
POLITECNICO DI TORINO

**Master of Science Course in**
**Materials Engineering for Industry 4.0**

Master's Degree Thesis

# AI-Based Classification Method Identifying Burns on Used PCB Boards for Reuse in a Circular Manufacturing Process

**Supervisors**
Prof. Daniele Ughues
Prof. Dr.-Ing. Martin Dix
Dr. Cédric Courbon
Dr.-Ing. Matthias Rehm

**Candidate**
Muhammad Junaid Ali

September 2025

**Erasmus Mundus Joint Master in Manufacturing 4.0**

**by intElligent and susTAinable technologies**


**MASTER's Degree Thesis**

# AI-Based Classification Method Identifying Burns on Used PCB Boards for Reuse in a Circular Manufacturing Process

**Supervisors**

Prof. Daniele Ughues
Prof. Dr.-Ing. Martin Dix
Dr. Cédric Courbon
Dr.-Ing. Matthias Rehm

**Candidate**

Muhammad Junaid Ali

**September 2025**

CENTRALE LYON ENISE · MINES Saint-Étienne Une école de l'IMT · NTNU · Politecnico di Torino · TECHNISCHE UNIVERSITÄT CHEMNITZ · Univerza v Ljubljani

# Abstract

The rapid growth of EOL and used products has intensified the need for circular manufacturing strategies that enable reuse and remanufacturing of products. One of the most critical challenges in this context is the visual inspection of EOL/used products to determine reusability. Traditional approaches rely either on manual inspection, which is often slow, inconsistent, and dependent on individual judgment, or on automated inspection systems based on rule-based image processing, which lack adaptability to new defect types, new products, and changing environmental conditions. This thesis addresses these challenges in the electronics domain by developing an AI-based classification method to classify burnt PCBs from reusable printed circuit boards (PCBs).

A dataset is developed by combining burnt and good PCB images collected from literature, online sources, and manually verified cases with synthetic images generated using generative AI tools such as ChatGPT (DALL·E) and Gemini. Data integrity is ensured through perceptual hashing and deep feature filtering with a pre-trained ResNet50 model to remove duplicate and augmented images. Finally, six targeted augmentation pipelines are also applied to introduce realistic variations in geometry, lighting, occlusion, noise, background, and compression.

Several state-of-the-art deep learning architectures are fine-tuned on this dataset using transfer learning with pre-trained ImageNet weights. CNNs demonstrated the strongest performance: YOLOv11 and ResNet50 both achieve 98% accuracy with perfect precision for burnt PCBs, while EfficientNetB3 follows closely with an F1-score of 0.96 and perfect precision on good PCBs. In contrast, larger CNNs such as ResNet152 (F1 = 0.77) and EfficientNetB7 (F1 = 0.90) show weaker generalization despite their higher capacity. DeiT achieves competitive performance with an F1-score of 0.92, whereas the self-supervised DINO variants underperform (F1 = 0.72 for ViT, 0.54 for CaIT backbones). These findings highlight that in data-scarce industrial domains, lightweight CNNs outperform deeper or more complex models.

The results demonstrate that AI-based visual inspection can significantly improve PCB reuse decisions in remanufacturing.

# Acknowledgments

First of all, I am grateful to Allah Almighty for giving me wisdom, courage, and knowledge to reach this stage of my education. Secondly, I would like to deeply thank my family for their constant love, support, and prayers gave me the strength to keep going, even during the toughest times.

I would like to especially thank Dr. rer. med. Mario Lorenz for providing me with the opportunity to complete my master's thesis at the Department of Mechanical Engineering at Technische Universität Chemnitz (TUC). His supervision and guidance were instrumental in making this research possible.

I am also deeply thankful to M.Sc. Arwa Own. The guidance and supervision that she provided from the early stages of the thesis until the final results evaluation.

As someone with a mechanical engineering background, working on an AI-based topic was quite challenging for me at first. I am especially grateful to my Pakistani friends from the Department of Computer Science at TUC, who supported me during this journey. Whether it was helping me understand the basics of AI or guiding me through technical issues, they were always there when I needed them.

# Table of Contents

# List of Figures

# List of Tables

# Nomenclature

### Abbreviations & acronyms

| | |
|---|---|
| CE | Circular Economy |
| EOL | End of Life |
| PCB | Printed Circuit Board |
| AI | Artificial Intelligence |
| CV | Computer Vision |
| ML | Machine Learning |
| DL | Deep Learning |
| RL | Reinforcement Learning |
| SSL | Self-Supervised Learning |
| CNN | Convolutional Neural Network |
| ViT | Vision Transformer |
| GPU | Graphics Processing Unit |
| RGB | Red Green Blue |
| GAP | Global Average Pooling |
| EMA | Exponential Moving Average |
| SiLU | Sigmoid Linear Unit |
| GELU | Gaussian Error Linear Unit |
| mAP | Mean Average Precision |
| ReLU | Rectified Linear Unit |
| Yolov11 | You Only Look Once version 11 |
| DINO | Self-Distillation with No Labels |
| ResNet | Residual Network |
| EfficientNet | Efficient Convolutional Neural Network |
| DeiT | Data Efficient Image Transformer |
| CaiT | Class-Attention in Image Transformers |
| PR Curves | Precision–Recall curves |
| WA | Weighted average |
| IT | Inference Time |
| FPS | Frames Per Second |
| DOE | Design of Experiment |

# Chapter 1

# Introduction

## 1.1. Motivation

The amount of waste from electrical and electronic equipment (WEEE) generated globally increased by 82% from 2010 to 62 Mt in 2022, and is anticipated to reach 82 Mt by 2030 [1]. This growing e-waste problem calls for more sustainable and circular approaches in manufacturing. Remanufacturing has become an essential strategies for reducing waste, conserving resources, and minimizing environmental impact. 85% of the raw material and 55% of the energy can be saved through remanufacturing compared to the production of new parts [2].

AI has evolved over the years. The features of AI, such as data analytics, optimization algorithms, and image processing, can be applied to make CE more efficient. AI covers a wide range of techniques such as ML, DL, natural language processing, CV, and RL. These technologies help AI to match up with human intelligence and perform tasks such as pattern recognition, image recognition, sentiment analysis, language translation, facial recognition, and robotics control. These characteristics of AI have a huge impact on the transition of the traditional linear economics model of production and consumption to CE through closed-loop continuous resource recycling. The complex challenges of resource efficiency and sustainability can be addressed through AI to a considerable extent [3][4]. This motivates exploring how AI can be applied specifically to the visual inspection of used PCBs. This thesis aims to develop a method that uses AI to classify whether a PCB has surface burns or not, a curricle first step to determine its reusability. This kind of solution can support more efficient remanufacturing, reduce waste, and contribute to smarter circular manufacturing practices.

