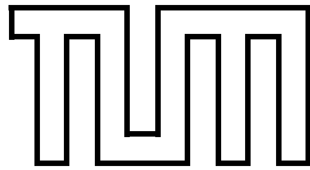


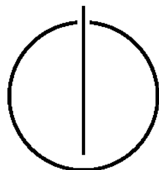
FAKULTÄT FÜR INFORMATIK

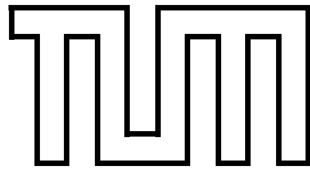
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

Using Synthetic Data for Classification of Small Parts

Amr Abdelraouf





FAKULTÄT FÜR INFORMATIK

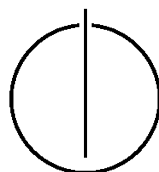
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

Using Synthetic Data for Classification of Small Parts

Verwendung synthetischer Daten für die Kleinteil-Erkennung

Author:	Amr Abdelraouf
Supervisor:	Prof. Bernd Brügge, Ph.D.
Advisor:	Sajjad Taheritanjni, M.Sc.
Date:	15.10.2018



I assure the single handed composition of this master thesis only supported by declared resources,

Munich, 15.10.2018 Amr Abdelraouf

Acknowledgements

I would first like to express my love and gratitude for my parents, Dr. Hatem Abdelraouf and Dr. Naglaa Hassan, for their unwavering support. Throughout my studies they have been my primary source of strength. The opportunity to study in Germany would have never been realized without their advice and encouragement. Furthermore, I would like to show appreciation for my fiancé Marwah Garib, whose support has been unfailing throughout the whole of my studies.

I would like to thank Prof. Bernd Brügge of the Applied Software Engineering chair at the Technical University of Munich. Prof. Bernd Brügge's comments and feedback were imperative to the quality of this document.

Last but most certainly not least, I would like to express thankfulness to my advisor, MSc. Sajjad Taheritanjni. The door to Sajjad's office was always open whenever I needed guidance, or had a question about the project, writing this document, academia in general, or even a non-thesis-related matter.

Abstract

Aircraft engines periodically undergo an overhaul process during which the engine parts are dismantled and then later reassembled. When the engine components are taken apart, thousands of small parts are unfastened. To facilitate the step of engine reassembly, the small parts have to be sorted and classified.

In this thesis we present a system to automatically classify small parts using convolutional neural networks. To train the neural networks, we use 3D models of the small parts to render synthetic images. The network is trained using a mixture of real images of small parts and synthetic images of their corresponding 3D models. We examine the use of a fully synthetic image set for training, as well as different ratios of real and synthetic images.

Our results indicate that a convolutional neural network trained on a dataset that contains as little as 5% real images can provide a comparable classification accuracy to networks that have been trained on purely real image sets. This is advantageous in supervised learning problems targeting novel domains, where an annotated dataset of images might not be available for training.

Keywords: image classification, convolutional neural networks, synthetic images, 3D models

Contents

1 Introduction **1**
1.1 Problem . . . 3
1.2 Motivation . . . 4
1.3 Outline . . . 4

2 Background **5**
2.1 Convolutional Neural Networks . . . 5
2.1.1 Training a Convolutional Neural Network . . . 8
2.2 Small Part Classification . . . 10
2.2.1 Real and Synthetic Images . . . 11

3 Related Work **12**
3.1 CNN Based Image Classification . . . 12
3.2 Industrial Use Cases for Image Classification . . . 14
3.3 Using Synthetic Images . . . 15

4 Analysis **17**
4.1 Requirements . . . 17
4.1.1 Functional Requirements . . . 18
4.1.2 Nonfunctional Requirements . . . 18
4.2 System Models . . . 19
4.2.1 Use Case Model . . . 19
4.2.2 Analysis Object Model . . . 29
4.2.3 Dynamic Model . . . 32

5 System design **35**
5.1 Design Goals . . . 35
5.2 Subsystem Decomposition . . . 36
5.3 Hardware Software Mapping . . . 38
5.4 Persistent Data Management . . . 40

6 Object Design **41**
6.1 Artec Space Spider and Artec Studio 41
6.1.1 Usage 42
6.2 Rhinoceros 42
6.2.1 Usage 42
6.3 Pillow 43
6.4 Keras and Tensorflow 43
6.4.1 Usage 43
6.5 VGG CNN Algorithm 43
6.6 Adam Optimization Algorithm 45

7 Evaluation **46**
7.1 Objectives 46
7.2 Methodology 46
7.2.1 Dataset Generation 47
7.2.2 Image Classification 51
7.3 Results 52
7.4 Findings 52
7.5 Limitations 53

8 Summary **54**
8.1 Status 54
8.2 Conclusion 54
8.3 Future Work 55

Chapter 1

Introduction

Aircraft engine overhaul is the process of removing, disassembling, inspecting, repairing, cleaning, reassembling and testing a used engine. Overhaul involves the intricate process of taking an engine apart, and in doing so, undoing up to thousands of small parts from the engine—eg. screws, nuts and bolts. In order to reassemble the engine in the overhaul process, the small parts have to be sorted and classified. Figure 1.1 shows an example of small parts that need to be classified.

Figure 1.1: Small parts that were taken apart during an aircraft engine overhaul, and have to be sorted for reassembling.

Manual Small Part Classification

Traditionally, the tasks of classifying and sorting small parts are executed manually by a human task force. The dismantled small parts are brought to the workers in a special work station, where the workers have access to manuals, pictures of each small part, magnifiers and measuring instruments.

The small parts are engraved with a part number which can be used for identification. However, sometimes the part number is indistinguishable due to erosion, rust or physical damage. In this case, the workers have to look up the small part in the manuals. The measuring instruments are used to compute the small part's distinguishing features as shown in 1.2. For classifying a small part with a distorted part number, the workers might need a few seconds to measure the different small part features and compare them against the models in the manual for classification. The time can varry depending on the small part and the exprience of the worker.

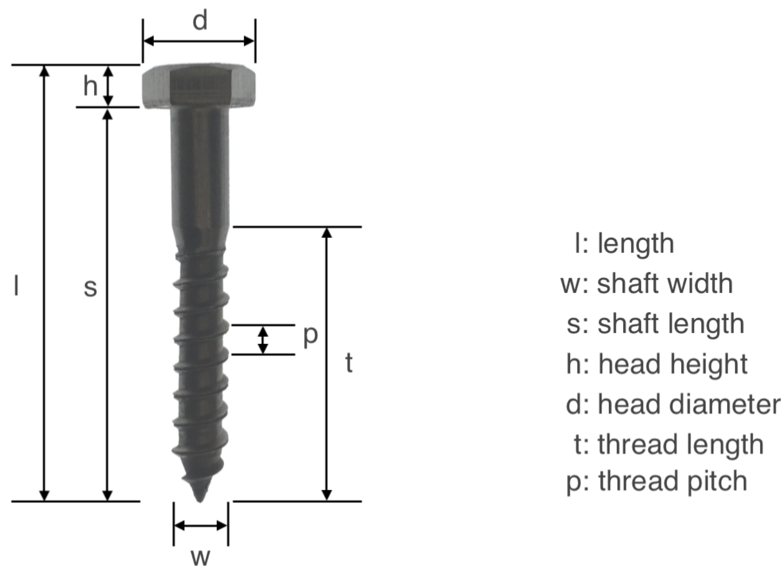


Figure 1.2: The structure of a sample screw displaying the different associated measurements.

Automatic Small Part Classification

An automatic small part classification system involves a setup where a camera and a robotic arm are placed over a conveyor belt. First, the small parts are placed on the conveyor belt. Next, the camera takes pictures of the small parts that are rolling underneath on the conveyor belt. The pictures are then sent to an image classification system which in turn classifies the images of the small parts. The classification system sends the labels of the small parts to the robotic arm, which uses those labels to sort the small parts accordingly. Figure 1.3 shows the automatic small part classification system.

The main brain behind automatic small part classification is the image classification system. In order to build a small part image classification sys-

tem, we employ convolutional neural networks (CNNs). CNNs have become the state-of-the-art approach to classical computer vision and image processing tasks such as image classification, object detection and many others [38].

The automatic approach involves minimal human interaction, and thus slashes the amount of man power required to sort and classify small parts. Moreover our automatic approach provides a quantifiable measurement for accuracy which can be used to assess and improve the classification system.

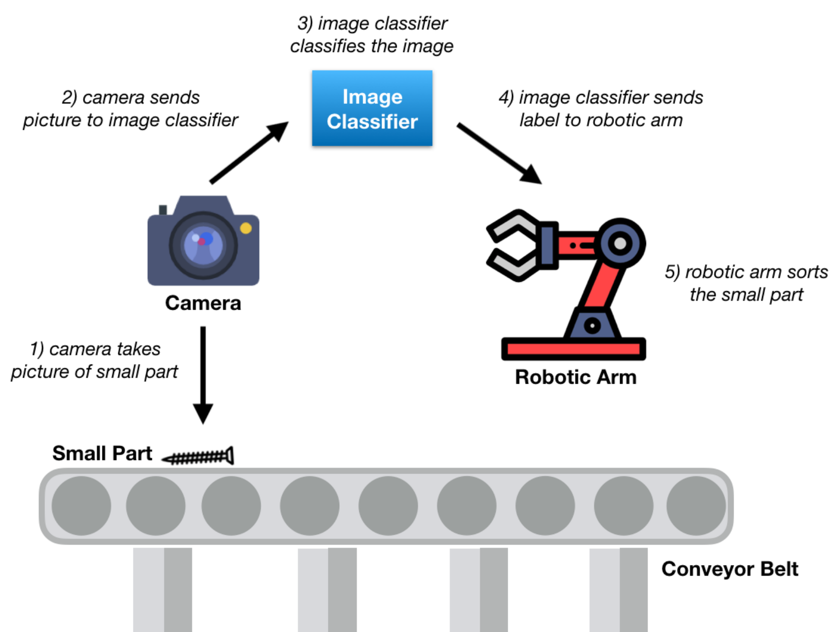


Figure 1.3: The automatic small part classification system.

1.1 Problem

Convolutional neural networks are characteristically data-centric algorithms. The performance of a CNN algorithm depends on the availability of a dataset of images that can capture each target object's intra-class variability [26]. In our case, we need to train a convolutional neural network using images that capture the distinguishing features of each of our small parts. Since aircraft engines can have up to thousands of small parts, and our CNN requires multiple images of each small part, the task of collecting an image set can be time consuming.

Our contribution in this thesis is twofold. 1) we describe a system to generate synthetic images for the classification of small parts. In addition to using pictures of the small parts, we obtain their respective 3D models and

use 3D modeling software to render 2D synthetic images of the small parts. The advantage being that the images are rendered rather than collected manually using a camera, and hence generation of the output image set requires less time and effort. 2) we train a convolutional neural network using synthetic images. We assess the use of synthetic data as input for CNN algorithms to perform image classification. We explore the use of a fully synthetic dataset and also a mixed dataset of both real and synthetic images in different ratios.

1.2 Motivation

Using synthetic images to classify small objects yields the power of convolutional neural network while overcoming the need to collect and annotate a dataset of images manually. This approach is extendable to image classification problems in other novel domains.

Furthermore, the process of generating synthetic images gives full control of the surrounding environment to the creator of the synthetic scene. The 3D models and the synthetic scenes can be easily adjusted to capture specific differentiating features of the small parts.

1.3 Outline

In chapter 2, convolutional neural networks are described and the necessary mathematical background needed to navigate this thesis is provided. We also define the terminology that we use to describe small parts and different types of images in our dataset. Chapter 3 transcribes the literature that was reviewed in preparation for this thesis. A review of convolutional neural networks based image classification, image classification in industrial use cases and the usage of synthetic data is provided. In chapter 4, the requirements of our system are broken down. The system use cases, object model and deployment diagrams are presented. In chapter 5, we describe our subsystems and the relationships between them. We also describe the software and hardware components used for implementation. Chapter 6 provides an in-depth explanation of the external frameworks, APIs, algorithms and software that we use out-of-the-box. Chapter 7 contains our implementation details, experiment design and results. Lastly, chapter 8 contains a recap of our work, the conclusions we have reached and the potential future work that can be based on our thesis.

Chapter 2

Background

In this chapter we provide the background needed for different concepts that are discussed in this thesis. In section 2.1 we introduce convolutional neural networks and how they are trained. Moreover we define the necessary terminology and provide the mathematical background required to navigate this thesis. Section 2.2 provides the terminology that we use throughout this thesis to describe small parts, synthetic images and real images.

2.1 Convolutional Neural Networks

A **neural network** is a massively parallel distributed processor made up of single processing units, which has a natural propensity for storing experiential knowledge and making it available for use [21]. Convolutional neural networks are a special type of neural network. They are typically used to solve computer vision problems like image classification and object recognition.

Regular neural networks consist of an **input layer**, **hidden layers** and an **output layer**. Every layer is made up of a set of **neurons**, where each neuron is fully connected to all neurons in the layer before. A neuron which has an input x of size n calculates the weighted sum z as follows:

$$z = \sum_{i=1}^{n} w_i x_i + b_i$$

where w is called **weight** and b is called **bias**. The weights and biases of all the neurons in a network are called the network **parameters**. Each neuron calculates its output y by applying a differentiable non-linear function $\varphi(\cdot)$, called the **activation function**, to the weighted sum of its input signals z.

$$y = \varphi(z)$$

A neural networks relays the output of its neurons through a series of hidden layers. Finally, the last fully-connected layer, called the output layer, computes the predictions of the neural network. Figure 2.1 shows an example structure of a regular neural network.

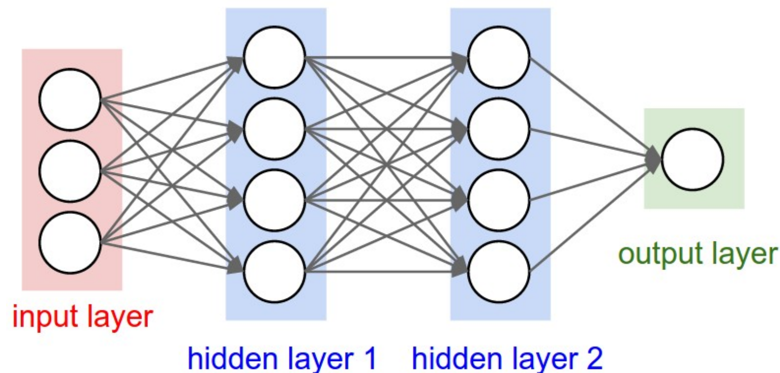


Figure 2.1: A neural network with an input layer, two hidden layers and an output layer. Each neuron is fully connected to all the neurons in the preceding layer.[1]

Convolutional neural networks (CNNs) are structured differently. CNNs have 3 types of layers: **convolution layers**, **pooling layers** and **fully connected** layers. Convolution layers slide a weighted matrix (called **filter** or **kernel**) over the previous layer, and calculate the sum of products. The size of the step that the filter takes while sliding is called a **stride**. Figure 2.2 shows how a convolution layer applies a 3x3 filter to the previous layer. A non-linear activation function is applied to the output of the convolution operation to create a **feature map**. It is common to add a pooling layer after the convolution layer for subsampling, which reduces the convolution layer dimensionality. A frequently used type of pooling is **max pooling** [47]. Max pooling divides the input layer into sections and computes the maximum activation of each section. Figure 2.3 shows how a max pooling layer applies a 2x2 filter to the preceding layer. Fully connected layers are similar to layers in a regular neural network. Each neuron in a fully connected layer is connected to all neurons in the previous layer.

