An Industry 4.0 Deep Learning Neural Network Machine Vision System Classifier Trained with Synthetic Computer-Aided Design Training Data

by

BERNARDUS CHRISTIAN BOTHMA

Thesis submitted in fulfillment of the requirements for the degree

DOCTOR of ENGINEERING in ELECTRICAL ENGINEERING

in the

Department of Electrical, Electronic and Computer Systems Engineering

of the

Faculty of Engineering, Built Environment and Information Technology

at the

Central University of Technology, Free State

Promoter:

Prof. Nicolaas Luwes, D.Tech.

BLOEMFONTEIN

2025

Declaration

I, Bernardus Christian Bothma, Identity Number: ############ and Student Number ########, do hereby declare that this research project, which has been submitted to the Central University of Technology for the degree, Doctor of Engineering in Electrical Engineering, is my own independent work, complies with the Code of Academic Integrity, as well as other relevant policies, procedures, rules, and regulations of the Central University of Technology, and has not been submitted before by any person in fulfillment (or partial fulfillment) of the requirements for the attainment of any qualification.

Acknowledgments

I would like to acknowledge the following individuals and institute, without whom the completion of this dissertation would not have been possible:

- Firstly, to my Creator, Lord and Savior, The God Almighty, for giving me the determination, perseverance and capability to complete this project and thesis.
- To my parents who never gave up on me, and through prayer, blood, sweat and tears, leveled the road of my life just enough to allow me to grow, overcome and succeed.
- To my wife and children for all your support and sacrifice.
- To my study leader, Prof. Luwes, for your guidance, endless knowledge and wisdom, and your friendship during my studies.
- To the Central University of Technology and RGEMS Research Unit, for granting me the opportunity to undertake this project, for monetary assistance, and for the knowledge and experience gained during the project.

Abstract

Industry 4.0 describes the digitalization of industrial manufacturing processes, the integration of information and communication technologies, and the use of smart products and machines. Two of the most impactful aims of Industry 4.0 are to facilitate rapid adaptation in the manufacturing process to design changes and the reduction of batch sizes to as little as a batch of one item while maintaining quality and profitability.

The implication being that manufacturing systems would need to deal with very fast turnaround times, possibly having many different products, variations of products, and constant addition of new products on the same assembly line. This would require quality control systems, especially Machine Vision Systems, which are capable of rapid reconfiguration.

Current Machine Vision Systems are not suited for such an environment. This is largely due to the requirement of an Image Processing Engineer to handcraft the feature extractor of the Machine Vision Systems and then train the artificial neural network that performs the classification. This process normally requires several iterations to ensure accuracy and reliability and can be significantly prolonged with larger amounts of features or products that the system needs to process.

Deep Learning has proven to be very successful in various image processing domains and holds great potential in developing more flexible Machine Vision Systems that can be retrained as required. With Deep Learning feature extraction and classification are integrated into the same structure, and features are automatically learned. Deep Learning requires significantly more data and processing power for learning. While these problems can be overcome in many general use cases by making use of techniques like transfer learning, image augmentation, and modern GPUs to train the Deep Learning network, there are many specialized cases, such as automated visual inspection of a product, where there is simply not enough data available.

The development of a Deep Learning Machine Vision Systems Classifier would be of great advantage to any Industry 4.0 manufacturing system. However, obtaining the large amounts of training data required for training such a system can be labor-intensive, expensive, and, in some cases, not possible. This, however, raises a question: Would it be possible to create synthetic training data using computer-aided design and modeling tools and to train a Deep Learning system with the synthetic data?

This study aims to create a synthetic image dataset of basic electronic components by using computer-aided design and modeling tools to automatically generate large amounts of synthetic images, use image augmentation to expand the synthetic dataset, and then train a Deep Learning classifier model on the resulting synthetic data.

To achieve this, a literature study was done on the role of Deep Learning in Industry 4.0, Deep Learning Classifiers, training data requirements, and Blender as a platform for creating synthetic training data. Three-dimensional models of the electronic components were created and textured; an automated workflow was developed to generate and augment the synthetic data; a Deep Learning classification model was trained using the synthetic data; and lastly, the performance of the Deep Learning model was evaluated using a set of real-world testing images.

The results show that it is feasible to use synthetic data to train a Deep Learning model, provided that the synthetic data accurately represents the real-world counterpart. However, it also shows that synthetic data that does not accurately represent the real-world counterpart is of no use and will reduce the overall performance of the classifier.

Table of Contents

List of Figures

List of Tables

Glossary

3D	Three-dimensional
ANN	Artificial Neural Network
API	Application Program Interface
BSDF	Bidirectional Scattering Distribution Function
CAD	Computer-aided design
CG	Computer Graphics
CNN	Convolutional Neural Networks
CPS	Cyber-Physical Systems
DL	Deep Learning
DNN	Deep Neural Network
ECM	Electronic component model
FN	False Negative
FP	False Positive
HDRI	High Dynamic Range Image
ICS	Image classification systems
IoT	Internet of Things
ML	Machine Learning
MVS	Machine Vision Systems
PBR	Physically-based rendering
PCB	Printed circuit board
ROI	Region of interest
TN	True Negative
TP	True Positive

Chapter 1: Introduction

This chapter is the introduction to the study. Section 1.1 provides background on image classification and Machine Vision Systems. Sections 1.2 and 1.3 cover the problem statement and research objectives respectively, and Section 1.4 states the expected contribution. Lastly, Section 1.5 provides an overview of the dissertation.

1.1 Image Classification and Machine Vision Systems

Image classification systems (ICS) are used in many domains like biomedical imaging, robot vision, surveillance, and industrial visual inspection, to name a few [1]. A basic ICS consists of a camera, a computing device, and image processing software.

The camera is responsible for capturing and processing the light from a specific region of interest (ROI) and then producing a digital image. The computing device can either be discrete, such as a computer workstation, or integrated into the camera. This computing device normally has the necessary hardware to store, manipulate, and evaluate the image as required by the image processing software. The software is responsible for image processing tasks like pre-processing, object detection, pattern matching, feature extraction and analysis, classification, and proving an output of some sort, for example, a class label.

Automatic industrial visual inspection systems, often referred to as Machine Vision Systems (MVS), have been effectively used in industry for quality control purposes [2] for several decades. These systems expand the basic ICS by adding lighting, optics, and a communication interface to external process control hardware. Proper lighting of the real-world object is vital to ensure that important features are well contrasted and visible.

It also reduces the possibility of false feature detection caused by shadows. The optics ensure that the ROI is in focus, enabling the camera to capture a sharp image in which the feature can clearly be identified. The communication interfaces allow the system to act on the outcome of the inspection.

With the fourth industrial revolution just beyond the horizon, it brings with it entirely new paradigms to the manufacturing industry, like smart factories, manufacturing as a service, and the proposed capability of producing a high variety of goods in very low volumes; even single item batches, while maintaining quality and profitability [3]. MVS, and especially their image processing capabilities, needs to be adapted to operate in such a manner that even one-off items can be successfully quality inspected in a fast and efficient manner.

1.1.1 Image Processing Software

In this section, the following aspects of the image processing software will be discussed:

- The tasks of image classification and object detection.
- An overview of the modern MVS approach to image processing software.
- An overview of the Deep Learning MVS approach to image processing software.

1.1.1.1 Image Classification and Object Detection

Image classification and object detection are common tasks in image processing and are often applied in MVS tasks. Image classification is a process in computer vision that can classify and label an image according to its visual content into one of a number of predefined categories [1] [4]. A common output of classification would be a class number or a class label.

Object detection expands image classification. Object detection is concerned with locating objects in an image and then classifying each of the objects [5]. This normally requires an additional algorithm to first identify regions of interest in an image that potentially contain objects of significance. The regions of interest are then individually classified to identify the objects that they contain. A common output is the original image with labeled bounding boxes around the detected objects.

Fig. 1-1 Below shows an example of classification and object detection. An animal classifier might take the image on the left as an input and produce “CAT” as the output label. An animal object detector might take the image on the right as an input and produce a modified image with colored bounding boxes along with the respective labels “CAT” and “DOG” as an output.

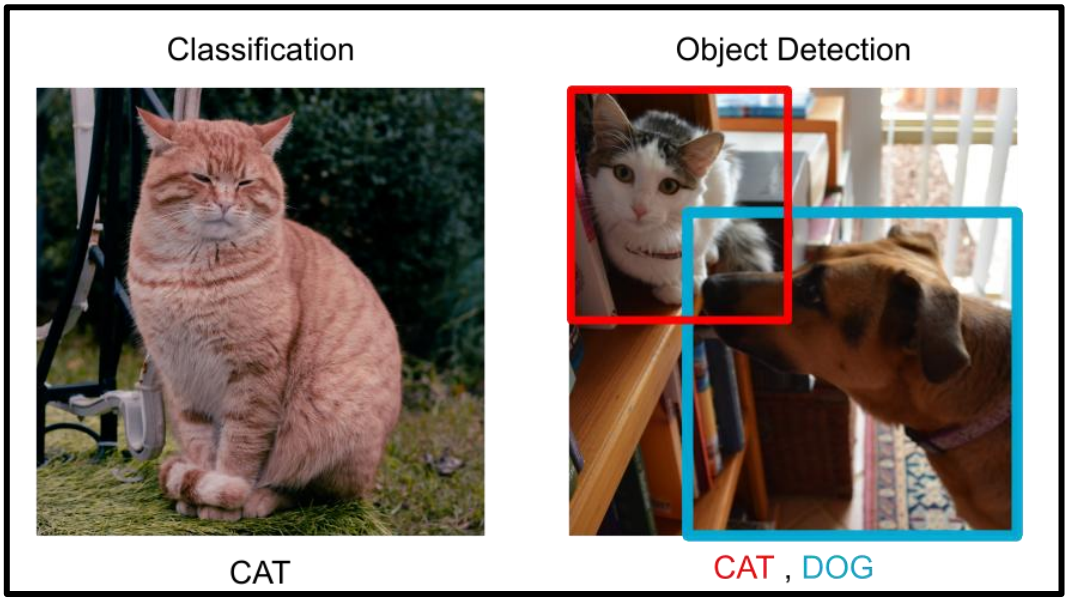


Fig. 1-1 Image Classification and Object Detection

Source: Adapted from [6] [7]

1.1.1.2 Modern MVS Approach to Image Processing Software

Modern MVS system image processing software consists of two main components. The first component is normally a handcrafted feature extractor that requires careful engineering and considerable domain knowledge to create [8] [9]. The second component is a classifier based on some sort of Machine Learning (ML) technique, often an Artificial Neural Network (ANN).

The feature extractor interprets the digital images captured by the MVS by making use of handcrafted features, transforming the raw pixel data in the input image into a feature vector. This feature vector is then used by the ML system to detect or classify patterns in the input image [8]. These MVS are very task-specific, are only able to perform quality control on a finite number of objects or configurations of objects, and can evaluate objects only under very strict conditions. These limitations make them unsuitable for an Industry 4.0 environment with a very high product variety and low production volumes.

1.1.1.3 Deep Learning MVS approach to Image Processing Software

Deep Learning (DL) MVS software consists of only one component, a deep artificial neural network, or Deep Neural Network (DNN). A DNN, although similar to a normal ANN, has two key differences: 1) It consists of many more layers, hence the "Deep" adjective; 2) It employs representation learning to automatically learn which features best represent a specific classification or object from the raw input image.

Fig. 1-2 contrasts a traditional ML-based approach to classification with a DL approach to classification as discussed below. The workflow at the top of the figure illustrates how the input data is passed to a handcrafted feature extractor, followed by the extracted features being passed to a trained ANN that will then produce an output vector that can be used to classify the input data.

In contrast with the DL classification approach, the input data is passed directly to a trained DNN that automatically identifies the relevant features and produces an output vector that can be used to classify the input data.

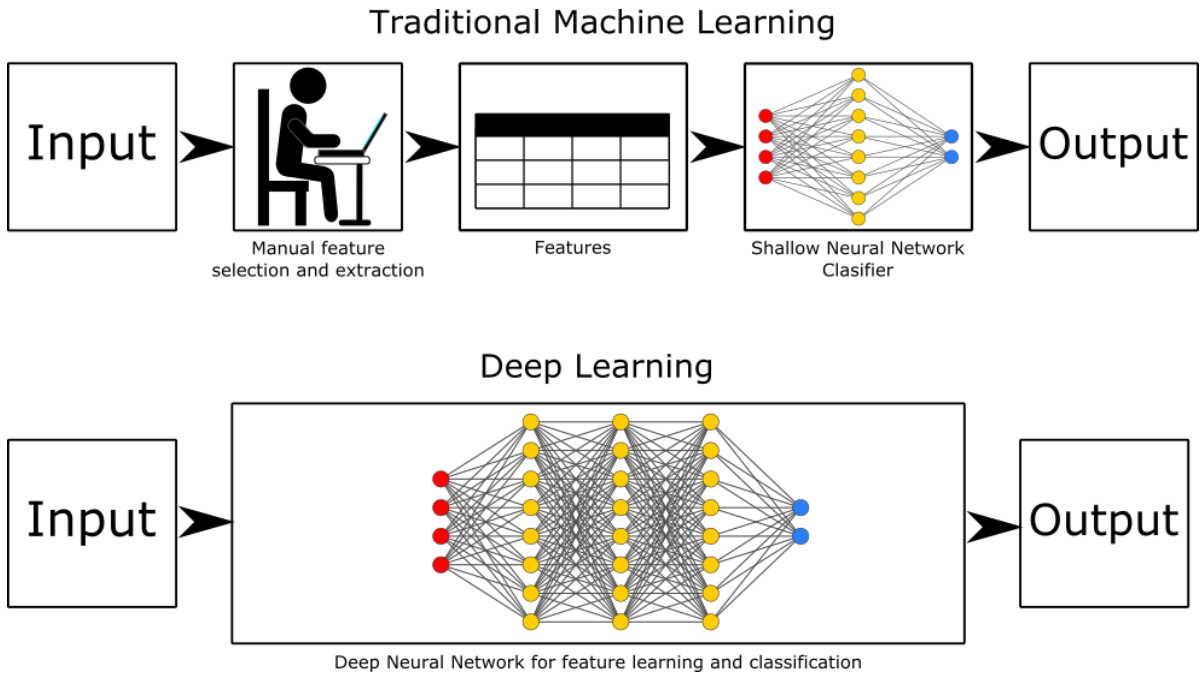


Fig. 1-2 Comparison of Traditional Machine Learning and Deep Learning

Convolutional neural networks (CNN), a DL technique developed for computer vision problems, have proven to be invariant to feature transformations like translation, rotation, and scale [10], allowing for better abstraction of features and more effective recognition of objects even if the actual image pixel values greatly differ. In recent years, DL models have proven to be more than capable of learning to classify large varieties of objects with high accuracy.

This ability to automatically learn relevant features does not come without cost, as DL has its own drawbacks. DL techniques require significantly larger training datasets compared to traditional ML systems; additionally, the deeper the network, the more processing power and other resources, such as memory, are required to train the network in reasonable time frames.

However, reduced training time due to the continual improvement of computational hardware (especially Graphical Processing Unit technology), the availability of larger training data sets, and improvements to the DL models and training techniques allow DL to be a viable approach [11] [12] [13]. In fact, MVS making use of DL has already proved to be effective for surface quality inspection [14] [15]. DL will be discussed in more detail in Chapter 2.

1.2 Problem Statement

MVS are time-consuming to develop and deploy, and in most cases, difficult to reconfigure. Additionally, MVS development can only be started when several samples of the product have been manufactured. Once sample products have been obtained, an Image Processing Engineer needs to develop a feature extractor and train an artificial neural network to perform the classification [8] [9]. The complexity and variety of the products have a direct effect on the time required to develop the system. Several iterations of the design process are normally needed to ensure accurate and reliable operation. This makes them unsuitable for a low-volume, high-variety production environment proposed by Industry 4.0.

DL offers a solution to these problems. DL algorithms allow for automatic feature learning and extraction [8]. However, these DL algorithms require large amounts of training data [16]. This can be addressed by using image augmentation techniques to increase whatever data is available and transfer learning to reduce the amount of data required [17] [18].

Furthermore, by using images rendered from 3D models created with computer-aided design (CAD) software, along with image augmentation techniques and transfer learning, it is possible to train a DL algorithm to perform tasks such as image classification and object detection on products, in this case electronic components on printed circuit board (PCB), before being exposed to a physical instance of the product. This also allows the MVS to be reconfigured for detecting and classifying new components and inspecting a new PCB while the design of the circuit is being finalized and before a physical PCB has been manufactured.