## 1.2. Problem Statement

One of the biggest challenges in remanufacturing is the amount of manual work involved in inspecting and sorting returned products [5]. Consequently, even if a company successfully implements remanufacturing processes, they are often carried out in low-wage countries [6]. A highly critical and value-adding step is the selection of returning used products, which requires initial visual inspection by workers who then make

technological and economic decisions regarding the reusability or remanufacturability of the returning used products [7]. For example, when printed circuit boards (PCBs) come back for reuse, usually workers visually inspect each one to decide whether it can still be used or burned and can not be reused/remanufactured. This process is time-consuming and also dependent on human judgment, especially when it comes to identifying issues like burn marks or physical damage. Although automation can help reduce costs and make remanufacturing more efficient, there's still a lot of uncertainty involved [8] due to the strict rules based on algorithmic automation. The quality of returned products varies [9], and most come in small batches [10]. Because of that, it's hard to standardize the inspection process in the context of automation, and companies often require human flexibility and adaptability[6].

## 1.3. Objectives

This thesis aims to develop a computer-vision-based Printed Circuit Board (PCB) classification system using AI to distinguish between burnt and good PCBs, assessing their suitability for reuse in a circular manufacturing framework. To achieve the goal, the first objective is to collect a dataset of Good PCBs and Burnt PCBs, then clean and prepare the data to improve quality and remove any repeated images. Next, we employ data augmentation techniques to create more realistic variations of the images, including different lighting conditions, rotations, backgrounds, and noise levels, thereby increasing the dataset's size and diversity. Once we have cleaned the dataset, a suitable DL classification model will be trained, validated using metrics such as accuracy, precision, recall, confusion matrix, and confidence analysis, and tested to distinguish between burnt and good PCBs. Finally, the classification pipeline is assessed for its scalability and potential to support resource-efficient practices in remanufacturing.

## 1.4. Thesis Structure

This thesis is organized as follows: Chapter 1 introduces the goals, motivation, and objectives. Chapter 2 reviews existing research in AI-based CV and state-of-the-art DL models. Chapter 3 explains data preparation, training, evaluation, and detailed DL model architectures. Chapter 4 presents the methodology, covering dataset collection, preprocessing, training, and evaluation. Chapter 5 reports results with metrics and visualizations. Chapter 6 discusses and interprets the results, highlighting limitations and future directions. Finally, Chapter 7 concludes with the key findings.

# Chapter 2

# Literature Review

## 2.1. Circular Economy and Strategies

Across 114 definitions, CE is most frequently described as a process of reduce, reuse, and recycle activities, whereas it is oftentimes not highlighted that CE necessitates a systemic shift[11]. Later work, based on 221 Definitions of CE, conceptualized the conservation of product, material, and resource value across cycles. In other words, it is a regenerative economic system that replaces the EOL/used products concept with reusing, recycling, and remanufacturing products or materials throughout the supply chain for sustainable development [12].

Figure 2.1 shows that 5.22 Gt of raw materials processed in the EU originate from natural resource extraction, 1.51 Gt from imports. 4.75 Gt of raw materials processed were used to make products; however, only 1.02 Gt from recycling and backfilling. The rest were mainly exported or used for producing energy. Moreover, the EU's overall circularity rate (the share of recycled materials in total material use) is 11.8 %, which means the European Union still relies on a linear economy [13] and aiming to increase it to 22.4% by 2030 [14].

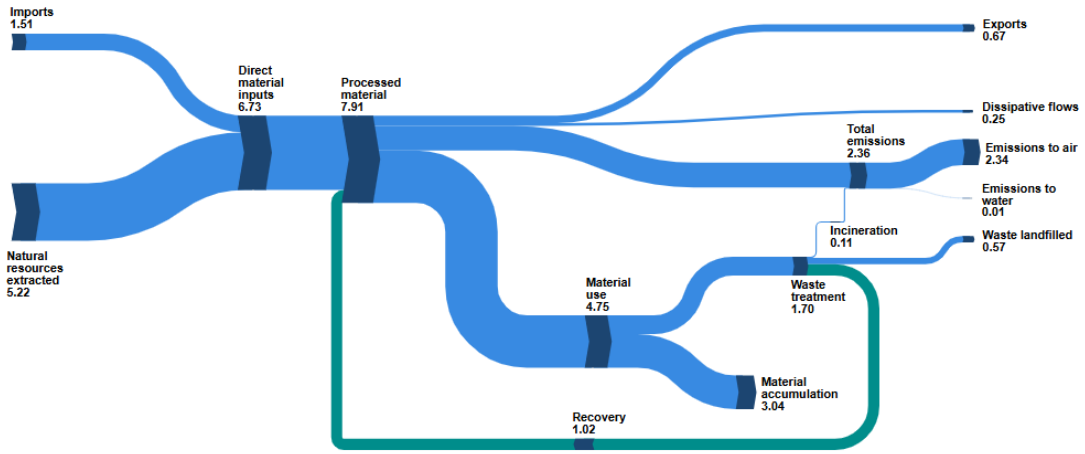



**Figure 2.1:** Material flows EU 2023 in Gigatonnes [15]

A meta-analysis of comparative life cycle assessment on mechanical products finds that remanufacturing can reduce global warming potential to 28.5% and primary energy consumption to 25.9% of new manufacture on average [16].

Global E-Waste Monitor reported that 62 million tonnes of e-waste were produced in 2022, and the generation of e-waste is on track to rise to 82 million tonnes in 2030. Waste electrical and electronic equipment (WEEE) poses a serious issue due to its complex mix of materials, making recycling both difficult and resource-intensive [1], [17].

Several scholars and organizations have proposed a broader set of strategies to operationalize circularity. Among the most widely adopted is the R9 framework, which extends beyond the traditional 3Rs to include nine complementary strategies as explained in Figure 2.2. These strategies, when combined, provide a hierarchical approach to conserving resources, prioritizing value retention at the highest possible level before resorting to material recycling or energy recovery [18] [19].

