph_node(requant_range_node)
    min_max_inputs = [
        requant_range_node.name + ":0", requant_range_node.name + ":1"
    ]
  requantize_node = create_node("Requantize",
                                original_node.name + "_eightbit_requantize",
                                quantized_outputs + min_max_inputs)
  set_attr_dtype(requantize_node, "Tinput", dtypes.qint32)
  set_attr_dtype(requantize_node, "out_type", dtypes.quint8)
  self.add_output_graph_node(requantize_node)
  return requantize_node.name


def add_dequantize_result_node(self,
                               quantized_output_name,
                               original_node_name,
                               min_tensor_index=1):
  min_max_inputs = [
      "%s:%s" % (quantized_output_name, min_tensor_index),
    "%s:%s" % (quantized_output_name, (min_tensor_index + 1))
  ]
  dequantize_name = original_node_name
  if self.should_merge_with_fake_quant_node():
    fake_quant_node = self.state.output_node_stack[-1][0]
    if original_node_name not in self.state.merged_with_fake_quant:
      min_max_inputs = [fake_quant_node.input[1], fake_quant_node.input[2]]
      self.state.merged_with_fake_quant[original_node_name] = True
    dequantize_name = fake_quant_node.name

  dequantize_node = create_node(
      "Dequantize", dequantize_name,
      [quantized_output_name, min_max_inputs[0], min_max_inputs[1]])
  set_attr_dtype(dequantize_node, "T", dtypes.quint8)
  set_attr_string(dequantize_node, "mode", b"MIN_FIRST")
  self.add_output_graph_node(dequantize_node)

def eightbitize_mat_mul_node(self, original_node):
  """Replaces a MatMul node with the eight bit equivalent sub-graph."""
  quantized_mat_mul_name = original_node.name + "_eightbit_quantized_mat_mul"
  all_input_names = self.add_eightbit_prologue_nodes(original_node)
  quantized_mat_mul_node = create_node("QuantizedMatMul",
                                       quantized_mat_mul_name,
                                       all_input_names)
  set_attr_dtype(quantized_mat_mul_node, "T1", dtypes.quint8)
  set_attr_dtype(quantized_mat_mul_node, "T2", dtypes.quint8)
  set_attr_dtype(quantized_mat_mul_node, "Toutput", dtypes.qint32)
  copy_attr(quantized_mat_mul_node, "transpose_a",
            original_node.attr["transpose_a"])
  copy_attr(quantized_mat_mul_node, "transpose_b",
            original_node.attr["transpose_b"])
  self.add_output_graph_node(quantized_mat_mul_node)
  quantize_down_name = self.add_quantize_down_nodes(original_node,
                                                    quantized_mat_mul_name)
  self.add_dequantize_result_node(quantize_down_name, original_node.name)

def eightbitize_conv_node(self, original_node):
  """Replaces a Conv2D node with the eight bit equivalent sub-graph."""
  all_input_names = self.add_eightbit_prologue_nodes(original_node)
  quantized_conv_name = original_node.name + "_eightbit_quantized_conv"
  quantized_conv_node = create_node("QuantizedConv2D", quantized_conv_name,
                                    all_input_names)
  copy_attr(quantized_conv_node, "strides", original_node.attr["strides"])
  copy_attr(quantized_conv_node, "padding", original_node.attr["padding"])
  set_attr_dtype(quantized_conv_node, "Tinput", dtypes.quint8)
  set_attr_dtype(quantized_conv_node, "Tfilter", dtypes.quint8)
  set_attr_dtype(quantized_conv_node, "out_type", dtypes.qint32)
  self.add_output_graph_node(quantized_conv_node)
  quantize_down_name = self.add_quantize_down_nodes(original_node,
                                                    quantized_conv_name)
```

```
    self.add_dequantize_result_node(quantize_down_name, original_node.name)

  def eightbitize_bias_add_node(self, original_node):
    """Replaces a BiasAdd node with the eight bit equivalent sub-graph."""
    quantized_bias_add_name = (
        original_node.name + "_eightbit_quantized_bias_add")
    all_input_names = self.add_eightbit_prologue_nodes(original_node)
    quantized_bias_add_node = create_node("QuantizedBiasAdd",
                                          quantized_bias_add_name,
                                          all_input_names)
    set_attr_dtype(quantized_bias_add_node, "T1", dtypes.quint8)
    set_attr_dtype(quantized_bias_add_node, "T2", dtypes.quint8)
    set_attr_dtype(quantized_bias_add_node, "out_type", dtypes.qint32)
    self.add_output_graph_node(quantized_bias_add_node)
    quantize_down_name = self.add_quantize_down_nodes(original_node,
                                                      quantized_bias_add_name)
    self.add_dequantize_result_node(quantize_down_name, original_node.name)

  def eightbitize_single_input_tensor_node(self, original_node,
                                           add_op_function):
    """Replaces a single-tensor node with the eight bit equivalent sub-graph.

    Converts a node like this:

       Shape(f)   Input(f)
  |         |
  +--------v v
       Operation
          |
          v
         (f)
```

Into a quantized equivalent:

```
          Input(f)          ReshapeDims
             +------v v-------------+
             |   Reshape
             |      |
             |      |         ReductionDims
             |      +-----+        |
             |      | +---c---------+
             |      v v   v v-------+
             |      Min   Max
             |  +----+     |
             v  v  v--------+
           Quantize
              |
              v
         QuantizedOperation
            |  |  |
              v   v   v
              Dequantize
                 |
                 v
                (f)


      Args:
        original_node: Float node to be converted.
        add_op_function: Function to create the actual node.

      Returns:
        Subgraph representing the quantized version of the original node.

      """
      quantized_op_name = original_node.name + "_eightbit_quantized"
      quantized_op_type = "Quantized" + original_node.op
      all_input_names = self.add_eightbit_prologue_nodes(original_node)
      quantized_op_node = create_node(quantized_op_type, quantized_op_name,
                                      all_input_names)
      add_op_function(original_node, quantized_op_node)
      self.add_output_graph_node(quantized_op_node)
      self.add_dequantize_result_node(quantized_op_name, original_node.name)

    def add_pool_function(self, original_node, quantized_op_node):
```

```
  set_attr_dtype(quantized_op_node, "T", dtypes.quint8)
  copy_attr(quantized_op_node, "ksize", original_node.attr["ksize"])
  copy_attr(quantized_op_node, "strides", original_node.attr["strides"])
  copy_attr(quantized_op_node, "padding", original_node.attr["padding"])

def add_relu_function(self, unused_arg_node, quantized_op_node):
  set_attr_dtype(quantized_op_node, "Tinput", dtypes.quint8)

def eightbitize_concat_node(self, original_node):
  """Replaces a Concat node with the eight bit equivalent sub-graph.

  Converts a node like this:

     Shape(f)   Input0(f)   Input1(f)
       |          |            |
       +--------v v v----------+
               Concat
                 |
                 v
                (f)

   Into a quantized equivalent:

     Shape(f)     Input0(f)             ReshapeDims                Input1(f)
       |             +------v v--------------+------------------v v------+
|          |   Reshape                    Reshape   |
|          |     |                          |     |
|          |     |         ReductionDims          |     |
|          |     +------+         |         +--------+     |
|          |     |  +---c---------+-----------c-----+  |     |
|          |     +v v   v v-------+---------v v     v v+     |
|          |      Min   Max              Min     Max      |
|          |  +----+     |              |      +-----+  |
|          v  v  v--------+                +----------v  v  v
|         Quantize                           Quantize
|            +------------------+   +----------------------+
+-------------------------------+   |  |
                         v   v   v
                       QuantizedConcat
                         |   |   |
                         v   v   v
                        Dequantize
                           |
                           v
                          (f)
```

Args:

original_node: Float node to be converted.

Returns:

Subgraph representing the quantized version of the original node.

```
    """
    namespace_prefix = original_node.name + "_eightbit"
    quantized_concat_name = namespace_prefix + "_quantized_concat"
    reshape_dims_name,                    reduction_dims_name                    =
self.add_common_quantization_nodes(
      namespace_prefix)
    shape_input_name = original_node.input[0]
    original_inputs = original_node.input[1:]
    input_names = []
    min_names = []
    max_names = []
    for original_input_name in original_inputs:
      quantize_input_name, min_input_name, max_input_name = (
          self.eightbitize_input_to_node(namespace_prefix, original_input_name,
                                         reshape_dims_name,
                                         reduction_dims_name))
      input_names.append(quantize_input_name)
      min_names.append(min_input_name)
      max_names.append(max_input_name)
    all_input_names = [shape_input_name]
    all_input_names.extend(input_names)
    all_input_names.extend(min_names)
    all_input_names.extend(max_names)
    quantized_concat_node = create_node("QuantizedConcat",
                              quantized_concat_name, all_input_names)
  set_attr_int(quantized_concat_node, "N", len(original_inputs))
  set_attr_dtype(quantized_concat_node, "T", dtypes.quint8)
  self.add_output_graph_node(quantized_concat_node)
  self.add_dequantize_result_node(quantized_concat_name, original_node.name)

def eightbitize_placeholder_node(self, current_node):
  """Replaces a placeholder node with a quint8 placeholder node+dequantize."""
  name = current_node.name

  # Convert the placeholder into a quantized type.
  output_node = node_def_pb2.NodeDef()
  output_node.CopyFrom(current_node)
  set_attr_dtype(output_node, "dtype", dtypes.quint8)
  output_node.name += "_original_input"
  self.add_output_graph_node(output_node)

  # Add a dequantize to convert back to float.
  dequantize_node = create_node("Dequantize", name, [
      output_node.name, "quantized_input_min_value",
      "quantized_input_max_value"
  ])
  set_attr_dtype(dequantize_node, "T", dtypes.quint8)
  set_attr_string(dequantize_node, "mode", b"MIN_FIRST")
  self.add_output_graph_node(dequantize_node)
```

```
    # For the descent over the graph to work, the dequantize node must be named
    # current_node.name.  However, for the feeding of the graph to work, the
    # placeholder must have the name current_node.name; so record a final set
    # of renames to apply after all processing has been done.
    self.final_node_renames[output_node.name] = name
    self.final_node_renames[dequantize_node.name] = name + "_dequantize"

  def eightbitize_reshape_node(self, original_node):
    """Replaces a Reshape node with the eight bit equivalent sub-graph.