Figure 2.4 shows the LeNet-5 convolutional neural network, which was designed by LeCun et al. [31] for handwritten digit recognition. LeNet-5 is an example of a CNN **architecture**. A CNN architecture is a set up of different types of layers which are combined to predict the class of the input image.

[1]Image: http://cs231n.github.io/convolutional-networks/

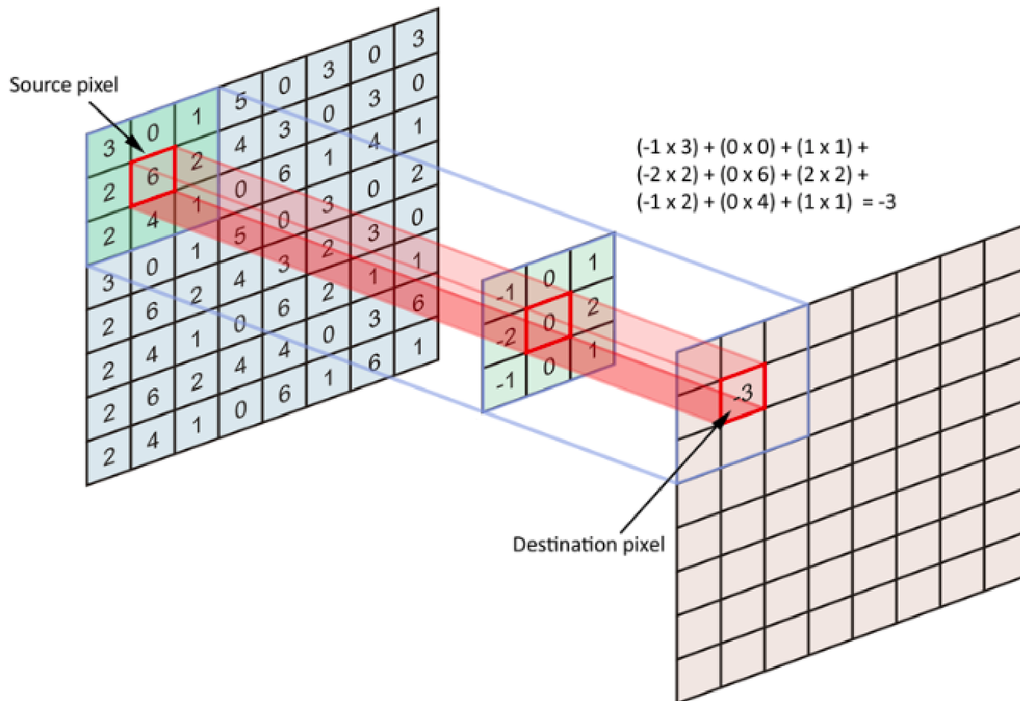


Figure 2.2: A 3x3 filter is applied to the source pixel. Each weight in the filter is multiplied by the source's neighboring pixel in the corresponding position. The sum of products is used to calculate the destination pixel's value in the output feature map.[2]

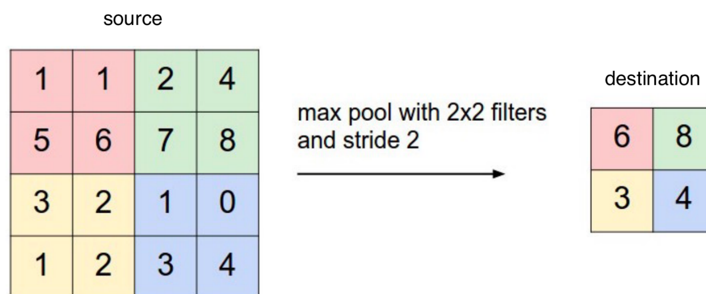


Figure 2.3: A max-pooling filter of size 2x2 and stride 2 is applied to the source feature map. The maximum value of each filter is used in the destination feature map.[3]

[2]Image: https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2

[3]Image: http://cs231n.github.io/convolutional-networks/

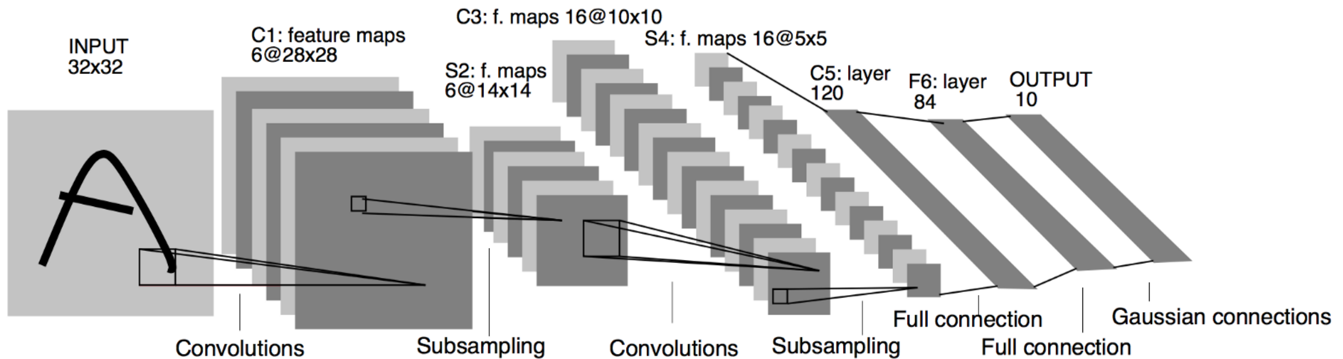


Figure 2.4: Architecture of the LeNet-5 convolutional neural network. [29]

The LeNet-5 network uses 3x3 filters with a stride of 1 for convolution layers and 2x2 filters with a stride of 2 for subsampling layers. LeNet-5 uses **average pooling**, which calculates the average of the values in the filter.

The network takes a 32x32 image as input. The first layer is a convolution layer (C1), which applies 6 filters, followed by a subsampling layer (S2). Next comes another convolution layer (C3) which applies 16 filter, followed by a subsampling layer (S4). A final convolution layer (C5) that applies 120 filters follows S4. Afterwards comes a fully connected (F6) layer with 84 neurons. Finally, an output layer of size 10 is connected to F6. Each neuron in the output layer corresponds to the probability of the input image to be one of the 10 digits (0-9).

2.1.1 Training a Convolutional Neural Network

We train a CNN using supervised learning. Supervised learning entails that our network learns from examples. The network is presented with a **dataset** of images. The dataset is divided into a **training set**, a **validation set** and a **testing set**. Each set contains images and their corresponding **labels** (also referred to as **classes**). We train a CNN to predict the labels of images as seen in the training set. The validation set is used for intermediate assessment of the CNN performance during the training phase, while the testing set is used to assess the performance of the CNN after the training phase is done.

CNNs process the training set in **mini-batches**. While the use of large mini-batches increases the available computational parallelism, small batch training has been shown to provide improved generalization performance [34]. It is also common for CNNs to process the whole training set multiple times. Each single iteration through the training set is called an **epoch**. The batch size and number of epochs are examples of CNN **hyperparameters**.

Hyperparameters are network parameters that are not learnable, but rather optimized manually.

Training a convolutional neural network is the process of learning the right parameters (filter weights) to compute the correct labels from the input images in the training set. To do so, each CNN has a **loss function**. A loss function calculates the error of the predictions made by the CNN. The loss of a CNN which predicts an output y on a training set that has the correct labels $\hat{y}$ is calculated as follows:

$$L(y, \hat{y}) = \frac{1}{m} \sum_{i=1}^{m} \mathscr{L}(y_i, \hat{y}_i)$$

where m is the number of examples in the training set, and $\mathscr{L}(\cdot)$ is a distance function that calculates the difference between pairwise instances in y and $\hat{y}$.

An **optimization function** (or **optimizer**) uses **back propagation** [29] to minimize the loss of a CNN during the training phase. The optimizer propagates the output of the loss function backwards by calculating the partial derivative of the loss function with respect to weights and using it to update the values of the weights.

The **stochastic gradient descent (SGD)** optimizer updates the weights of a CNN that uses the loss function L thusly:

$$w_{new} = w_{old} - \alpha \frac{\partial L}{\partial w_{old}}$$

where α is called the **learning rate**. The learning rate controls how fast the weights are updated during training. The learning rate is a tricky hyperparameter to tune. If it is too large, the optimizer might overshoot the optimum parameter values. If it is too small, the optimizer might never reach the optimal value.

Transfer learning [35] is sometimes employed to speed up the training of convolutional neural networks. Transfer learning involves pre-training a CNN on images from a different domain, and then retraining the network on an image set that is specific to the desired classification task. The advantage of transfer learning is the ability to learn features from a different dataset. This is useful when we have a classification task in one domain of interest, but we only have sufficient training data in another domain. A common technique is to freeze the weights of earlier layers while training a CNN, meaning that the weights are set to be untrainable. The number of **frozen layers** is a network hyperparameter.

2.2 Small Part Classification

We describe a system for the automatic classification of **small parts** in aircraft engines. Small parts refer to the fasteners used to build the engine like screws, bolts and nuts.

There is a set of classification challenges that are specific to small parts. A number of small parts can only be separated by subtle distinctions. This difference can be a slight change in size, which means that small parts, unlike scale invariant categories, are sensitive to differences in size, length and width. Figure 2.5 shows 2 screws that are identical except for a 6mm difference in length.

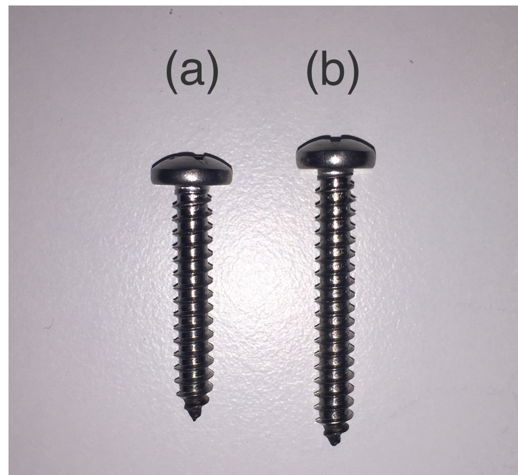


Figure 2.5: Screw (a) is identical to screw (b) except that screw (a) is 6mm shorter than screw (b). An overhead light shines on the 2 screws. The reflection changes the natural color of their surface.

Aircraft fasteners are metallic and have a shiny surface. The light reflection off the fastener surface disturbs the natural surface color as shown in figure 2.5, and may hide important features. Furthermore, the effect of the light reflection depends on the direction of the light, which may cause variance between images of a single small part.

Unlike cases where the background can provide information, the background is non-informative to the classification of small parts. Additionally, exposure to a regular lighting condition, like an overhead light in a room, causes the fasteners to cast a shadow on their background. A fastener shadow is not a discriminative feature. Both the background and the fastener shadow are a source of noise for the classifier.

2.2.1 Real and Synthetic Images

To automatically classify small parts, we build a convolutional neural network that is trained on both **real images** and **synthetic images**. We define real images as the natural images of the small parts, taken using a camera. Synthetic images, on the other hand, are artificially created images. They are 2-dimensional renditions of **3D models** of small parts. To generate synthetic images, we must create a **synthetic scene**: an artificial environment created in 3D modeling software. Figure 2.6 shows an example of a synthetic scene. A 3D model is placed in a synthetic scene in 3D modeling software. The software renders a 2D image of the environment to create a synthetic scene.

The real and synthetic images should capture the small parts and their corresponding 3D models from different angles and in different positions. To do so, we apply **transformations** to the small parts and 3D models. There are two types of transformations: **translation** to change the position of the target object, and **rotation** to change the angle. Each transformation has a range. This prevents the target objects from being translated or rotated away from the camera view.

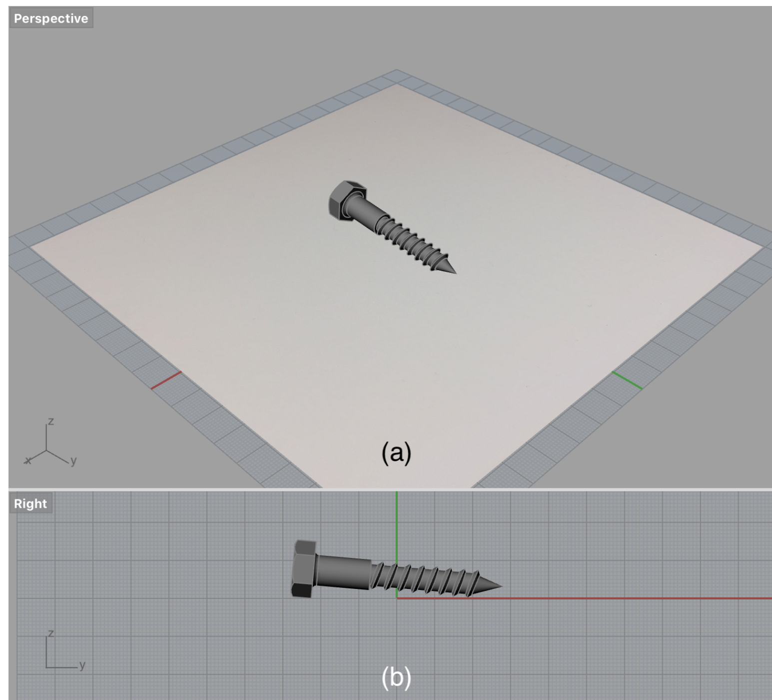


Figure 2.6: A synthetic scene where a screw lies on a horizontal plane. (a) shows the isometric view of the scene while (b) shows the side view.

Chapter 3

Related Work

We provide in this chapter the literature review about topics which touch different aspects of our work. Section 3.1 discusses CNN based image classification, inspired by the overview of deep learning in neural networks presented by Schmidhuber [41]. Section 3.2 provides an overview of different industrial use cases of image classification. Finally, in section 3.3 we examine the use of synthetic images for different computer vision problems.