1.3 Research Aim and Objectives

The aim is to develop a DL-based MVS that is trained on synthetic data and can classify electronic components on a PCB. The system's performance will be evaluated to determine whether it is viable to train a DL-based MVS using synthetic training data based on image renderings of 3D CAD models, as this would improve flexibility and reduce reconfiguration turnaround time.

This aim will be achieved by pursuing the following objectives:

- Select a DL model for classification.
- Generate the synthetic training data set. This includes:
    - Gathering or creating the 3D CAD models of the electronic components.
    - Creating the base training image dataset by rendering the images based on the 3D models of the components.
    - Expanding the training image dataset using image augmentation techniques.
- Train and fine-tune the DL model using the synthetic training data.
- Collect real-world images to create a test for evaluating the performance of the DL-based MVS.
- Evaluate the performance of the DL classifier using the real-world image test dataset.

1.4 Expected Contribution

The original contribution of this study is training a DL classification model that can be used as part of an MVS by using a completely synthetic image training dataset that will be created using image renderings of 3D CAD models based on a combination of through-hole and surface mount electronic components on PCB. This is ideally aimed at the Industry 4.0 environment, where the quality control system would be ready before the new manufacturing setup is done. This system leads to fast turnaround, flexibility, virtual commissioning, and artificial intelligence, which are all key aspects of Industry 4.0, making this a true Industry 4.0 DL neural network machine vision system classifier.

1.5 Hypothesis

A Deep Learning classifier trained on a synthetically generated image dataset can achieve an acceptable level of accuracy in classifying real-world images of electronic components, potentially enabling more adaptable Machine Vision Systems for Industry 4.0 applications despite the limited availability of real-world training data.

1.6 Overview of the Dissertation

The dissertation consists of five chapters. The following outline provides a brief overview of each chapter:

- Chapter 1 – Introduction: An introduction to the study. It includes the problem statement, research aim and objectives, and original contribution.
- Chapter 2 – Literature Review: A study of relevant literature to acquire preliminary knowledge needed for a DL MVS classifier. It covers the role of DL in Industry 4.0, DL image classifiers, the considerations regarding training data in DL, and Blender as a tool for creating synthetic data.
- Chapter 3 – Methodology: The hardware and software used for the research and the process of creating the 3D model, rendering and augmenting the synthetic training datasets, and validating the synthetic image training dataset.
- Chapter 4 – Results: The results from the synthetic data rendering and augmentation process and from training and evaluating the YoloV5 Classifier to validate the synthetic training data.
- Chapter 5 – Conclusion and Future Work: Conclusions drawn from the study are discussed along with possible future research.

Chapter 2: Literature Review

This chapter covers the literature study done to acquire preliminary knowledge needed for a DL MVS classifier. Section 2.1 covers the role of DL in Industry 4.0; Section 2.2 and its subsection reviews DL image classifiers; Section 2.3 covers the considerations regarding training data for DL and Blender as a tool for creating synthetic data.

2.1 Deep Learning in Industry 4.0

Industry 4.0, or the fourth industrial revolution, refers to the next phase in the digitization of the manufacturing sector, aiming to realize low-volume, high-mix production in a cost-efficient way [19]. Here Smart Factories [20] utilize Cyber-Physical Systems (CPS) [21] for the monitoring of physical processes, virtualizing the physical world, and decentralizing decision-making. The Internet of Things (IoT) [22] allows CPS to communicate and cooperate, in real-time, with each other and humans; and the Internet of Services [23] allows internal and cross-organizational services to be offered and utilized by the value chain's participants [24]. With Industry 4.0's focus on technology-driven automation and smart and intelligent systems, DL has the potential to play a vital role in building intelligent data-driven systems and has become core to achieving Industry 4.0's goals [25].

DL is a subset of ML based on artificial neural networks that makes use of multiple processing layers [8] [26]. This allows the DL neural networks (DNN) to make use of representation learning to capture complex structures of large datasets [8] [26].

Representation learning allows the DL techniques to learn and represent data at multiple abstraction levels [10]. Several low-level representations (features) are extracted at the first layer. These are then passed on to the next layer as input. This process is repeated for every layer in the network; however, each time, representation of a higher, slightly more abstract level is extracted [8] [9].

As an example, the first layer could extract edges, the second layer could extract corners and contours, the next, basic shapes, etc. until the nth layer, where complex features like arms, legs, or faces can be differentiated. The key aspect of this process is that the relevant features are learned from data using a general-purpose learning procedure and not designed by human engineers [8]. This is a great advantage, especially in the context of smart manufacturing [9].

The first DL models were based on Feedforward Multilayer Perceptron networks [11], with multiple hidden layers. It became apparent that as the number of layers increased, it became increasingly difficult and time-consuming to train the models and optimize the parameters [9]. In many cases, this led to overfitting or local optimization problems. However, the Deep Belief Network introduced by Hinton et al. in 2006 reported satisfactory performance [9] [10], which led to renewed interest in DL, and since then the field has seen significant development and improvement, especially in terms of training [5] [27], with modern DL models outperforming traditional Artificial Neural Network (ANN)-based techniques in a variety of domains, including natural language processing, image classification, and object detection.

In the context of the fourth industrial revolution, the literature suggests that these modern DL algorithms are considered a key technology for enabling smart manufacturing [9], and they have a significant role to play as they prove to have superior performance over traditional ML systems in several key scenarios [28]. Considering the potential of this new generation of DL algorithms, Hernavs et al. provide several examples of the application of DL to different manufacturing scenarios, including: descriptive analytics for product quality inspection, diagnostic analytics for fault assessment, and predictive analytics for defect prognosis [9] [28].

2.2 Deep Learning Image Classifiers

Discriminative DL architectures are normally designed to classify visual data into a set of classes [25]. Popular discriminative architectures include Recurrent Neural Networks, Multi-Layer Perceptron, and CNN.

Inspired by the neuron arrangement of the visual cortex [29], CNNs have historically shown excellent performance at tasks such as hand-written digit classification and face detection; however, in the last decade, their deep variations have shown that they can do the same with much more challenging visual classification tasks [12].

In 2012, Krizhevsky et al. proposed a Deep CNN architecture, often referred to as AlexNet [30], that won the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) that year with higher accuracy than contemporary ML models of the time [13]. It introduced several new concepts and approaches, such as making use of multiple convolutional kernels to extract features from an image [31], introducing dropout layers [32] and the ReLU activation function [33], while using model parallelization to allow model training on GPUs [13] [34], and using augmentation to artificially enlarge the training dataset and reduce overfitting in training [13] [29].

This caused a significant interest and research in Deep CNNs leading to the rapid development of new models, each bringing forth improvements and, in some cases, new concepts. Some notable mentions include: VGG (2014) [35], GoogLeNet/Inception (2014) and its successors [36] [29] [37] [38], ResNets (2016) [39], and EfficientNet (2019) [40].

With the exception of a few unique types of layers, like the Inception module used in Google's GoogLeNet, all the models are constructed from similar layers [41]. As an example, consider the basic CNN model as illustrated in Fig. 2-1. It uses three different types of layers, namely convolutional-, pooling-, and fully-connected layers. Several convolutional and pooling layers are combined and used for feature learning, while the fully connected layer is used to perform the classification. The subsections below will discuss each of these layers in more detail.

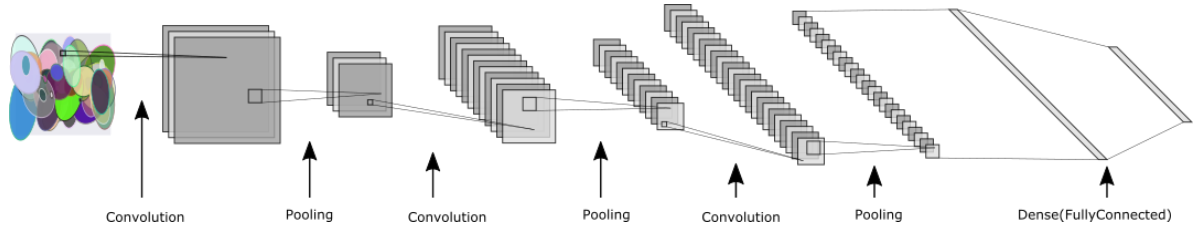


Fig. 2-1 Example of a Simple Deep CNN Model

2.2.1 The Convolution Layers

The convolution layers are used to extract the features. This is achieved by employing various convolution kernel filters to convolve with the raw input data and subsequent feature maps to generate multiple higher-level feature maps containing invariant local features [9] [10]. The raw numeric information of the feature maps at any given layer is normally stored in a third-rank tensor [42]. Similar to the neurons of traditional ANNs, each kernel filter contains a bias and an activation function. The latter introduces nonlinearities that assist the Deep CNN in detecting nonlinear features [41]. Unlike traditional ML algorithms that made use of the more computationally expensive tanh and sigmoid activation functions, many Deep CNNs make use of the faster ReLU activation function [33]. Fig. 2-2 illustrates the operation of a convolutional kernel filter.

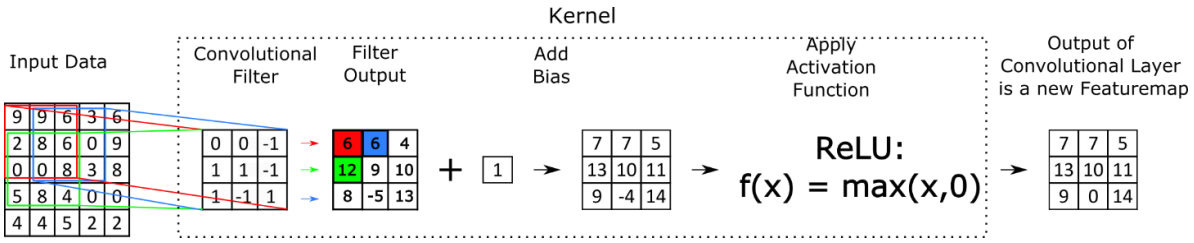


Fig. 2-2 Operation of a Convolutional Kernel with a ReLU Activation

2.2.2 The Pooling Layers

A pooling layer normally follows each convolutional layer. The purpose of pooling layers is to promote shift-invariance by subsampling of the input data [10] [41]. Although pooling leads to information loss, it has the benefits of reducing computational overhead in higher layers and reducing the risk of overfitting [10].

2.2.3 The Fully Connected Layer

After feature extraction is performed by several convolutional and pooling layers, one or more fully connected layers are added to the network to perform high-level reasoning [41] [9]. Here the 2D feature maps are converted into a feature vector that can either be used as categories for classification or utilized for further processing [10].

2.2.4 The YOLO Object Detector Family

While many of the classification models mentioned in Section 2.2 were later used as the backbones for two-stage object detectors [43], the selected classifier for this study started a dedicated single-stage object detector [44] and was later adapted to support classification.

Redmond et al. introduced the You Only Look Once, or YOLO [45], a family of single-stage object detectors in 2016 and followed by improved revisions in the following years [46] [44] [47] [48]. YoloV3 was faster than most of the leading object detectors of its time while maintaining comparable performance. It did, however, struggle to detect smaller objects.

Even though Redmond et al. stopped the development of the YOLO model after YoloV3, its open-source nature resulted in the continual development and improvement by several different groups. The YoloV5 model [49], developed and released by Ultralytics, became popular because of its ease of use and integration with the PyTorch framework [50], relatively fast training times, and implementation flexibility. The latter comes from the availability of several different implementations of the model offering a tradeoff between hardware requirements, speed, and accuracy. Later revisions of YoloV5 added support for image classification and segmentation, increasing application flexibility by offering a single solution for different visual identification problems. This makes it easy to test a concept using classification and the later transition to an object detection or segmentation approach.

2.3 Training Data

DL models require significantly more training data than traditional ML algorithms. Literature suggests that more data generally leads to better-performing DL models with more accurate results, especially with more complex models [51] [16] [17]. It is also clear that attempting to train a DL model with small datasets can lead to overfitting [52], a condition where the DL model overtrains and performs well on the training but is unable to generalize the problem, resulting in a reduction in its accuracy or even failure with unseen data [53] [46] [29].

The training data sets for DL classifiers and object detectors are made up of large sets of labeled or annotated images. For DL classifiers, the images only need to be labeled according to the class they represent. This is normally done by making the class label part of the image file name or by using the class label as the name of the folder that contains all of the images related to that class.

Obtaining a large enough training dataset can be challenging. Even with the availability of several open image databases like Coco [54] [55] and ImageNet [56] [57], there are many domains where there are simply very few or no image samples available [17] [58] [59]. Creating such large datasets is often labor-intensive and very expensive to the point of not being economically feasible, and in certain scenarios, such as bioinformatics information, it might even be very difficult or impossible to access in practice [59] [58] [60] [32]. Even if these problems are overcome and a costly tailor-made dataset is acquired, it may have serious biases and become out of date [32] [58].

Literature suggests several strategies to address this problem of overfitting [17], however data augmentation [61] [17] and transfer learning [18] [62] are most commonly used with limited training data. Transfer learning involves training a DL model on available large datasets, such as COCO, that are not necessarily related to the problem domain and then fine-tuning the training on a smaller domain-specific dataset. Data augmentation utilizes random image manipulation techniques, for instance, translation, rotation, applying noise and other filters, etc., to the images of the base training dataset, thereby creating a more diverse and larger training dataset.

Another approach that can be taken is to artificially enlarge the training dataset with synthetic data. Using three-dimensional (3D) modeling tools, one can create a 3D Computer Graphics (CG) scene composed of 3D models of the objects related to the problem, background information, lighting information, etc. 2D images can then be rendered, and accurate ground truth labels or bounding box annotations can be extracted from the 3D CG scene [63]. This process still involves some manual work. However, it is a one-time task that can result in the generation of nearly unlimited labeled or annotated training images. The CG scene can also be updated, and new annotated or labeled images can easily be generated. This ensures that the training dataset can be adjusted to different problems and that any biases identified in the future can be addressed.

2.3.1 Blender as a Platform for Creating Synthetic Train Data

When selecting the software to create the synthetic training data several cost, 3D modeling capability, scene creation tools, rendering capabilities, and scripting tools for automatic annotation generation and process automation. Popular commercial software like Maya and 3ds Max offer a verity of tools for professional 3D modeling, texturing and animation. Unreal Engine and Unity primarily used for game development and although they do provide tools for basic 3D modeling their strengths are more focused on level design, animation, and rendering. They normally make use of existing 3D models that are either part of commercial 3D asset libraries or created using software like Maya, 3ds Max, or Blender. For cases where synthetic video data is required Unreal Engine and Unity are good contenders. However, for this study where advanced modeling capabilities are required and synthetic images are the goal, they only present additional complexity in the workflow.

Blender is a cross-platform, free, and open-source 3D creation suite that offers features comparable to Maya and 3ds Max [64] [65]. It provides a complete set of 3D pipeline tools for a wide range of tasks such as modeling, texturing, rendering, rigging and animation, compositing, etc. [65]. Blender also supports task automation and the development of specialized tools through its Application Program Interface (API) for Python [64]. Using this API, Python scripts can be created to programmatically manipulate all aspects of the models, scene, lighting, camera position angles, etc. This makes Blender a good platform for creating synthetic data for DL classifiers and object doctors. In fact, several projects have already used Blender to produce synthetic data [59] [63].

The following sections will provide a brief overview of Blender version 3.4's features that are most relevant to developing synthetic data for DL image classifiers and object detectors.

2.3.1.1 Workspaces

Blender provides several standard workspaces that facilitate specific tasks in the 3D pipeline. Each of these workspaces is preconfigured with specific editors, toolbars, and headers depending on the purpose of the workspace. The workspaces can be modified, and new custom workspaces can be configured to suit the required workflow. The workspaces of note for this study include:

- The Layout workspace is used to set up the 3D scene. Here objects, such as meshes, surfaces, text, lights, cameras, etc. [66], can be added, moved (translated), rotated, or scaled as is needed to create the desired scene. Objects are often viewed as solid, untextured shapes in this workspace.
- The Modeling workspace provides tools, such as Bevel, Loop Cut, Extrude, Smooth, etc. [66], to manipulate the shape of individual objects. In this workspace, objects are often viewed in 'Wireframe' mode [67] that displays the object with transparent faces and emphasizes the edges and vertices. This allows more precise modification of the object.
- The Shading workspace is concerned with the visualization and material characteristics of the object. This workspace provides a Shader Editor that allows one to configure the color, material, and texture of the object using a node system and Material nodes [67]. The Shading workspace makes use of the Viewport Shading Mode to display objects, allowing an immediate preview of the material characteristics that are applied to the objects [67]. The node system, Material and Shader nodes, and shaders will be discussed in more detail in Section 2.3.1.2.
- The Geometry Nodes workspace also makes use of a nodes system; however, it utilizes Geometry nodes to procedurally model an object's mesh [68]. This allows for a much more flexible modeling process and dynamic modifications to a model's mesh. Objects are displayed as either solid, untextured shapes or as wireframes. Geometry nodes will be discussed in more detail in Section 2.3.1.2.