    Args:
      original_node: Float node to be converted.

    Returns:
      Subgraph representing the quantized version of the original node.

    """
    namespace_prefix = original_node.name + "_eightbit"
    quantized_reshape_name = namespace_prefix + "_quantized_reshape"
    reshape_dims_name,                    reduction_dims_name                    =
self.add_common_quantization_nodes(
        namespace_prefix)
    shape_input_name = original_node.input[1]
    quantize_input_name, min_input_name, max_input_name = (
        self.eightbitize_input_to_node(namespace_prefix, original_node.input[0],
                                       reshape_dims_name, reduction_dims_name))
    quantized_reshape_node = create_node(
        "QuantizedReshape", quantized_reshape_name,
        [quantize_input_name, shape_input_name, min_input_name, max_input_name])
    set_attr_dtype(quantized_reshape_node, "T", dtypes.quint8)
    self.add_output_graph_node(quantized_reshape_node)
    self.add_dequantize_result_node(quantized_reshape_name, original_node.name)

  def eightbitize_batch_norm_node(self, original_node):
    """Replaces a MatMul node with the eight bit equivalent sub-graph."""
    namespace_prefix = original_node.name + "_eightbit"
    original_input_name = original_node.input[0]
    original_mean_name = original_node.input[1]
    original_variance_name = original_node.input[2]
    original_beta_name = original_node.input[3]
    original_gamma_name = original_node.input[4]
    quantized_batch_norm_name = namespace_prefix + "_quantized_batch_norm"

    reshape_dims_name,                reduction_dims_name                =
self.add_common_quantization_nodes(
        namespace_prefix)
    quantize_input_name, min_input_name, max_input_name = (
        self.eightbitize_input_to_node(namespace_prefix, original_input_name,
                                       reshape_dims_name, reduction_dims_name))
```

```
quantize_mean_name, min_mean_name, max_mean_name = (
    self.eightbitize_input_to_node(namespace_prefix, original_mean_name,
                                   reshape_dims_name, reduction_dims_name))
quantize_variance_name, min_variance_name, max_variance_name = (
    self.eightbitize_input_to_node(namespace_prefix, original_variance_name,
                                   reshape_dims_name, reduction_dims_name))
quantize_beta_name, min_beta_name, max_beta_name = (
    self.eightbitize_input_to_node(namespace_prefix, original_beta_name,
                                   reshape_dims_name, reduction_dims_name))
quantize_gamma_name, min_gamma_name, max_gamma_name = (
    self.eightbitize_input_to_node(namespace_prefix, original_gamma_name,
                                   reshape_dims_name, reduction_dims_name))
quantized_batch_norm_node = create_node(
    "QuantizedBatchNormWithGlobalNormalization", quantized_batch_norm_name,
    [
        quantize_input_name, min_input_name, max_input_name,
        quantize_mean_name, min_mean_name, max_mean_name,
        quantize_variance_name, min_variance_name, max_variance_name,
        quantize_beta_name, min_beta_name, max_beta_name,
        quantize_gamma_name, min_gamma_name, max_gamma_name
    ])
set_attr_dtype(quantized_batch_norm_node, "Tinput", dtypes.quint8)
set_attr_dtype(quantized_batch_norm_node, "out_type", dtypes.qint32)
copy_attr(quantized_batch_norm_node, "scale_after_normalization",
          original_node.attr["scale_after_normalization"])
```

```
  copy_attr(quantized_batch_norm_node, "variance_epsilon",
            original_node.attr["variance_epsilon"])
  self.add_output_graph_node(quantized_batch_norm_node)
  quantize_down_name = self.add_quantize_down_nodes(original_node,
                                                    quantized_batch_norm_name)
  self.add_dequantize_result_node(quantize_down_name, original_node.name)

def add_output_graph_node(self, output_node):
  """Inserts one node into the new graph."""
  self.output_graph.node.extend([output_node])

def remove_redundant_quantization(self, old_graph):
  """Removes unneeded pairs of quantize/dequantize ops from the graph.

  This is a bit of a tricky function, because it's attempting to spot the
  pattern of dequantizing from eight-bit up to float, and then immediately
  quantizing back down to eight bits again, that's introduced by previous
  passes that do 'key-hole' conversions of individual nodes but have to
  convert back to float to match the previous output interface, since they
  don't know that the next op can handle quantized tensors.
  It works by:
   - Looking for Quantize nodes.
   - Checking to see if their first input is a Dequantize node.
   - Seeing if their min/max inputs come from Min/Max nodes.
   - Making sure those Min/Max nodes are being fed from the same Dequantize.
```

```
 - Or that the Min is indirectly being fed from the same Dequantize as Max.
 - Making sure the Dequantize is going through a Reshape (which we add
   during the previous pass when we create the quantize sub-graph).
 - Looking for the dims Const op for the Min/Max dims.
If all of these conditions are met, then it's a sub-graph pattern that
we know how to optimize out (and is likely the common one we've introduced).
We then rewire the graph to skip it entirely, and then rely on the dead node
removal pass to get rid of any nodes that are no longer needed.

Args:
  old_graph: The model we'll be stripping redundant nodes from.

Returns:
  A graph with the unnecessary nodes removed.

Raises:
  ValueError: Two nodes with the same name were found in the graph.
"""
old_nodes_map = self.create_nodes_map(old_graph)
self.output_graph = graph_pb2.GraphDef()
inputs_to_rename = {}
# We go through all the nodes, looking for any that match the patterns we
# know how to optimize away.
for node in old_graph.node:
  # We always start with a Quantize node, and examine its inputs to see if
# they are in a form that can be removed.
if node.op not in ["Quantize", "QuantizeV2"]:
  continue
dequantize_node_name = node_name_from_input(node.input[0])
if dequantize_node_name not in old_nodes_map:
  raise ValueError("Input node name '" + dequantize_node_name +
                   "' not found in node '" + node.name + "'")
dequantize_node = old_nodes_map[dequantize_node_name]
# Do we have a Dequantize feeding in, with the same type as the Quantize?
if dequantize_node.op != "Dequantize":
  continue
if node.attr["T"] != dequantize_node.attr["T"]:
  continue
# Now look at the other inputs, and ensure they're Min/Max nodes.
min_node_name = node_name_from_input(node.input[1])
max_node_name = node_name_from_input(node.input[2])
min_node = old_nodes_map[min_node_name]
max_node = old_nodes_map[max_node_name]
is_min_right_type = (min_node.op in ["Min", "Dequantize"])
is_max_right_type = (max_node.op in ["Max", "Dequantize"])
if not is_min_right_type or not is_max_right_type:
  print("Didn't find expected types on inputs : %s, %s." % (min_node.op,
                                                            max_node.op))
  continue
min_node_input_name = node_name_from_input(min_node.input[0])
```

```
max_node_input_name = node_name_from_input(max_node.input[0])
# There are two different patterns for Min nodes we can recognize, one
# where the input comes directly from the same one as the Max, and
# another where we run it through another Min first, so check for both.
is_same_input = False
if min_node_input_name == max_node_input_name:
  is_same_input = True
else:
  first_min_node_input = old_nodes_map[min_node_input_name]
  if first_min_node_input.op == "Concat":
    second_min_node_name = node_name_from_input(
        first_min_node_input.input[1])
    second_min_node = old_nodes_map[second_min_node_name]
    if second_min_node.op == "Min":
      second_min_node_input_name = node_name_from_input(
          second_min_node.input[0])
      is_same_input = (second_min_node_input_name == max_node_input_name)
if not is_same_input:
  print("Different min/max inputs: " + min_node_input_name)
  continue
# We recognize this pattern, so mark the graph edges to be rewired to
# route around it entirely, since we know it's a no-op.
dequantize_source_name = node_name_from_input(dequantize_node.input[0])
node_tensor_name = ensure_tensor_name_has_port(node.name)
min_tensor_name = node.name + ":1"
```