3.1 CNN Based Image Classification

The *Neocognitron*, introduced by Fukushima [14], is the first neural network that resembles the modern convolutional neural network. It was based on the neurophysical insights of Hubel et al. [23] [24], where simple cells and complex cells were observed to respond differently to stimulation in the cat's visual cortex. Subsequently, Fukushima introduced *S-cells* (corresponding to simple cells), where a weighted filter is shifted across a 2 dimensional array to produce activations of higher order, in order to detect a particular pattern (eg. edge, shape, etc..). Fukushima also introduced *C-cells* (corresponding to complex cells), which are in turn wired to the output activations of a set of S-cells. C-cells are activated if any of the preceding S-cells are activated, hence they were used as subsampling layers. The Neocognitron is a multilayered network of S-cells and C-cells. The structure of a Neocognitron is similar to what we currently know as a convolutional neural network. Fukushima applied the Neocognitron to hand-written character recognition.

In 1989, LeCun et al. used the backpropagation algorithm to train a 5-layer network (dubbed *LeNet-5*) [32] [31]. Similar to the Neocognitron, there are two types of layers. Convolution layers use a weighted sliding window to across the 2-dimensional input to calculate the layer activations. On the

other hand, subsampling layers calculate the average of neighboring input activations. LeCun et al. trained LeNet-5 on 16x16 images of handwritten digits [30], and later applied this concept to develop an application for automatically reading zip codes [29]. This work also introduced the MNIST dataset, which is a database of handwritten digits [28].

Inspired by the Neocognitron, Weng et al. introduced the Cresceptron [47], which adapts the same topology during training. To implement the subsampling layers, the Cresceptron uses max pooling. Here, the 2-dimensional input to the subsampling layer is partitioned into smaller rectangular arrays. Each is replaced in the subsampling layer by the maximum activation value within the partition.

Traditionally, convolutional neural networks were implemented and computed on central processing units (CPU). To speed up the processing of CNNs Chellapilla et al. provided a graphical processing unit (GPU) based implementation of convolutional neural networks [5]. The GPU-based implementation was 4.11 times faster than the CPU-based implementation. GPUs have become more and more important for deep learning in subsequent years.

In 2011, Ciresan et al. [7] described a GPU-implementation [5] of a CNN with max pooling layers [47], trained using backpropagation [29]. Ciresan et al. used this to achieve superhuman visual pattern recognition performance in the IJCNN 2011 traffic sign recognition contest in San Jose, California [44] [8] [9]. Ciresan et al. obtained a 0.56% error rate, while the human performance on the test set was 1.19%. Furthermore, Ciresan et al. achieved human-competitive performance (around 0.2%) on MNIST handwriting benchmark [10].

The ImageNet Large Scale Visual Recognition Challenge is a benchmark in object category classification and detection on hundreds of object categories and millions of images [38]. The challenge has been conducted annually from 2010 to present. In 2012, Krizhevsky et al. [26] used a CNN (dubbed *Alexnet*) to achieve the best results on the image classification benchmark. Alexnet achieved 16.4% classification error, a significant improvement over the 2011 winning team, which achieved 25.8%. Zeiler and Fergus [48] won the classification challenge in 2013. They presented a network which was based on Alexnet (dubbed *ZF Net*) and were able to achieve 11.6% classification error. Based on the work of Zeiler et al. [49], Zeiler and Fergus developed a visualization technique called *Deconvolutional Network*, which projects the feature activations back to the input pixel space. 2014 saw yet another CNN-based winning system. Szegedy et al. [45] created a network called *GoogLeNet* (also known as *Inception*) that achieved 6.7% classification error. In second place during the same year, Simonyan and Zisserman [42] reached 7.3% using the *VGG* network.

3.2 Industrial Use Cases for Image Classification

Marino et al. [33] [11] describe Visual Inspection System for Railway maintenance (VISyR), which detects the presence of hexagonal-headed bolts. VISyR uses two 3-layer neural networks (NN) running in parallel. The input features to the two networks are extracted using discrete wavelet transform (DWT). The first NN uses Daubechies wavelets, while the second one uses Haar wavelets. Both neural networks are trained with same examples and fined tuned using back propagation. The system signals the presence of a hexagonal-headed bolts if both networks detect its presence.

Gibert et al. [18] describes a method to detect the presence of a fastener in an image, determine whether the fastener is broken, and further subcategorize the fastener into one of five classes. The method uses Histogram of Gradients (HOG) to extract features, and a Support Vector Machine (SVM) for classification. In the same year, Gibert et al. [17] presented a way to classify materials in images of railway tracks. Gibert et al. use a 4-layered convolutional neural network to perform segmentation on the input images. The network classifies each pixel in the image into one of 10 materials. The CNN was trained to detect chipped or crumbling railway cross ties. In 2017, Gibert et al. [19] presented an approach to perform the tasks simultaneously: classify both the materials and the fasteners in an input image. Gibert et al. extended the material classification network [17] by extracting the features from the 3rd network layer. The features were used as input for an SVM classifier to detect the presence of a fastener and classify its type if found.

In 2015, Aytekin et al. [2] describe a real-time railway fastener detection system using a high-speed laser range finder camera, which is placed under a railway carriage designed for railway quality inspection. Aytekin et al. present multiple pixel-similarity-based approaches, namely principal component analysis (PCA), linear discriminant analysis (LDA), random-forest (RF), sparse representation (SR), and multitemplate matching (MTM). Moreover, the histogram-based approaches histogram matching (HM) and depth peeks (DP) approaches are also implemented. Aytekin et al. focused on false alarm rate (FAR) as an evaluation metric since they wanted to minimize false positives. Aytekin et al. concluded that a combination of PCA and DP produce the least false alarm rate.

Feng et al. [13] describes a method to perform automatic fastener classification and defect detection. Their system which places 2 cameras under a train coach, which in turn takes pictures of the track and sends it to an onboard processor which performs fastener localization, classification and de-

fect assessment by ranking the status of the fasteners. Feng et al. propose a novel classification technique based on Latent Dirichlet Allocation (LDA). Their system can model different types of fasteners using unlabeled data and is robust to illumination changes.

3.3 Using Synthetic Images

Synthetic images have been used for training in a variety of problems. Peng et al. [36] uses 3D CAD models to generate synthetic images. These images are then used to tests the invariance of convolutional neural networks to low level cues, namely, object texture, color, 3D pose and 3D shape, as well as background scene texture and color. A region-based convolutional neural network (RCNN) [20] is used to test cue invariance in object detection. Peng et al. present the results of training a CNN on novel domains using synthetic images. They use part of the Office dataset [39], which has the same 31 categories of common objects (cups, keyboards, etc.) in each domain. They compare the usage of 3D models with real texture versus 3D models with uniform gray texture. Peng et al. conclude that a CNN trained on synthetic images with real texture perform better than images with a gray texture.

Gerogakis et al. [15] use a mixture of synthetic images and real images to train a network for object detection in indoor scenes. The synthetic images are created by superimposing images of the target objects on indoor backgrounds. Instead of rendering images from 3D models, cropped real images from the BigBird dataset [43] are used. Additionally, the background images have depth map information (RGB-D images), and come from the GMU-Kitchen Scenes dataset [16] and the Washington RGB-D Scenes v2 datasets [27]. Gerogakis et al. place the target object in the scene using a method they call *selective positioning*. First, they use image segmentation to find counters and tables in the background image. Next, they use the depth map of the background image to scale the width and height of the target object. Lastly, they blend the object with the background image in order to mitigate the effects of changes in illumination and contrast. Gerogakis et al. conclude that training an object classifier using a mixture of 90% synthetic images and 10% real images can produce comparable results to a classifier trained on only real images.

Rajpura et al. [37] used 3D models of refrigerators and and products that are placed in refrigerators from Archive3D[1] and ShapeNet [4] to perform object detection in refrigerator scenes. Synthetic images were generated by

[1] http://archive3d.net/

rendering 2D images of a 3D synthetic scene were the products are placed inside the refrigerators. Rajpura et al. trains a CNN using a fully real training set, a fully synthetic training set, and a synthetic training set containing 10% real data. The CNN fully trained with only synthetic images underperforms against one with real images but the mixed training set boosts the detection performance by 12% which signifies the importance of transferable cues from synthetic to real.

Sarkar et al. [40] train a CNN image classifier on synthetic images rendered from 3D models of 5 different objects. The 3D models are created by scanning the real target objects using a 3D scanner. Synthetic images are created by rendering the 3D models on 3 different types of backgrounds: a plain white background, a random indoor background taken from indoor categories of PASCAL dataset [12], and a chosen background similar to that of test images. Additionally, images are rendered with two object texture settings: full texture and no texture. Sarkar et al. find that synthetic images of fully textured objects overlaid on a mixture of white and chosen backgrounds produce the best results compared to other texture and background settings.

Chapter 4

Analysis

The analysis process results in a model of the system that aims to be correct, complete, consistent and unambiguous [3]. In this chapter, we focus on scenario-based analysis. Section 4.1, identifies the functional and nonfunctional requirements of the system. In section 4.2, we describe a use case model that depicts the different interactions between the users and the system. Furthermore, we identify and study the objects of our system in the Analysis Object Model. Finally, 3 dynamic models that depict workflows in our system are presented.

4.1 Requirements

A requirement is a feature that the system must have or a constraint that it must satisfy to be accepted by the client [3]. Requirements elicitation requires domain knowledge of the problem statement as well as experience in building software systems.

We design a system that uses synthetic and real images to classify different small parts. To do so, we create 3D models of our small parts. We subsequently generate a real image set for each small part, and a synthetic image set from the small part's 3D model. The real and synthetic image sets are combined to form a dataset, which we use to train a classification model based on convolutional neural networks. In this section, the concrete functional and nonfunctional requirements needed to build our system are defined.

4.1.1 Functional Requirements

Functional requirements (FRs) describe the interactions between the system and the its environment independent of its implementation [3].

The system's main function is to classify images of different small parts. Furthermore, the system needs to be dynamic and scalable to accommodate new small parts. To ensure that the system is able to classify images accurately, while leveraging synthetic input images, we introduce the following Functional Requirements.

FR1 **Create 3D Model**: The system should use a 3D scanner to create a 3D model of a given small part.

FR2 **Generate Synthetic Image Set**: Given a 3D model of a small part, the system should place the 3D model in a synthetic scene and subsequently render a set of 2D synthetic images of the scene. The synthetic images should capture the 3D model in different positions.

FR3 **Set 3D Model Transformation Range**: To capture the model from different positions the model should be rotated and translated in the synthetic scene. The system should allow the user to define a rotation and translation range for each 3D model to avoid the risk of placing the 3D model in unrealistic positions.

FR4 **Generate Real Image Set**: Given a small part, the system should generate a set of real images of the small part.

FR5 **Split Input Dataset**: The synthetic and real image sets are combined to form a dataset. The system should split the dataset into a training set, a testing set and a validation set to train and evaluate a classification model. Moreover, the system should allow the user to set the ratio of synthetic to real images in the training set.

FR6 **Train Classification Model**: The system should have the capacity to train a classification model given a training and a validation set.

FR7 **Evaluate Classification Model**: The system should provide a mechanism to evaluate the performance of a given classification model.

4.1.2 Nonfunctional Requirements

Nonfunctional Requirements (NFRs) are key system requirements that apply to the system as a whole [3]. To maintain the system's general requirements we define the following nonfunctional requirements.

NFR1 **Performance**: A performance requirement is the measure of a quantifiable attribute of our system. In our case we would like to track our system's **classification accuracy**.
We first define our system's accuracy for a single class to be the percentage of the given class's testing images that the classification model correctly predicts. We consequently define our system's classification accuracy to be the classification model's average prediction accuracy over each class.

NFR2 **Adaptability**: Adaptability is the ability to change a system to deal with additional application domain concepts [3]. Our system should accommodate for the addition of new small parts to the set of target classes that our classification model is trained to identify. This addition can be done even after the system has been deployed.

4.2 System Models

In this section we take a closer look at the system models. In section 4.2.1, we establish the main actors of our system and determine the actions that they can undertake. Section 4.2.2, identifies the objects of our system, examines their behavior and studies the relationship between them. Lastly, in section 4.2.3 we present activity diagrams that describe the workflows of our system.

4.2.1 Use Case Model

Use case models represent the relationship between the user group of a system and the general functions that they can execute. A use case model can reduce the complexity of a system and increase its understandability.

Figure 4.1 depicts the use case model of our system. There are three main protagonists in our system. The *3DModelCreator* is responsible for creating 3D models for the given small parts. The *DataGenerator* uses the small parts and their corresponding 3D models to generate synthetic and real image sets. Finally, the *MachineLearningEngineer* is responsible for creating a training, a validation and a testing sets. Moreover, the *MachineLearningEngineer* trains the classification model to identify a set of small parts, and evaluates the model's performance.

The use case model can be divided into 4 main functions; namely creating 3D models, generating synthetic images, generating real images, and training/evaluating a classification model.

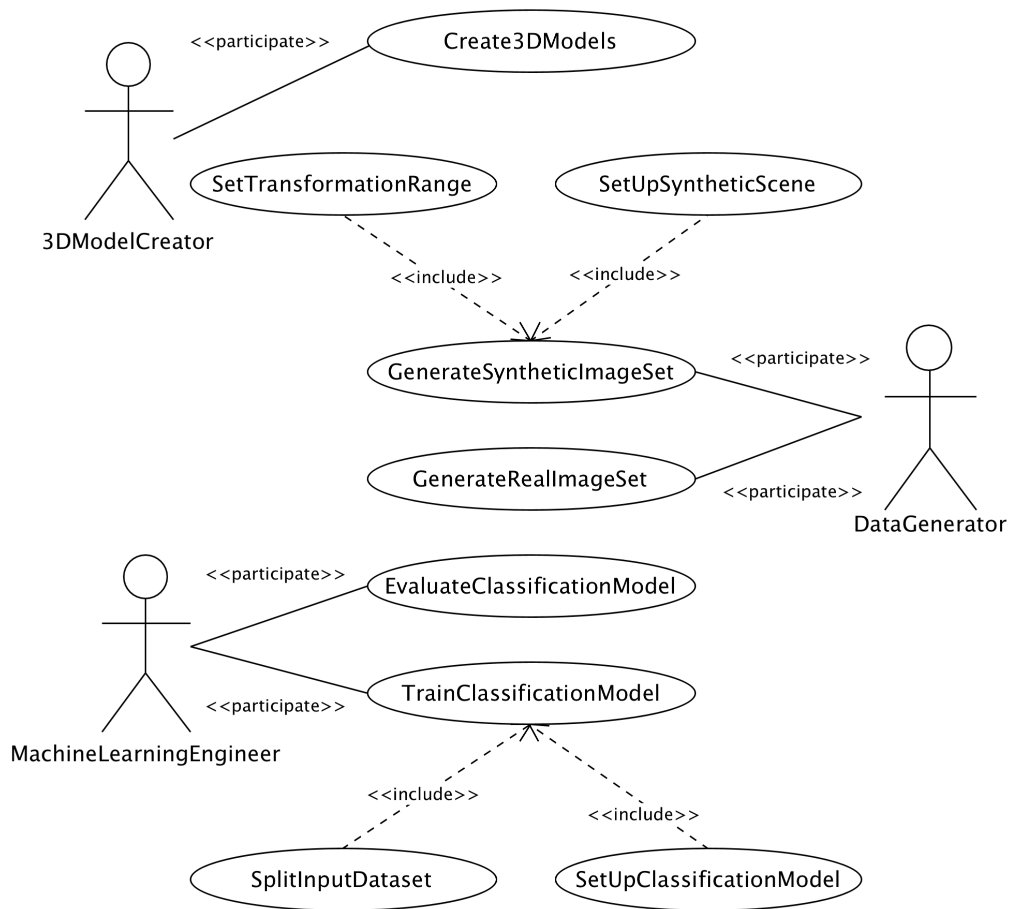


Figure 4.1: The use case model displays the DataGenerator and the MachineLearningEngineer and the actions that they can undertake.