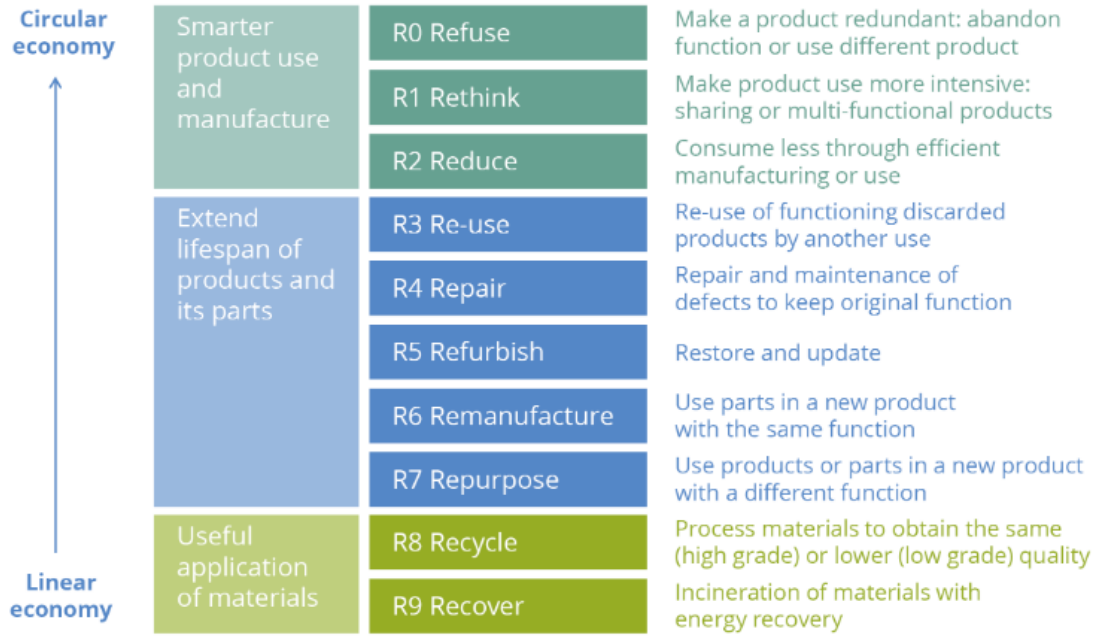



**Figure 2.2:** Circular economy strategies [20]

### 2.1.1. The Remanufacturing Process

Remanufacturing is defined as a manufacturing process in which end-of-life products are rebuilt to be used in a new lifecycle. Unlike recycling, which mainly focuses on material recovery, remanufacturing restores full functionality and can also manufacture an upgraded product comparable to or better than a new product [21]. Remanufacturing distinguishes itself from reconditioning or repair because it is the process that can compete with a new product in terms of quality [22]. There are also international

standards to define remanufacturing processes, which include BS8887-2:2009 [23], ANSI RIC001.2-2021 [24], and the latest publication issued by the German Institute, DIN SPEC 91472:2023 [25]. The general remanufacturing processes are shown in Figure 2.3.

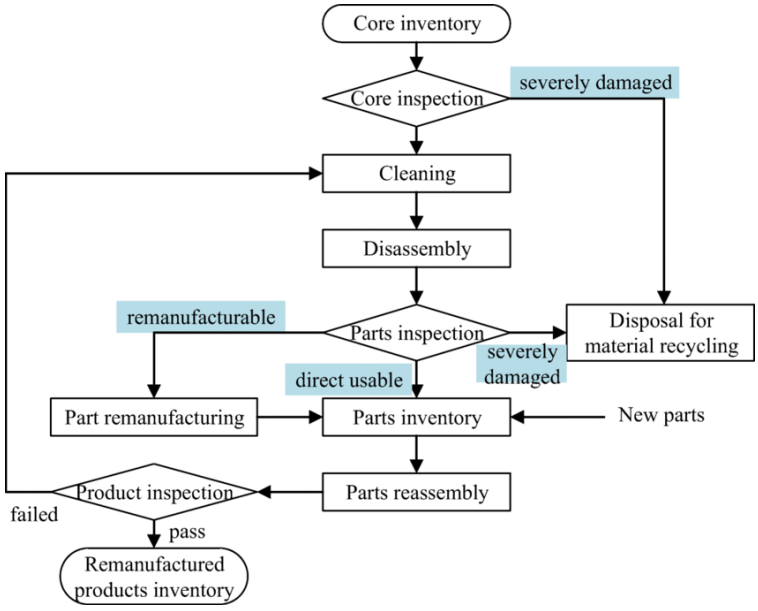



**Figure 2.3:** Remanufacturing Process [26]

The key sequences of operations involved in remanufacturing differ depending on the constraints and requirements of the product and the industry [27]. An overview of the remanufacturing process's key operations for different industries are given in Table 2.1.

**Table 2.1:** Remanufacturing Processes Across Diverse Industries

| Industry / Scope | Key Processes in Remanufacturing | Reference |
|---|---|---|
| **Generic Industrial Remanufacturing** | Cores → Inspection and cleaning → Disassembly → Component Reprocessing → Reassembly → Testing | [26][28] |
| **Automotive (General framework)** | Cores → Cores Cleaning → Disassembly → Cleaning → Inspection and Sorting → Refurbishment → Reassembly → Testing | [29] [30] [31] [32] |
| **Industrial Equipment (Design phases)** | Design & development → Product Collection → Disassembly → Condition assessment and sorting → cleaning → Repair → Re-assembly → Testing | [33] |
| **Industrial Electronics (PCBs, EEE, etc.)** | Inspection → Disassembly → Cleaning → Troubleshooting → Component replacement/refurbishment → Reassembly → Testing | [34] [35] |
| **Appliances Industry** | Inspection → Disassembly → Parts replacement/refurbishment → Cleaning → Reassembly → Final testing | [36] [37] |

Despite its significant environmental and economic benefits, remanufacturing faces persistent challenges such as end-of-life product sourcing, component variability, and

unpredictable failure rates that hinder process efficiency. To address these issues, functional requirements and associated performance measures have been defined. One of these functional requirements is the supply of cores and their inspection [38].

## 2.2. Inspection of Used/EOL parts

Inspection is defined as the testing of the conformity of the relevant characteristics of a product by observation, measurement, or functional testing [39]. Inspection is an essential part of the remanufacturing process, as illustrated in Table 2.1. EOL products and their components must be carefully examined to identify visible defects or damage [40]. The uncertain condition of returned products constitutes a major source of complexity in remanufacturing, distinguishing it significantly from linear manufacturing [41]. Additionally, unlike conventional manufacturing, where sampling-based inspection methods are used, remanufacturing requires 100% inspection of all components to guarantee that the final product meets quality and performance specifications [42]. During the initial inspection, products are typically classified into three categories: (1) reusable, (2) remanufacturable, and (3) non-remanufacturable [43].

### 2.2.1. Visual Inspection

The remanufacturing process typically begins with an initial visual inspection to determine whether a returned product can proceed further in the process chain. For this purpose, remanufacturers often define core acceptance criteria that outline the conditions under which a product or its subassemblies are considered suitable for remanufacturing [44]. Therefore, these criteria vary across companies and product types, specifying tolerances for mechanical and electrical defects, permissible levels of corrosion, and the acceptable extent of wear, disassembly, or missing components. Despite such guidelines, inspection outcomes are still largely qualitative, since they rely on the operator's sensory perception and judgment. In practice, the inspector assesses the product's reusability based on observable characteristics and, when necessary, supplements this with haptic inspection [45]. From a methodological perspective, visual inspection is regarded as a non-dimensional inspection method, as performed along the process of remanufacturing, as it does not provide measurable data but instead produces results based on subjective human evaluation. While this makes it low-cost, it also introduces a high degree of variability and dependence on operator expertise, which is a known limitation in the remanufacturing industry.