```
      max_tensor_name = node.name + ":2"
      inputs_to_rename[node_tensor_name] = dequantize_source_name
      inputs_to_rename[min_tensor_name] = dequantize_node.input[1]
      inputs_to_rename[max_tensor_name] = dequantize_node.input[2]
    # Finally we apply all the rewiring we've marked to the graph.
    for node in old_graph.node:
      for index, input_full_name in enumerate(node.input):
        input_name = ensure_tensor_name_has_port(input_full_name)
        if input_name in inputs_to_rename:
          node.input[index] = inputs_to_rename[input_name]
      self.add_output_graph_node(node)
    return self.output_graph

  def apply_final_node_renames(self):
    """Applies node renames in self.final_node_renames to self.output_graph."""
    old_graph = self.output_graph
    self.output_graph = graph_pb2.GraphDef()
    for node in old_graph.node:
      node.name = self.final_node_renames.get(node.name, node.name)
      for index, input_name in enumerate(node.input):
        node_name = node_name_from_input(input_name)
        input_full_name = ensure_tensor_name_has_port(input_name)
        if node_name in self.final_node_renames:
          node.input[index] = "%s%s" % (self.final_node_renames[node_name],
                                        input_full_name[len(node_name):])
      self.add_output_graph_node(node)
    return self.output_graph

  def remove_dead_nodes(self, output_names):
    """Removes nodes that are no longer needed for inference from the graph."""
    old_output_graph = self.output_graph
    self.output_graph = graph_util.extract_sub_graph(old_output_graph,
                                                     output_names)

  def quantize_weights(self, input_graph, quantization_mode):
    """Quantize float Const ops.

    There are two modes of operations, both replace float Const ops with
    quantized values.
    1. If quantization_mode is "weights_rounded", this function replaces float
    Const ops with quantized float Const ops - same as the original op, but
    float values being mapped to the center of one of 1<<FLAGS.bitdepth buckets.
    This does not change the raw model size, but compression algorithms such as
    zip (as used for compressing apks) or bzip2 will achieve a very good
    compression ratio.
    2. For other quantization modes ("MIN_COMBINED" or "MIN_FIRST"), float
    Const ops are quantized and replaced by a tuple of four ops to perform
    the dequantization at runtime:
    * eight-bit Const (bucket indices, same shape as original float Const op
    * two float Const ops (min and max value of original float Const op)
```

```
* Dequantize op to convert the eight-bit consts to float tensors.
The quantization mode is important because we see accuracy problems when
quantizing weights for different situations depending on the algorithm
used. We haven't figured out exactly what the underlying cause is yet,
unfortunately.

Args:
  input_graph: A GraphDef of the model containing float Const ops.
  quantization_mode: How to quantize and dequantize the values.

Returns:
  A GraphDef of the converted graph.

Raises:
  ValueError: If quantization_mode is unsupported.
"""
output_graph = graph_pb2.GraphDef()
for input_node in input_graph.node:
  should_quantize = False
  if input_node.op == "Const":
    dtype = dtypes.as_dtype(input_node.attr["dtype"].type)
    if dtype == dtypes.float32:
      should_quantize = True
  if should_quantize:
    if quantization_mode == "weights_rounded":
          output_graph.node.extend(quantize_weight_rounded(input_node))
        elif quantization_mode in (b"MIN_COMBINED", b"MIN_FIRST"):
          output_graph.node.extend(
              quantize_weight_eightbit(input_node, quantization_mode))
        else:
          raise ValueError("Unsupported quantization mode %s." %
                           quantization_mode)
      else:
        output_node = node_def_pb2.NodeDef()
        output_node.CopyFrom(input_node)
        output_graph.node.extend([output_node])
    return output_graph

  def set_input_graph(self, new_input_graph):
    self.input_graph = new_input_graph
    self.nodes_map = self.create_nodes_map(self.input_graph)


def main(unused_args):
  if not gfile.Exists(FLAGS.input):
    print("Input graph file '" + FLAGS.input + "' does not exist!")
    return -1

  known_modes = [
      "round", "quantize", "eightbit", "weights", "test", "weights_rounded"
]
if not any(FLAGS.mode in s for s in known_modes):
  print("mode is '" + FLAGS.mode + "', not in " + ", ".join(known_modes) +
        ".")
  return -1

tf_graph = graph_pb2.GraphDef()
with gfile.Open(FLAGS.input, "rb") as f:
  data = f.read()
  tf_graph.ParseFromString(data)

graph = ops.Graph()
with graph.as_default():
  importer.import_graph_def(tf_graph, input_map={}, name="")

quantized_input_range = None
if FLAGS.quantized_input:
  quantized_input_range = [
      FLAGS.quantized_input_min, FLAGS.quantized_input_max
  ]

fallback_quantization_range = None
if (FLAGS.quantized_fallback_min is not None or
    FLAGS.quantized_fallback_max is not None):
  assert FLAGS.quantized_fallback_min is not None
```

```
    assert FLAGS.quantized_fallback_max is not None
    fallback_quantization_range = [
        FLAGS.quantized_fallback_min, FLAGS.quantized_fallback_max
    ]

  rewriter = GraphRewriter(tf_graph, FLAGS.mode, quantized_input_range,
                           fallback_quantization_range)

  output_graph = rewriter.rewrite(FLAGS.output_node_names.split(","))

  f = gfile.FastGFile(FLAGS.output, "wb")
  f.write(output_graph.SerializeToString())

  return 0


if __name__ == "__main__":
  os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
  app.run()
```

retrain.py

```
# Copyright 2015 The TensorFlow Authors. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
=====================================================
============================
r"""Simple transfer learning with Inception v3 or Mobilenet models.

With support for TensorBoard.

This example shows how to take a Inception v3 or Mobilenet model trained on
```

ImageNet images, and train a new top layer that can recognize other classes of images.

*The top layer receives as input a 2048-dimensional vector (1001-dimensional for Mobilenet) for each image. We train a softmax layer on top of this representation. Assuming the softmax layer contains N labels, this corresponds to learning N + 2048*N (or 1001*N) model parameters corresponding to the learned biases and weights.*

Here's an example, which assumes you have a folder containing class-named subfolders, each full of images for each label. The example folder flower_photos should have a structure like this:

```
~/flower_photos/daisy/photo1.jpg
~/flower_photos/daisy/photo2.jpg
...
~/flower_photos/rose/anotherphoto77.jpg
...
~/flower_photos/sunflower/somepicture.jpg
```

The subfolder names are important, since they define what label is applied to each image, but the filenames themselves don't matter. Once your images are prepared, you can run the training with a command like this:

```bash
bazel build tensorflow/examples/image_retraining:retrain && \
bazel-bin/tensorflow/examples/image_retraining/retrain \
    --image_dir ~/flower_photos
```

Or, if you have a pip installation of tensorflow, `retrain.py` can be run
without bazel:

```bash
python tensorflow/examples/image_retraining/retrain.py \
    --image_dir ~/flower_photos
```

You can replace the image_dir argument with any folder containing subfolders of
images. The label for each image is taken from the name of the subfolder it's
in.

This produces a new model file that can be loaded and run by any TensorFlow
program, for example the label_image sample code.

By default this script will use the high accuracy, but comparatively large and
slow Inception v3 model architecture. It's recommended that you start with this
to validate that you have gathered good training data, but if you want to deploy
on resource-limited platforms, you can try the `--architecture` flag with a

~~~
Mobilenet model. For example:

```bash
python tensorflow/examples/image_retraining/retrain.py \
    --image_dir ~/flower_photos --architecture mobilenet_1.0_224
```

There are 32 different Mobilenet models to choose from, with a variety of file
size and latency options. The first number can be '1.0', '0.75', '0.50', or
'0.25' to control the size, and the second controls the input image size, either
'224', '192', '160', or '128', with smaller sizes running faster. See
https://research.googleblog.com/2017/06/mobilenets-open-source-models-for.html
for more information on Mobilenet.

To use with TensorBoard:

By default, this script will log summaries to /tmp/retrain_logs directory

Visualize the summaries with this command:

tensorboard --logdir /tmp/retrain_logs

"""
from __future__ import absolute_import
from __future__ import division
~~~

```
from __future__ import print_function

import argparse
import collections
from datetime import datetime
import hashlib
import os.path
import random
import re
import sys
import tarfile

import numpy as np
from six.moves import urllib
import tensorflow as tf

from tensorflow.python.framework import graph_util
from tensorflow.python.framework import tensor_shape
from tensorflow.python.platform import gfile
from tensorflow.python.util import compat

FLAGS = None

# These are all parameters that are tied to the particular model architecture
# we're using for Inception v3. These include things like tensor names and their
# sizes. If you want to adapt this script to work with another model, you will
# need to update these to reflect the values in the network you're using.
MAX_NUM_IMAGES_PER_CLASS = 2 ** 27 - 1  # ~134M



def create_image_lists(image_dir, testing_percentage, validation_percentage):
  """Builds a list of training images from the file system.

  Analyzes the sub folders in the image directory, splits them into stable
  training, testing, and validation sets, and returns a data structure
  describing the lists of images for each label and their paths.

  Args:
    image_dir: String path to a folder containing subfolders of images.
    testing_percentage: Integer percentage of the images to reserve for tests.
    validation_percentage: Integer percentage of images reserved for validation.