Creating 3D Models

In order to create synthetic images, we need to create 3D models of our small parts. To do so, the 3DModelCreator uses a 3D scanner and compatible 3D scanning software to create the model.

Use Case	**Create3DModel**
Participating Actors	3DModelCreator
Flow of Events	1. The 3DModelCreator places the small part on a flat surface. 2. The 3DModelCreator uses the 3D scanner to scan images of the small part from all directions. 3. The 3D scanning software combines the scanned images to create a 3D model.
Entry Condition	• A small part has been chosen for scanning. • The compatible 3D scanning software is open. • The 3D scanner is turned on and recognized by the 3D scanning software.
Exit Condition	• A 3D model of the chosen small part has been created.
Quality Requirements	• The 3D model is as realistic as possible.

Generating Synthetic Images

The created 3D models are used to generate a synthetic image set. To increase the understandability of our model, we separate the use case that concerns modeling the 3D model as *SetTransformationRange*, and the use case that deals with the synthetic scene environment as *SetUpSyntheticScene*. Both are included in the use case *GenerateSyntheticImageSet*.

Use Case	**SetTransformationRange**
Participating Actors	DataGenerator
Flow of Events 1. In the modeling software, the DataGenerator chooses the transformations that are applicable to the small part's 3D model. 2. For each transformation, the DataGenerator sets the range of each transformation attribute that will be later used to generate a random position for the synthetic image of the small part.	
Entry Condition	• A 3D model of a small part has been created.
Exit Condition	• All the transformation ranges for the 3D model have been set.
Quality Requirements	• The 3D transformation ranges should reflect the real-life transformations that the small parts can undergo. • The 3D model should remain within camera viewfield.

<table>
<tr><th>Use Case</th><th>SetUpSyntheticScene</th></tr>
<tr><td>Participating Actors</td><td>DataGenerator</td></tr>
<tr><td colspan="2">Flow of Events
1. In the modeling software, the DataGenerator places the 3D model of the small part on a horizontal plane as if lying on a table.
2. The DataGenerator sets the background of the horizontal plane to mimic the background upon which the small part's real images are taken.
3. The DataGenerator sets the lighting condition of the scene to reflect the lighting condition of the environment where the small part's real images are taken.</td></tr>
<tr><td>Entry Condition</td><td>• A 3D model of a small part has been created.</td></tr>
<tr><td>Exit Condition</td><td>• In the modeling software, the 3D model of the small part should be lying on a horizontal plane.
• The background and lighting condition should reflect the environment of the real image set.</td></tr>
<tr><td>Quality Requirements</td><td>• The synthetic scene should look as photo-realistic as possible. This helps the classification model achieve a higher classification accuracy using the synthetic image set.</td></tr>
</table>

Use Case	**GenerateSyntheticImageSet**
Participating Actors	DataGenerator
Flow of Events 1. The DataGenerator specifies the desired number of output images in the synthetic image set. 2. The modeling software generates the desired number of output synthetic images. Each image is a rendition of the synthetic scene after the transformations have been applied to the 3D model.	
Entry Condition	• DataGenerator has set the transformation ranges of the 3D model. • DataGenerator has set up the synthetic scene.
Exit Condition	• The DataGenerator obtains a set of synthetic images for the specified small part. The image set can be later used to train a classification model.
Quality Requirements	• The generated image dimensions should match the input dimensions required by the classification model.

Generating Real Images

The DataGenerator should strive to create an environment that is easy to model in 3D software. This will enable the creation of photo-realistic synthetic images.

Use Case	**GenerateRealImageSet**
Participating Actors	DataGenerator
Flow of Events 1. The DataGenerator places the camera over a plane. 2. The DataGenerator sets the small part over the plane in a random position, while maintaining that the small part remains horizontal. The full small part body should remain within the viewfield of the camera. 3. The DataGenerator takes a picture of the small part. 4. The DataGenerator repeats steps 2 and 3 until the desired number of real images of the small part is reached. 5. The DataGenerator resizes the images to correspond to the input image size of the classification model.	
Entry Condition	• A small part has been chosen.
Exit Condition	• The DataGenerator obtains a set of real images for the specified small part that can be later used to train the classification model.
Quality Requirements	• The lighting of the environment should eliminate shadows and light reflections on the small part surface.

Training the Classification Model

The synthetic and real image sets are combined to form a dataset. Training a classification model requires the dataset to be split into a training, a validation, and a testing set. We model this requirement as the use case *SplitInputDataset*. Furthermore the actions required for setting up the classification model are modeled as the use case *SetUpClassificationModel*.

Use Case	**SplitInputDataset**
Participating Actors	MachineLearningEngineer
Flow of Events	1. The MachineLearningEngineer chooses the number of images for the training set, validation set and testing set. 2. The MachineLearningEngineer chooses the ratio of synthetic images to real images in the training set.
Entry Condition	• For each small part set to be classified by the classification model, the corresponding real and synthetic image sets should be generated and ready for use.
Exit Condition	• The images in the dataset are split into a training, a validation and a testing set, ready to be used by the classification model.

Use Case	**SetUpClassificationModel**
Participating Actors	MachineLearningEngineer
Flow of Events 1. The MachineLearningEngineer chooses a convolutional neural network architecture for the classification model. 2. The MachineLearningEngineer sets the hyperparameters: number of batches, number of epochs, number of frozen layers in the CNN, the optimizer and the hyperparameters associated with the chosen optimizer.	
Exit Condition	• The classification model architecture, optimizer and corresponding hyperparameters should be selected.

<table>
<tr><th>Use Case</th><th>TrainClassificationModel</th></tr>
<tr><td>Participating Actors</td><td>MachineLearningEngineer</td></tr>
<tr><td colspan="2">Flow of Events
1. The MachineLearningEngineer trains the classification model on the given training and validation sets.
2. The MachineLearningEngineer evaluates the accuracy of the classification model by studying the model's predictions on the testing set.
3. The MachineLearningEngineer fine-tunes the hyperparameters and re-trains the system until the maximum possible accuracy is reached.</td></tr>
<tr><td>Entry Condition</td><td>• The dataset should be split.
• The CNN model architecture, optimizer and initial hyperparameters should be chosen.</td></tr>
<tr><td>Exit Condition</td><td>• The classification model outputs the maximum possible accuracy given the input dataset.</td></tr>
<tr><td>Quality Requirements</td><td>• The classification model's hyperparameters should be fine-tuned until the maximum possible accuracy has been reached.</td></tr>
</table>

4.2.2 Analysis Object Model

The analysis object model is a visual dictionary of the main concepts visible to the user [3]. The analysis object model depicts a system's entities, their corresponding attributes and functions, and illustrates the relationship between said entities. Figure 4.2 and figure 4.3 represent the analysis object models of our system. We divide our model into 2 UML packages, namely, *Dataset Generation* and *Image Classification* depicted in figures 4.2 and 4.3 respectively. Dataset Generation presents the objects required to create 3D models of small parts and subsequently generate a dataset of real and synthetic images. Image Classification displays the objects used to utilize the generated dataset for the training and evaluation of a CNN model, which we use to classify our small parts.

Dataset Generation

To generate our dataset, we first need to create **3DModels** of our **SmallParts**. A **3DScanner** and a compatible **3DScanningSoftware** are used to scan a SmallPart and generate the corresponding 3DModel. Each SmallPart and its reciprocal 3DModel have a label string which it later used by the image classifier as the object class.

SmallParts and 3DModels both implement the **Transformable** interface. Each Transformable has multiple **Transformations**, and can apply the transform function. Transformation is an abstraction of **Translation** and **Rotation**. Each Transformation has an axis (x, y or z) and a range; Translations define minimum and maximum distances, while Rotations define minimum and maximum angles. Additionally, each transformation has a function to generate a random value within the defined range. This function is used to transform the object randomly before generating a real or synthetic image.

ImageGenerator is a generalization of **RealImageProcessor**, which uses the **Camera** to generate **RealImages**, and **3DModelingSoftware**, which generates **SyntheticImages**. ImageGenerators define the output number of images, as well as their width and height. Moreover, they apply random transformations to their target transformable before generating each image. Each 3DModelingSoftware has a **SyntheticScene** where the background and lighting conditions of the synthetic images are defined.

RealImage and SyntheticImage are both specializations of the **Image** class. Each Image has a width, height, label (corresponding to the label of the depicted SmallPart of 3DModel), and a data buffer which holds the values of the image pixels. The **Dataset** is a composition of multiple Images.

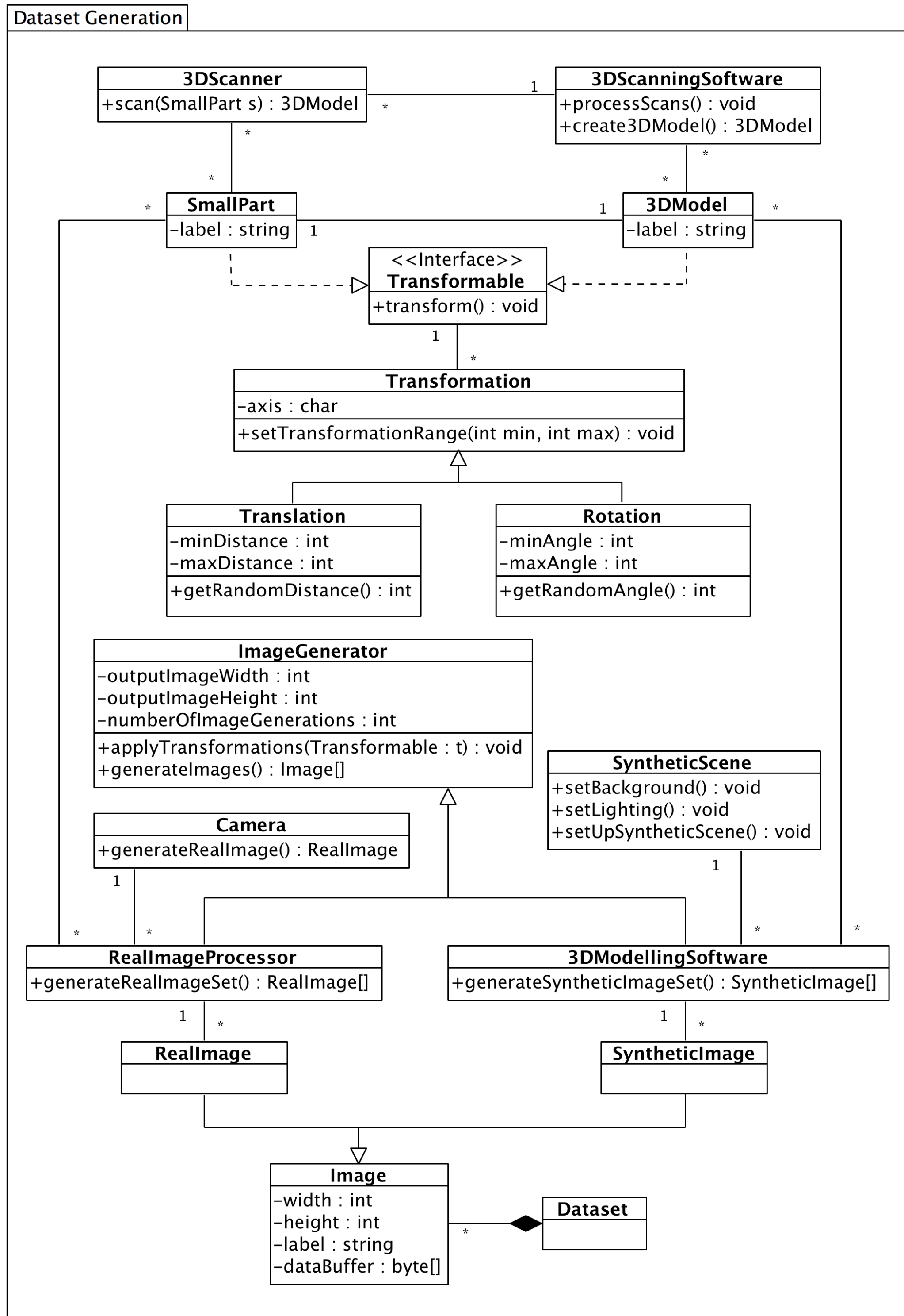


Figure 4.2: UML package of the analysis object model which models dataset generation.

Image Classification

After the dataset has been generated, it is split into a training set, a validation set and a testing set by the **DatasetSplitter**. The DatasetSplitter sets the size of each set as well as the ratio of synthetic to real images in the training set. The **ImageClassifier** uses the split dataset to train and evaluate a **CNNModel**. The CNNModel defines the model weights as well as the hyperparameters: batch size, number of epochs, and, number of frozen layers. CNNModel uses the **Optimizer** to adjust the model weights in the training phase.