- The Rendering workspace allows one to watch how the final image is rendered and then to view and analyze the result [66].
- The Scripting workspace is where one can make use of Python and Blender's Python API to write scripts and even automate tasks.

2.3.1.2 The Blender Node System, Shader Nodes and Geometry Nodes

Blender's node system is a visual programming system that allows easy configuration and manipulation of material shaders and meshes with a node graph. Each node has a specific function, and most nodes have a combination of input and output sockets [67]. Input sockets are used to adjust the parameters that control the behavior of a node, while output nodes are used to connect the output of one node to the input of another node. These nodes can then be connected to form a complex network that describes a data-processing pipeline where data flows from nodes describing data sources to other nodes that process the data, and then to nodes that output the result [69].

These node networks allow one to easily create dynamic, complex, and procedurally generated geometry or shader materials. Node graphs can be created in different workspaces by using the appropriate node editor, e.g., the Shader editor for Material and Shader nodes [66].

2.3.1.2.1 Materials and Shader Nodes

Blender's rendering engines, CYCLES and EEVEE, make use of a shading and rendering methodology called physically-based rendering (PBR). PBR makes use of realistic shading and lighting models and takes into account how light interacts with surfaces, aiming to reproduce real-world materials as accurately as possible [67].

Material nodes are used to define these PBR-based materials and their characteristics. These nodes influence how the objects are visualized in the 3D scene and can be used to define sophisticated photorealistic materials [67]. Blender provides many nodes to manipulate lighting, texture, color, etc. However, the primary type of node that is used when creating a material is the Shader node.

Shader nodes determine the characteristics and shading of a material [67]. Many shaders make use of a Bidirectional Scattering Distribution Function (BSDF) that defines aspects such as light transmission, reflection, and refraction of surfaces [69]. Blender provides several different Shader nodes, for instance the Diffuse BSDF, Glossy BSDF, Glass BSDF, Specular BSDF, etc. [67]. However, the default and most widely used is the Principled BSDF node [66]. The Principled BSDF node combines the Anisotropic BSDF, Diffuse BSDF, Glass BSDF, Glossy BSDF, and Subsurface Scattering nodes into a single node that can be used to create materials ranging from glass to iron to concrete and much more [67].

2.3.1.2.2 Geometry Nodes

Geometry nodes work similarly to material nodes; however, they are used to procedurally create complex and dynamic geometries. These nodes are a powerful tool for editing objects' geometry, allowing for greater flexibility and control over objects' geometry through procedural modeling [67] [68]. Geometry Nodes can be used to modify and create meshes, curves, point clouds, volumes, and instances [68].

Geometry nodes are useful in a variety of cases, from creating single objects with dynamic geometry features to entire scenes with a large number of objects that need some variation, e.g., a model of a tree that can be tweaked to have different appearances based on several parameters or a forest of trees where each tree is slightly different from the others.

2.3.1.3 Lighting and Cameras

Besides the materials that are assigned to objects, lights and cameras have a significant influence on the final render of a scene.

Camera objects provide the functionality to render images from the 3D scene [66]. They can be precisely positioned to frame the exact part of the scene that will be rendered [67]. Similar to real cameras, Blender's camera object provides several settings that can influence parameters such as focal length, aperture, depth of field, the dimensions of the rendered image, etc. [66].

Blender provides several light sources that can be used separately or together to create very specific lighting conditions.

- Point lights are like lightbulbs. Depending on the radius setting of the point light, light radiates uniformly in all directions from either a point or a spherical surface. Light intensity is influenced by the power setting, in Watts, of the light, the radius setting of the light, and the distance that the light is from an object [66].
- Spotlights work similarly to point lights but produce conical-shaped directional light that can be used to light a specific part of a scene. The area that is covered by the spotlight is determined

by a combination of the size and blend settings [66]. The size setting controls the emission angle of the light cone at the light source, which determines the area that will be covered. The blend setting, in turn, controls the boundary point at which the spotlight's intensity starts falling off and starts to blur and soften [66].

- Area lights radiate light from the area of an elliptical, rectangular, or similarly shaped pane [67] and are used where soft diffused light sources are needed, e.g., studio lights used in photography. These lights process shadows with soft borders and can be in different shapes [66].
- The Sun light is used to simulate natural daylight. It emits light from a single direction with a constant intensity that does not fall off regardless of the distance between the light source and the objects in the scene [66]. The intensity of the light can be controlled with the Strength setting in Watts/m^2 [67].
- High Dynamic Range Image (HDRI) Lighting provides a form of indirect lighting [67]. HDRI maps have a far greater dynamic range than digital imaging techniques, making them ideal for representing the natural lighting conditions of real-world scenes and transmitting them to the scene as lighting and reflection [66] [67]. Unlike other light sources, HDRI maps are not assigned using light objects. HDRI maps are assigned to the world environment by using the Shader editor and the Environment texture node [67]. This allows one to fine-tune the environmental lighting effects of the HDRI using the nodal system.

2.3.1.4 Blender's Render Engines

A render engine takes the information from the materials, lights, and camera and performs the actual rendering process. Blender provides three builds in render engines, i.e. Eevee, Cycles, and Workbench [66]. The Workbench Engine is not meant to render final images; rather, it is optimized for fast rendering and displaying a scene in the 3D Viewport while it is being worked [66].

The Eevee and Cycle engines are used for rendering the final output images. Although Eevee and Cycles have a few things in common, like using the same Shader nodes to define materials and using the same light and camera objects, they have different approaches to performing the rendering process [66].

Eevee is an OpenGL-based real-time render engine focused on speed and interactivity and is able to render PBR materials [70]. Eevee uses rasterization to render images. Rasterization utilizes several different algorithms to estimate how light interacts with objects and materials [66] [71]. Eevee also has several limitations when compared to Cycles [66].

Cycles is a physically-based path tracer [66] that utilizes Ray tracing [72]. While this allows for more accurate lighting simulation, leading to more photorealistic renders [73], it is computationally more expensive and often more time-consuming. To compensate for the computational overhead, Cycles supports GPU-accelerated rendering, which allows using a graphics card to speed up rendering [66].

2.4 Conclusion

In conclusion, DL is crucial for achieving Industry 4.0's goals by building intelligent data-driven systems for smart manufacturing. DL's representation learning extracts complex features from large datasets, outperforming traditional ML techniques. The YoloV5 model is highly flexible DL model that can be applied to different visual identification problems, offering the ability to perform object detection, image classification and segmentation. DL models require large amounts of training data. Blender, an open-source 3D modelling animation software package, provides several tools and Python scripting capabilities that can be used to model, texture, render and automatically annotate images that can be used create a synthetics training dataset for DL models like YOLO.

Chapter 3: Methodology

This chapter focuses on the process of creating and validating the synthetic image training dataset for five electronic components, namely a through-hole resistor, a through-hole electrolytic capacitor, a through-hole diode, a surface mount resistor, and a surface mount ceramic capacitor.

Section 3.1 briefly covers the computer hardware and software that is used. Section 3.2 and its subsections cover the process of creating the 3D models of the electronic components and setting up the pipeline to render the synthetic images. Section 3.3 covers the process of rendering and augmenting the synthetic training datasets. Section 3.4 covers the process of evaluating the synthetic training datasets by using them to train a YoloV5 classifier and then evaluating the model's performance with a test dataset that consists of real-world images.

3.1 Hardware and Software

The entire process of creating, augmenting, and evaluating the synthetic dataset is done using a computer with an AMD Ryzen 5 3600 CPU, 32GB RAM, an NVIDIA RTX 2070 Super GPU, 1TB hard disk, and Windows 10 as the operating system.

Blender 3.4 is used to create the 3D models of the electronic components and render the images based on the models. The Blender API for Python is used to create the Python script to automate the process of rendering images and create the synthetic dataset.

The Imgaug Python library is used to create additional Python scripts to augment and expand the synthetic dataset.

Yolov5 was selected as the classification model as it is a state-of-the-art object detection model that can be configured for image classification and object segmentation, making it very flexible and adaptable.

3.2 Creating the Synthetic Data

The following sections cover the process of creating the synthetic dataset. Section 3.2.1 covers the process followed to create the 3D models of the electronic components; Section 3.2.2 covers the preparation of the different elements needed to make up a 3D scene and the configuration of the rendering options; and lastly Section 3.2.3 covers the logic of the Python script that automates the rendering process.

3.2.1 Creating the 3D Models of the Electronic Components

Using a combination of Blender's tools and layouts, 3D models of five electronic components were created. Different workflows were used for creating the geometry of each electronic component model (ECM), depending on the complexity and the dynamic features of each model. However, in most cases, the different parts of each EMC's geometry, e.g., the connector leads and bodies, were modeled as separate geometries, which were then combined into a single geometry using geometry nodes. This was done so that the geometries of the different parts could be manipulated independently from each other and assigned different materials or textures.

This is illustrated in Fig. 3-1, where, at the top-left, the different parts used to make up the resistor ECM are shown as separate geometries, each with their own PBR material; the resistor ECM to the right is the result of the geometry node map at the bottom of the figure that combines the parts into a single geometry.

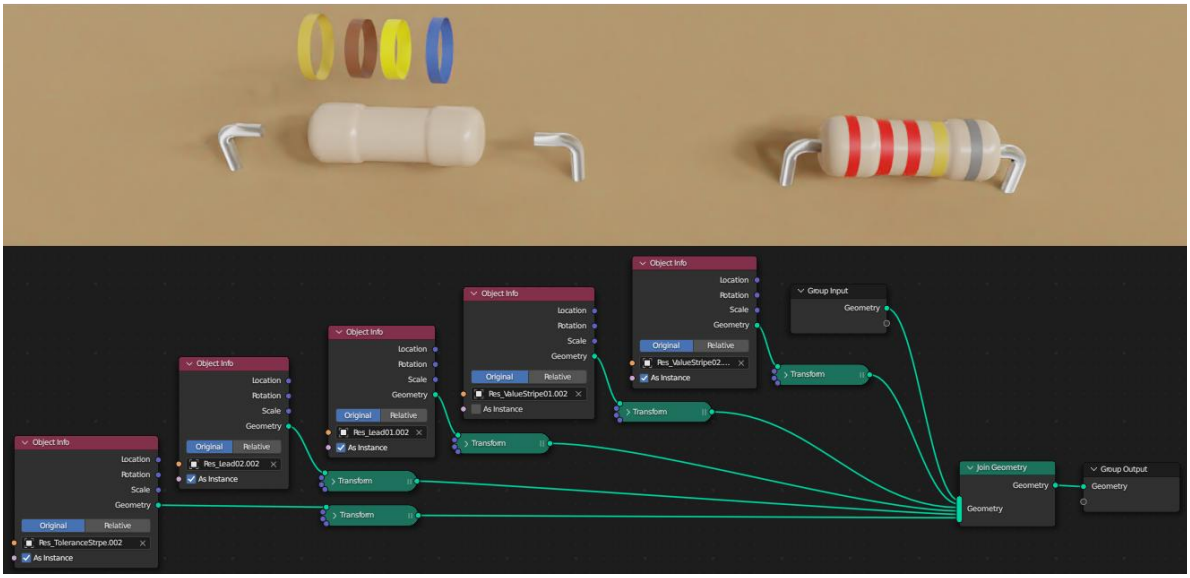


Fig. 3-1 Separate Parts Combined into a Single 3D Model Using Geometry Nodes

3.2.1.1 The Through-Hole Diode and Surface Mount Resistor Models

The ECMs of the through-hole diode (diode) and surface mount resistor (SMD resistor) were created in a similar fashion using Blender's modeling tools. First, their individual subparts, as illustrated in Fig. 3-2., were modeled. The part was then combined using geometry nodes to form the untextured models in Fig. 3-3. The shader node maps in Fig. 3-4 was then used to apply PRR materials to each part of the diode and SMD resistor. The Metallic Matte material was applied to metallic parts like the terminals, the Black Plastic Matte material to the bodies of the models, and the White Marking material was used to texture the marking that was applied to each model.

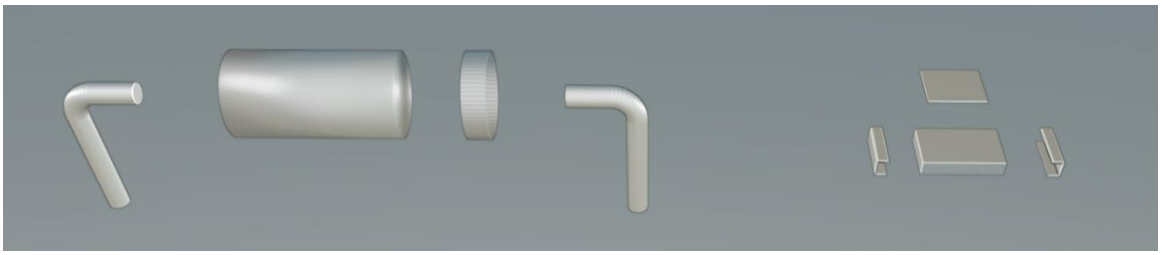


Fig. 3-2 The Separate Parts of the Diode Model (Left) and the SMD Resistor (Right)

Fig. 3-3 The Untextured Models of the Diode (Left) and the SMD Resistor (Right)

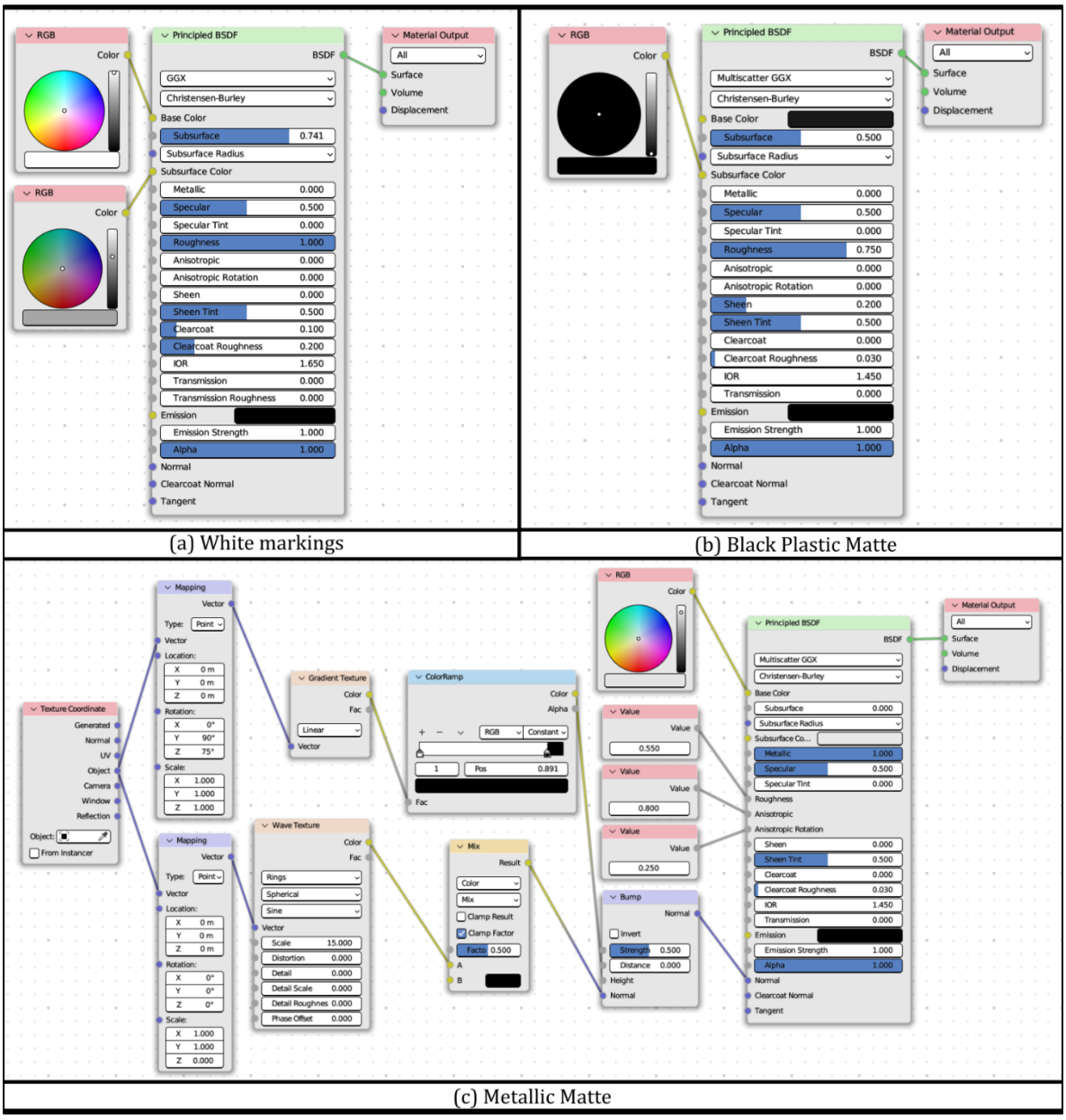


Fig. 3-4 Shader Node Maps of the White Marking, Black Plastic Matte and Metallic Matte PBR Materials

Some dynamic features were also added to each model. For the SMD resistor model, the geometry node map in Fig. 3-5 was used to generate a randomized marking. The node map generates a random number using the model's location as a seed. The number is then converted to a mesh, and White Markings PBR material is applied to it before it is joined with the geometry of the rest of the model. The geometry node map also makes use of a second random number that is used to determine if the markings should be visible or not.