  Returns:
    A dictionary containing an entry for each label subfolder, with images split
    into training, testing, and validation sets within each label.
  """
  if not gfile.Exists(image_dir):
    tf.logging.error("Image directory '" + image_dir + "' not found.")
    return None
  result = collections.OrderedDict()
```

```
sub_dirs = [
  os.path.join(image_dir,item)
  for item in gfile.ListDirectory(image_dir)]
sub_dirs = sorted(item for item in sub_dirs
                  if gfile.IsDirectory(item))
for sub_dir in sub_dirs:
  extensions = ['jpg', 'jpeg', 'JPG', 'JPEG']
  file_list = []
  dir_name = os.path.basename(sub_dir)
  if dir_name == image_dir:
    continue
  tf.logging.info("Looking for images in '" + dir_name + "'")
  for extension in extensions:
    file_glob = os.path.join(image_dir, dir_name, '*.' + extension)
    file_list.extend(gfile.Glob(file_glob))
  if not file_list:
    tf.logging.warning('No files found')
    continue
  if len(file_list) < 20:
    tf.logging.warning(
        'WARNING: Folder has less than 20 images, which may cause issues.')
  elif len(file_list) > MAX_NUM_IMAGES_PER_CLASS:
    tf.logging.warning(
        'WARNING: Folder {} has more than {} images. Some images will '
        'never be selected.'.format(dir_name, MAX_NUM_IMAGES_PER_CLASS))
```

```
label_name = re.sub(r'[^a-z0-9]+', ' ', dir_name.lower())
training_images = []
testing_images = []
validation_images = []
for file_name in file_list:
  base_name = os.path.basename(file_name)
  # We want to ignore anything after '_nohash_' in the file name when
  # deciding which set to put an image in, the data set creator has a way of
  # grouping photos that are close variations of each other. For example
  # this is used in the plant disease data set to group multiple pictures of
  # the same leaf.
  hash_name = re.sub(r'_nohash_.*$', '', file_name)
  # This looks a bit magical, but we need to decide whether this file should
  # go into the training, testing, or validation sets, and we want to keep
  # existing files in the same set even if more files are subsequently
  # added.
  # To do that, we need a stable way of deciding based on just the file name
  # itself, so we do a hash of that and then use that to generate a
  # probability value that we use to assign it.
  hash_name_hashed = hashlib.sha1(compat.as_bytes(hash_name)).hexdigest()
  percentage_hash = ((int(hash_name_hashed, 16) %
                      (MAX_NUM_IMAGES_PER_CLASS + 1)) *
                     (100.0 / MAX_NUM_IMAGES_PER_CLASS))
  if percentage_hash < validation_percentage:
    validation_images.append(base_name)
```

```
      elif percentage_hash < (testing_percentage + validation_percentage):
        testing_images.append(base_name)
      else:
        training_images.append(base_name)
    result[label_name] = {
        'dir': dir_name,
        'training': training_images,
        'testing': testing_images,
        'validation': validation_images,
    }
  return result


def get_image_path(image_lists, label_name, index, image_dir, category):
  """"Returns a path to an image for a label at the given index.

  Args:
    image_lists: Dictionary of training images for each label.
    label_name: Label string we want to get an image for.
    index: Int offset of the image we want. This will be moduloed by the
    available number of images for the label, so it can be arbitrarily large.
    image_dir: Root folder string of the subfolders containing the training
    images.
    category: Name string of set to pull images from - training, testing, or
    validation.
  Returns:
    File system path string to an image that meets the requested parameters.

  """
  if label_name not in image_lists:
    tf.logging.fatal('Label does not exist %s.', label_name)
  label_lists = image_lists[label_name]
  if category not in label_lists:
    tf.logging.fatal('Category does not exist %s.', category)
  category_list = label_lists[category]
  if not category_list:
    tf.logging.fatal('Label %s has no images in the category %s.',
                     label_name, category)
  mod_index = index % len(category_list)
  base_name = category_list[mod_index]
  sub_dir = label_lists['dir']
  full_path = os.path.join(image_dir, sub_dir, base_name)
  return full_path


def get_bottleneck_path(image_lists, label_name, index, bottleneck_dir,
                        category, architecture):
  """"Returns a path to a bottleneck file for a label at the given index.
  Args:
    image_lists: Dictionary of training images for each label.
    label_name: Label string we want to get an image for.
    index: Integer offset of the image we want. This will be moduloed by the
    available number of images for the label, so it can be arbitrarily large.
    bottleneck_dir: Folder string holding cached files of bottleneck values.
    category: Name string of set to pull images from - training, testing, or
    validation.
    architecture: The name of the model architecture.

  Returns:
    File system path string to an image that meets the requested parameters.
  """
  return get_image_path(image_lists, label_name, index, bottleneck_dir,
                        category) + '_' + architecture + '.txt'



def create_model_graph(model_info):
  """"Creates a graph from saved GraphDef file and returns a Graph object.

  Args:
    model_info: Dictionary containing information about the model architecture.

  Returns:
    Graph holding the trained Inception network, and various tensors we'll be
    manipulating.
  """
  with tf.Graph().as_default() as graph:
    model_path = os.path.join(FLAGS.model_dir, model_info['model_file_name'])
    with gfile.FastGFile(model_path, 'rb') as f:
      graph_def = tf.GraphDef()
      graph_def.ParseFromString(f.read())
      bottleneck_tensor, resized_input_tensor = (tf.import_graph_def(
          graph_def,
          name='',
          return_elements=[
              model_info['bottleneck_tensor_name'],
              model_info['resized_input_tensor_name'],
          ]))
  return graph, bottleneck_tensor, resized_input_tensor


def run_bottleneck_on_image(sess, image_data, image_data_tensor,
                            decoded_image_tensor, resized_input_tensor,
                            bottleneck_tensor):
  """Runs inference on an image to extract the 'bottleneck' summary layer.

  Args:
    sess: Current active TensorFlow Session.
    image_data: String of raw JPEG data.
    image_data_tensor: Input data layer in the graph.
    decoded_image_tensor: Output of initial image resizing and  preprocessing.
    resized_input_tensor: The input node of the recognition graph.
    bottleneck_tensor: Layer before the final softmax.

  Returns:
    Numpy array of bottleneck values.
  """
  # First decode the JPEG image, resize it, and rescale the pixel values.
  resized_input_values = sess.run(decoded_image_tensor,
                                  {image_data_tensor: image_data})
  # Then run it through the recognition network.
  bottleneck_values = sess.run(bottleneck_tensor,
                               {resized_input_tensor: resized_input_values})
  bottleneck_values = np.squeeze(bottleneck_values)
  return bottleneck_values


def maybe_download_and_extract(data_url):
  """Download and extract model tar file.

  If the pretrained model we're using doesn't already exist, this function
  downloads it from the TensorFlow.org website and unpacks it into a directory.

  Args:
    data_url: Web location of the tar file containing the pretrained model.
  """
  dest_directory = FLAGS.model_dir
  if not os.path.exists(dest_directory):
    os.makedirs(dest_directory)
  filename = data_url.split('/')[-1]
  filepath = os.path.join(dest_directory, filename)
  if not os.path.exists(filepath):

    def _progress(count, block_size, total_size):
      sys.stdout.write('\r>> Downloading %s %.1f%%' %
                       (filename,
                        float(count * block_size) / float(total_size) * 100.0))
      sys.stdout.flush()

    filepath, _ = urllib.request.urlretrieve(data_url, filepath, _progress)
    print()
    statinfo = os.stat(filepath)
    tf.logging.info('Successfully downloaded', filename, statinfo.st_size,
                    'bytes.')
  tarfile.open(filepath, 'r:gz').extractall(dest_directory)


def ensure_dir_exists(dir_name):
  """Makes sure the folder exists on disk.
  Args:
    dir_name: Path string to the folder we want to create.
  """
  if not os.path.exists(dir_name):
    os.makedirs(dir_name)


bottleneck_path_2_bottleneck_values = {}


def create_bottleneck_file(bottleneck_path, image_lists, label_name, index,
                           image_dir, category, sess, jpeg_data_tensor,
                           decoded_image_tensor, resized_input_tensor,
                           bottleneck_tensor):
  """Create a single bottleneck file."""
  tf.logging.info('Creating bottleneck at ' + bottleneck_path)
  image_path = get_image_path(image_lists, label_name, index,
                              image_dir, category)
  if not gfile.Exists(image_path):
    tf.logging.fatal('File does not exist %s', image_path)
  image_data = gfile.FastGFile(image_path, 'rb').read()
  try:
    bottleneck_values = run_bottleneck_on_image(
        sess, image_data, jpeg_data_tensor, decoded_image_tensor,
        resized_input_tensor, bottleneck_tensor)
  except Exception as e:
    raise RuntimeError('Error during processing file %s (%s)' % (image_path,
                                                                 str(e)))
  bottleneck_string = ','.join(str(x) for x in bottleneck_values)
  with open(bottleneck_path, 'w') as bottleneck_file:
    bottleneck_file.write(bottleneck_string)



def get_or_create_bottleneck(sess, image_lists, label_name, index, image_dir,
                             category, bottleneck_dir, jpeg_data_tensor,
                             decoded_image_tensor, resized_input_tensor,
                             bottleneck_tensor, architecture):
  """Retrieves or calculates bottleneck values for an image.