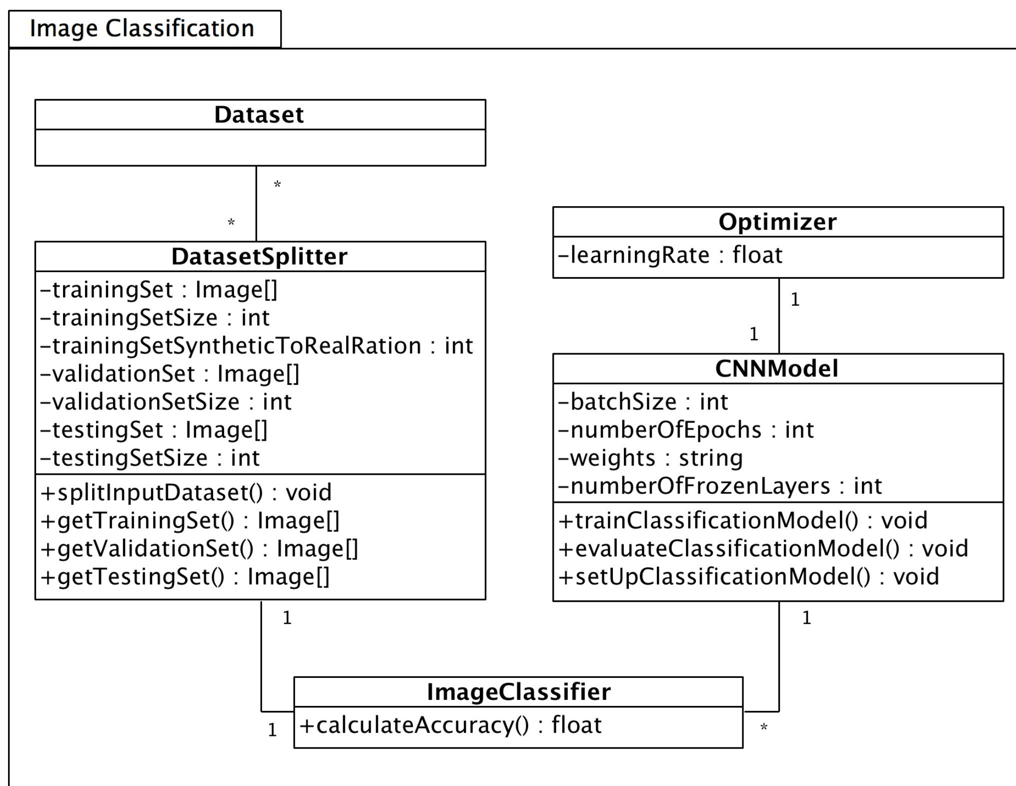


Figure 4.3: UML package of the analysis object model which depicts the objects that are used for image classification.

4.2.3 Dynamic Model

The dynamic model is focused on the behavior of the system. Sequence diagrams, activity diagrams and state machines are usually used to depict dynamic models [3]. In this section we present 3 activity diagrams, each depicts a workflow in our system. Namely, the workflow to create a real image set from a small part, the workflow to create the small part's 3D model and subsequently generate the synthetic image set of the small part, and the workflow to train a classification model using the dataset that combines both generated image sets.

Synthetic Image Set Generation

Figure 4.4 depicts the workflow of activities required to generate a synthetic image set. The 3DModelCreator uses a 3D scanner to create a 3D model of a small part. Next, the DataGenerator sets the 3D model transformations and creates the synthetic scene in the 3D modeling software. Finally, the software is used to generate a synthetic image set.

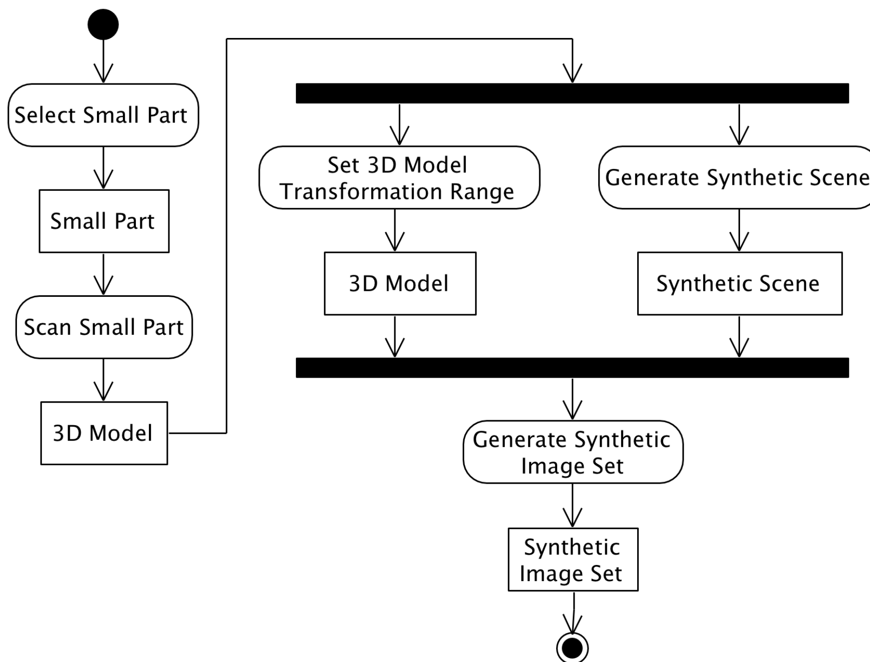


Figure 4.4: Activity Diagram depicting the workflow required for synthetic image set generation.

Real Image Set Generation

The workflow of activities required to generate a real image set is depicted in figure 4.5. The DataGenerator selects a small part. The small part is transformed and the camera is used to take a picture. The process is repeated until the desired number of real images is reached.

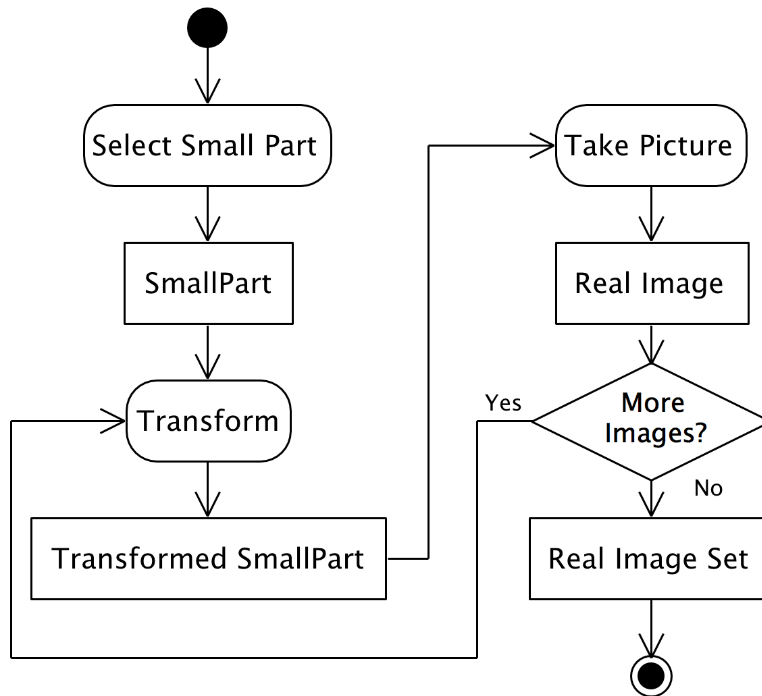


Figure 4.5: Activity Diagram depicting the workflow required for real image set generation.

Image Classification

Figure 4.6 describes the workflow of activities required to create the image classifier. The MachineLearningEngineer splits the data into training, validation and testing sets. The MachineLearningEngineer then creates a CNN model. The training and validation sets are used to train the CNN model. Next, the model accuracy is evaluated using the testing set. If the accuracy of the trained CNN model is not sufficient, the hyperparameters are fine tuned and the CNN model re-trained.

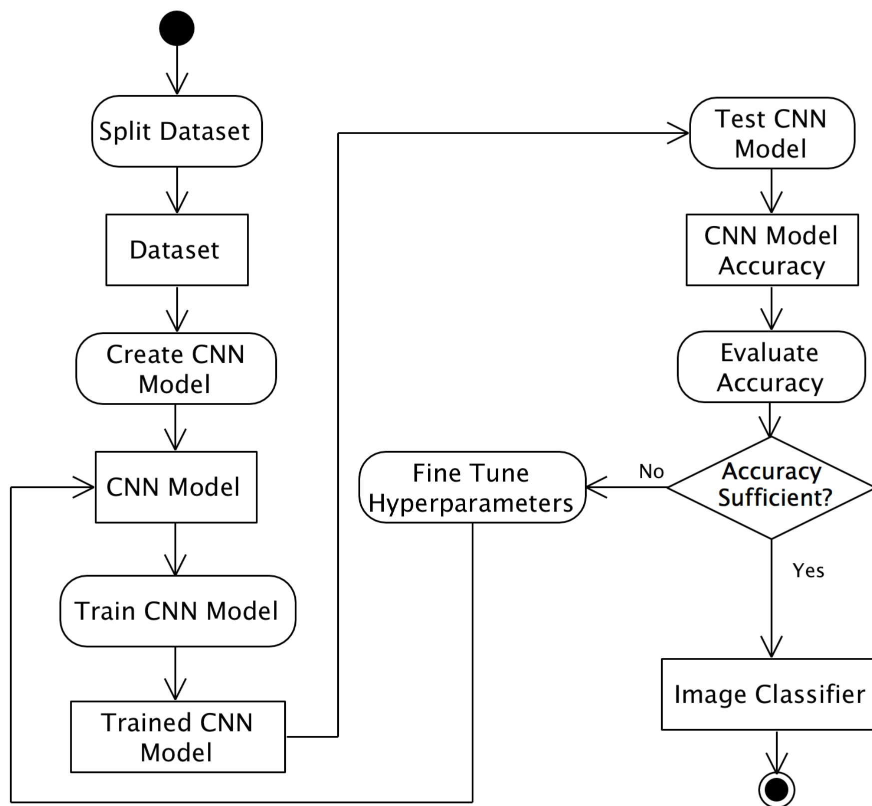


Figure 4.6: Activity Diagram describing the workflow required to train a CNN model and create an image classifier.

Chapter 5

System design

In the previous chapter, we have extracted the functional and non-functional requirements of our system. Subsequently, we broke down our system into use cases in the use case model. Furthermore, we have described the entities of our system and the relationships between them using an Analysis Object Models and we presented 3 activity diagrams that depict the workflows in our system.

In this chapter we map our analysis into the solution domain. We first identify our design goals in section 5.1. In section 5.2, we group the objects that we have identified in the analysis object models into subsystems, and we discuss the dependencies between those subsystems. In section 5.3 we present the hardware and software components used to implement our system. Lastly, section 5.4 presents the structure of our system's persistent data.

5.1 Design Goals

In the previous chapter we have defined our non-functional requirements. In this section, we use those non-functional requirements to extract our design goals, and we use those design goals as a compass for our system design. Defining our design goals allows us to make consistent design decisions across our different subsystems. Table 5.1 lists our non-functional requirements and their corresponding design goal. There are 5 types of possible design criteria from which the design goals can be selected: performance, dependability, cost, maintenance and end user criteria [3].

Requirement	Design Goal	Criteria
Adaptability	The system should create a convolutional neural network that can accommodate the addition of new classes.	Performance
Accuracy	The system should utilize real and synthetic images to train a CNN model to classify different small parts with high accuracy.	Performance

Table 5.1: Non-functional requirements and their corresponding design goals.

5.2 Subsystem Decomposition

Figure 5.1 depicts our system's subsystem decomposition. Our system is divided into 3 subsystems. **SyntheticImageSubsystem** is responsible for providing our system with synthetic images. It consists of 4 main components: **3DScanner**, **3DScanningSoftware**, **3DScanningSoftware**, **SyntheticScene** and **3DModelingSoftware**. The 3DScanner scans **SmallParts** and outputs raw 3D model, which are processes by the 3DModelingSoftware to create a **3DModel**. The SyntheticScene generates an artificial environment. The 3DModelingSoftware places the 3D models in the environment created by SyntheticScene, and renders **SyntheticImages** of the 3D models.

On the other hand, **RealImageSubsystem** is responsible for providing our system with real images. The **Camera** is used to take real images of the SmallParts. Next, the **RealImageProcessor** resizes the raw images to provide our system with **RealImages** that can be used out of the box.

The synthetic images created by the SyntheticImageSubsystem and the real images created by the RealImageSubsystem are both sent to the **ImageClassifierSubsystem**. The ImageClassifierSubsystem in turn combines both image sets to create a **Dataset**. The images are then fed to the **DatasetSplitter**, which is in turn responsible for splitting the data into a training set, a validation set and a testing set. The **ImageClassiefier** uses the training and validation sets to train a **CNNModel**. The CNNModel uses an **Optimizer** to adjust the model weights during the training phase. The testing set is then used to evaluate the accuracy of the trained CNNModel.

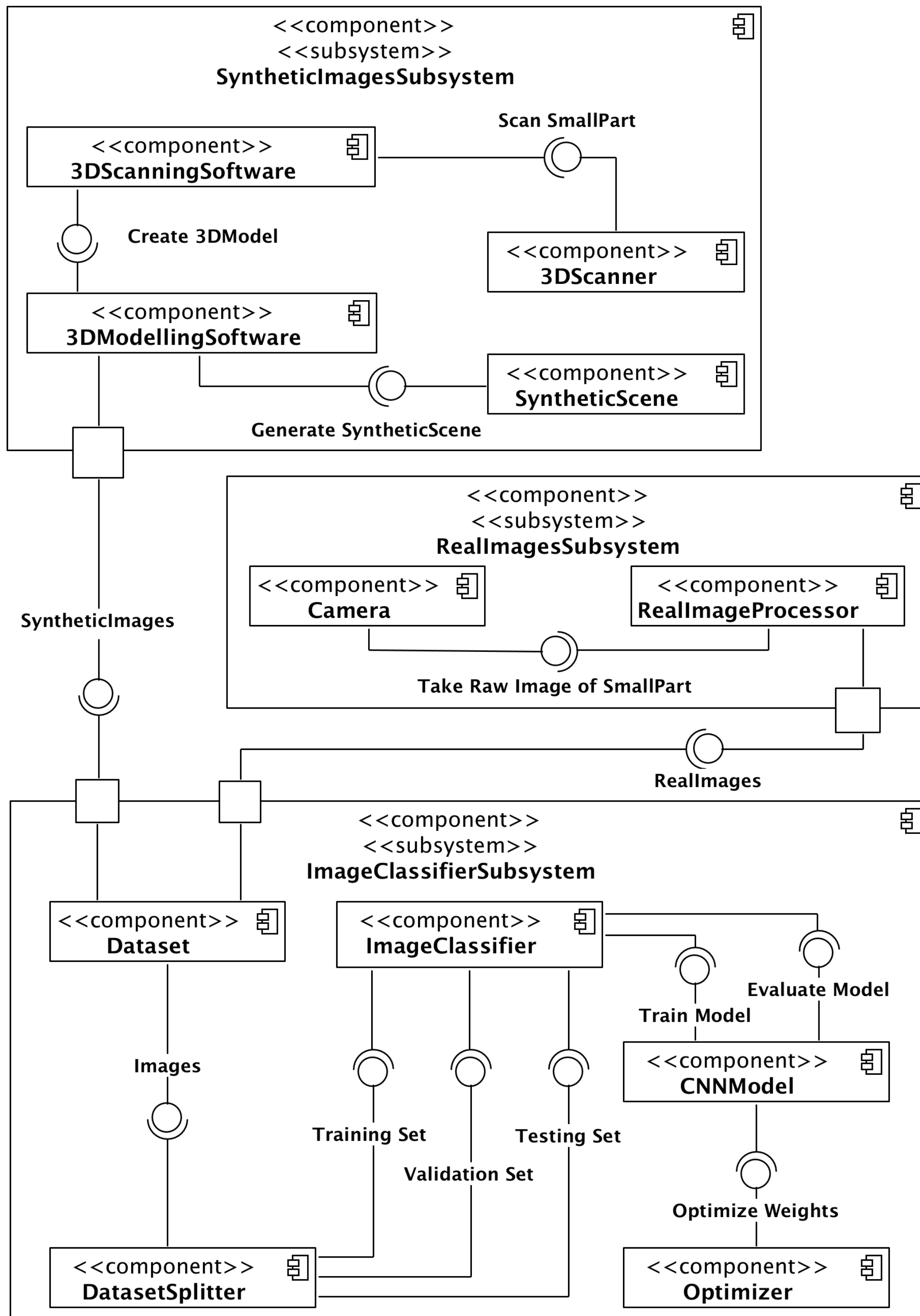


Figure 5.1: Subsystem decomposition showing the SyntheticImageSubsystem, the RealImageSubsystem, the ImageClassifierSubsystem and the services that they provide and consume.