### 2.2.2. Automated Visual Inspection

Manual visual inspection by a worker is susceptible to errors and can reach error rates of up to 30% [46]. Extensive research has recently been conducted on automated systems for visual inspection because it improve the accuracy and efficiency of inspection. At a fundamental level, these systems typically comprise a light source, an image acquisition unit/vision sensor systems, and a processing module (Figure 2.4) [47].

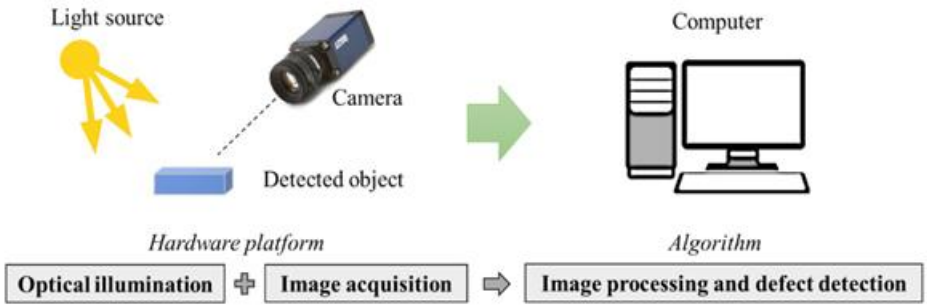



**Figure 2.4:** Automated Visual Inspection System[48]

The acquisition system generates digital signals in various formats, including binary, grayscale, color images, depth maps, and point clouds. Each type of data has a specific purpose: binary images detect object presence, grayscale images analyze texture, color images identify surface defects, and point clouds are used for geometric inspections [49].

Moreover, automated visual inspection is an image processing approach that involves various techniques for modifying captured images of an object to make a reasonable decision based on the photos. The methods involve filtering, projection, learning, hybrid, and other algorithm-based automated visual inspection systems [50].

Automated inspection reduces process complexity, improves material utilization, and minimizes non-remanufacturable parts. It ensures better value recovery, consistent quality, and lower costs. Additionally, automation increases process throughput and accuracy, enhances worker safety by reducing risks, and shortens factory lead times, making the overall remanufacturing process more efficient [51].

### 2.2.3. AI-based Visual Inspection

Rigid rule-based algorithms used in automated inspection lack adaptability to new or complex defect types, especially under variable lighting conditions, surface textures, or product orientations. As a result, their performance drops in a realistic environment. To overcome these variable issues in Industry 4.0, AI-based algorithms are used in visual inspection, which relies on data-driven models to automatically learning features and patterns directly from image data (defective and non-defective samples), instead of

depending on pre-defined rulesm, enabling the system to perform accurate classification, localization, or segmentation of faults [52]. Additionally, AI-based inspection systems are based on the same fundamental hardware configuration as automated inspection (vision sensors and a processing computer system).

## 2.3. AI-Based PCB Inspection: Datasets and Models

The field of Printed Circuit Board (PCB) inspection has undergone a significant transformation through the integration of DL CV. These advancements range across multiple inspection domains, including defect detection on bare boards, component localization on assembled boards, and anomaly detection. Each domain benefits from specialized datasets and leading AI models that push the boundaries of speed and accuracy.

The first step for DL model is the requirement of a dataset. Existing datasets for DL based PCB visual analysis mainly contain good or defective PCBs. The PCB-DSLR dataset [53] is the first publicly available dataset to facilitate research on CV-based PCB analysis, comprising 748 images of PCBs (165 different PCBs, 3 to 5 images per PCB) captured under representative conditions using a professional DSLR camera at a recycling facility. The images are annotated for bounding boxes for 2048 unique labeled ICs, 9313 labeled ICs in total. The other publicly available dataset at [54] includes 480 images of 80 different PCBs, but consists of low-quality images that are inadequate for analysis at the detail level. The authors in another work [55] performed text recognition on both boards and its components. The dataset used by them consists of 860 PCB segments with text. However, no specific details were available on the number of boards used. PCB-METAL [56] provides PCB high-resolution image dataset that can be utilized for DL based component analysis. The dataset consists of 984 high-resolution images of 123 unique PCBs with bounding box annotations for ICs(5844), Capacitors(3175), Resistors(2670), and Inductors(542). The dataset is useful for image-based PCB analysis, such as component detection, PCB classification, circuit design extraction, etc. PCBA-Defect [57] dataset for defect detection and classification, containing a PCB dataset containing 1386 images with 6 kinds of defects (missing hole, mouse bite, open circuit, short, spur, and spurious copper), 1742 for the use of detection and classification of PCB using a CNN, achieving 99.40% average precision. DeepPCB [58] contains 1,500 image pairs with annotations, including positions of 6 common types of PCB defects. Experiment results validate the effectiveness and efficiency of the proposed CNN-based model by the author, namely group pyramid pooling, achieving 98.6% mAP at 62 FPS on the DeepPCB dataset.

FICS-PCB dataset [59] collected at the SeCurity and AssuraNce lab at the University of Florida, and it is designed to support the evaluation of automated PCB visual inspection systems. This dataset consists of 9,912 PCB sample images collected from a DSLR camera and a digital optical microscope, and the annotation results in 77,347 component images. This dataset was used for performance evaluation on state-of-the-art PCB component classification methods. PCB-Vision [60] contains Multiscene RGB-Hyperspectral PCB dataset includes 53 hyperspectral data cubes and their corresponding 53 RGB images of PCBs, collected under industrial-like conditions with dual segmentation annotations. The Multi-Perspective and Illumination PCB (MPI-PCB) [61] dataset contains 1742 images with dimensions of 4096 × 2816 pixels showing an unmodified PCB.

After the literature review of the PCB dataset, the next step is DL models, which have been used for PCB visual inspections. A defect detection network based on Coordinate Feature Refinement (CFR) [62] proposed by modifying the YOLOv5s framework. The model integrates four CFR modules to adaptively suppress conflicting multi-scale features, uses a Content-Aware ReAssembly of Features (CARAFE) upsampler for better semantic aggregation, and adds an extra detection layer for tiny objects. The method was tested on PCB dataset containing 1,386 images across six defect classes, expanded to 8,316 samples after augmentation. The approach achieved a mAP of 97.9%, outperforming YOLOv4 (85.3%), YOLOv7 (94.7%), and EfficientNet (78.2%), while maintaining a compact model size of 32.8 MB and real-time speed of 52.7 FPS. The improvements were particularly significant for small defect types such as spurs, mouse bites, and open circuits, making it suitable for industrial real-time PCB defect detection.

[63] studied a PCB surface defect detection network, YOLO-HMC, based on YOLOv5, introducing a single-detection head approach, achieving mean average precision of 98.6%. This framework also trains fewer parameters, 5.94 million as compared to YOLOb8, 30.07 million parameters. The model integrates HorNet in the backbone for enhanced feature extraction, CARAFE, and multiple convolutional block attention module modules that detect defects from highly similar PCB backgrounds.