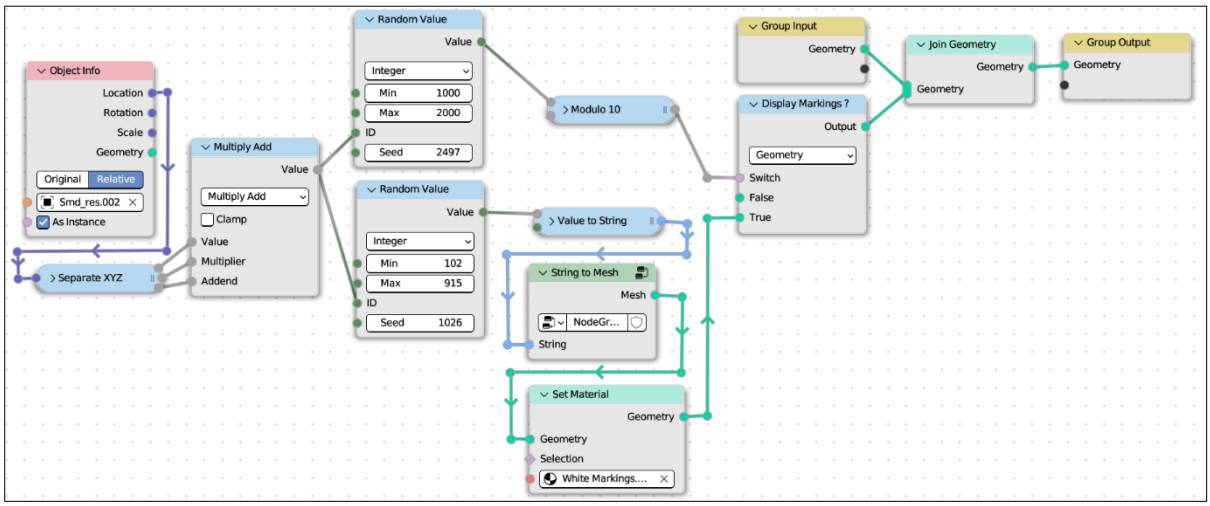


Fig. 3-5 Geometry Node Map to Create the Dynamic Marking of the SMD Resistor Model

For the diode ECM, the dynamic features were added using a modified Black Plastic Matte shader node map in Fig. 3-6. This was done because the geometry nodes in Blender 3.4 cannot curve a string along the mesh of another object. The shader node loads a random image from a set of pre-created diode marking images and inverts its colors before using it as the base color map for the Principle BSDF node. As a result, the diode body will be black with randomized white markings.

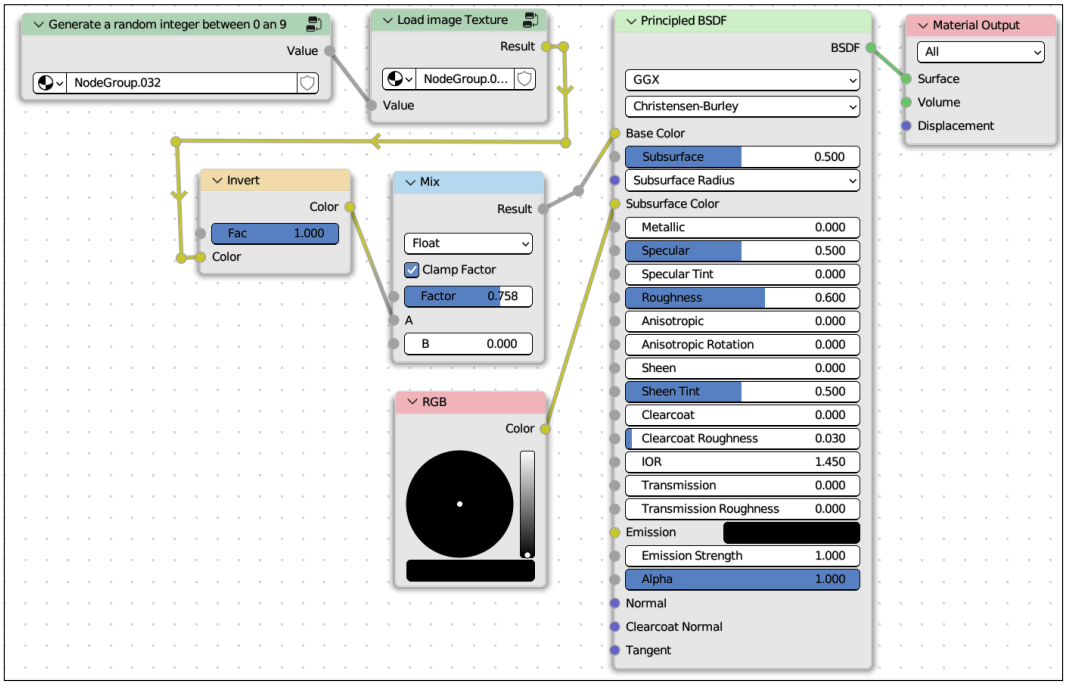


Fig. 3-6 Modified Black Plastic Matte Texture that Generates the Random Diode Markings

3.2.1.2 The Through-Hole Resistor and Through-Hole Electrolytic Capacitor Models

The through-hole resistor (resistor) and through-hole electrolytic capacitor (capacitor) ECMs were created in a similar manner; however, they both have dynamic texture and mesh elements that are randomized. For these ECMs the mesh elements in Fig. 3-7 were created using the modeling tools.

The geometries for these ECMs are dynamically created using geometry nodes to select which mesh elements make up the final geometry of the respective model. For the capacitor model the end-disk designs, different polarity indicator styles, and dynamic voltage and capacitance rating markings are randomized and dynamically selected based on the location model's body in the 3D space. In a similar manner, the resistor model's number of value stripes are randomized and spaced.

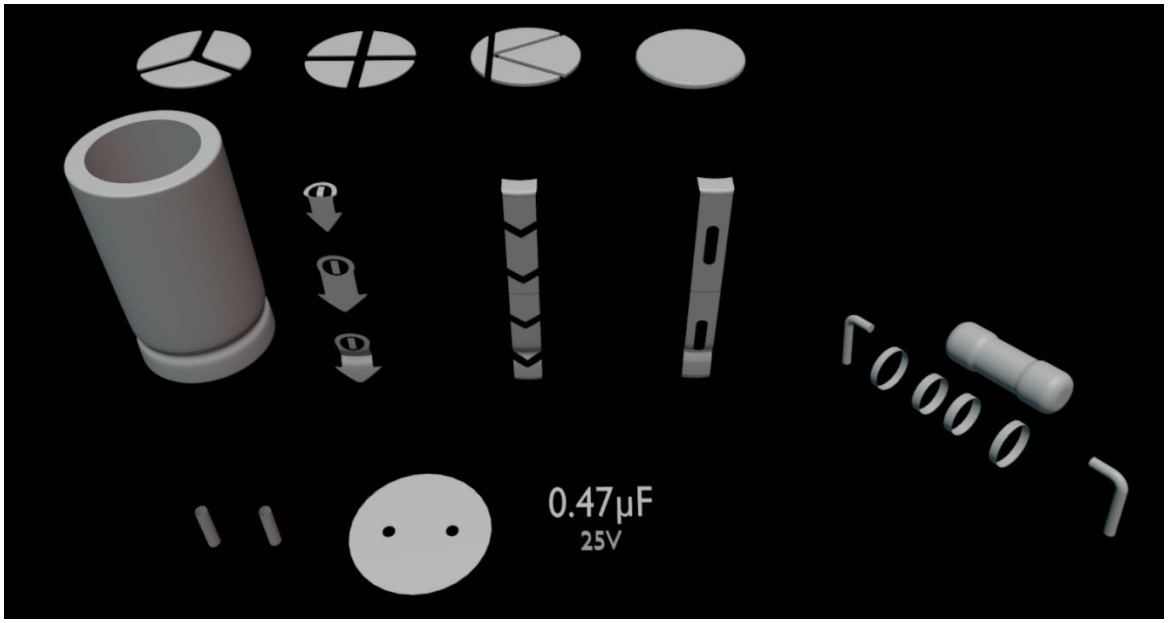


Fig. 3-7 Mesh Elements of the Capacitor and Resistor Models

The White Markings and Metallic Matte PBR material were used for basic texturing. The White Markings material was assigned to the capacitor's polarity indicators and voltage and capacitance rating markings, while the Metallic Matte material was assigned to the capacitor's end-disc designs and the connector terminals of both models.

The dynamic textures for the capacitor body, resistor body, and resistor value stripes all used similar shader maps, illustrated in Fig. 3-8, with only the ColorRamp node being customized according to the range of colors that are required for the respective mesh element. The shader node map changes the color of the mesh element that it is assigned to, based on the element's location in 3D space and the range colors assigned to the ColorRamp node.

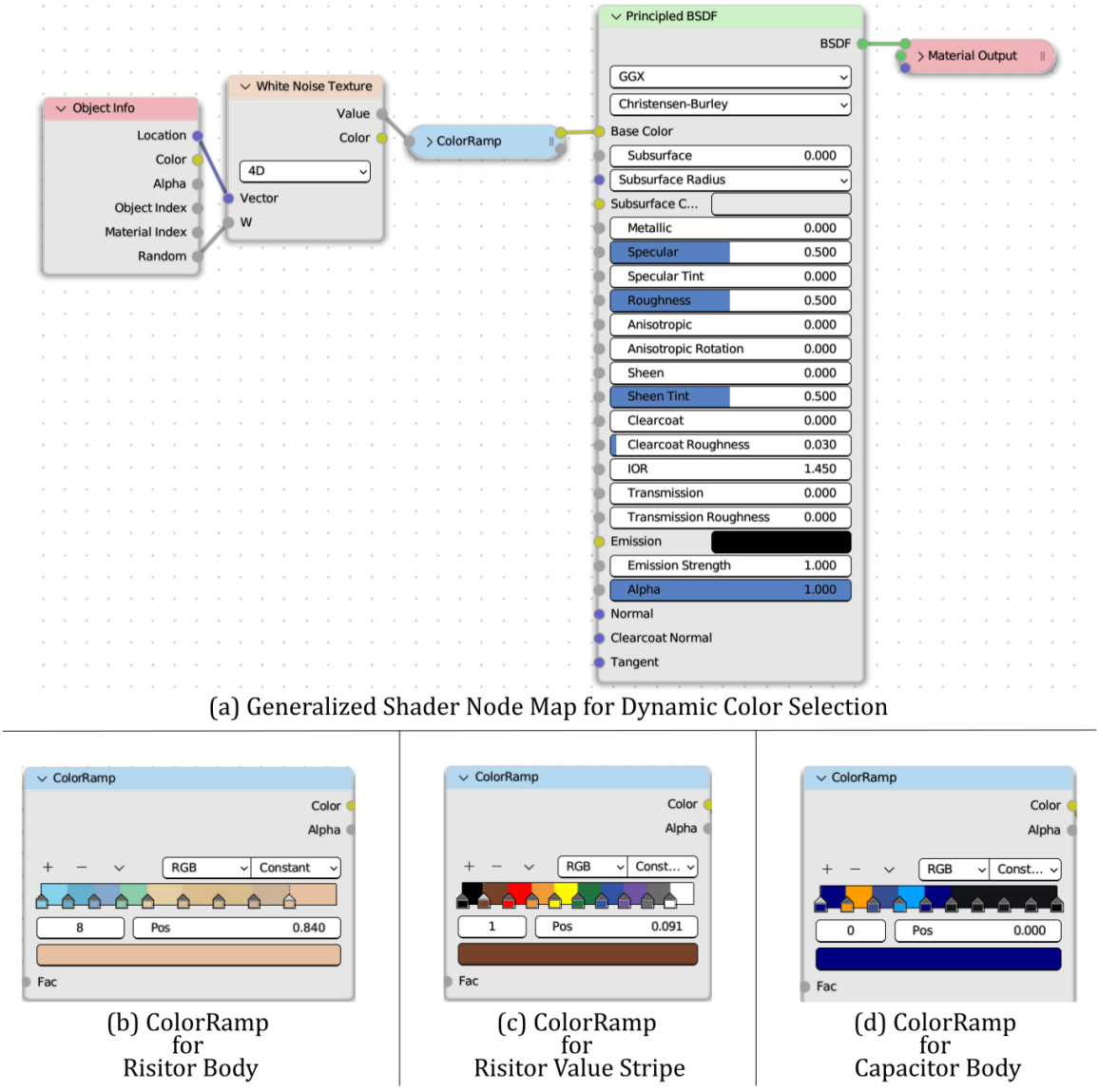


Fig. 3-8 Geometry Node Map and ColorRamp Nodes used to Dynamically Randomize the Color of Mesh Elements of the Resistor and Capacitor Models

3.2.1.3 The Surface Mount Ceramic Capacitor Model

The model of the surface mount ceramic capacitor (SMD capacitor) was modeled entirely with the geometry nodes. The geometry map uses the Set Material node to assign the Metallic Matte PBR material to the connection terminal meshes and SMD Cap Body PBR material, shown in Fig. 3-9, to the body mesh of the model. The geometry map shown in Fig. 3-10, manipulates the dimensions and locations of three cube meshes that form the combined geometry of the SMD capacitor. This allows an SMD capacitor model's size to be randomized according to different standard packages while creating a dimensionally accurate model. Fig. 3-11 illustrates some examples of the variation of each ECM.