5.3 Hardware Software Mapping

This section describes how the subsystems are mapped onto existing hardware and software components. Figure 5.2 depicts the deployment diagram of our system. The **SyntheticImageGenerator** device is used to run 2 execution environments: the **Artec Studio**[1] 3D scanning software, and the **Rhinoceros**[2] 3D modeling software. Artec Studio is used to process the scans produced by the **3DScanner**[3] device. The 3DScanner is the Artec Space Spider. The Windows version of Rhinoceros is chosen because it supports extra features that are not available on the Ubuntu or Mac versions. We use Rhinoceros to construct the SyntheticScene. Moreover, Rhinoceros hosts a **Python** execution environment. We use the Python interpreter to render synthetic images on Rhinoceros.

The **Camera** is used to capture raw images of the small parts. We use the Camera application on an iPhone 6. The **RealImageGenerator** runs a Python environment. We use the Python commands to resize the the raw images of the small parts captured by the camera. The resized real images are then ready to be fed to the image classifier.

The Dataset is stored in the file system of the **ImageClassifier** device. We use the device's Python environment to run the DatasetSplitter, the ImageClassifier, the CNNModel and the Optimizer. Furthermore, we use **Keras** [6], a neural networks API that provides out-of-the-box CNN implementations. The ImageClassifier device is equipped with a graphics processing unit (GPU). The GPU is used for parallel processing of the Convolutional Neural Network training algorithm. This enhancement significantly slashes down the time needed to train a CNN.

[1]https://www.artec3d.com/3d-software/artec-studio
[2]https://www.rhino3d.com
[3]https://www.artec3d.com/portable-3d-scanners/artec-spider

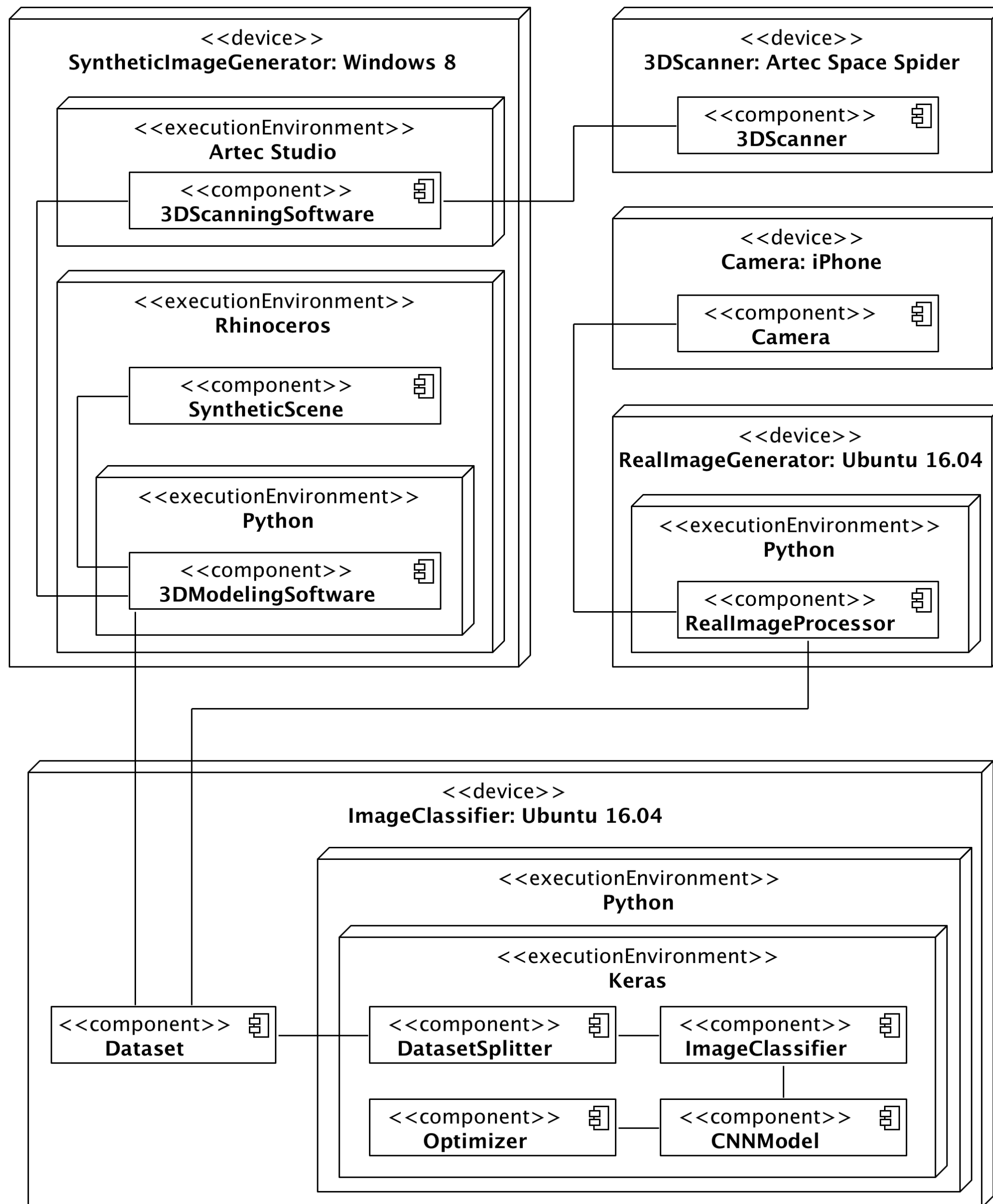


Figure 5.2: System Deployment Diagram depicting the different hardware and software components used to implement our system.

5.4 Persistent Data Management

The implementation of the ImageClassifier requires the split dataset to be organized in a certain way. The file structure of the dataset is shown in figure 5.3. The root folder contains 3 folders: one for training images, one for validation and one for testing. Each folder, in turn, contains one folder per class, and each of these folders are named after their respective class's label. Lastly, each class folder contains a set its own images.

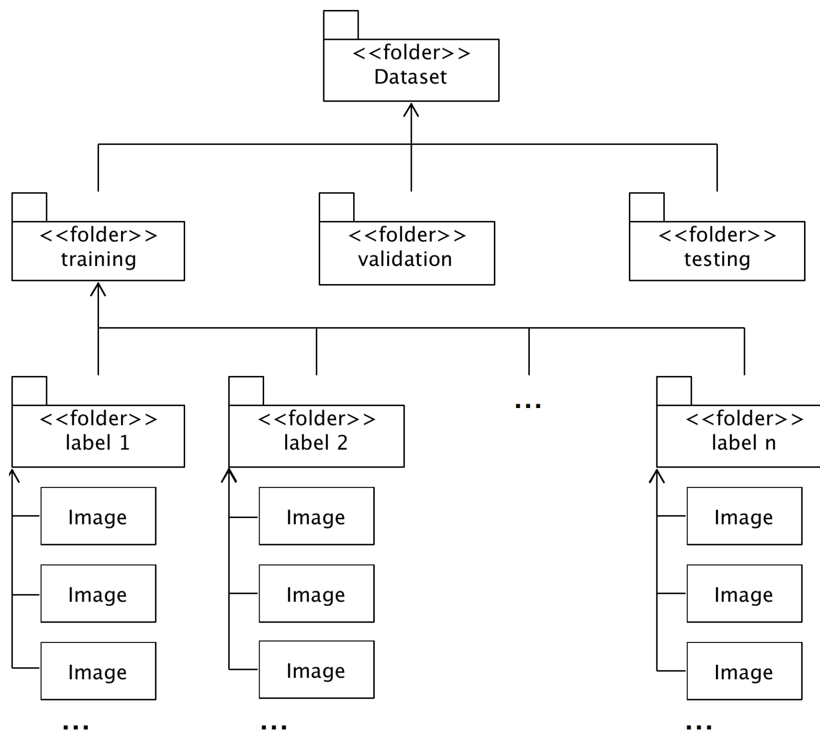


Figure 5.3: Dataset file structure model displaying how the split dataset is organized before being fed as input to the image classifier. Each subfolder in the training/validation/testing directory is named after a small part label.

Chapter 6

Object Design

In chapter 4, we identified the objects in our system by creating the analysis object model. In chapter 5, we mapped our objects to components and we defined the hardware and software platforms that host those components. During hardware/software mapping, off-the-shelf components that we use throughout our system have been identified. In this chapter, we close the gap between the application objects and the off-the-shelf components by identifying additional solution objects and refining existing objects [3].

Section 6.1 presents the Artec Space Spider 3D scanner and the Artec Studio 3D scanning software. It also describes how they are used to create 3D models. Section 6.2 presents the 3D modeling software Rhinoceros and how it is used to create synthetic images. Section 6.3 describes Pillow the python imaging library, and how it is used to process real images. In section 6.4, we introduce Keras and Tensorflow, and we describe how they are used to create an image classifier. Section 6.5 describes the VGG convolutional neural network algorithms used to implement the CNNModel. Finally, Section 6.6 presents the Adam optimization algorithm.

6.1 Artec Space Spider and Artec Studio

Artec Space Spider[1] is a high-resolution 3D scanner based on blue light technology. It is built for capturing small objects or intricate details of large industrial objects in high resolution, with steadfast accuracy and precise color.

Artec Studio[2] is a 3D scanning and data processing software. It is used hand in hand with the Artec Space Spider 3D scanner to create 3D models.

[1] https://www.artec3d.com/portable-3d-scanners/artec-spider

[2] https://www.artec3d.com/3d-software/artec-studio

6.1.1 Usage

We use the Artec Space Spider 3D scanner to scan our small part from multiple angles. The different scans are then sent to Artec Studio for processing. Artec Studio provides a functionality that detects features in multiple scans and automatically combines the scans to form a 3D model. Moreover, Artec Studio automatically removes outlier points in the scans. We also use Artec Studio to identify the horizontal plane where the small part was lying during the scan. The plane is then removed from the output 3D model. After the scans have been combined to construct a 3D model, Artec Studio automatically fills the holes in the object where there are missing scans.

6.2 Rhinoceros

Rhinoceros[3] (Rhino for short) is a 3D modeling software. Rhino can create, edit, analyze, document, render, animate, and translate curves, surfaces, solids, point clouds, and polygon meshes.

We choose Rhino to create our synthetic data because it has a python library called *RhinoScriptSyntax*. RhinoScriptSyntax provides an API for object transformation, scene manipulation and image rendering in python. Moreover, the Rhino software hosts a python interpreter and consequently supports running python scripts directly in the software. RhinoScriptSyntax and the hosted python interpreter are used to automate repetitive tasks that can otherwise consume more time if executed manually.

6.2.1 Usage

We use Rhino to create our synthetic scene. We set the background and the lighting conditions of the environment, then we import our 3D model and place it horizontally on the background.

Furthermore, RhinoSyntaxScript over python to is used to create our synthetic images. After setting up the synthetic scene in Rhino, we write a python script that executes random transformations over our 3D model. Next, a RhinoSyntaxScript function that renders and saves a 2D image is applied to the synthetic scene. In the python script, the desired number of output images is specified. Rhino repeats the transformation and rendering process accordingly.

[3]https://www.rhino3d.com

6.3 Pillow

Pillow is a python imaging library built on top of PIL (Python Imaging Library). Pillow is used for image manipulation in python. The *Image* module provided by Pillow has a function to open an existing image, resize it and save the new resized image. We use the Image module to implement our RealImageProcessor.

6.4 Keras and Tensorflow

Keras is a high-level neural networks API, written in Python and capable of running on top different backends such as Tensorflow, CNTK or Theano [6]. Keras focuses on being a user friendly API. It minimizes the number of actions required to develop common neural networks. Furthermore, Keras is designed modularly. Neural layers, optimizers, cost functions and regularization schemes are shipped as standalone components that can be easily used to create new models.

Our system runs Keras over Tensorflow. Tensorflow is an open source software library for high performance numerical computation and a machine learning framework [1]. Tensorflow is a flexible library that can be used to express neural network algorithms in a wide variety of domains.

6.4.1 Usage

Keras provides an out-of-the-box implementation of different CNN models, optimizers and cost functions. We took advantage of Keras's modular components to build and fine tune our CNN models and their respective optimizers.

Furthermore, Keras provides an image loader, called *ImageDataGenerator*. ImageDataGenerator feeds the training, validation and testing data to the CNN models from a folder directory. Consequently, we built our DatasetSplitter using Keras's image loader.

6.5 VGG CNN Algorithm

VGG is a convolutional neural network model created in 2014 by Karen Simonyan and Andrew Zisserman of the University of Oxford [42]. VGG was introduced in the ILSVRC 2014 competition, where the team came second on the image classification benchmark with a 7.3% error rate.

As shown in figure 6.1, the VGG CNN algorithm has different configurations. For example configuration D has 16 layers while E has 19 layers. The

convolution layers used in the VGG network use a 3x3 filter with a stride and a pad of 1, while the max pooling layers use a 2x2 filter with a stride of 2.

Figure 6.1 shows the different VGG architecture configurations presented by Simonyan and Zisserman [42]. We use configurations D and E to implement our CNNModel. We refer to them as **VGG16** and **VGG19** respectively.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 6.1: Different VGG architectures. We use configurations D and E.