  If a cached version of the bottleneck data exists on-disk, return that,
  otherwise calculate the data and save it to disk for future use.

  Args:
    sess: The current active TensorFlow Session.
    image_lists: Dictionary of training images for each label.
    label_name: Label string we want to get an image for.
    index: Integer offset of the image we want. This will be modulo-ed by the
    available number of images for the label, so it can be arbitrarily large.
    image_dir: Root folder string  of the subfolders containing the training
    images.
    category: Name string of which  set to pull images from - training, testing,
    or validation.
    bottleneck_dir: Folder string holding cached files of bottleneck values.
    jpeg_data_tensor: The tensor to feed loaded jpeg data into.
    decoded_image_tensor: The output of decoding and resizing the image.
    resized_input_tensor: The input node of the recognition graph.
    bottleneck_tensor: The output tensor for the bottleneck values.
    architecture: The name of the model architecture.

  Returns:
    Numpy array of values produced by the bottleneck layer for the image.
  """
  label_lists = image_lists[label_name]
  sub_dir = label_lists['dir']
  sub_dir_path = os.path.join(bottleneck_dir, sub_dir)
  ensure_dir_exists(sub_dir_path)
  bottleneck_path = get_bottleneck_path(image_lists, label_name, index,
                                        bottleneck_dir, category, architecture)
  if not os.path.exists(bottleneck_path):
    create_bottleneck_file(bottleneck_path, image_lists, label_name, index,
                           image_dir, category, sess, jpeg_data_tensor,
                           decoded_image_tensor, resized_input_tensor,
                           bottleneck_tensor)
  with open(bottleneck_path, 'r') as bottleneck_file:
      bottleneck_string = bottleneck_file.read()
    did_hit_error = False
    try:
      bottleneck_values = [float(x) for x in bottleneck_string.split(',')]
    except ValueError:
      tf.logging.warning('Invalid float found, recreating bottleneck')
      did_hit_error = True
    if did_hit_error:
      create_bottleneck_file(bottleneck_path, image_lists, label_name, index,
                             image_dir, category, sess, jpeg_data_tensor,
                             decoded_image_tensor, resized_input_tensor,
                             bottleneck_tensor)
      with open(bottleneck_path, 'r') as bottleneck_file:
        bottleneck_string = bottleneck_file.read()
      # Allow exceptions to propagate here, since they shouldn't happen after a
      # fresh creation
      bottleneck_values = [float(x) for x in bottleneck_string.split(',')]
    return bottleneck_values


  def cache_bottlenecks(sess, image_lists, image_dir, bottleneck_dir,
                        jpeg_data_tensor, decoded_image_tensor,
                        resized_input_tensor, bottleneck_tensor, architecture):
    """Ensures all the training, testing, and validation bottlenecks are cached.
```

```
Because we're likely to read the same image multiple times (if there are no
distortions applied during training) it can speed things up a lot if we
calculate the bottleneck layer values once for each image during
preprocessing, and then just read those cached values repeatedly during
training. Here we go through all the images we've found, calculate those
values, and save them off.

Args:
  sess: The current active TensorFlow Session.
  image_lists: Dictionary of training images for each label.
  image_dir: Root folder string of the subfolders containing the training
  images.
  bottleneck_dir: Folder string holding cached files of bottleneck values.
  jpeg_data_tensor: Input tensor for jpeg data from file.
  decoded_image_tensor: The output of decoding and resizing the image.
  resized_input_tensor: The input node of the recognition graph.
  bottleneck_tensor: The penultimate output layer of the graph.
  architecture: The name of the model architecture.

Returns:
  Nothing.
"""
how_many_bottlenecks = 0
ensure_dir_exists(bottleneck_dir)
for label_name, label_lists in image_lists.items():
  for category in ['training', 'testing', 'validation']:
   category_list = label_lists[category]
   for index, unused_base_name in enumerate(category_list):
    get_or_create_bottleneck(
      sess, image_lists, label_name, index, image_dir, category,
      bottleneck_dir, jpeg_data_tensor, decoded_image_tensor,
      resized_input_tensor, bottleneck_tensor, architecture)

    how_many_bottlenecks += 1
    if how_many_bottlenecks % 100 == 0:
     tf.logging.info(
       str(how_many_bottlenecks) + ' bottleneck files created.')



def get_random_cached_bottlenecks(sess, image_lists, how_many, category,
                 bottleneck_dir, image_dir, jpeg_data_tensor,
                 decoded_image_tensor, resized_input_tensor,
                 bottleneck_tensor, architecture):
 """Retrieves bottleneck values for cached images.

 If no distortions are being applied, this function can retrieve the cached
 bottleneck values directly from disk for images. It picks a random set of
 images from the specified category.

 Args:
  sess: Current TensorFlow Session.
  image_lists: Dictionary of training images for each label.
  how_many: If positive, a random sample of this size will be chosen.
  If negative, all bottlenecks will be retrieved.
  category: Name string of which set to pull from - training, testing, or
  validation.
  bottleneck_dir: Folder string holding cached files of bottleneck values.
  image_dir: Root folder string of the subfolders containing the training
  images.
  jpeg_data_tensor: The layer to feed jpeg image data into.
  decoded_image_tensor: The output of decoding and resizing the image.
  resized_input_tensor: The input node of the recognition graph.
  bottleneck_tensor: The bottleneck output layer of the CNN graph.
  architecture: The name of the model architecture.

Returns:
  List of bottleneck arrays, their corresponding ground truths, and the
  relevant filenames.
"""
class_count = len(image_lists.keys())
bottlenecks = []
ground_truths = []
filenames = []
if how_many >= 0:
  # Retrieve a random sample of bottlenecks.
  for unused_i in range(how_many):
    label_index = random.randrange(class_count)
    label_name = list(image_lists.keys())[label_index]
    image_index = random.randrange(MAX_NUM_IMAGES_PER_CLASS + 1)
    image_name = get_image_path(image_lists, label_name, image_index,
                                image_dir, category)
    bottleneck = get_or_create_bottleneck(
        sess, image_lists, label_name, image_index, image_dir, category,
        bottleneck_dir, jpeg_data_tensor, decoded_image_tensor,
        resized_input_tensor, bottleneck_tensor, architecture)
    ground_truth = np.zeros(class_count, dtype=np.float32)
    ground_truth[label_index] = 1.0
    bottlenecks.append(bottleneck)
    ground_truths.append(ground_truth)
    filenames.append(image_name)
else:
  # Retrieve all bottlenecks.
  for label_index, label_name in enumerate(image_lists.keys()):
    for image_index, image_name in enumerate(
        image_lists[label_name][category]):
      image_name = get_image_path(image_lists, label_name, image_index,
                                  image_dir, category)
      bottleneck = get_or_create_bottleneck(
          sess, image_lists, label_name, image_index, image_dir, category,
          bottleneck_dir, jpeg_data_tensor, decoded_image_tensor,
        resized_input_tensor, bottleneck_tensor, architecture)
      ground_truth = np.zeros(class_count, dtype=np.float32)
      ground_truth[label_index] = 1.0
      bottlenecks.append(bottleneck)
      ground_truths.append(ground_truth)
      filenames.append(image_name)
  return bottlenecks, ground_truths, filenames


def get_random_distorted_bottlenecks(
    sess, image_lists, how_many, category, image_dir, input_jpeg_tensor,
    distorted_image, resized_input_tensor, bottleneck_tensor):
  """Retrieves bottleneck values for training images, after distortions.

  If we're training with distortions like crops, scales, or flips, we have to
  recalculate the full model for every image, and so we can't use cached
  bottleneck values. Instead we find random images for the requested category,
  run them through the distortion graph, and then the full graph to get the
  bottleneck results for each.

  Args:
    sess: Current TensorFlow Session.
    image_lists: Dictionary of training images for each label.
    how_many: The integer number of bottleneck values to return.
    category: Name string of which set of images to fetch - training, testing,
  or validation.
  image_dir: Root folder string of the subfolders containing the training
  images.
  input_jpeg_tensor: The input layer we feed the image data to.
  distorted_image: The output node of the distortion graph.
  resized_input_tensor: The input node of the recognition graph.
  bottleneck_tensor: The bottleneck output layer of the CNN graph.

Returns:
  List of bottleneck arrays and their corresponding ground truths.
"""
class_count = len(image_lists.keys())
bottlenecks = []
ground_truths = []
for unused_i in range(how_many):
  label_index = random.randrange(class_count)
  label_name = list(image_lists.keys())[label_index]
  image_index = random.randrange(MAX_NUM_IMAGES_PER_CLASS + 1)
  image_path = get_image_path(image_lists, label_name, image_index, image_dir,
                              category)
  if not gfile.Exists(image_path):
    tf.logging.fatal('File does not exist %s', image_path)
  jpeg_data = gfile.FastGFile(image_path, 'rb').read()
  # Note that we materialize the distorted_image_data as a numpy array before
  # sending running inference on the image. This involves 2 memory copies and
```

```
    # might be optimized in other implementations.
    distorted_image_data = sess.run(distorted_image,
                                    {input_jpeg_tensor: jpeg_data})
    bottleneck_values = sess.run(bottleneck_tensor,
                                 {resized_input_tensor: distorted_image_data})
    bottleneck_values = np.squeeze(bottleneck_values)
    ground_truth = np.zeros(class_count, dtype=np.float32)
    ground_truth[label_index] = 1.0
    bottlenecks.append(bottleneck_values)
    ground_truths.append(ground_truth)
  return bottlenecks, ground_truths


def should_distort_images(flip_left_right, random_crop, random_scale,
                          random_brightness):
  """Whether any distortions are enabled, from the input flags.