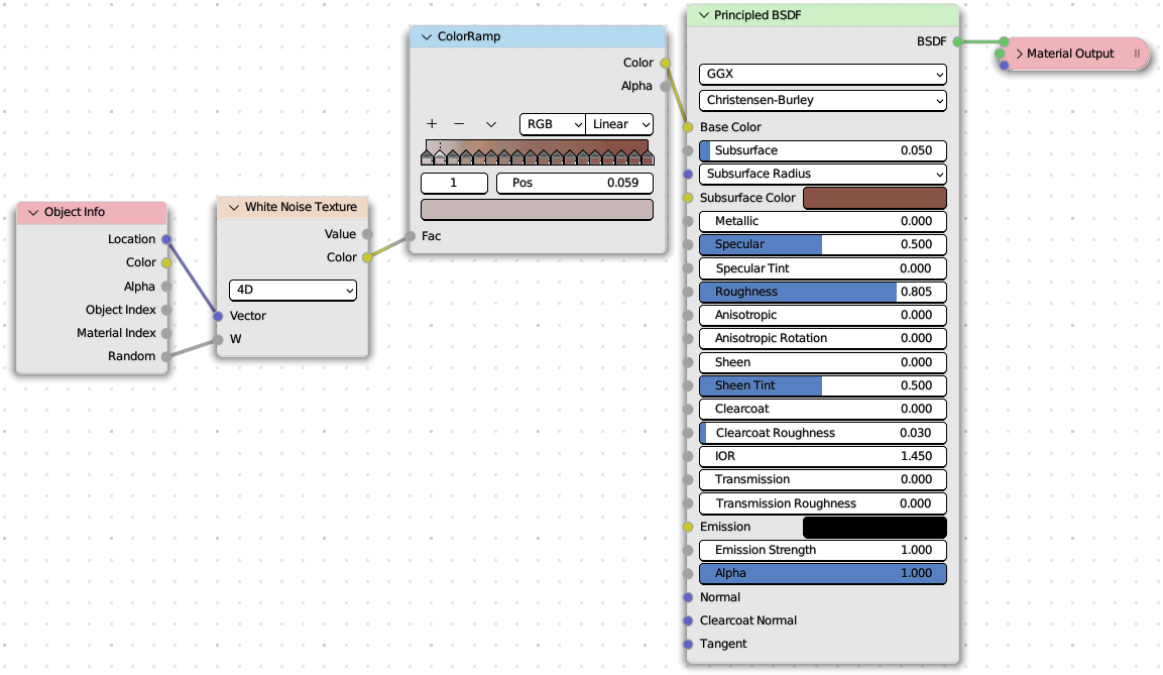


Fig. 3-9 SMD Cap Body Shader Node Map used to Dynamically Randomize the Color of the SMD Resistor Model

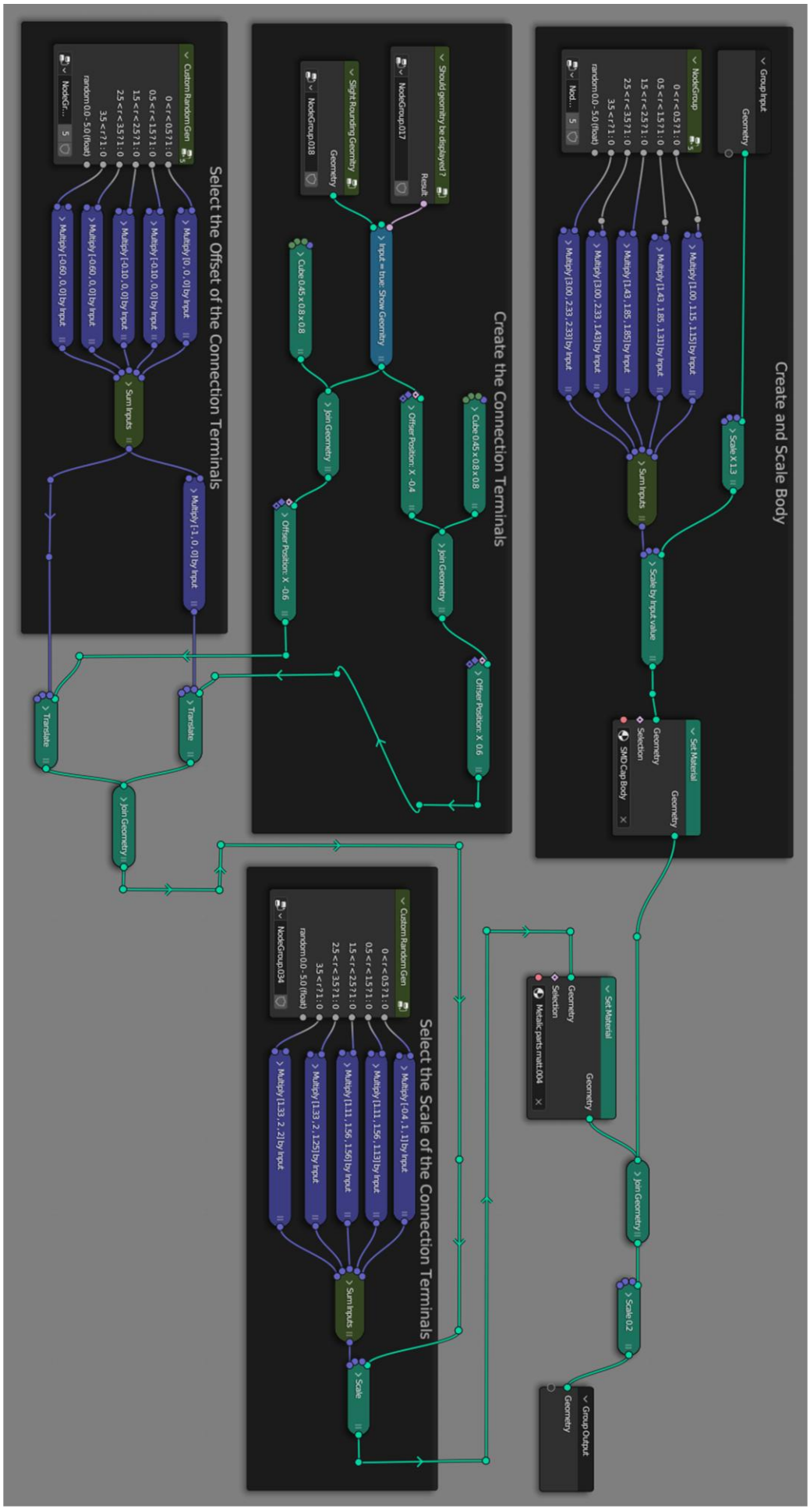


Fig. 3-10 Geometry Node Map Used to Dynamically Create the SMD Capacitor Model

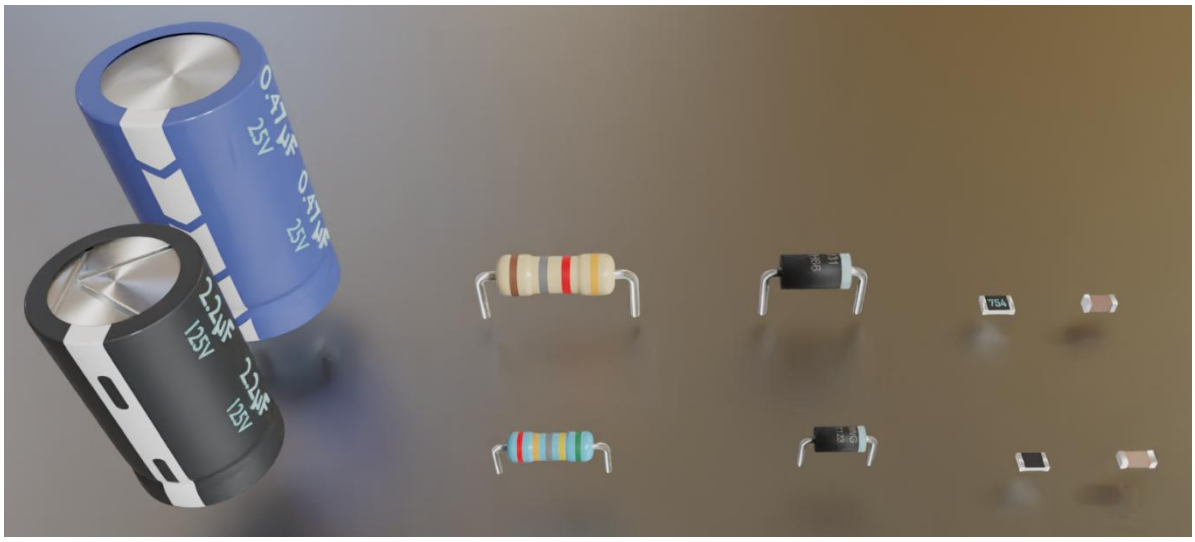


Fig. 3-11 Variations of the Rendered Electronic Component Models

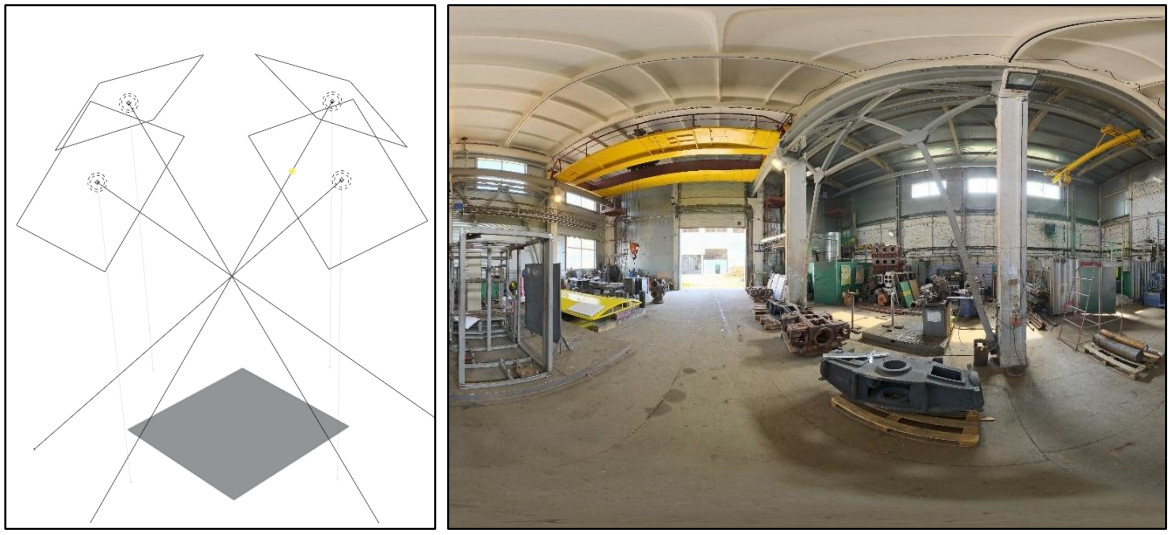
3.2.2 Setup of the Scene Elements and Rendering Configuration

Once all the electronic component models were created, some of the scene's basic elements were configured. A scene is made up of the objects that are to be rendered, light sources, cameras, and backgrounds. This section will focus on the configuration of the scene's lighting and camera elements, as well as the preparation of the backgrounds that will be used for the scenes. The ECM and background that will be used when creating and rendering the scene will be selected and moved into place by the Python script that is discussed in Section 3.2.3.

3.2.2.1 Configuring the Scene Elements

Two different lighting configurations were set up. The first lighting configuration uses four area light panes to produce soft diffused light similar to studio lights used in photography. Fig. 3-12 (a) shows how the lights are positioned relative to the scene. This configuration minimizes shadows in the scene while still providing sufficient illumination.

The second lighting configuration disables the light panes and rather makes use of a High Dynamic Range Image (HDRI) to simulate indirect natural lighting conditions. Here the 'Machine Shop 02' HDRI [74], shown in Fig. 3-12 (b) was used as it provides lighting similar to what can be expected in an industrial building.



(a) (b)

Fig. 3-12 Lighting Configurations. (a) 12 Pane Lights at a 45˚ Angle Relative to the Edges of The Scene. (b) Preview of the HDRI used for the Second Lighting Configuration

Three cameras were configured to allow images to be rendered from different perspectives, as visualized in Fig. 3-13. The primary camera is positioned directly below the scene and looks down on it, providing a top-down perspective of the scene. The two secondary cameras were offset slightly on the X and Y axes and tilted at a 34˚ angle relative to the scene.

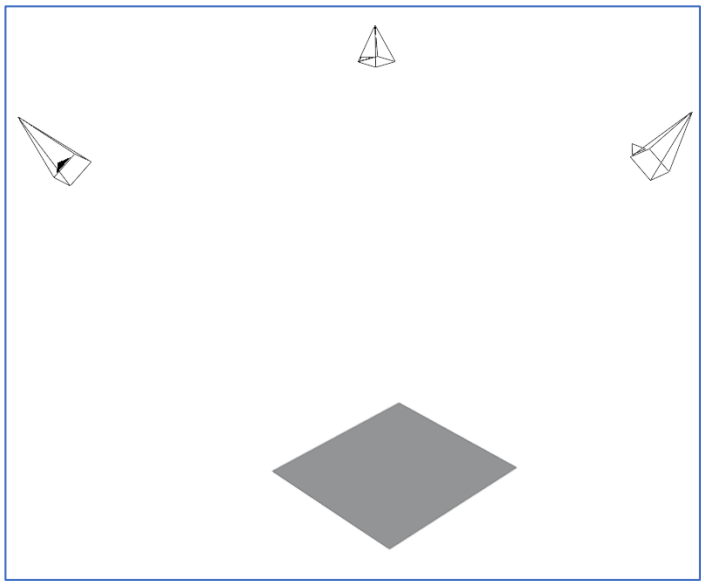


Fig. 3-13 The Camera Locations Relative to the Scene

The four different scene backgrounds in Fig. 3-14 were created using mesh planes. Each mesh plane was assigned a different texture. When the scene is created by the Python script one of the mesh planes will be selected at random to act as the background of the scene.

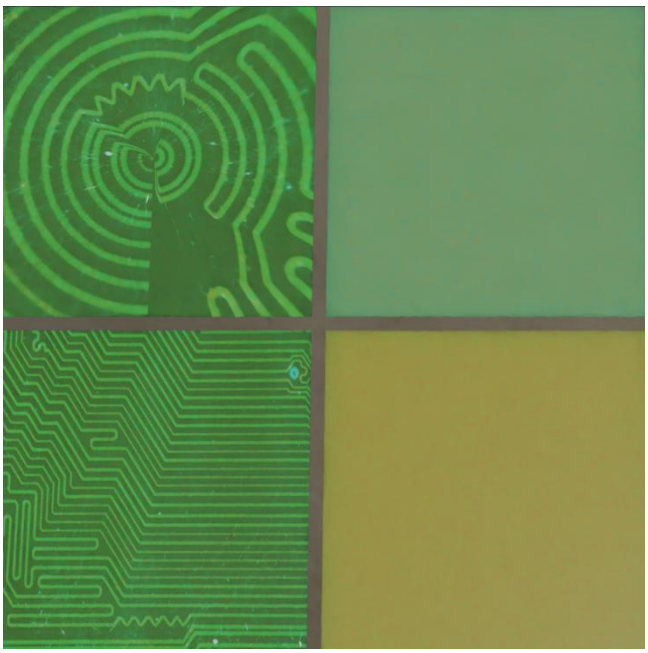


Fig. 3-14 Scene Backgrounds

3.2.2.2 Configuring the Rendering options

Blender was configured to use both the CPU and GPU to render the images. The output file setting was set to use a 224x224 resolution with the PNG image format, with an 8-bit color depth per channel. The rendering setting was configured as follows:

- Render engine: Cycles
- Device: GPU Compute
- Denoise: Selected
- Denoiser: OptiX [66]
- Passes: Albedo and Normal

3.2.3 Automating Rendering Using Python Scripts

A Python script, using Blender's Python API, was created to manage the generation of a randomized scene from the different scene elements, the rendering of an image of the scene, and saving the image to the specified location. Before the script is run, the following needs to be selected and configured:

- The lighting configuration to be used.
- The camera to be used for the rendering process.
- The number of images that need to be rendered per ECM.
- The output folder where the rendered images will be saved.

The flowchart in Fig. 3-15 shows the logic of the script. Starting at the first model in the list of ECMs, a scene is created. This involves randomly selecting one of the backgrounds and moving it into the scene space; moving the ECM to the scene space; randomizing the location, scale, and rotation of the ECM; and if the primary camera is used to render the scene, its location is adjusted depending on ECM's scale.

The scene is then rendered, and the resulting image is saved in the pacified output folder. After the image is saved, the scene is cleared, and then the process is repeated *n* times for each of the ECMs; here *n* is the number of images that need to be created per ECM.