6.6 Adam Optimization Algorithm

Adam [25] is a gradient descent optimization algorithm. It is used during the training phase of a network to update the model weights. Adam differs than the classical stochastic gradient descent (SGD) optimizer. While SGD maintains a constant learning rate for all weight updates, Adam calculates the exponential moving average of the gradient and the squared gradient. The Adam optimizer updates the weight of a network that uses the loss function L as follows:

$$g_t = \frac{\partial L}{\partial w_{t-1}}$$

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1 \cdot g_t)$$

$$s_t = \beta_2 \cdot s_{t-1} + (1 - \beta_2 \cdot g_t^2)$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{s}_t = \frac{s_t}{1 - \beta_2^t}$$

$$w_t = w_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{s}_t} + \epsilon}$$

where α is the learning rate, β_1 and β_2 are decay rates for the moving average of the gradient and the square gradient respectively, t is a time-step that starts at 0 and ϵ is a small constant (usually 10^{-8}) used to avoid division by zero.

The Adam optimization algorithm is more sophisticated than the classical stochastic gradient descent. However, we use both algorithms to implement the Optimizer class, which is in turn used to update the weights of the CNNModel during the training phase.

Chapter 7

Evaluation

In this chapter we describe how our system is utilized to run our experiments. We provide a quantitative analysis of the data that we gathered and the comparison between the different experiment settings. Section 7.1 defines out experiment objectives. Section 7.2 describes the implementation details of how we generate our dataset and perform image classification. In section 7.3 we provide the results of our experiment. Our findings and experiment limitations are described in sections 7.4 and 7.5 respectively.

7.1 Objectives

We hypothesize that we are able to enhance the classification accuracy of our image classifier using synthetic data for training. Multiple ratios of synthetic to real images for training are examined. Moreover, we run our experiments on multiple CNN architectures and fine-tune their hyperparameters accordingly.

7.2 Methodology

Our methodology is divided into 2 main sections. Section 7.2.1 describes 3D model creation and dataset generation. Moreover, it discusses generating real and synthetic images, and how they're collected from small parts and their corresponding 3D models. Section 7.2.2 presents image classification and how the collected images are utilized to maximize the classification accuracy of the image classifier.

7.2.1 Dataset Generation

During dataset generation, we strive to create a synthetic dataset that has a high degree of photo-realism. Simultaneously, we want to create 3D models as fast as possible to decrease the timing of the whole workflow. To achieve this balance, we aim to create an environment that is easy to model on 3D software. We attempt to eliminate light reflections and shadows and reduce the overall complexity of the scene.

Creating 3D Models

We first choose a small part that we wish for our image classifier to recognize. A 3D scanner is used to create a 3D model of the chosen small part.

We were unable to create 3D models with satisfactory quality for two reasons. Firstly, the small size of the target small parts requires a great level of precision to create an accurate 1:1 model. Secondly, the reflective surface of the small parts bounces the scanner's light off and causes the scanner to capture a lot of clutter. Figure 7.1 shows a small part and its corresponding scanned 3D model. The scanner is not able to accurately scan fine details like the small part threads.

To counter this problem, readily available 3D models are obtained. We download 3D model by searching the Traceparts website[1] for the small part's code name. Figure 7.2 shows each small part and its corresponding 3D model.

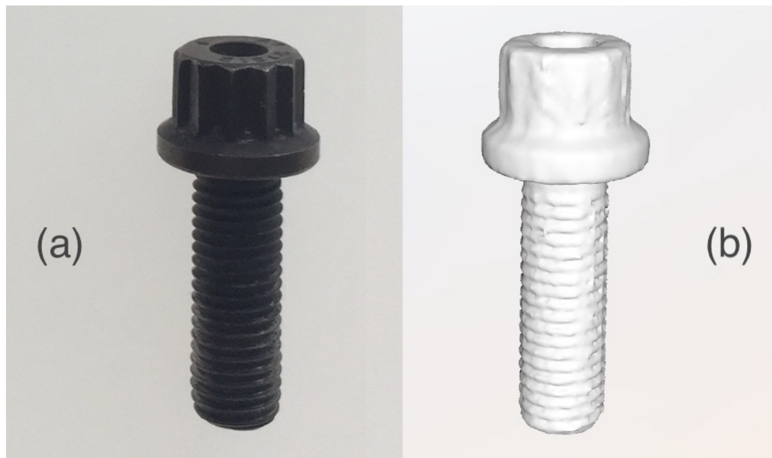


Figure 7.1: (b) is a 3D model of the small part (a) created by 3D scanning. The 3D scanner is unable to accurately capture fine details like the small part threads as shown in (b). The 3D model is rendered without texture for clarity.

[1] https://www.traceparts.com

Figure 7.2: Samples of images for each class that we use in our experiments. For each pair, the image on the left is a sample real image, while the image on the right is a sample synthetic image. The label under each image pair is the label we assign to the class.

Generating Real Images

We place the chosen small part on a horizontal plane and place a light source underneath. A plain, white, semi-transparent plane covered with opal foil is used as background to disperse the light source coming from underneath, and distribute the light evenly across the plane. We place the light source beneath the plane, rather than above, to eliminate any shadow that the small parts might cast. Moreover the dispersed backlight provides a lighting source without casting a direct light on the objects. A direct light might cause glare on the small part's reflective surface.

Next, an iPhone is placed over the plane, such that the small object is fully within the viewfield of the iPhone's camera. A clamp places the iPhone 133 mm over the plane as shown in figure 7.3.

Afterwards, we rotate and change the position of the small part randomly, while maintaining that the small part is fully within the viewfield of the camera. The iPhone's camera is used to take a picture of the small part. These steps are repeated until we obtain the desired number of real images.

Lastly, the raw real images taken by the iPhone camera are resized to conform with the height and width required by the image classifier.

Generating Synthetic Images

We start by creating the synthetic scene in the Rhinoceros 3D modeling software. First, a picture of the real horizontal plane is taken and used as a background for our synthetic scene. A synthetic lighting source is placed underneath the plane to mimic the lighting effect of the real environment. The 3D model is then placed on the horizontal plane of the environment.

Next, we generate a python script that uses the Rhinoceros library to manipulate 3D objects in the Rhinoceros software. The python script is the controller for the synthetic data generator. The synthetic data generator obtains the rotation and translation ranges of the 3D model. It also specifies how many images are to be generated. For each new image, the script generates a random rotation and translation value from within the defined ranges. It then applies those transformation to the 3D model and renders a new 2D synthetic image.

Dataset

The real images generated from the small part and the synthetic images generated from the corresponding 3D model are transferred to the Image-Classifier device, where they are placed in the dataset folder.

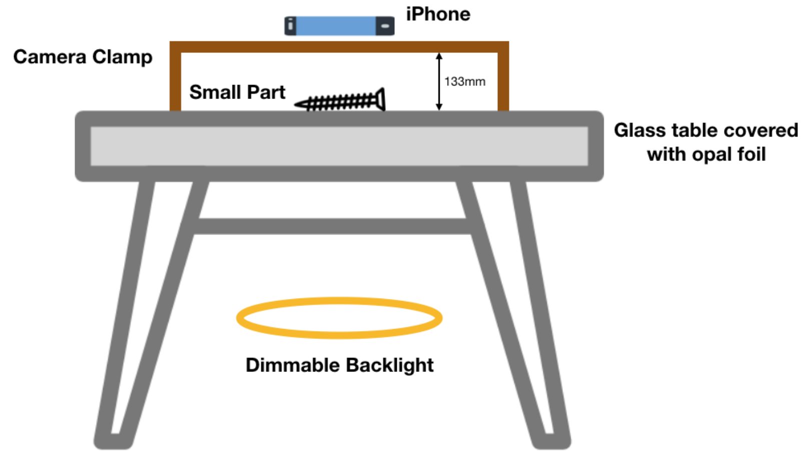


Figure 7.3: Sketch of the setup to generate real images. A clamp places the iPhone over a horizontal, backlit glass table with an opal foil. The small part rests on the table within the viewfield of the iPhone camera.

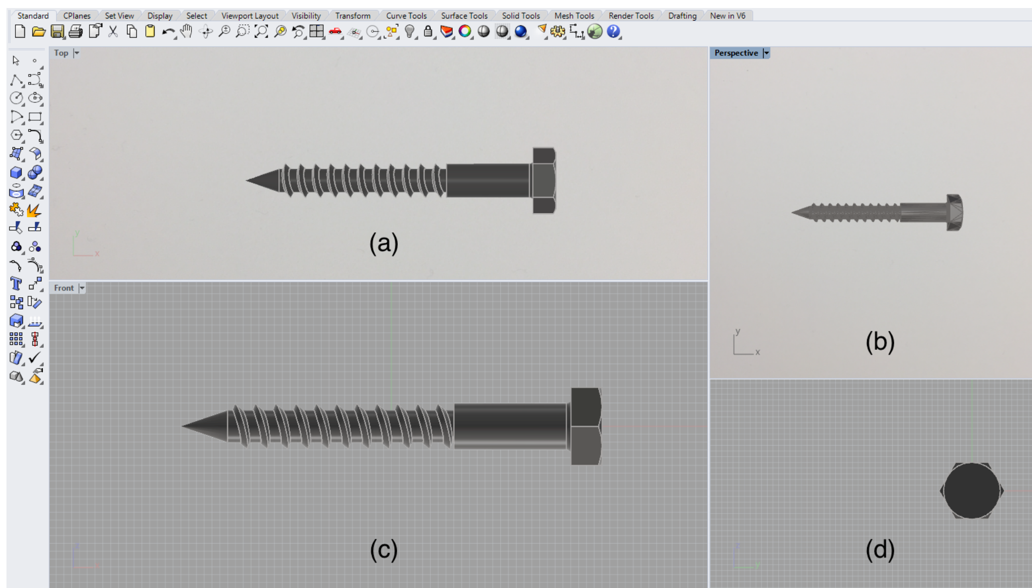


Figure 7.4: Synthetic scene in Rhino. The 3D model rests horizontally on a backlit plane to mimic the environment of the real setup. Sections (a) and (b) of the image depict the top view of the scene.

7.2.2 Image Classification

The image classifier uses the images that were generate to train a convolutional neural network model to classify images of our small parts. We describe the different dataset splits that were used in our experiment. Furthermore, we present the convolutional neural models and optimizers used for image classification.

Dataset Splitting

Firstly, the dataset is split into a training set, a validation set and a testing set. Moreover, we determine the number of synthetic images and the number of real images in our training set. The dataset is split into folders named after the label of their respective small part. The images are organized as shown in figure 5.3.

5 different ratios of synthetically enhanced training sets are compared Table 7.1 describe our different dataset splits, detailing the number of images per class for each split. We generate a fully synthetic training set $\boldsymbol{R_0 S_{100}}$, a 2.5% real training set $\boldsymbol{R_{2.5} S_{97.5}}$, a 5% real training set $\boldsymbol{R_5 S_{95}}$, a 10% real training set $\boldsymbol{R_{10} S_{90}}$, and a fully real training set $\boldsymbol{R_{100} S_0}$.

Dataset	Training (Synthetic)	Training (Real)	Training (Total)	Validation	Testing
$\boldsymbol{R_0 S_{100}}$	600	0	600	100	150
$\boldsymbol{R_{2.5} S_{97.5}}$	585	15	600	100	135
$\boldsymbol{R_5 S_{95}}$	570	30	600	100	120
$\boldsymbol{R_{10} S_{90}}$	540	60	600	100	90
$\boldsymbol{R_{100} S_0}$	0	600	600	100	100

Table 7.1: Different dataset splits that we use in our experiment. Each column details the number of images per class for the specified data split.

CNN Model

For each data split, the VGG16 and the VGG19 [42] convolutional neural network models are used. Moreover, the Stochastic Gradient Descent (SGD) and the Adam optimization algorithms [25] are used to optimize the network weights. We leverage the power of transfer learning [35] by using CNN models that are pre-trained on the ImageNet dataset [38]. Furthermore, our CNN models are tweaked to train and evaluate images of size 500px by 500px.

Hyperparameter Tuning

For each split dataset, the hyperparameters of the CNN model and the optimizer are fine tuned to maximize the classification accuracy of the output. Below is a list of hyperparameters that are optimized in the fine-tuning phase.

- **Number of Frozen Layers**: Our CNN models are pre-trained on the ImageNet dataset, which means that each layer is pre-loaded with optimized weights. During training, some of the early network layers are frozen to preserve these optimized weights. We optimize our classification accuracy by fine tuning the number of frozen layers.
  For VGG16, the first 10 or 14 layers are frozen. For VGG19, the first 12 or 16 layers are frozen.
- **Batch Size**: During training, our CNN model processes the training set in batches. We fine tune our batch size between 4, 8 and 16.
- **Number of Epochs**: Due to the iterative nature of the training process, a CNN has to train for multiple epochs until it reaches maximum classification accuracy. We train each model for 30 epochs and observe the peak accuracy during this range.
- **Optimizer Learning Rate**: For stochastic gradient descent and Adam, a learning rate of 0.0001 is used.

7.3 Results

Tables 7.2 and 7.3 provide the resulting class-wise classification accuracy of training a VGG16 network and a VGG19 network respectively. The displayed results are reached after fine-tuning the hyperparameters described in the previous section. The provided classification accuracy is the result of evaluating each CNN model using the respective testing set.

7.4 Findings

In our experiments, purely synthetic training sets are found to generate inferior results compared to purely real training sets. A VGG16 model trained on the $\boldsymbol{R_0 S_{100}}$ dataset has an accuracy of 83.556%, a whole 16.444% less that a VGG16 model trained on the $\boldsymbol{R_{100} S_0}$ dataset. However, A VGG16 trained on a training set with 10% real data produces comparable results to a training set with purely real data. We note that while using $\boldsymbol{R_{100} S_0}$ dataset,

the VGG16 network reached an accuracy of 100% on the test set, while using the $\boldsymbol{R_{10}S_{90}}$ dataset resulted in an accuracy of 97.037%. Moreover, the VGG19 network achieved an accuracy of 99.333% using the $\boldsymbol{R_{100}S_{0}}$ dataset, compared to an accuracy of 97.222% as a result of using the $\boldsymbol{R_{5}S_{95}}$ dataset.

Furthermore, we notice that the small parts with the highest class-wise accuracy are *Flat Head* and *Longhex Large*. Compared to their counterparts, those two small parts have a unique aesthetic. Hex Large and Hex small look similar, and the same goes for Mushroom Large and Mushroom Small.