  Args:
    flip_left_right: Boolean whether to randomly mirror images horizontally.
    random_crop: Integer percentage setting the total margin used around the
    crop box.
    random_scale: Integer percentage of how much to vary the scale by.
    random_brightness: Integer range to randomly multiply the pixel values by.

  Returns:
    Boolean value indicating whether any distortions should be applied.
  """
  return (flip_left_right or (random_crop != 0) or (random_scale != 0) or
          (random_brightness != 0))


def add_input_distortions(flip_left_right, random_crop, random_scale,
                          random_brightness, input_width, input_height,
                          input_depth, input_mean, input_std):
  """Creates the operations to apply the specified distortions.

  During training it can help to improve the results if we run the images
  through simple distortions like crops, scales, and flips. These reflect the
  kind of variations we expect in the real world, and so can help train the
  model to cope with natural data more effectively. Here we take the supplied
  parameters and construct a network of operations to apply them to an image.

  Cropping
  ~~~~~~~~

  Cropping is done by placing a bounding box at a random position in the full
  image. The cropping parameter controls the size of that box relative to the
  input image. If it's zero, then the box is the same size as the input and no
  cropping is performed. If the value is 50%, then the crop box will be half the
  width and height of the input. In a diagram it looks like this:
```

```
<      width       >
+---------------------+
|                 |
|  width - crop%     |
|    <      >        |
|    +------+        |
|    |      |       |
|    |      |       |
|    |      |       |
|    +------+        |
|                 |
|                 |
+---------------------+
```

Scaling

*~~~~~~~*

Scaling is a lot like cropping, except that the bounding box is always centered and its size varies randomly within the given range. For example if the scale percentage is zero, then the bounding box is the same size as the input and no scaling is applied. If it's 50%, then the bounding box will be in a random range between half the width and height and full size.