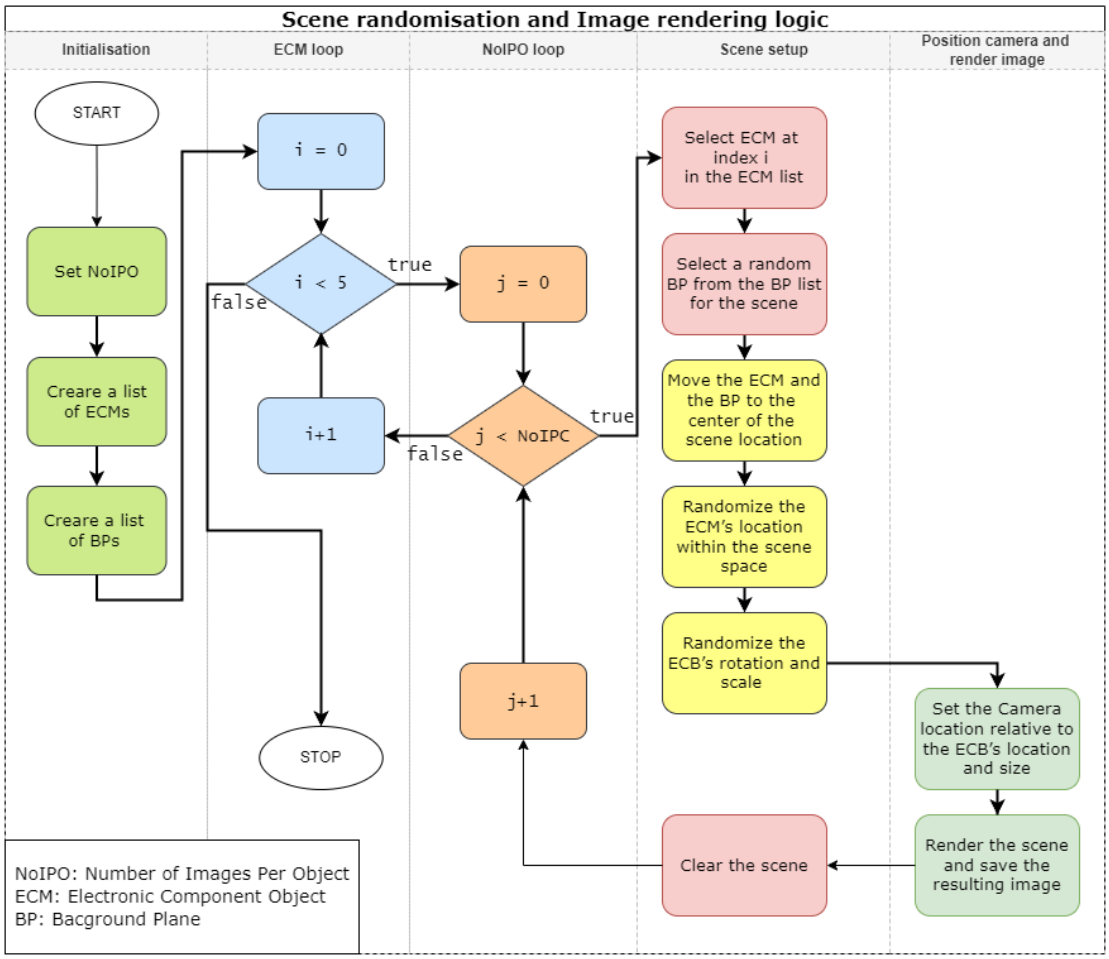


Fig. 3-15 The Logic of the Synthetic Data Generation Script

3.3 The Final Synthetic Datasets

Two synthetic datasets were created using the Python script described in Section 3.2.3. The first dataset consists of only rendered synthetic images, referred to as the *synthetic dataset*. The second, referred to as the *augmented dataset*, extends the synthetic dataset with additional rendered synthetic images and augmented images.

3.3.1 Rendering the Synthetic Image Training Data

For the synthetic dataset, the Python script was executed several times with the following configurations:

- Run 1:
    - Lighting: HDRI
    - Camera: Primary camera
    - Number of images per ECM: 900
- Run 2:
    - Lighting: Area light panes
    - Camera: Primary camera
    - Number of images per ECM: 900
- Run 3:
    - Lighting: Area light panes
    - Camera: Both secondary cameras
    - Number of images per ECM: 450 per camera (900 Total)
- Run 4:
    - Lighting: HDRI
    - Camera: Both secondary cameras
    - Number of images per ECM: 450 per camera (900 Total)

The resulting dataset consisted of 3,600 images per ECM class and a total of 18,000 synthetic images. The synthetic dataset was then randomized and divided into a validation dataset with 3,600 images and a training dataset with 14,400 images.

The augmented dataset uses a copy synthetic dataset as its base, expanding it by running the Python script with the following configurations to render the additional synthetic images:

- Run 1:
    - Lighting: HDRI
    - Camera: Primary camera
    - Number of images per ECM: 1000
- Run 2:
    - Lighting: Area light panes
    - Camera: Primary camera
    - Number of images per ECM: 1000
- Run 3:
    - Lighting: Area light panes
    - Camera: Both secondary cameras
    - Number of images per ECM: 136 per camera (272 Total)

The resulting dataset, referred to as the *base augmented dataset*, contains 5,872 images per ECM class and a total of 29,360 images.

3.3.2 Augmenting the Second Synthetic Dataset

The base augmented dataset was expanded and enhanced using the Imgaug library for Python [75] to apply color modification and affine image transformations image augmentation techniques [76].

Data augmentation was done in two phases using Python scripts created with the Imgaug library (See Appendix). In the first phase, the Python script randomly selects a combination of one or more transformation augmentations for each image in the synthetic dataset. It then applies the augmentations to the images before saving them to a specified output folder. The transformation augmentations include:

- Rotate the image by 90, 180, or 270 degrees
- Vertical Flip
- Horizontal Flip

These augmented images were then added to the base synthetic dataset to form the augmented dataset, consisting of 11,744 images per ECM and 58,720 images in total. The augmented dataset was then randomized and divided into a base-training dataset and a validation dataset, with an approximate 19 to 81% ratio. The base training dataset consists of 44,770 images, and the validation dataset consists of 13,950 images.

The second phase of augmentation was applied only to a randomized subset of 18,490 images from the base-training dataset. The Python script for this phase follows the same logic as that of the first phase, but it applies a random combination of one or more of the following color modifications:

- Adding Gaussian noise
- Adjusting brightness
- Adjusting hue and saturation
- Grayscale conversion
- Blurring the image

These augmented images were then added to the base-training dataset to form the final training dataset of the augmented dataset, containing 12,652 images per ECM class and a total of 63,260 images. Fig. 3-16 shows some samples from the training dataset.

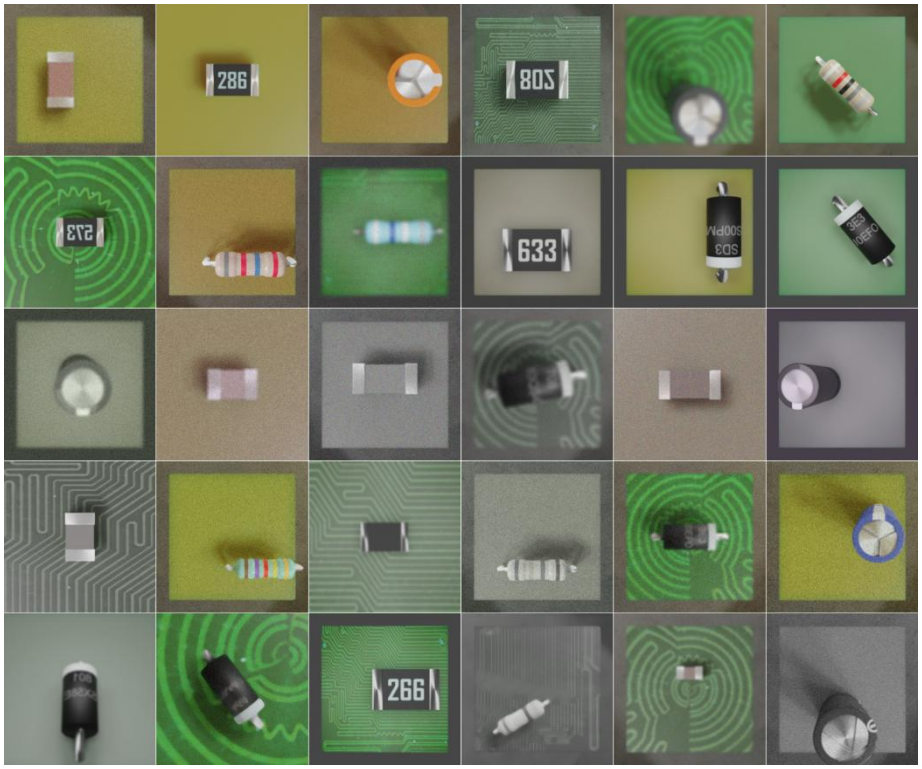


Fig. 3-16 Sample of Training Images from the Augmented Dataset

3.4 Training and Evaluating the YoloV5 Classifier

To evaluate the efficacy of the synthetic dataset and the augmented dataset, four instances of the YoloV5 classification model (YoloV5m-cls) [49], each with slightly different hyperparameter configurations, were trained using PyTorch [50].

Instance three was trained with real-time augmentations enabled. This allowed the YoloV5 model to make use of the preconfigured augmentation in real-time to augment the training images as it trained. The default augmentations as defined in dataloaders.py and augmentations.py files in the .../yolov5/utils directory were used.

Table 3-1 Hyperparameter Configuration for the Different Instances of the YoloV5 Classifier

	Training and Validation Dataset	Batch Size	Epochs	Initial Learning Rate	Learning Decay Rate	Dropout	Realtime Augmentation
Instance 1	Synthetic	100	144	0.001	5.00E-05	0.005	No
Instance 2	Augmented	172	368	0.001	5.00E-05	0.005	No
Instance 3	Augmented	172	368	0.001	5.00E-05	None	Yes

The training algorithm saves two sets of weights as it runs. The latest set of weights, referred to as the last weights, are saved at the end of every epoch. The best-performing weights save the set of weights with the highest accuracy and lowest valuation- and training-loss. Table 3-1 shows the configurations of the different YoloV5 instances.

The best weight set from each of the three YoloV5 instances was evaluated using a testing dataset that consists of 30 real-world images of each electronic component, totaling 150 images. Fig. 3-17 shows a sample subset of the testing dataset.

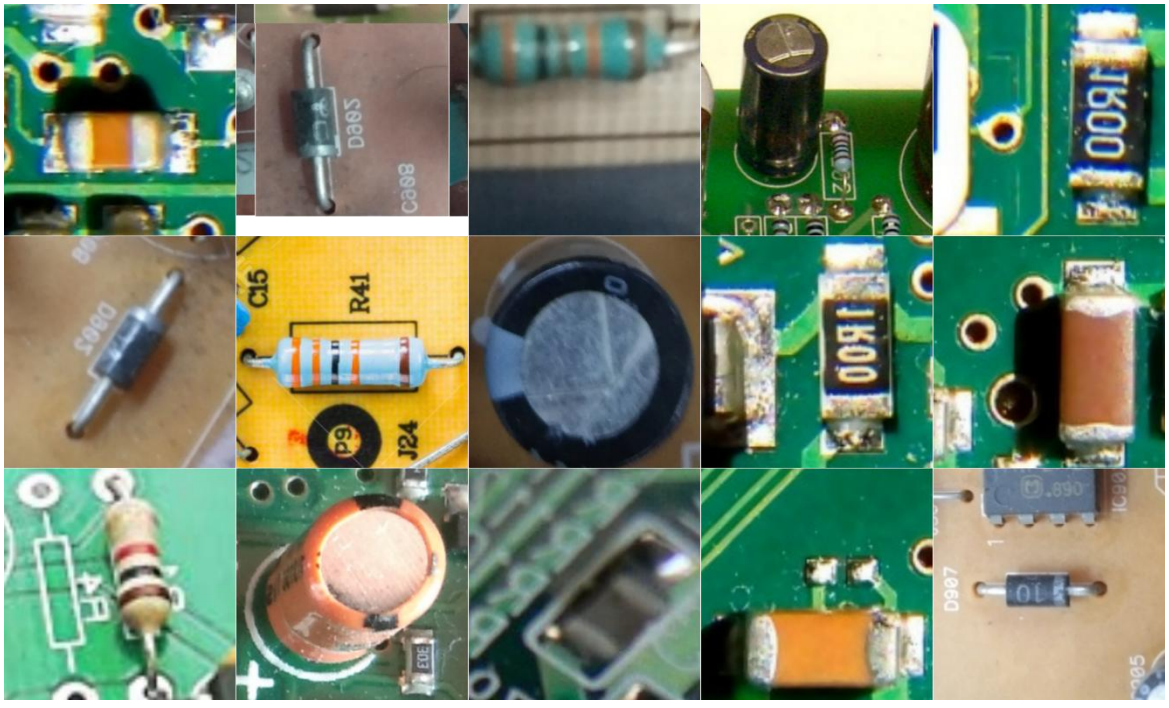


Fig. 3-17 Samples from the Test Dataset

The performance of the model was evaluated using a confusion matrix along with the Accuracy, Precision, Recall, and F1-Score metrics [77] [78] [79]. The confusion matrix is a special table that is used to visualize the classification performance of an algorithm. It compares predicted classes with actual or known classes, allowing one to clearly identify the True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN) predictions for each class. Using this information, the Accuracy, Precision, Recall and F1 Score can be derived.

The Accuracy metric measures the overall classification correctness for a given class and can be calculated as:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad \textbf{(3-1)}$$

The Precision metric measures how many of the positive classifications for a given class were correct and can be calculated as:

$$Precision = \frac{TP}{TP + FP} \quad \textbf{(3-2)}$$

The Recall metric is used to measure how many instances of a given class were correctly classified and can be calculated as:

$$Recall = \frac{TP}{TP + FN} \quad \textbf{(3-3)}$$

The F1 score takes both the Precision and Recall into account by calculating their harmonic mean. It is calculated as:

$$F1\ Score = 2 \times \frac{(Precision\ \times Recall)}{(Precision\ +\ Recall)} \quad \textbf{(3-4)}$$

3.5 Conclusion

In conclusion, a synthetic training dataset of electronic component was created using Blender and Python scripts. Blender was used to create the textured models and configure the elements needed to create 3d scenes. Python scripts were used to automate the processes the randomize, render, and augment the synthetic images. A YoloV5 classifier model was trained be on using the synthetic images and evaluated using real-world images. The results were evaluated with a confusion matrix along with the Accuracy, Precision, Recall, and F1-Score metrics

Chapter 4: Results

This chapter covers the results of the research. Section 4.1 covers the results from synthetic data rendering and augmentation process, and Section 4.2 covers the results from training and evaluating the YoloV5 Classifier to validate the synthetic training data.

4.1 Synthetic Data Rendering and Augmentation

The 18,000 images of the synthetic dataset took 46 hours 40 minutes to render. The additional 11,360 images for the augmented dataset took 43 hours 21 minutes to render. Phase 1 and Phase 2 augmentation took a total of 1 hour 38 minutes to complete.

4.2 Classifier Model Training and Testing

The first instance of the YoloV5m-cls classifier model took 2 hours 15 minutes to train, the second instance took 4 hours 2 minutes to train, and the third instance took 4 hours 37 minutes to train.

The best weights for the first and second instances were found after their respective last training epoch was completed; epoch 144 and 368 respectively. The third instance's best weight was found at epoch 260.

Confusion matrices were created from the respective classification results on the test dataset. Thereafter, the evaluation metrics for each instance were derived.

From the confusing matrix for Instance 1 in Fig. 4-1 (a), it can be seen that the model performed poorly. While Instance 1 had a high number of TP classifications and a low number of FN classifications for the resistor class, it also had a high number of FP classifications for the resistor class, confusing most of the real-world images of the other classes as a resistor. This shows that the model was not able to learn sufficient unique features or properly differentiate the classes.

From the confusion matrix for Instance 2 in Fig. 4-1 (b) it is clear that the larger and more diverse augmented dataset, larger batch size and increased epochs resulted in the model performing significantly better. While the TP classifications of the resistor and SMD capacitor classes are slightly lower, those of the capacitor, diode, and SMD resistor classes all improved. The number of FP and FN classifications also decreased. While the model's performance improved, it was still not able to properly classify the SMD classes, confusing them as either a capacitor, a resistor, or a diode.

The confusion matrix for Instance 3 in Fig. 4-1 (c) shows another improvement in performance; however, it is still far from ideal. Here it is clear that additional diversification of the training data due to the real-time augmentation allowed the model to better learn unique features to properly differentiate the classes. The TP classifications improved for all the classes except the SMD capacitor class.

The number of FP and FN classifications also decreased further. Unfortunately, there was no improvement in the classification of the SMD capacitor images, and most of the images of the SMD classes were confused as capacitors.