Dataset	**Flat Head**	**Hex Large**	**Hex Small**	**Long-hex Large**	**Mush-room Large**	**Mush-room Small**	**Total**
$\boldsymbol{R_{0}S_{100}}$	99.333	48.000	98.667	96.667	80.667	78.000	**83.556**
$\boldsymbol{R_{2.5}S_{97.5}}$	100.00	79.259	99.259	99.259	83.704	80.741	**90.370**
$\boldsymbol{R_{5}S_{95}}$	100.00	95.833	100.00	97.500	90.833	75.833	**93.333**
$\boldsymbol{R_{10}S_{90}}$	100.00	94.444	98.889	100.00	98.889	90.000	**97.037**
$\boldsymbol{R_{100}S_{0}}$	100.00	100.000	100.00	100.000	100.000	100.000	**100.000**

Table 7.2: Class-wise classification accuracy of a VGG16 network trained on different ratios of synthetic data to real data.

Dataset	**Flat Head**	**Hex Large**	**Hex Small**	**Long-hex Large**	**Mush-room Large**	**Mush-room Small**	**Total**
$\boldsymbol{R_{0}S_{100}}$	100.00	81.333	88.667	92.000	98.667	52.667	**85.556**
$\boldsymbol{R_{2.5}S_{97.5}}$	100.00	100.00	86.667	99.259	99.259	90.370	**95.926**
$\boldsymbol{R_{5}S_{95}}$	100.00	100.00	96.667	98.333	100.00	88.333	**97.222**
$\boldsymbol{R_{10}S_{90}}$	100.00	90.000	96.667	98.889	98.889	96.667	**96.852**
$\boldsymbol{R_{100}S_{0}}$	100.00	100.000	100.000	100.000	99.000	97.000	**99.333**

Table 7.3: Class-wise classification accuracy of a VGG19 network trained on different ratios of synthetic data to real data.

7.5 Limitations

Our system is dependent on having access to the 3D model of the small parts in order to generate synthetic images. Moreover, the model's classification accuracy is dependent on the shapes of the small parts. As observed in the previous section, the class-wise accuracy for an object that is similar to other objects in the dataset is lower than an object that is aesthetically unique.

Chapter 8

Summary

In this chapter we recap the work that we have done. We summarize the status of the project by relating to the functional that were previously identified in section 8.1. In section 8.2 we present a conclusion of our results and in section 8.3 propose a direction for future work that can further advance the usage of synthetic data in image classification.

8.1 Status

We look back at the functional requirements that we have defined in chapter 4. We are able to create 3D models of our small parts using the 3DScanner device. However the output 3D models were imprecise and could not capture the details of the small parts. We instead downloaded readily available 3D models. Generating the synthetic image set is implemented using ready made 3D models, the creation of synthetic scenes and rendering 2D synthetic images in Rhinoceros. The transformation ranges for each 3D model are set in the python script that Rhino executes to generate the synthetic images. A camera is used to take real images of the small parts which are then resized by the RealImageProcessor component in the RealImageGenerator device. In the ImageClassifier device, the DatasetSplitter divides the images in the Dataset component into a training set, a validation set and a testing set. The ImageClassifier trains a CNNModel using the training and validation sets, and evaluates its classification accuracy using the testing set.

8.2 Conclusion

The main objective of our work was to decrease the amount of manual labor required to label a dataset of small parts. We described a system that

facilitates the usage of synthetic data in image classification. Furthermore, we carried out an experiment to evaluate the usage of synthetic images to train a convolutional neural network to classify images of small parts. Our results indicate that a CNN trained on a mixture of synthetic and real images can provide a comparable classification accuracy to CNNs that have been trained on purely real images. This is advantageous in situations where a large dataset of images is not available for new classes.

8.3 Future Work

We propose 3 possible directions for future improvements over the work that has been presented in our thesis. Firstly, our results indicate that, when adding new classes, the output classification accuracy is affected by the aesthetic similarity of small parts in the dataset. A logical next step is to evaluate the CNNs after increasing the number of output classes. Secondly, a possible improvement is to evaluate the usage of deeper CNN models such as the ones presented by He et al. [22] and Szegedy et al. [46]. Thirdly, the total number of synthetic images in the training set has not been varied throughout our experiments. A possible improvement in accuracy could be achieved by training the CNN models on a bigger training set containing a larger amount of synthetic images.

List of Figures

1.1 Small parts that were taken apart during an aircraft engine overhaul, and have to be sorted for reassembling. 1
1.2 The structure of a sample screw displaying the different associated measurements. 2
1.3 The automatic small part classification system. 3

2.1 A neural network with an input layer, two hidden layers and an output layer. Each neuron is fully connected to all the neurons in the preceding layer. 6
2.2 A 3x3 filter is applied to the source pixel. Each weight in the filter is multiplied by the source's neighboring pixel in the corresponding position. The sum of products is used to calculate the destination pixel's value in the output feature map. 7
2.3 A max-pooling filter of size 2x2 and stride 2 is applied to the source feature map. The maximum value of each filter is used in the destination feature map. 7
2.4 Architecture of the LeNet-5 convolutional neural network. . . 8
2.5 Screw (a) is identical to screw (b) except that screw (a) is 6mm shorter than screw (b). An overhead light shines on the 2 screws. The reflection changes the natural color of their surface. 10
2.6 A synthetic scene where a screw lies on a horizontal plane. (a) shows the isometric view of the scene while (b) shows the side view. 11

4.1 The use case model displays the DataGenerator and the MachineLearningEngineer and the actions that they can undertake. 20
4.2 UML package of the analysis object model which models dataset generation. 30
4.3 UML package of the analysis object model which depicts the objects that are used for image classification. 31

4.4 Activity Diagram depicting the workflow required for synthetic image set generation. 32
4.5 Activity Diagram depicting the workflow required for real image set generation. 33
4.6 Activity Diagram describing the workflow required to train a CNN model and create an image classifier. 34

5.1 Subsystem decomposition showing the SyntheticImageSubsystem, the RealImageSubsystem, the ImageClassifierSubsystem and the services that they provide and consume. 37
5.2 System Deployment Diagram depicting the different hardware and software components used to implement our system. . . . 39
5.3 Dataset file structure model displaying how the split dataset is organized before being fed as input to the image classifier. Each subfolder in the training/validation/testing directory is named after a small part label. 40

6.1 Different VGG architectures. We use configurations D and E. 44

7.1 (b) is a 3D model of the small part (a) created by 3D scanning. The 3D scanner is unable to accurately capture fine details like the small part threads as shown in (b). The 3D model is rendered without texture for clarity. 47
7.2 Samples of images for each class that we use in our experiments. For each pair, the image on the left is a sample real image, while the image on the right is a sample synthetic image. The label under each image pair is the label we assign to the class. 48
7.3 Sketch of the setup to generate real images. A clamp places the iPhone over a horizontal, backlit glass table with an opal foil. The small part rests on the table within the viewfield of the iPhone camera. 50
7.4 Synthetic scene in Rhino. The 3D model rests horizontally on a backlit plane to mimic the environment of the real setup. Sections (a) and (b) of the image depict the top view of the scene. 50

List of Tables

5.1 Non-functional requirements and their corresponding design goals. 36

7.1 Different dataset splits that we use in our experiment. Each column details the number of images per class for the specified data split. 51
7.2 Class-wise classification accuracy of a VGG16 network trained on different ratios of synthetic data to real data. 53
7.3 Class-wise classification accuracy of a VGG19 network trained on different ratios of synthetic data to real data. 53

Bibliography

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Aytekin, Ç., Rezaeitabar, Y., Dogru, S., and Ulusoy, I. Railway fastener inspection by real-time machine vision. *IEEE Transactions on Systems, Man, and Cybernetics: Systems 45*, 7 (2015), 1101–1107.

[3] Bruegge, B., and Dutoit, A. H. *Object Oriented Software Engineering Using UML, Patterns, and Java 3rd Edition.* Prentice Hall, 2009.

[4] Chang, A. X., Funkhouser, T., Guibas, L., Hanrahan, P., Huang, Q., Li, Z., Savarese, S., Savva, M., Song, S., Su, H., et al. Shapenet: An information-rich 3d model repository. *arXiv preprint arXiv:1512.03012* (2015).

[5] Chellapilla, K., Puri, S., and Simard, P. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition* (2006), Suvisoft.

[6] Chollet, F., et al. Keras. `https://keras.io`, 2015.

[7] Ciresan, D., Meier, U., Masci, J., Maria Gambardella, L., and Schmidhuber, J. Flexible, high performance convolutional neural

networks for image classification. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence* (2011), vol. 22, Barcelona, Spain, p. 1237.

[8] Cireşan, D., Meier, U., Masci, J., and Schmidhuber, J. A committee of neural networks for traffic sign classification. In *Neural Networks (IJCNN), The 2011 International Joint Conference on* (2011), IEEE, pp. 1918–1921.

[9] Cireşan, D., Meier, U., Masci, J., and Schmidhuber, J. Multi-column deep neural network for traffic sign classification. *Neural networks 32* (2012), 333–338.

[10] Cireşan, D., Meier, U., and Schmidhuber, J. Multi-column deep neural networks for image classification. *arXiv preprint arXiv:1202.2745* (2012).

[11] De Ruvo, P., Distante, A., Stella, E., and Marino, F. A gpu-based vision system for real time detection of fastening elements in railway inspection. In *Proceedings of the 16th IEEE international conference on Image processing* (2009), IEEE Press, pp. 2309–2312.

[12] Everingham, M., Van Gool, L., Williams, C. K., Winn, J., and Zisserman, A. The pascal visual object classes (voc) challenge. *International journal of computer vision 88*, 2 (2010), 303–338.

[13] Feng, H., Jiang, Z., Xie, F., Yang, P., Shi, J., and Chen, L. Automatic fastener classification and defect detection in vision-based railway inspection systems. *IEEE Trans. Instrumentation and Measurement 63*, 4 (2014), 877–888.

[14] Fukushima, K. Neocognitron: A self-organizing neural network for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics 36* (1980), 193–202.

[15] Georgakis, G., Mousavian, A., Berg, A. C., and Kosecka, J. Synthesizing training data for object detection in indoor scenes. *arXiv preprint arXiv:1702.07836* (2017).

[16] Georgakis, G., Reza, M. A., Mousavian, A., Le, P.-H., and Kosecka, J. Multiview rgb-d dataset for object instance detection. *arXiv preprint arXiv:1609.07826* (2016).

[17] Gibert, X., Patel, V. M., and Chellappa, R. Material classification and semantic segmentation of railway track images with deep convolutional neural networks. In *Image Processing (ICIP), 2015 IEEE International Conference on* (2015), IEEE, pp. 621–625.

[18] Gibert, X., Patel, V. M., and Chellappa, R. Robust fastener detection for autonomous visual railway track inspection. In *Applications of Computer Vision (WACV), 2015 IEEE Winter Conference on* (2015), IEEE, pp. 694–701.

[19] Gibert, X., Patel, V. M., and Chellappa, R. Deep multitask learning for railway track inspection. *IEEE Transactions on Intelligent Transportation Systems 18*, 1 (2017), 153–164.

[20] Girshick, R., Donahue, J., Darrell, T., and Malik, J. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2014), pp. 580–587.

[21] Haykin, S. *Neural networks: a comprehensive foundation.* Prentice Hall PTR, 1994.

[22] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 770–778.

[23] Hubel, D. H., and Wiesel, T. N. Receptive fields of single neurones in the cat's striate cortex. *The Journal of physiology 148*, 3 (1959), 574–591.

[24] Hubel, D. H., and Wiesel, T. N. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology 160*, 1 (1962), 106–154.

[25] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[26] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (2012), pp. 1097–1105.

[27] Lai, K., Bo, L., and Fox, D. Unsupervised feature learning for 3d scene labeling. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on* (2014), IEEE, pp. 3050–3057.

[28] LeCun, Y. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/* (1998).

[29] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. Backpropagation applied to handwritten zip code recognition. *Neural computation 1*, 4 (1989), 541–551.

[30] LeCun, Y., Boser, B. E., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. E., and Jackel, L. D. Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems* (1990), pp. 396–404.

[31] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11 (1998), 2278–2324.

[32] LeCun, Y., et al. Generalization and network design strategies. *Connectionism in perspective* (1989), 143–155.

[33] Marino, F., Distante, A., Mazzeo, P. L., and Stella, E. A real-time visual inspection system for railway maintenance: automatic hexagonal-headed bolts detection. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 37*, 3 (2007), 418–428.

[34] Masters, D., and Luschi, C. Revisiting small batch training for deep neural networks. *arXiv preprint arXiv:1804.07612* (2018).

[35] Pan, S. J., Yang, Q., et al. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering 22*, 10 (2010), 1345–1359.

[36] Peng, X., Sun, B., Ali, K., and Saenko, K. Learning deep object detectors from 3d models. In *Proceedings of the IEEE International Conference on Computer Vision* (2015), pp. 1278–1286.

[37] Rajpura, P. S., Bojinov, H., and Hegde, R. S. Object detection using deep cnns trained on synthetic images. *arXiv preprint arXiv:1706.06782* (2017).

[38] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M.,

Berg, A. C., and Fei-Fei, L. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV) 115*, 3 (2015), 211–252.

[39] Saenko, K., Kulis, B., Fritz, M., and Darrell, T. Adapting visual category models to new domains. In *European conference on computer vision* (2010), Springer, pp. 213–226.

[40] Sarkar, K., Varanasi, K., and Stricker, D. Trained 3d models for cnn based object recognition. In *VISIGRAPP (5: VISAPP)* (2017), pp. 130–137.

[41] Schmidhuber, J. Deep learning in neural networks: An overview. *Neural networks 61* (2015), 85–117.

[42] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[43] Singh, A., Sha, J., Narayan, K. S., Achim, T., and Abbeel, P. Bigbird: A large-scale 3d database of object instances. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on* (2014), IEEE, pp. 509–516.

[44] Stallkamp, J., Schlipsing, M., Salmen, J., and Igel, C. The german traffic sign recognition benchmark: a multi-class classification competition. In *Neural Networks (IJCNN), The 2011 International Joint Conference on* (2011), IEEE, pp. 1453–1460.

[45] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2015), pp. 1–9.

[46] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 2818–2826.

[47] Weng, J., Ahuja, N., and Huang, T. S. Cresceptron: a self-organizing neural network which grows adaptively. In *Neural Networks, 1992. IJCNN., International Joint Conference on* (1992), vol. 1, IEEE, pp. 576–581.

[48] Zeiler, M. D., and Fergus, R. Visualizing and understanding convolutional networks. In *European conference on computer vision* (2014), Springer, pp. 818–833.

[49] Zeiler, M. D., Taylor, G. W., and Fergus, R. Adaptive deconvolutional networks for mid and high level feature learning. In *Computer Vision (ICCV), 2011 IEEE International Conference on* (2011), IEEE, pp. 2018–2025.