Args:

```
  flip_left_right: Boolean whether to randomly mirror images horizontally.
  random_crop: Integer percentage setting the total margin used around the
  crop box.
  random_scale: Integer percentage of how much to vary the scale by.
  random_brightness: Integer range to randomly multiply the pixel values by.
  graph.
  input_width: Horizontal size of expected input image to model.
  input_height: Vertical size of expected input image to model.
  input_depth: How many channels the expected input image should have.
  input_mean: Pixel value that should be zero in the image for the graph.
  input_std: How much to divide the pixel values by before recognition.

Returns:
  The jpeg input layer and the distorted result tensor.
"""

jpeg_data = tf.placeholder(tf.string, name='DistortJPGInput')
decoded_image = tf.image.decode_jpeg(jpeg_data, channels=input_depth)
decoded_image_as_float = tf.cast(decoded_image, dtype=tf.float32)
decoded_image_4d = tf.expand_dims(decoded_image_as_float, 0)
margin_scale = 1.0 + (random_crop / 100.0)
resize_scale = 1.0 + (random_scale / 100.0)
margin_scale_value = tf.constant(margin_scale)
resize_scale_value = tf.random_uniform(tensor_shape.scalar(),
                                       minval=1.0,
                          maxval=resize_scale)
scale_value = tf.multiply(margin_scale_value, resize_scale_value)
precrop_width = tf.multiply(scale_value, input_width)
precrop_height = tf.multiply(scale_value, input_height)
precrop_shape = tf.stack([precrop_height, precrop_width])
precrop_shape_as_int = tf.cast(precrop_shape, dtype=tf.int32)
precropped_image = tf.image.resize_bilinear(decoded_image_4d,
                            precrop_shape_as_int)
precropped_image_3d = tf.squeeze(precropped_image, squeeze_dims=[0])
cropped_image = tf.random_crop(precropped_image_3d,
                    [input_height, input_width, input_depth])
if flip_left_right:
  flipped_image = tf.image.random_flip_left_right(cropped_image)
else:
  flipped_image = cropped_image
brightness_min = 1.0 - (random_brightness / 100.0)
brightness_max = 1.0 + (random_brightness / 100.0)
brightness_value = tf.random_uniform(tensor_shape.scalar(),
                         minval=brightness_min,
                         maxval=brightness_max)
brightened_image = tf.multiply(flipped_image, brightness_value)
offset_image = tf.subtract(brightened_image, input_mean)
mul_image = tf.multiply(offset_image, 1.0 / input_std)
distort_result = tf.expand_dims(mul_image, 0, name='DistortResult')
return jpeg_data, distort_result
```

```
def variable_summaries(var):
  """Attach a lot of summaries to a Tensor (for TensorBoard visualization)."""
  with tf.name_scope('summaries'):
    mean = tf.reduce_mean(var)
    tf.summary.scalar('mean', mean)
    with tf.name_scope('stddev'):
      stddev = tf.sqrt(tf.reduce_mean(tf.square(var - mean)))
    tf.summary.scalar('stddev', stddev)
    tf.summary.scalar('max', tf.reduce_max(var))
    tf.summary.scalar('min', tf.reduce_min(var))
    tf.summary.histogram('histogram', var)


def add_final_training_ops(class_count, final_tensor_name, bottleneck_tensor,
                           bottleneck_tensor_size):
  """Adds a new softmax and fully-connected layer for training.

  We need to retrain the top layer to identify our new classes, so this function
  adds the right operations to the graph, along with some variables to hold the
  weights, and then sets up all the gradients for the backward pass.

  The set up for the softmax and fully-connected layers is based on:
  https://www.tensorflow.org/versions/master/tutorials/mnist/beginners/index.html
```

```
Args:
  class_count: Integer of how many categories of things we're trying to
  recognize.
  final_tensor_name: Name string for the new final node that produces results.
  bottleneck_tensor: The output of the main CNN graph.
  bottleneck_tensor_size: How many entries in the bottleneck vector.

Returns:
  The tensors for the training and cross entropy results, and tensors for the
  bottleneck input and ground truth input.
"""
with tf.name_scope('input'):
  bottleneck_input = tf.placeholder_with_default(
      bottleneck_tensor,
      shape=[None, bottleneck_tensor_size],
      name='BottleneckInputPlaceholder')

  ground_truth_input = tf.placeholder(tf.float32,
                                      [None, class_count],
                                      name='GroundTruthInput')

# Organizing the following ops as `final_training_ops` so they're easier
# to see in TensorBoard
layer_name = 'final_training_ops'
```

```
with tf.name_scope(layer_name):
  with tf.name_scope('weights'):
    initial_value = tf.truncated_normal(
        [bottleneck_tensor_size, class_count], stddev=0.001)

    layer_weights = tf.Variable(initial_value, name='final_weights')

    variable_summaries(layer_weights)
  with tf.name_scope('biases'):
    layer_biases = tf.Variable(tf.zeros([class_count]), name='final_biases')
    variable_summaries(layer_biases)
  with tf.name_scope('Wx_plus_b'):
    logits = tf.matmul(bottleneck_input, layer_weights) + layer_biases
    tf.summary.histogram('pre_activations', logits)

final_tensor = tf.nn.softmax(logits, name=final_tensor_name)
tf.summary.histogram('activations', final_tensor)

with tf.name_scope('cross_entropy'):
  cross_entropy = tf.nn.softmax_cross_entropy_with_logits(
      labels=ground_truth_input, logits=logits)
  with tf.name_scope('total'):
    cross_entropy_mean = tf.reduce_mean(cross_entropy)
tf.summary.scalar('cross_entropy', cross_entropy_mean)
```

```
  with tf.name_scope('train'):
    optimizer = tf.train.GradientDescentOptimizer(FLAGS.learning_rate)
    train_step = optimizer.minimize(cross_entropy_mean)

  return (train_step, cross_entropy_mean, bottleneck_input, ground_truth_input,
          final_tensor)


def add_evaluation_step(result_tensor, ground_truth_tensor):
  """Inserts the operations we need to evaluate the accuracy of our results.

  Args:
    result_tensor: The new final node that produces results.
    ground_truth_tensor: The node we feed ground truth data
    into.

  Returns:
    Tuple of (evaluation step, prediction).
  """
  with tf.name_scope('accuracy'):
    with tf.name_scope('correct_prediction'):
      prediction = tf.argmax(result_tensor, 1)
      correct_prediction = tf.equal(
          prediction, tf.argmax(ground_truth_tensor, 1))
    with tf.name_scope('accuracy'):
```

```
    evaluation_step = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
  tf.summary.scalar('accuracy', evaluation_step)
  return evaluation_step, prediction



def save_graph_to_file(sess, graph, graph_file_name):
  output_graph_def = graph_util.convert_variables_to_constants(
      sess, graph.as_graph_def(), [FLAGS.final_tensor_name])
  with gfile.FastGFile(graph_file_name, 'wb') as f:
    f.write(output_graph_def.SerializeToString())
  return



def prepare_file_system():
  # Setup the directory we'll write summaries to for TensorBoard
  if tf.gfile.Exists(FLAGS.summaries_dir):
    tf.gfile.DeleteRecursively(FLAGS.summaries_dir)
  tf.gfile.MakeDirs(FLAGS.summaries_dir)
  if FLAGS.intermediate_store_frequency > 0:
    ensure_dir_exists(FLAGS.intermediate_output_graphs_dir)
  return



def create_model_info(architecture):
  """Given the name of a model architecture, returns information about it.
  There are different base image recognition pretrained models that can be
  retrained using transfer learning, and this function translates from the name
  of a model to the attributes that are needed to download and train with it.

  Args:
    architecture: Name of a model architecture.

  Returns:
    Dictionary of information about the model, or None if the name isn't
    recognized

  Raises:
    ValueError: If architecture name is unknown.
  """
  architecture = architecture.lower()
  if architecture == 'inception_v3':
    # pylint: disable=line-too-long
    data_url    =    'http://download.tensorflow.org/models/image/imagenet/inception-
2015-12-05.tgz'
    # pylint: enable=line-too-long
    bottleneck_tensor_name = 'pool_3/_reshape:0'
    bottleneck_tensor_size = 2048
    input_width = 299
    input_height = 299
  input_depth = 3
  resized_input_tensor_name = 'Mul:0'
  model_file_name = 'classify_image_graph_def.pb'
  input_mean = 128
  input_std = 128
elif architecture.startswith('mobilenet_'):
  parts = architecture.split('_')
  if len(parts) != 3 and len(parts) != 4:
    tf.logging.error("Couldn't understand architecture name '%s'",
                     architecture)
    return None
  version_string = parts[1]
  if (version_string != '1.0' and version_string != '0.75' and
      version_string != '0.50' and version_string != '0.25'):
    tf.logging.error(
        """"The Mobilenet version should be '1.0', '0.75', '0.50', or '0.25',
but found '%s' for architecture '%s'""",
        version_string, architecture)
    return None
  size_string = parts[2]
  if (size_string != '224' and size_string != '192' and
      size_string != '160' and size_string != '128'):
    tf.logging.error(
        """The Mobilenet input size should be '224', '192', '160', or '128',
but found '%s' for architecture '%s'""",
      size_string, architecture)
  return None
if len(parts) == 3:
  is_quantized = False
else:
  if parts[3] != 'quantized':
    tf.logging.error(
        "Couldn't understand architecture suffix '%s' for '%s'", parts[3],
        architecture)
    return None
  is_quantized = True
data_url = 'http://download.tensorflow.org/models/mobilenet_v1_'
data_url += version_string + '_' + size_string + '_frozen.tgz'
bottleneck_tensor_name = 'MobilenetV1/Predictions/Reshape:0'
bottleneck_tensor_size = 1001
input_width = int(size_string)
input_height = int(size_string)
input_depth = 3
resized_input_tensor_name = 'input:0'
if is_quantized:
  model_base_name = 'quantized_graph.pb'
else:
  model_base_name = 'frozen_graph.pb'
model_dir_name = 'mobilenet_v1_' + version_string + '_' + size_string
model_file_name = os.path.join(model_dir_name, model_base_name)
```

```
    input_mean = 127.5
    input_std = 127.5
  else:
    tf.logging.error("Couldn't understand architecture name '%s'", architecture)
    raise ValueError('Unknown architecture', architecture)

  return {
      'data_url': data_url,
      'bottleneck_tensor_name': bottleneck_tensor_name,
      'bottleneck_tensor_size': bottleneck_tensor_size,
      'input_width': input_width,
      'input_height': input_height,
      'input_depth': input_depth,
      'resized_input_tensor_name': resized_input_tensor_name,
      'model_file_name': model_file_name,
      'input_mean': input_mean,
      'input_std': input_std,
  }


def add_jpeg_decoding(input_width, input_height, input_depth, input_mean,
                      input_std):
  """Adds operations that perform JPEG decoding and resizing to the graph..

  Args:
    input_width: Desired width of the image fed into the recognizer graph.
    input_height: Desired width of the image fed into the recognizer graph.
    input_depth: Desired channels of the image fed into the recognizer graph.
    input_mean: Pixel value that should be zero in the image for the graph.
    input_std: How much to divide the pixel values by before recognition.

  Returns:
    Tensors for the node to feed JPEG data into, and the output of the
      preprocessing steps.
  """
  jpeg_data = tf.placeholder(tf.string, name='DecodeJPGInput')
  decoded_image = tf.image.decode_jpeg(jpeg_data, channels=input_depth)
  decoded_image_as_float = tf.cast(decoded_image, dtype=tf.float32)
  decoded_image_4d = tf.expand_dims(decoded_image_as_float, 0)
  resize_shape = tf.stack([input_height, input_width])
  resize_shape_as_int = tf.cast(resize_shape, dtype=tf.int32)
  resized_image = tf.image.resize_bilinear(decoded_image_4d,
                                           resize_shape_as_int)
  offset_image = tf.subtract(resized_image, input_mean)
  mul_image = tf.multiply(offset_image, 1.0 / input_std)
  return jpeg_data, mul_image


def main(_):
  # Needed to make sure the logging output is visible.
  # See https://github.com/tensorflow/tensorflow/issues/3047
  tf.logging.set_verbosity(tf.logging.INFO)

  # Prepare necessary directories  that can be used during training
  prepare_file_system()

  # Gather information about the model architecture we'll be using.
  model_info = create_model_info(FLAGS.architecture)
  if not model_info:
    tf.logging.error('Did not recognize architecture flag')
    return -1

  # Set up the pre-trained graph.
  maybe_download_and_extract(model_info['data_url'])
  graph, bottleneck_tensor, resized_image_tensor = (
      create_model_graph(model_info))

  # Look at the folder structure, and create lists of all the images.
  image_lists = create_image_lists(FLAGS.image_dir, FLAGS.testing_percentage,
                                   FLAGS.validation_percentage)
  class_count = len(image_lists.keys())