An enhanced YOLOv5s framework named YOLO-MBBi [64] was introduced for PCB surface defect detection. The improvements include the use of MBConv modules from EfficientNetB0 in the backbone, CBAM attention to strengthen feature learning, BiFPN for multiscale feature fusion, depth-wise convolutions to reduce computation, and replacement of the CIoU loss with SIoU loss. The method was evaluated on the PKU-Market-PCB dataset (1,200 images, six defect types) and the DeepPCB dataset (1,500

pairs of normal and defect images). On PKU-Market-PCB, the model achieved 95.3% mAP50, 95.8% precision, and 94.6% recall at 48.9 FPS, while reducing FLOPs to 12.8 G, lower than YOLOv7's 103.2 G. On DeepPCB, it reached a mAP50 of 99.0%, precision of 98.7%, and recall of 97.5%, with an inference speed of 69.5 FPS. Compared to Faster R-CNN, YOLOv4, YOLOv7, and the baseline YOLOv5s, YOLO-MBBi consistently achieved higher detection accuracy with much lower computational complexity, making it well-suited for industrial PCB inspection.

A transformer-based detection model named MFAD-RTDETR [65] extends RT-DETR with modules designed for small PCB defect detection. It incorporates a Detail Feature Retainer (DFR) for local detail preservation, Visual State Space (VSS) for efficient global attention, Deformable Attention for precise localization, and a Multi-Frequency Aggregate Diffusion (MFAD) paradigm for fine-grained multi-scale feature fusion. On the Peking University PCB dataset (1,386 images, six defect types), the model achieved 97.0% mAP, with an F1-score of 0.955 and an 18.2% parameter reduction compared to RT-DETR. Detection accuracy was strong for missing holes (AP 99.5%) and remained above 92% for other defect types, outperforming YOLOv5, YOLOv8, and Faster R-CNN.

The REDef-DETR [66] model introduced an efficient and real-time transformer-based detector for industrial surface defects, including PCB inspection. It extends RT-DETR by adding three modules: a Multi-scale Contextual Information Dilated (MCID) module with large kernel convolutions to expand the receptive field, a Feature Enhancement with Cascaded Group Attention (FECA) module to improve semantic feature extraction, and a Content-Aware Efficient Feature Fusion (CEFF) module for multi-scale fusion using a content-aware mechanism with discrete wavelet transforms. The model was evaluated on NEU-DET (steel defects) and PCB-DET (693 PCB images, six defect types). On PCB-DET, REDef-DETR achieved 98.0%mAP at 79.4 FPS, outperforming RT-DETR (96.1%), YOLOv8m (97.3%), and DsP-YOLO (95.8%). Detection accuracy for missing holes and missing bites exceeded 99% AP, while other defect types remained above 96% AP.

An inspection method based on EfficientDetD1 [67] with an EfficientNetB1 backbone was proposed for PCB surface defect detection. The model integrates BiFPN for bidirectional multi-scale feature fusion, K-means clustering for optimized anchor ratios, and focal loss to tackle class imbalance. The method was evaluated on the HRIPCB dataset (693 images, six defect types). It achieved 89.5% mAP and a recall of 70.1% and a detection speed of 47.3 FPS when the batch size was increased, while requiring only 0.9 GB GPU memory. Compared to Faster R-CNN and RetinaNet, EfficientDet achieved

similar accuracy with much lower computational cost, making it attractive for real-time industrial PCB inspection.

The CS-ResNet [68] model was designed for PCB cosmetic defect detection by extending ResNet with a cost-sensitive adjustment layer to handle class imbalance and different misclassification costs. It was trained on the PCB dataset (40,706 images) and achieved sensitivity of 0.89, G-mean of 0.91, and the lowest total misclassification cost, while maintaining a fast detection speed of 0.007s per image.

A study compared VGG16, InceptionV3, and ResNet50 for PCB defect image classification using a dataset containing defective and non-defective images [69]. Among the three models, ResNet50 achieved the highest classification accuracy of 95.7%, with precision of 97.3%, sensitivity of 93.9%, specificity of 97.4%, and F1-score of 95.6%. After applying data augmentation, the ResNet50 performance further improved to 97.8% accuracy, 99.8% precision, 95.8% sensitivity, 99.8% specificity, and 97.7% F1-score. To enhance small defect recognition, the architecture was modified by integrating Res2Net modules into the residual blocks. The improved ResNet50 achieved 98.3% accuracy and 98.7% sensitivity, while recognition accuracy for small defects increased substantially from 33.9% to 88.4%. These results confirm that ResNet50, particularly with Res2Net enhancements, provides superior feature extraction and defect classification performance for PCB images compared to other CNN models.

A lightweight transformer framework called Lite-DETR [70] designed for the detection of surface tiny defects of PCB. The model is based on RT-DETR and introduces a Lightweight Efficient Backbone Network (LEBN) optimized from ResNet-18, an Image Feature Augmentation Module (IFAM) to improve generalization, and a Refined Cross-Scale Feature Fusion Module (RCFFM) for effective multi-scale defect representation. To improve bounding box matching, a hybrid WMPDIoU loss (Wasserstein distance + MPDIoU) was employed. Evaluation was carried out on the HRIPCB dataset (693 PCB images, six defect types) and the DeepPCB dataset (1,500 images, six defect types). Lite-DETR achieved a 99.0% mAP on HRIPCB and 99.2% on DeepPCB, outperforming YOLOv7, YOLOv8, YOLOv9, and RT-DETR, with an F1-score of 0.97. The model contained only 5.2M parameters (74% fewer than RT-DETR) and maintained an inference speed of 25–30 ms per image, making it both highly accurate and computationally efficient for industrial deployment.

The reviewed literature shows that AI-based PCB inspection has been dominated by supervised detection and segmentation models, focusing on defect localization or component recognition. A variety of improved CNN and transformer-based architectures (e.g., YOLO variants, DET-based models, EfficientDet, and ResNet adaptations) have achieved high accuracies on public datasets such as DeepPCB, HRIPCB, PCB-METAL, and FICS-PCB. However, all existing datasets are limited to standard defect categories or good PCBs, and there is no publicly available dataset containing burnt boards. Additionally, modified architectures are complex to use and not available publicly to train. Furthermore, self-supervised methods have not been applied to PCB-level condition assessment [71]. This gap in both data and methodology justifies the work presented in this thesis, which introduces and evaluates AI models for the classification of good versus burnt PCBs within a circular manufacturing context.

## 2.4. AI in Computer Vision for Classification

The application of AI in CV has progressed rapidly, particularly in image classification tasks that can be used for industrial applications such as medical diagnostics, industrial quality control, and autonomous navigation.

Due to the insufficient amount of input data, as well as the need to speed up the development process, the Transfer Learning principle is applied in a designed system (section 3.2.4). The following AI-based classification models, pretrained on large datasets, can thus be adapted for PCB visual inspection.