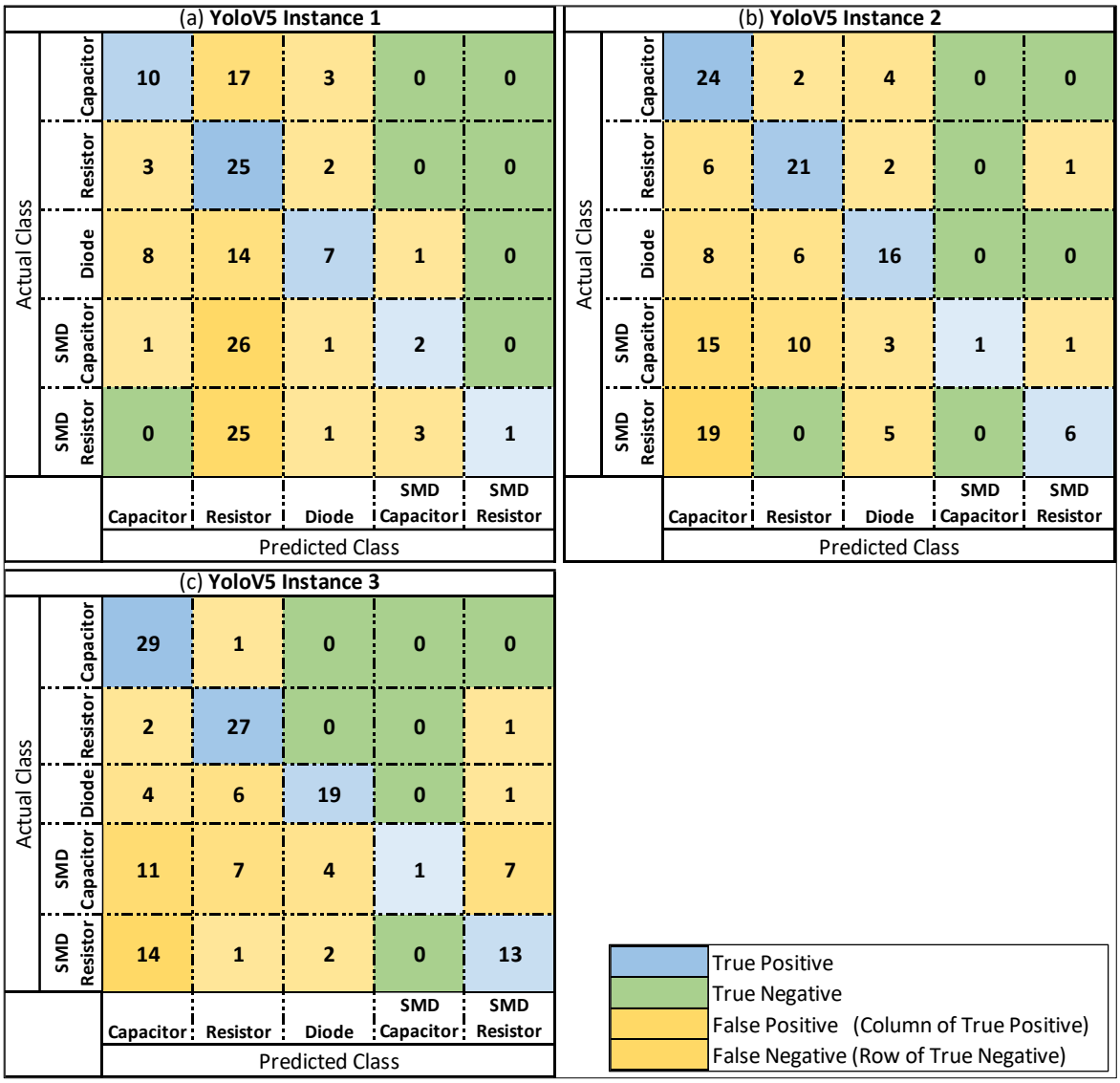


Fig. 4-1 Confusion Matrices for the Three Yolov5 Training Instances

Error! Reference source not found. shows a data table and a chart for each of the four derived metrics d iscussed in Section 3.4. In each case, the chart compares the performance of the three YoloV5 instances for each of the ECM classes. It is clear that the Accuracy metric on its own is not meaningful, e.g., the SMD capacitor class has a high accuracy across all three YoloV5 instances, yet only a few of the related real-world images were correctly classified. The Precision metric, being tied to the FP classification, changed as the FP distribution of the individual classes changed. The F1 score and Recall metrics show a gradual improvement in the performance of most of the classes, with a clear pattern of a decrease in FN classifications as the diversity and number of training images increased due to augmentation.

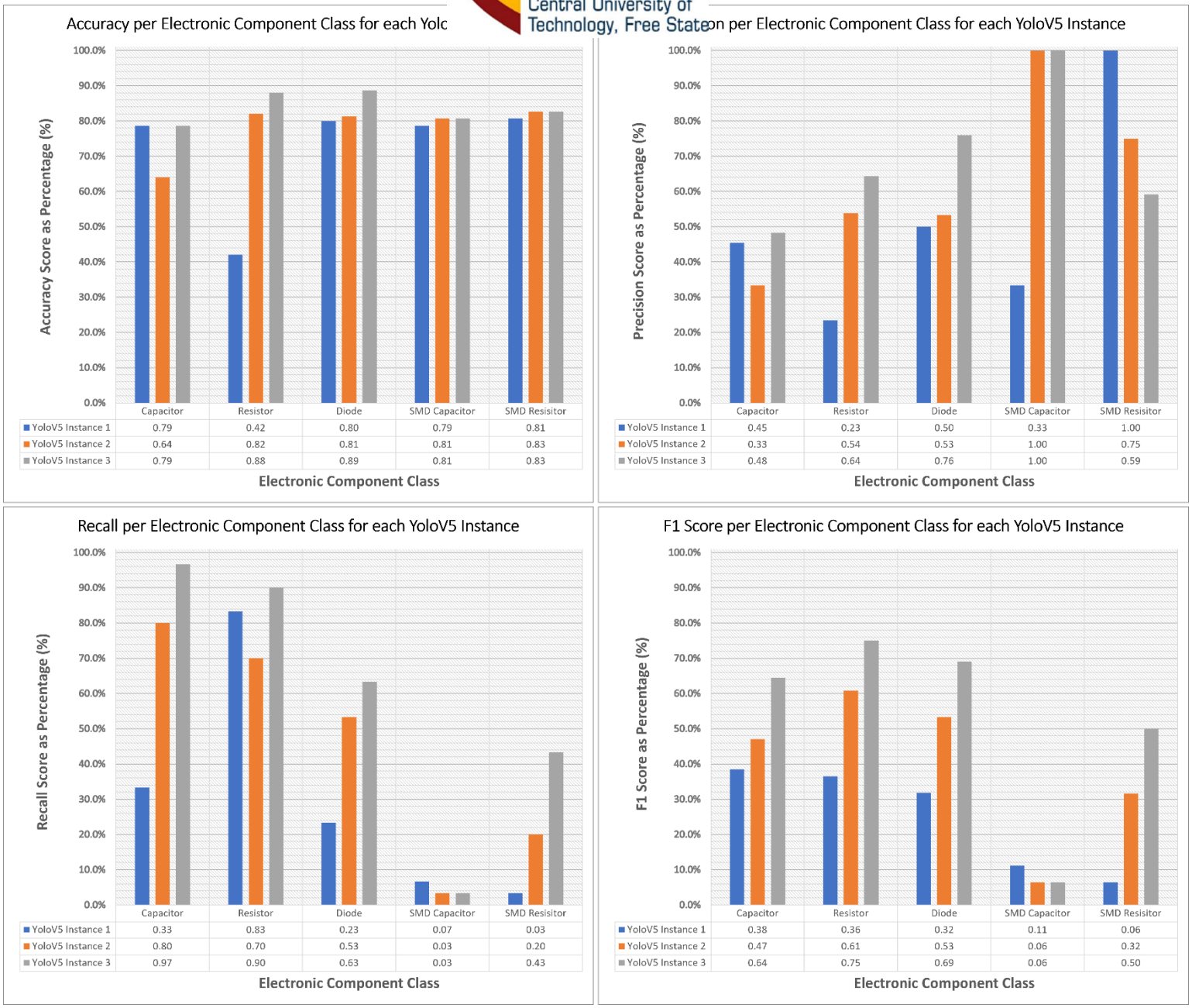


Fig. 4-2 Classification Performance Metrics and Their Comparison per Electronic Component Class for the Three YoloV5 instances

Fine-tuning the hyperparameters might lead to slightly better performance, but as the YoloV5 model is well established and proven across multiple applications, a larger gain is most probable by revisiting and adjusting the synthetic data, especially in the case of the SMD classes.

With the low FP classifications of the SMD classes, it is clear that the model was able to clearly identify which images did not contain SMD capacitors or SMD resistors, yet it struggled to correctly classify these two classes. This indicates that the SMD classes' synthetic images do not represent the real-world version accurately enough, and as a result, it does not allow the classifier to discover enough unique feature combinations during training to differentiate it from the other ECM classes.

The second cause might be a mismatch between the 3D models and the actual real-world electronic components in terms of geometry and visual appearance, especially in the case of the SMD capacitor. As the SMD capacitor has fewer visual features to uniquely distinguish it from other electronic components, it might require a significantly more accurate 3D model and texturing for synthetic images to more closely resemble the real-world counterpart.

Examples of what can be done to improve the quality of the synthetic data include:

- Adding solder pads to the scene where SMD capacitors and SMD resistors are located.
- Rounding the corners and edges of the SMD capacitor slightly to more closely match the real-world components.
- Fine-tuning the color and texture of the 3D models of the ECMs.
- Adding PCB marking to the backgrounds, where applicable, to some of the synthetic images.

Chapter 5: Conclusion and Future Work

The purpose is to explore the use of synthetic training data for the classification of electronic components. This is achieved by demonstrating a method for generating synthetic image datasets using Blender and then augmenting those datasets, allowing for the rapid creation of training data for a YoloV5 classifier.

It is clear from the results that the synthetic images for the visual feature-rich classes like the capacitor and resistor classes contain enough unique features from their real-world counterparts to allow the YoloV5 model to learn to classify them accurately. The model struggled to discover sufficient unique features for classes that contain less distinct or unique visual features and thus had difficulty correctly classifying them and confusing them as other classes. This is especially evident with SMD capacitors, where only one of its real-world images could be correctly classified.

This indicates that it is viable to create and train a DL classifier on synthetic data, but there are limitations. It is challenging for a DL classifier to successfully train on synthetic data from classes with fewer unique visual features. To overcome this, more careful feature engineering is required when creating the synthetic data for classes with fewer visual features. This will ensure that they match their real-world equivalent as closely as possible, allowing the DL classifier to more easily learn the unique features of the classes in question, generalize better, and perform better. However, this necessitates a higher level of 3D modeling skill and possibly more time to construct and texture the object based on the classes in question, as well as to create a more realistic scene before rendering the synthetic images. In the context of this study, the 3D models that the synthetic data for the diodes, SMD resistors, and especially the SMD capacitors are based on need to be revisited and adjusted to better match the real-world equivalent.

Several areas of future research have been identified, including:

- Evaluating the effect of using a mix of real-world and synthetic training data affects the performance of the classifier once sufficient real-world data is available.

- Enhancing the synthetic data generation process to generate synthetic data for a broader range of electronic components, including bounding-box and segmentation annotations. Then evaluating the effectiveness of using synthetic data for train object detection and segmentation models aimed at identifying various electronic components on circuit boards.
- Evaluating the effectiveness of using synthetic data to train several different classification and object detection models, including the latest YOLO models.
- Exploring different techniques and software to create more realistic renders of SMD components.

Considering that there is a tremendous advancement in AI in general and that state-of-the-art AI models need ever larger training datasets, the use of synthetic training data for visual classifiers is a viable solution.

References

[1] P. Kamavisdar, S. Saluja, and S. Agrawal, “A Survey on Image Classification Approaches and Techniques,” International Journal of Advanced Research in Computer and Communication Engineering, vol. 2, no. 1, pp. 1005–1009, Jan. 2013.

[2] T.-H. Sun, C.-C. Tseng, and M.-S. Chen, “Electric Contacts Inspection Using Machine Vision,” Image and Vision Computing, vol. 28, no. 6, pp. 890–901, Jun. 2010, doi: https://doi.org/10.1016/j.imavis.2009.11.006.

[3] H. Kagermann, J. Helbig, A. Hellinger, and W. Wahlster, “Recommendations for Implementing the Strategic Initiative Industrie 4.0: Securing the Future of German Manufacturing Industry; Final Report of the Industrie 4.0 Working Group,” Forschungsunion, 2013.

[4] D. Kaeli, P. Mistry, D. Schaa, and D. P. Zhang, “Chapter 9 - Case Study: Image Clustering,” in Heterogeneous Computing with OpenCL 2.0, Morgan Kaufmann, 2015, pp. 213–228. doi: https://doi.org/10.1016/b978-0-12-801414-1.00009-0.

[5] Z.-Q. Zhao, P. Zheng, S.-T. Xu, and X. Wu, “Object Detection with Deep Learning: A Review,” IEEE Transactions on Neural Networks and Learning Systems, vol. 30, no. 11, pp. 3212–3232, Nov. 2019, doi: https://doi.org/10.1109/tnnls.2018.2876865.

[6] A. Kozakova, Selective Focus Photography of Orange Cat Beside Plant During Daytime. Unsplash, 2020. Accessed: Jul. 15, 2020. [Online]. Available: https://unsplash.com/photos/selective-focus-photography-of-orange-cat-beside-plant-during-daytime-o9bBI24BM0Q

[7] D. Callister, Tan Dog Near Cat on Bookshelf. Unsplash, 2019. Accessed: Jun. 17, 2020. [Online]. Available: https://unsplash.com/photos/tan-dog-near-cat-on-bookshelf-TqLI2LiJxLM

[8] Y. LeCun, Y. Bengio, and G. Hinton, “Deep Learning,” Nature, vol. 521, no. 7553, pp. 436–444, May 2015, doi: https://doi.org/10.1038/nature14539.

[9] J. Wang, Y. Ma, L. Zhang, R. X. Gao, and D. Wu, “Deep Learning for Smart Manufacturing: Methods and Applications,” Journal of Manufacturing Systems, vol. 48, pp. 144–156, Jul. 2018, doi: https://doi.org/10.1016/j.jmsy.2018.01.003.

[10] A. Voulodimos, N. Doulamis, A. Doulamis, and E. Protopapadakis, “Deep Learning for Computer Vision: A Brief Review,” Computational Intelligence and Neuroscience, vol. 2018, pp. 1–13, Feb. 2018, doi: https://doi.org/10.1155/2018/7068349.

[11] J. Schmidhuber, “Deep Learning in Neural Networks: An Overview,” Neural Networks, vol. 61, no. 61, pp. 85–117, Jan. 2015, doi: https://doi.org/10.1016/j.neunet.2014.09.003.

[12] M. D. Zeiler and R. Fergus, “Visualizing and Understanding Convolutional Networks,” Computer Vision – ECCV 2014, vol. 8689, pp. 818–833, 2014, doi: https://doi.org/10.1007/978-3-319-10590-1_53.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” Communications of the ACM, vol. 60, no. 6, pp. 84–90, May 2012, doi: https://doi.org/10.1145/3065386.

[14] X. Xie, "A Review of Recent Advances in Surface Defect Detection Using Texture Analysis Techniques," ELCVIA Electronic Letters on Computer Vision and Image Analysis, vol. 7, no. 3, pp. 1–22, Jun. 2008, doi: https://doi.org/10.5565/rev/elcvia.268.

[15] Gwang Myong Go, S.-J. Bu, and Sung Bae Cho, "A Deep Learning-Based Surface Defect Inspection System for Smartphone Glass," Intelligent Data Engineering and Automated Learning – IDEAL 2019, pp. 375–385, Jan. 2019, doi: https://doi.org/10.1007/978-3-030-33607-3_41.

[16] B. Juba and H. S. Le, "Precision-Recall Versus Accuracy and the Role of Large Data Sets," Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, no. 1, pp. 4039–4048, Jul. 2019, doi: https://doi.org/10.1609/aaai.v33i01.33014039.

[17] C. Shorten and T. M. Khoshgoftaar, "A Survey on Image Data Augmentation for Deep Learning," Journal of Big Data, vol. 6, no. 1, pp. 1–48, Jul. 2019, doi: https://doi.org/10.1186/s40537-019-0197-0.

[18] M. Hussain, J. J. Bird, and D. R. Faria, "A Study on CNN Transfer Learning for Image Classification," Advances in Computational Intelligence Systems, vol. 840, pp. 191–202, Aug. 2018, doi: https://doi.org/10.1007/978-3-319-97982-3_16.

[19] M. A. Kamarul Bahrin, M. F. Othman, N. H. Nor Azli, and M. F. Talib, "Industry 4.0: A Review on Industrial Automation and Robotic," Jurnal Teknologi, vol. 78, no. 6–13, pp. 137–143, Jun. 2016, doi: https://doi.org/10.11113/jt.v78.9285.

[20] V. Roblek, M. Meško, and A. Krapež, "A Complex View of Industry 4.0," SAGE Open, vol. 6, no. 2, p. 215824401665398, Apr. 2016, doi: https://doi.org/10.1177/2158244016653987.