  if class_count == 0:
    tf.logging.error('No valid folders of images found at ' + FLAGS.image_dir)
    return -1
  if class_count == 1:
    tf.logging.error('Only one valid folder of images found at ' +
                     FLAGS.image_dir +
                     ' - multiple classes are needed for classification.')
    return -1

  # See if the command-line flags mean we're applying any distortions.
  do_distort_images = should_distort_images(
      FLAGS.flip_left_right, FLAGS.random_crop, FLAGS.random_scale,
      FLAGS.random_brightness)

  with tf.Session(graph=graph) as sess:
    # Set up the image decoding sub-graph.
    jpeg_data_tensor, decoded_image_tensor = add_jpeg_decoding(
        model_info['input_width'], model_info['input_height'],
        model_info['input_depth'], model_info['input_mean'],
        model_info['input_std'])

    if do_distort_images:
      # We will be applying distortions, so setup the operations we'll need.
      (distorted_jpeg_data_tensor,
       distorted_image_tensor) = add_input_distortions(
           FLAGS.flip_left_right, FLAGS.random_crop, FLAGS.random_scale,
           FLAGS.random_brightness, model_info['input_width'],
           model_info['input_height'], model_info['input_depth'],
           model_info['input_mean'], model_info['input_std'])
```

```
else:
  # We'll make sure we've calculated the 'bottleneck' image summaries and
  # cached them on disk.
  cache_bottlenecks(sess, image_lists, FLAGS.image_dir,
                    FLAGS.bottleneck_dir, jpeg_data_tensor,
                    decoded_image_tensor, resized_image_tensor,
                    bottleneck_tensor, FLAGS.architecture)

# Add the new layer that we'll be training.
(train_step, cross_entropy, bottleneck_input, ground_truth_input,
 final_tensor) = add_final_training_ops(
     len(image_lists.keys()), FLAGS.final_tensor_name, bottleneck_tensor,
     model_info['bottleneck_tensor_size'])

# Create the operations we need to evaluate the accuracy of our new layer.
evaluation_step, prediction = add_evaluation_step(
    final_tensor, ground_truth_input)

# Merge all the summaries and write them out to the summaries_dir
merged = tf.summary.merge_all()
train_writer = tf.summary.FileWriter(FLAGS.summaries_dir + '/train',
                                     sess.graph)

validation_writer = tf.summary.FileWriter(
    FLAGS.summaries_dir + '/validation')
```

```
# Set up all our weights to their initial default values.
init = tf.global_variables_initializer()
sess.run(init)

# Run the training for as many cycles as requested on the command line.
for i in range(FLAGS.how_many_training_steps):
  # Get a batch of input bottleneck values, either calculated fresh every
  # time with distortions applied, or from the cache stored on disk.
  if do_distort_images:
    (train_bottlenecks,
     train_ground_truth) = get_random_distorted_bottlenecks(
         sess, image_lists, FLAGS.train_batch_size, 'training',
         FLAGS.image_dir, distorted_jpeg_data_tensor,
         distorted_image_tensor, resized_image_tensor, bottleneck_tensor)
  else:
    (train_bottlenecks,
     train_ground_truth, _) = get_random_cached_bottlenecks(
         sess, image_lists, FLAGS.train_batch_size, 'training',
         FLAGS.bottleneck_dir, FLAGS.image_dir, jpeg_data_tensor,
         decoded_image_tensor, resized_image_tensor, bottleneck_tensor,
         FLAGS.architecture)
  # Feed the bottlenecks and ground truth into the graph, and run a training
  # step. Capture training summaries for TensorBoard with the `merged` op.
  train_summary, _ = sess.run(
      [merged, train_step],
      feed_dict={bottleneck_input: train_bottlenecks,
                 ground_truth_input: train_ground_truth})
  train_writer.add_summary(train_summary, i)

  # Every so often, print out how well the graph is training.
  is_last_step = (i + 1 == FLAGS.how_many_training_steps)
  if (i % FLAGS.eval_step_interval) == 0 or is_last_step:
    train_accuracy, cross_entropy_value = sess.run(
        [evaluation_step, cross_entropy],
        feed_dict={bottleneck_input: train_bottlenecks,
                   ground_truth_input: train_ground_truth})
    tf.logging.info('%s: Step %d: Train accuracy = %.1f%%' %
                    (datetime.now(), i, train_accuracy * 100))
    tf.logging.info('%s: Step %d: Cross entropy = %f' %
                    (datetime.now(), i, cross_entropy_value))
    validation_bottlenecks, validation_ground_truth, _ = (
        get_random_cached_bottlenecks(
            sess, image_lists, FLAGS.validation_batch_size, 'validation',
            FLAGS.bottleneck_dir, FLAGS.image_dir, jpeg_data_tensor,
            decoded_image_tensor, resized_image_tensor, bottleneck_tensor,
            FLAGS.architecture))
    # Run a validation step and capture training summaries for TensorBoard
    # with the `merged` op.
    validation_summary, validation_accuracy = sess.run(
        [merged, evaluation_step],
        feed_dict={bottleneck_input: validation_bottlenecks,
                   ground_truth_input: validation_ground_truth})
      validation_writer.add_summary(validation_summary, i)
      tf.logging.info('%s: Step %d: Validation accuracy = %.1f%% (N=%d)' %
                      (datetime.now(), i, validation_accuracy * 100,
                       len(validation_bottlenecks)))

    # Store intermediate results
    intermediate_frequency = FLAGS.intermediate_store_frequency

    if (intermediate_frequency > 0 and (i % intermediate_frequency == 0)
        and i > 0):
      intermediate_file_name = (FLAGS.intermediate_output_graphs_dir +
                                'intermediate_' + str(i) + '.pb')
      tf.logging.info('Save intermediate result to : ' +
                      intermediate_file_name)
      save_graph_to_file(sess, graph, intermediate_file_name)

  # We've completed all our training, so run a final test evaluation on
  # some new images we haven't used before.
  test_bottlenecks, test_ground_truth, test_filenames = (
      get_random_cached_bottlenecks(
          sess, image_lists, FLAGS.test_batch_size, 'testing',
          FLAGS.bottleneck_dir, FLAGS.image_dir, jpeg_data_tensor,
            decoded_image_tensor, resized_image_tensor, bottleneck_tensor,
            FLAGS.architecture))
    test_accuracy, predictions = sess.run(
        [evaluation_step, prediction],
        feed_dict={bottleneck_input: test_bottlenecks,
                   ground_truth_input: test_ground_truth})
    tf.logging.info('Final test accuracy = %.1f%% (N=%d)' %
                    (test_accuracy * 100, len(test_bottlenecks)))

    if FLAGS.print_misclassified_test_images:
      tf.logging.info('=== MISCLASSIFIED TEST IMAGES ===')
      for i, test_filename in enumerate(test_filenames):
        if predictions[i] != test_ground_truth[i].argmax():
          tf.logging.info('%70s  %s' %
                          (test_filename,
                           list(image_lists.keys())[predictions[i]]))

    # Write out the trained graph and labels with the weights stored as
    # constants.
    save_graph_to_file(sess, graph, FLAGS.output_graph)
    with gfile.FastGFile(FLAGS.output_labels, 'w') as f:
      f.write('\n'.join(image_lists.keys()) + '\n')


if __name__ == '__main__':
```

```
parser = argparse.ArgumentParser()
parser.add_argument(
    '--image_dir',
    type=str,
    default='',
    help='Path to folders of labeled images.'
)
parser.add_argument(
    '--output_graph',
    type=str,
    default='/tmp/output_graph.pb',
    help='Where to save the trained graph.'
)
parser.add_argument(
    '--intermediate_output_graphs_dir',
    type=str,
    default='/tmp/intermediate_graph/',
    help='Where to save the intermediate graphs.'
)
parser.add_argument(
    '--intermediate_store_frequency',
    type=int,
    default=0,
    help="""\
      How many steps to store intermediate graph. If "0" then will not
      store.\
    """
)
parser.add_argument(
    '--output_labels',
    type=str,
    default='/tmp/output_labels.txt',
    help='Where to save the trained graph\'s labels.'
)
parser.add_argument(
    '--summaries_dir',
    type=str,
    default='/tmp/retrain_logs',
    help='Where to save summary logs for TensorBoard.'
)
parser.add_argument(
    '--how_many_training_steps',
    type=int,
    default=4000,
    help='How many training steps to run before ending.'
)
parser.add_argument(
    '--learning_rate',
    type=float,
    default=0.01,
    help='How large a learning rate to use when training.'
)
parser.add_argument(
    '--testing_percentage',
    type=int,
    default=10,
    help='What percentage of images to use as a test set.'
)
parser.add_argument(
    '--validation_percentage',
    type=int,
    default=10,
    help='What percentage of images to use as a validation set.'
)
parser.add_argument(
    '--eval_step_interval',
    type=int,
    default=10,
    help='How often to evaluate the training results.'
)
parser.add_argument(
    '--train_batch_size',
    type=int,
    default=100,
    help='How many images to train on at a time.'
```

```
)
parser.add_argument(
    '--test_batch_size',
    type=int,
    default=-1,
    help="""\
    How many images to test on. This test set is only used once, to evaluate
    the final accuracy of the model after training completes.
    A value of -1 causes the entire test set to be used, which leads to more
    stable results across runs.\
    """
)
parser.add_argument(
    '--validation_batch_size',
    type=int,
    default=100,
    help="""\
    How many images to use in an evaluation batch. This validation set is
    used much more often than the test set, and is an early indicator of how
    accurate the model is during training.
    A value of -1 causes the entire validation set to be used, which leads to
    more stable results across training iterations, but may be slower on large
    training sets.\
    """
)
```

```
parser.add_argument(
    '--print_misclassified_test_images',
    default=False,
    help="""\
    Whether to print out a list of all misclassified test images.\
    """,
    action='store_true'
)
parser.add_argument(
    '--model_dir',
    type=str,
    default='/tmp/imagenet',
    help="""\
    Path to classify_image_graph_def.pb,
    imagenet_synset_to_human_label_map.txt, and
    imagenet_2012_challenge_label_map_proto.pbtxt.\
    """
)
parser.add_argument(
    '--bottleneck_dir',
    type=str,
    default='/tmp/bottleneck',
    help='Path to cache bottleneck layer values as files.'
)
parser.add_argument(
    '--final_tensor_name',
    type=str,
    default='final_result',
    help="""\
    The name of the output classification layer in the retrained graph.\
    """
)
parser.add_argument(
    '--flip_left_right',
    default=False,
    help="""\
    Whether to randomly flip half of the training images horizontally.\
    """,
    action='store_true'
)
parser.add_argument(
    '--random_crop',
    type=int,
    default=0,
    help="""\
    A percentage determining how much of a margin to randomly crop off the
    training images.\
    """
)
parser.add_argument(
    '--random_scale',
    type=int,
    default=0,
    help="""\
    A percentage determining how much to randomly scale up the size of the
    training images by.\
    """
)
parser.add_argument(
    '--random_brightness',
    type=int,
    default=0,
    help="""\
    A percentage determining how much to randomly multiply the training image
    input pixels up or down by.\
    """
)
parser.add_argument(
    '--architecture',
    type=str,
    default='inception_v3',
    help="""\
    Which model architecture to use. 'inception_v3' is the most accurate, but
    also the slowest. For faster or smaller models, chose a MobileNet with the
    form 'mobilenet_<parameter size>_<input_size>[_quantized]'. For example,
      'mobilenet_1.0_224' will pick a model that is 17 MB in size and takes 224
      pixel input images, while 'mobilenet_0.25_128_quantized' will choose a much
      less accurate, but smaller and faster network that's 920 KB on disk and
      takes 128x128 images. See https://research.googleblog.com/2017/06/mobilenets-
open-source-models-for.html
      for more information on Mobilenet.\
      """)
  FLAGS, unparsed = parser.parse_known_args()
  tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)
```

show_image.py

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
import os

from IPython.display import Image, HTML, display


root = "tf_files/flower_photos/"
with open(root+"/LICENSE.txt") as f:
    attributions = f.readlines()[4:]
attributions = [line.split(' CC-BY') for line in attributions]
attributions = dict(attributions)

def show_image(image_path):
    display(Image(image_path))

    image_rel = image_path.replace(root,'')
    caption = "Image " + ' - '.join(attributions[image_rel].split(' - ')[:-1])
    display(HTML("<div>%s</div>" % caption))
```