### 2.4.1. CNN-Based Models

Multiple CNN-based models are built to improve computational efficiency compared to fully connected networks. They can be divided into two categories: two-stage methods (e.g., Faster R-CNN) and one-stage methods (e.g., YOLO series). CNN architectures differ mainly in their depth (number of layers), kernel or filter sizes, scaling strategies, and the incorporation of residual connections, and are trained on the ImageNet dataset for transfer learning. Figure 2.5 illustrates CNN architectures along with their Top-1 accuracy (model prediction with the highest probability) on the ImageNet dataset [72], as well as the number of parameters and computational operations required.

Figure 2.6 [73] shows the performance of these models on the COCO dataset [74]. YOLOv11 uses fewer parameters than other models, which means it's lighter and faster to run. Also, despite its smaller size, it achieves a higher mean average precision (mAP) on the COCO dataset.

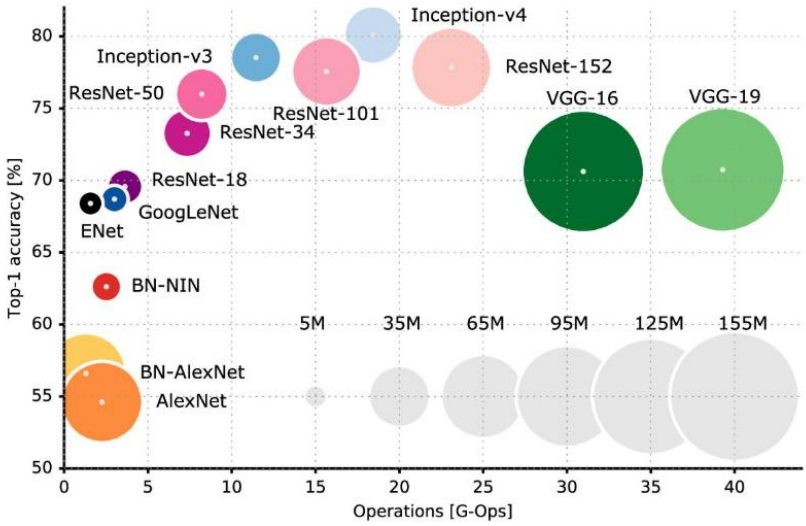



**Figure 2.5:** Comparison of CNN-based DL Models [75]

### 2.4.2. Transformer-based models

Transformer-based models differ from CNNs by replacing convolutional filters with patch embeddings and self-attention mechanisms, enabling the capture of both local and long-range dependencies across the image. Their variations are mainly defined by embedding size, number of attention heads, hierarchical design, and windowing strategies. These models are trained on large datasets such as ImageNet-1k or ImageNet-21k [76], either in supervised or hybrid training schemes. Table 2.2 summarized transformer architectures, comparing their Top-1 accuracy and model size.

**Table 2.2:** Transformers-based Models Comparison

| Transformer | Sub-models / Variants | Parameters (M) | Top-1 Accuracy (ImageNet-1k) |
|---|---|---|---|
| **ViT** [76] | ViT-Tiny (Ti/16), ViT-Small (S/16), ViT-Base (B/16), ViT-Large (L/16), ViT-Huge (H/14) | 5.7M / 22M / 86M / 307M / 632M | 72.0% - 88.0% |
| **DeiT** [77] | DeiT-Tiny, DeiT-Small, DeiT-Base | 5M / 22M / 86M | 74.0% – 85.2% |
| **Swin Transformer** [78] | Swin-Tiny, Swin-Small, Swin-Base, Swin-Large | 29M / 50M / 88M / 197M | 81.3% - 87.3% |
| **Swin Transformer V2** [79] | Swin-B, Swin-L, Swin-G | 88M / 197M / 3B | 87.1% - 90.17% |
| **CaiT** [80] | CaiT-S24, CaiT-M36, CaiT-M48 | 47M / 270M / 356M | 82.7% - 86.5% |
| **CCT (Compact Convolutional Transformer)** [80] | CCT-Tiny, CCT-Small, CCT-Base, CCT-Large | 3M / 12M / 45M / 150M | 75% - 85. 4% |

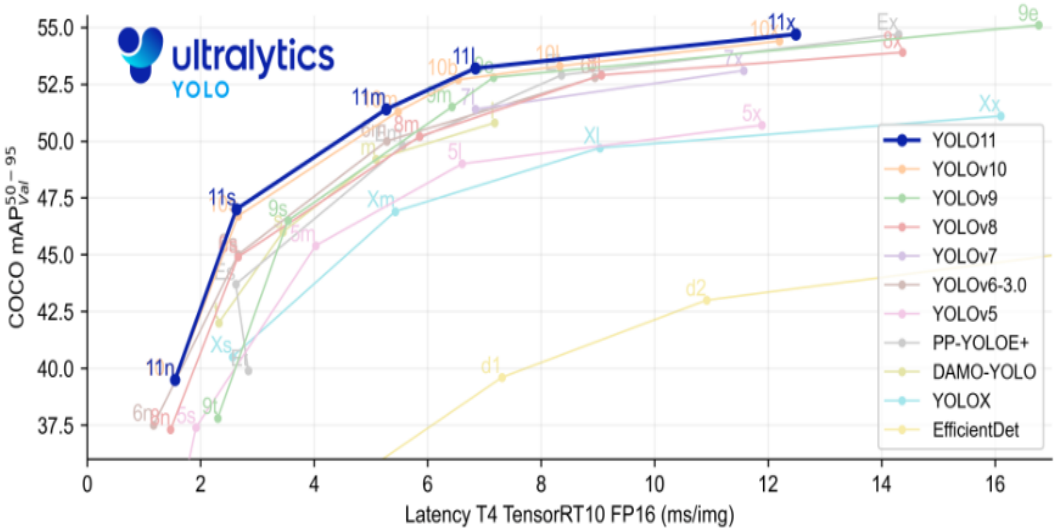



**Figure 2.6:** Comparison of YOLO Family [73]

### 2.4.3. Self-Supervised Learning models

SSL models are designed to reduce dependence on large label datasets by learning visual representations from unlabeled images. SSL models are evaluated using linear classification on ImageNet or through fine-tuning on downstream tasks, where they achieve competitive accuracy compared to fully supervised methods. Table 2.3 summarize SSL models with ViT backbone only ( Appendix B.5 describes a detailed comparison of all SSL models and respective backbones) and their performance in terms of Top-1 accuracy under linear evaluation and k-Nearest Neighbors (K-NN).

**Table 2.3:** SSL Methods with ViT Backbone comparison

| Method | Backbone | Parameters (M) | Linear | k-NN |
|---|---|---|---|---|
| BYOL | ViT-S | 21 | 71.4 | 66.6 |
| MoCoV2 | ViT-S | 21 | 72.7 | 66.3 |
| SwAV | ViT-S | 21 | 73.5 | 66.4 |
| DINO | ViT-S/ViT-B | 21/85 | **77/80.1** | **74.5/77.4** |

## 2.5. Approaches to Address Limited Training Data in AI

A major challenge in training DL models is the need for large, diverse datasets to achieve high performance and prevent overfitting. In practice, especially in specialized fields, large labeled datasets are often unavailable. To address these issues, data augmentation and synthetic data generation using generative AI are effective strategies used in DL models.