[21] A. Gilchrist, Industry 4.0: The Industrial Internet of Things. Berkeley, CA: Apress, 2016. doi: https://doi.org/10.1007/978-1-4842-2047-4.

[22] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A Survey," Computer Networks, vol. 54, no. 15, pp. 2787–2805, Oct. 2010, doi: https://doi.org/10.1016/j.comnet.2010.05.010.

[23] J. Z. Reis and R. F. Gonçalves, "The Role of Internet of Services (IoS) on Industry 4.0 Through the Service Oriented Architecture (SOA)," Advances in Production Management Systems. Smart Manufacturing for Industry 4.0, pp. 20–26, 2018, doi: https://doi.org/10.1007/978-3-319-99707-0_3.

[24] M. Hermann, T. Pentek, and B. Otto, "Design Principles for Industrie 4.0 Scenarios," 2016 49th Hawaii International Conference on System Sciences (HICSS), Jan. 2016, doi: https://doi.org/10.1109/hicss.2016.488.

[25] I. H. Sarker, "Deep Learning: A Comprehensive Overview on Techniques, Taxonomy, Applications and Research Directions," SN Computer Science, vol. 2, no. 6, Aug. 2021, doi: https://doi.org/10.1007/s42979-021-00815-1.

[26] A. Jamwal, R. Agrawal, and M. Sharma, "Deep Learning for Manufacturing Sustainability: Models, Applications in Industry 4.0 and Implications," International Journal of Information Management Data Insights, vol. 2, no. 2, p. 100107, Nov. 2022, doi: https://doi.org/10.1016/j.jjimei.2022.100107.

[27] M. Z. Alom et al., "The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches," arXiv, Mar. 2018, doi: https://doi.org/10.48550/arxiv.1803.01164.

[28] J. Hernavs, M. Ficko, L. Berus, R. Rudolf, and S. Klančnik, "Deep Learning in Industry 4.0 – Brief," Journal of Production Engineering, vol. 21, no. 2, pp. 1–5, Dec. 2018, doi: https://doi.org/10.24867/jpe-2018-02-001.

[29] T. Amaratunga, Deep Learning on Windows, 1st ed. Berkeley, CA: Apress, 2020. doi: https://doi.org/10.1007/978-1-4842-6431-7.

[30] J.-P. Briot, "From Artificial Neural Networks to Deep Learning for Music Generation: History, Concepts and Trends," Neural Computing and Applications, vol. 33, no. 1, pp. 39–65, Oct. 2020, doi: https://doi.org/10.1007/s00521-020-05399-0.

[31] S. S. A. Zaidi, M. S. Ansari, A. Aslam, N. Kanwal, M. Asghar, and B. Lee, "A Survey of Modern Deep Learning Based Object Detection Models," Digital Signal Processing, vol. 126, p. 103514, Jun. 2022, doi: https://doi.org/10.1016/j.dsp.2022.103514.

[32] S. I. Nikolenko, Synthetic Data for Deep Learning, 1st ed. Springer International Publishing, 2021. doi: https://doi.org/10.1007/978-3-030-75178-4.

[33] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for Activation Functions," arXiv, Oct. 2017, doi: https://doi.org/10.48550/arXiv.1710.05941.

[34] H. Wang and B. Raj, "On the Origin of Deep Learning," arXiv, Mar. 2017, doi: https://doi.org/10.48550/arXiv.1702.07800.

[35] K. Simonyan and A. Zisserman, "3rd Very Deep Convolutional Networks for Large-Scale Image Recognition," 2015, Accessed: Mar. 15, 2022. [Online]. Available: https://iclr.cc/archive/www/2015.html

[36] C. Szegedy et al., "Going Deeper with Convolutions," 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 1–9, 2015, doi: https://doi.org/10.1109/cvpr.2015.7298594.

[37] L. Alzubaidi et al., "Review of Deep Learning: Concepts, CNN Architectures, Challenges, Applications, Future Directions," Journal of Big Data, vol. 8, no. 1, Mar. 2021, doi: https://doi.org/10.1186/s40537-021-00444-8.

[38] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," Proceedings of the AAAI Conference on Artificial Intelligence, vol. 31, no. 1, Feb. 2017, doi: https://doi.org/10.1609/aaai.v31i1.11231.

[39] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778, Jun. 2016, doi: https://doi.org/10.1109/cvpr.2016.90.

[40] M. Tan and Q. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," Proceedings of the 36th International Conference on Machine Learning, vol. 97, pp. 6105–6114, May 2019.

[41] J. Gu et al., "Recent Advances in Convolutional Neural Networks," Pattern Recognition, vol. 77, pp. 354–377, May 2018, doi: https://doi.org/10.1016/j.patcog.2017.10.013.

[42] M. A. Qureshi, M. Deriche, and A. Beghdadi, "Quantifying Blur in Colour Images Using Higher Order Singular Values," Electronics Letters, vol. 52, no. 21, pp. 1755–1757, Oct. 2016, doi: https://doi.org/10.1049/el.2016.1792.

[43] P. Soviany and R. T. Ionescu, “Optimizing the Trade-Off between Single-Stage and Two-Stage Deep Object Detectors Using Image Difficulty Prediction,” 2018 20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), Sep. 2018, doi: https://doi.org/10.1109/synasc.2018.00041.

[44] J. Redmon and A. Farhadi, “YOLO9000: Better, Faster, Stronger,” 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Jul. 2017, doi: https://doi.org/10.1109/cvpr.2017.690.

[45] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection,” 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 779–788, 2016, doi: https://doi.org/10.1109/cvpr.2016.91.

[46] M. Elgendy, Deep Learning for Vision Systems. Shelter Island, Ny Manning Publications Co, 2020.

[47] Y. Xiao et al., “A Review of Object Detection Based on Deep Learning,” Multimedia Tools and Applications, vol. 79, no. 33, pp. 23729–23791, Sep. 2020, doi: https://doi.org/10.1007/s11042-020-08976-6.

[48] J. Redmon and A. Farhadi, “YOLOv3: An Incremental Improvement,” arXiv, Apr. 2018, doi: https://doi.org/10.48550/arXiv.1804.02767.

[49] G. Jocher, “Ultralytics YOLOv5,” GitHub, Aug. 21, 2020. https://github.com/ultralytics/yolov5 (accessed Mar. 2023).

[50] A. Paszke et al., “PyTorch: An Imperative Style, High-Performance Deep Learning Library,” Neural Information Processing Systems, vol. 32, pp. 8024–8035, 2019.

[51] X. Zhu, C. Vondrick, C. C. Fowlkes, and D. Ramanan, “Do We Need More Training Data?,” International Journal of Computer Vision, vol. 119, no. 1, pp. 76–92, Mar. 2015, doi: https://doi.org/10.1007/s11263-015-0812-2.

[52] M. Sato et al., “Application of Deep Learning to the Classification of Images from Colposcopy,” Oncology Letters, vol. 15, no. 3, pp. 3518–3523, Mar. 2018, doi: https://doi.org/10.3892/ol.2018.7762.

[53] G. Chassagnon, M. Vakalopolou, N. Paragios, and M.-P. Revel, “Deep Learning: Definition and Perspectives for Thoracic Imaging,” European Radiology, vol. 30, no. 4, pp. 2021–2030, Dec. 2019, doi: https://doi.org/10.1007/s00330-019-06564-3.

[54] T.-Y. Lin et al., “Microsoft COCO: Common Objects in Context,” Computer Vision – ECCV 2014, pp. 740–755, 2014, doi: https://doi.org/10.1007/978-3-319-10602-1_48.

[55] COCO Consortium, “COCO - Common Objects in Context,” 2018. https://cocodataset.org (accessed Jan. 18, 2020).

[56] L. Fei-Fei, J. Deng, and K. Li, “ImageNet: Constructing a Large-Scale Image Database,” Journal of Vision, vol. 9, no. 8, pp. 1037–1037, Mar. 2010, doi: https://doi.org/10.1167/9.8.1037.

[57] Stanford Vision Lab, Stanford University, Princeton University, “ImageNet,” 2009. https://www.image-net.org/index.php (accessed 01, 2020).

[58] K. Weiss, T. M. Khoshgoftaar, and D. Wang, "A Survey Of Transfer Learning," Journal of Big Data, vol. 3, no. 1, May 2016, doi: https://doi.org/10.1186/s40537-016-0043-6.

[59] R. Sieczka and M. Pańczyk, "Blender as a Tool for Generating Synthetic Data," Journal of Computer Sciences Institute, vol. 16, pp. 227–232, Sep. 2020, doi: https://doi.org/10.35784/jcsi.2086.

[60] M. A. Nielsen, "Neural Networks and Deep Learning," 2015. https://neuralnetworksanddeeplearning.com/ (accessed Oct. 2021).

[61] L. Perez and J. Wang, "The Effectiveness of Data Augmentation in Image Classification Using Deep Learning," arXiv, Dec. 2017, doi: https://doi.org/10.48550/arXiv.1712.04621.

[62] E. A. Hadhrami, M. A. Mufti, B. Taha, and N. Werghi, "Transfer learning with convolutional neural networks for moving target classification with micro-Doppler radar spectrograms," 2018 International Conference on Artificial Intelligence and Big Data (ICAIBD), pp. 148–154, May 2018, doi: https://doi.org/10.1109/ICAIBD.2018.8396184.

[63] S. Basak, Hossein Javidnia, F. Khan, R. McDonnell, and M. Schukat, "Methodology for Building Synthetic Datasets with Virtual Humans," Trinity's Access to Research Output (TARA) (Trinity College Dublin), Jun. 2020, doi: https://doi.org/10.1109/issc49989.2020.9180188.

[64] Blender Foundation, "blender.org - Home of the Blender project - Free and Open 3D Creation Software," blender.org, 2019. https://www.blender.org/ (accessed Jan. 2024).

[65] D. Senkic, "Dynamic simulation in a 3D-environment : A comparison between Maya and Blender," Dissertation, 2010.

[66] Blender Documentation Team, "Blender 3.4 Reference Manual," 2023. https://docs.blender.org/manual/en/3.4/index.html (accessed Jan. 2024).

[67] G. Moioli, Introduction to Blender 3.0. Apress, 2022. doi: https://doi.org/10.1007/978-1-4842-7954-0.

[68] S. Lens and W. Adriaensen, Procedural 3D Modeling Using Geometry Nodes in Blender. Packt Publishing Ltd, 2023.

[69] Blender Documentation Team, "Blender 2.79 Reference Manual," 2017. https://docs.blender.org/manual/en/2.79/index.html (accessed Nov. 2023).

[70] I. A. Astuti, I. H. Purwanto, T. Hidayat, D. A. Satria, and R. Purnama, "Comparison of Time, Size and Quality of 3D Object Rendering Using Render Engine Eevee and Cycles in Blender," 2022 5th International Conference of Computer and Informatics Engineering (IC2IE), pp. 54–59, Sep. 2022, doi: https://doi.org/10.1109/ic2ie56416.2022.9970186.

[71] S. Crowder, Shading, Lighting, And Rendering with Blender's EEVEE: Learn How to Create and Iterate Amazing Concept Art Using a Real-Time Rendering Engine. Birmingham: Packt Publishing, Limited, 2022.

[72] D. Garcia, J. Orteu, L. Robert, B. Wattrisse, and F. Bugarin, "A Generic Synthetic Image Generator Package for the Evaluation Of 3D Digital Image Correlation and Other Computer Vision-Based Measurement Techniques," PhotoMechanics 2013, p. Clé USB, May 2013, doi: https://hal.science/hal-00836324.

[73] D. Schraml and G. Notni, “Synthetic Training Data in AI-Driven Quality Inspection: The Significance of Camera, Lighting, and Noise Parameters,” Sensors, vol. 24, no. 2, p. 649, Jan. 2024, doi: https://doi.org/10.3390/s24020649.

[74] S. Majboroda, “Machine Shop 02 HDRI,” Poly Haven: The public 3D asset library, 2020. https://polyhaven.com/a/machine_shop_02 (accessed Jun. 2023).

[75] A. Jung, “imgaug,” GitHub, May 19, 2020. https://github.com/aleju/imgaug (accessed Feb. 2024).

[76] A. Mikołajczyk and M. Grochowski, “Data Augmentation for Improving Deep Learning in Image Classification Problem,” 2018 International Interdisciplinary PhD Workshop (IIPhDW), pp. 117–122, May 2018, doi: https://doi.org/10.1109/IIPHDW.2018.8388338.

[77] P. Kumar and M. Shingala, “Native Monkey Detection Using Deep Convolution Neural Network,” in Advanced Machine Learning Technologies and Applications: Proceedings of AMLTA 2020, Springer Nature, 2021, pp. 373–383. doi: https://doi.org/10.1007/978-981-15-3383-9_34.

[78] V. Miglani and M. Bhatia, “Skin Lesion Classification: A Transfer Learning Approach Using EfficientNets,” in Advanced Machine Learning Technologies and Applications: Proceedings of AMLTA 2020, 2021, pp. 315–324. doi: https://doi.org/10.1007/978-981-15-3383-9_29.

[79] P. Singh, N. Singh, K. K. Singh, and A. Singh, “Diagnosing of Disease Using Machine Learning,” Machine Learning and the Internet of Medical Things in Healthcare, pp. 89–111, 2021, doi: https://doi.org/10.1016/b978-0-12-821229-5.00003-3.

Appendix

Python script for phase one of the augmentations.

```
import imgaug as ia
import imgaug.augmenters as iaa
import cv2
import glob
import imageio
import os
from scipy import ndimage

def main():
    # path of the folder containing the raw images
    inPath ="D:/dlproj/data/cassification/debug"

    # path of the folder that will contain the modified image
    outPath ="D:/dlproj/data/cassification/debug_aug"

    # 1. Configure Image Augmentation

    transformations = iaa.SomeOf((1, None), [
        iaa.Fliplr(1.0),
        iaa.Flipud(1.0),
        iaa.Rot90((1, 3))
    ],  random_order=True
    )

    for folder in os.listdir(inPath):
        # 2. Load Image Dataset

        # Read the images into the in_images list
        in_images = []
        for fname in os.listdir(inPath + '\\' + folder):
            img = imageio.imread(inPath + '\\' + folder + '\\' \
             + fname, pilmode="RGB")
            in_images.append(img)

        # 3. Apply the augmentations to all the images in the list and save the
        # augmented images to the images_aug list
        images_aug = transformations(images=in_images)

        # iterate over every image in the images_aug list and save
        # the iumages to the appropriate output folder
        i=2001
        for im1 in images_aug:
            imageio.imwrite(outPath + '/'
                            + folder + '/' + folder+ "%05d.png" % (i,),im1)
            i+=1

# Driver Function
if __name__ == '__main__':
    main()
```

Python script for phase two of the augmentations.

```
#import numpy as np
import imgaug as ia
import imgaug.augmenters as iaa
import cv2
import glob
import imageio
import os
from scipy import ndimage

def main():

    # path of the folder containing the raw images
    inPath ="D:/dlproj/data/cassification/debug"

    # path of the folder that will contain the modified image
    outPath ="D:/dlproj/data/cassification/debug_aug"

    # 1. Configure Image Augmentation
    colouraug = iaa.Sequential(
    [
        # don't execute all of them, as that would often be way too strong
        iaa.SomeOf((1, None),
        [
            iaa.OneOf([
            iaa.GaussianBlur((0, 3.0)), # blur images with a sigma
            iaa.AverageBlur(k=(2, 2)), # blur image using local averages
            iaa.MedianBlur(k=(3, 7)), # blur image using local medians
            ]),
        # blend the result with the original image
        iaa.AdditiveGaussianNoise(loc=0, scale=(0.0, 0.04*255), per_channel=0.5),
        iaa.Add((-7, 7), per_channel=0.5), # change brightness of images
        iaa.AddToHueAndSaturation((-10, 10)), # change hue and saturation
        iaa.Grayscale(alpha=(0.5, 1.0))
        ],
    random_order=True
    )
    ],
    random_order=True
    )

    for folder in os.listdir(inPath):
        # 1. Load Dataset
        #ensure you are in training image folder before running
        in_images = []
        for fname in os.listdir(inPath + '\\' + folder):
          img = imageio.imread(inPath + '\\' + folder + '\\' + fname, pilmode="RGB")
          in_images.append(img)

        images_aug = colouraug(images=in_images)

        # iterate over every example image and save it as jpg files
        i=5000
        for im1 in images_aug:
            imageio.imwrite(outPath + '/'
                            + folder + '/' + folder+ "%06d.png" % (i,),im1)
            i+=1

# Driver Function
if __name__ == '__main__':
    main()
```