### 2.5.1. Data Augmentation

Data augmentation artificially increases the size of the dataset by extracting more information from it, and helps reduce overfitting and improve the performance of models [81]. Table 2.4 summarizes selected works that demonstrate the use of and effectiveness of augmentation in enhancing model performance.

**Table 2.4:** Data Augmentation in deep learning

| Domain / Study | Augmentation Techniques | Purpose | Model Performance | Reference |
|---|---|---|---|---|
| Visual inspection - Reman (Automotive) | Geometric (Flip, Perspective, Shift, Scale, Rotate)<br>Colour (Colour Jitter, RGB Shift, Grayscale Conversion, HSV Shift)<br>Noise (ISO Noise, Gaussian Noise)<br>Blurring (Random, Gaussian)<br>Contrast (CLAHE, Gamma) | Increase the size of the dataset | Best model accuracy 93.9% | [82] |
| (Autonomous Driving) Situation Awareness | Geometric Transformation<br>Colour Transformation<br>Blur Transformation<br>Noise Transformation | Increase model accuracy for unknown data | Achieved model accuracy 92.3% | [83] |
| Image classification (ImageNet subset) | Rotation, Flipping, Cropping, Shading with a hue, NeuralNET Augmentation | Increase the accuracy of classification tasks | Accuracy improved to 77.0% from 70.5% compared to no augmentation | [84] |
| Medical imaging (Chest X-rays) | Rotation, flipping, Downscale, Normalization | Increase the performance of the model | Achieves an F1 score of 0.435. | [85] |
| Medical image data augmentation techniques | Geometric Transformations, Cropping, Occlusion, Intensity Operations, Noise Injection, Filtering | Increase the size of the dataset | Model performance improved by up to 12 – 47% | [86] |
| Image classification (Augmentation Strategies) | Mix-up, Cutout, Auto Augment | Improve generalization | On ImageNet and CIFAR-10, performance increases by 0.4% and 0.6% respectively | [87] |
| Image classification | Traditional, GAN-based synthetic data | Improve robustness, reduce overfitting | Accuracy increases from 85.5% to 91.5% | [81] |

### 2.5.2. Synthetic data generation using generative AI

Generative AI models, particularly text-to-image diffusion models, are not limited to modifying existing datasets but can also create new samples that reflect the statistical properties of real-world data. Recent work has applied this approach to the ImageNet-1K dataset [88], where synthetic images were generated and combined with real data for training. The results show that while models trained only on synthetic data perform

worse than those trained on real images, combining the two sources leads to higher accuracy. As shown in Table 2.5, models trained on generated images perform worse than those trained on real datasets; however, combining them increases accuracy [89].

**Table 2.5:** Accuracy of DL models trained on real, AI-gen, and combined datasets [89]

| Model | Real Data Accuracy (%) | Generated Data Accuracy (%) | Real + Generated Accuracy (%) | Δ Performance |
|---|---|---|---|---|
| ResNet-50 | 76.39 | 69.24 | 78.17 | 1.78 |
| ResNet-152 | 78.59 | 72.38 | 80.15 | 1.56 |
| ViT-S/16 | 79.89 | 71.88 | 81.00 | 1.11 |
| DeiT-B | 81.79 | 74.55 | 82.84 | 1.05 |
| DeiT-L | 82.22 | 74.6 | 83.05 | 0.83 |

Recent research conducted by Google and MIT has demonstrated that synthetic images generated through generative AI diffusion models can be effectively used for training DL models. Table 2.6 shows that models trained on synthetic images achieved accuracy comparable to, and in some cases higher than, those trained with real datasets, with reported improvements of around 1–4%. The study highlighted that a model trained with only 20 million synthetic images was able to surpass the performance of a counterpart trained with 50 million real images, proving that generative AI data not only improves accuracy but also provides significant data efficiency [90].

**Table 2.6:** Accuracy improvement with Generative AI Images [90]

| Model | Real Data Accuracy | Synthetic Data Accuracy | Δ Performance |
|---|---|---|---|
| SimCLR | 60.40% | 62.00% | 1.60% |
| MAE | 51.80% | 56.00% | 4.20% |
| StableRep | 70.30% | 73.50% | 3.20% |
| StableRep | 71.90% | 74.50% | 2.60% |

These findings highlight that AI-generated data can provide additional diversity and support better generalization across both convolutional networks and transformer-based model.

### 2.5.3. Image editing software-based dataset creation

Researchers have employed image-editing software (Adobe Photoshop) to generate synthetic datasets for DL. In [91], a large dataset of manipulated facial images was created by Photoshop. In [92], artificial defects were introduced onto PCB images using Photoshop, producing a dataset of 693 boards with 2953 annotated defects. Using this dataset, a skip-connected convolutional autoencoder achieved 98% accuracy in defect classification, confirming the utility of synthetic data in PCB quality assurance.

# Chapter 3

# State-of-the-Art

This chapter describes the state-of-the-art methods that will be used, based on the literature review, focusing on addressing the key challenges outlined in the problem statement. This section presents the best DL classification methodologies that will be employed to achieve the objectives of this thesis. Additionally, it offers a detailed overview of these approaches to determine which model is the most suitable for the specific research context after analyzing the results.

## 3.1. AI Taxonomy

AI is the ability of computer systems to perform tasks that require human intelligence to think and learn. It includes techniques that allow machines to perceive their environment, reason over data, and make decisions. AI has enabled the design of production systems, automation, predictive maintenance, defect detection, quality control, and enhanced decision-making in the context of Industry 4.0.

### 3.1.1. Machine Learning

ML is a subfield of AI that uses statistics to enable computers to learn and make predictions or decisions from data without being explicitly programmed. The main objective of ML is to allow computers to learn from experience and improve performance over time. A computer program is said to learn from experience for some tasks if its performance at tasks improves with experience. ML has different strategies explained as follows:

a) Supervised learning uses labeled datasets to train the algorithms, where each dataset is associated with a label or outcome. The algorithm learns to develop a relation of the input data to the corresponding output data by minimizing the error between its predictions and the true data. Examples of supervised learning algorithms include linear regression, logistic regression, decision trees, support vector machines (SVM), Random Forest (RF), K-Nearest Neighbors (K-NN), and neural networks.

b) Unsupervised Learning uses unlabeled datasets to train the algorithms to identify hidden patterns or groupings, or correlations in the data. Unsupervised

learning techniques include clustering algorithms (K-means clustering, Gaussian mixture models, fuzzy and hierarchical clustering) and dimensionality reduction strategies (principal component analysis, Independent component analysis, and Singular value decomposition).

c) By using a dataset that includes both labeled and unlabeled data to train the algorithm and combines aspects of supervised and unsupervised learning. The algorithm leverages the labeled data to guide its learning process, while also exploiting the unlabeled data to discover additional patterns or information. Semi-supervised learning is useful when labeled data is limited or expensive to obtain.

### 3.1.2. Deep Learning

DL is a subset of ML that has artificial neural networks with multiple layers to learn from large datasets. DL algorithms compose numerous nonlinear transformations. These transformations help the algorithm in learning hierarchical representations of data. These hierarchical representations enable DL models to capture intricate patterns and relationships in the data, leading to state-of-the-art performance in various tasks such as image recognition, speech recognition, natural language processing, and many others. Figure 3.1 shows the underlying conceptual operation of DL where hidden layers undergo iterative processing. Simple information that machines can understand is the main focus at first, and as more layers are explored, the foundation is progressively built upon. At each subsequent layer, new information is integrated with the existing knowledge, resulting in a cumulative representation of the input data by the final layer.

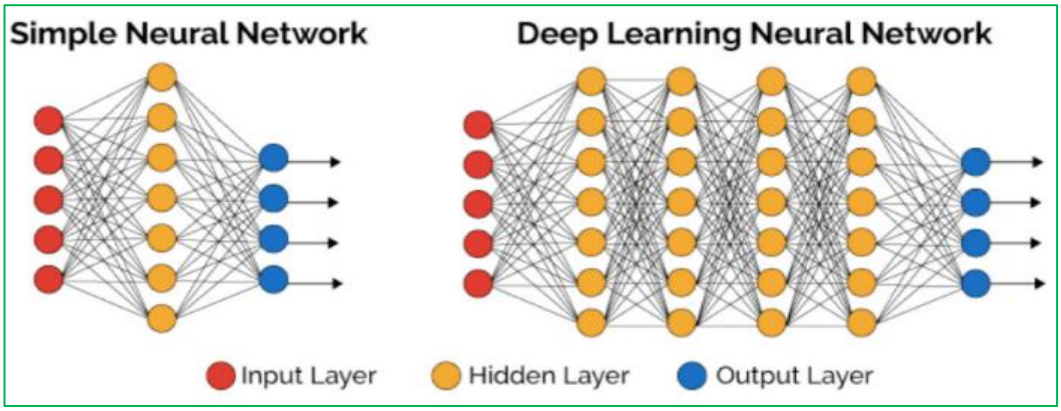



**Figure 3.1:** Deep Learning Model Examples [93]

## 3.2. DL Model Development Steps

DL model development follows a structured workflow that starts with data collection, where relevant datasets are gathered for training. This is followed by data preparation, including cleaning, class organization, augmentation, and dataset splitting. The next stage involves model training and validation to optimize the learning process. Evaluation is then performed to assess performance using defined metrics. Finally, the model is deployed in the industry/real word to make predictions or decisions [94].

### 3.2.1. Data Acquisition

Data acquisition is the first and critical step in developing DL models, as the quality and diversity of data directly affect model performance. For DL-based models for CV, this involves collecting images that represent all classes with realistic variations. Existing datasets from literature or open sources are often used as a starting point [72].

Furthermore, data can be gathered through web searches or domain-specific case studies, followed by manual verification to ensure reliability. Additionally, in many applications, synthetic data generation using generative models is widely adopted to enrich datasets and balance classes [95]. A general decision flow chart of the data collection is shown in Figure 3.2.

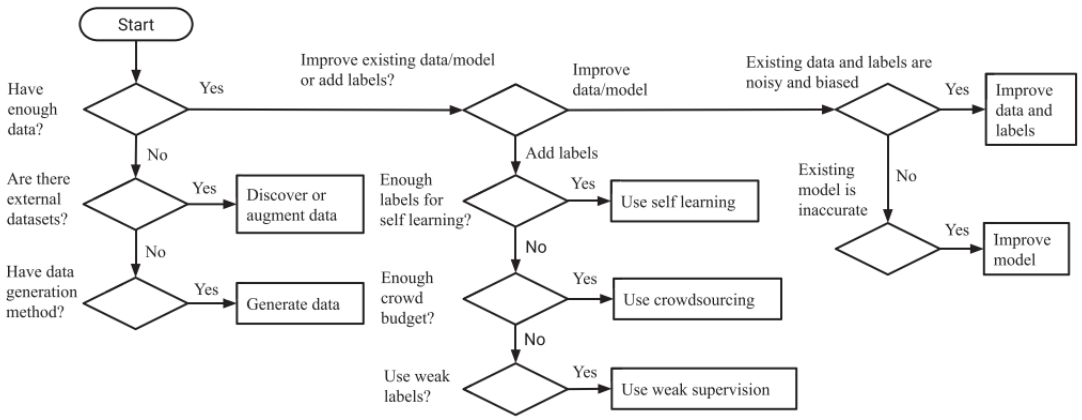



**Figure 3.2:** Decision flow chart for data collection [95]

Specifically, images are capturing a high-quality frame from the scene using a sensor (digital cameras, industrial vision systems, scanners, specialized imaging devices, etc.). The captured image is a digital numerical representation of that scene, and its accuracy depends on factors such as lighting, focal length, and quantum efficiency, etc. Proper lighting is crucial for high-quality image acquisition, as poor lighting can significantly degrade results. Image quality can also be affected by noise and camera distortions [96].

### 3.2.2. Data Augmentation

DL models heavily rely on large datasets to learn effectively and avoid overfitting, a condition where the network memorizes the training data instead of generalizing, resulting in high variance. If the dataset is not large enough, data augmentation is applied to artificially increase the diversity and size of the training dataset by applying transformations to input images [95] (Table 3.1).

**Table 3.1: Types of Image Augmentation** [97] [98]

| Augmentation Type | Description | Augmentation Methods |
|---|---|---|
| **Geometric Transformations** | Change the spatial arrangement of pixels without altering colour or texture | Rotation, Translation, Scaling, Flipping, Cropping, Shearing, Perspective warping |
| **Photometric / Colour Transformations** | Modify pixel intensity values while keeping spatial structure intact | Brightness, Contrast, Saturation adjustment, Hue shift, Colour jittering, random shadow |
| **Noise and Blur Injection & Filtering** | Add or remove random variations in pixel values | Gaussian noise, Salt-and-pepper noise, Gaussian and Motion blur, Sharpening filters |
| **Occlusion & Region Manipulation** | Hide or replace parts of the image to enhance feature learning | Cutout, Random Erasing, Hide-and-seek patches, Mix-up, CutMix |
| **Synthetic & Generative Augmentation** | Create new images from learned distributions or style transfers | GAN-generated images, Diffusion models, Neural Style Transfer, Inpainting, |
| **Image Quality & Compression** | Reduce image quality to mimic real-world low-quality inputs | JPEG compression artifacts, WebP compression, Bit-depth reduction |

### 3.2.3. Dataset Preparation

After being captured, the dataset is commonly divided into three subsets: training (70–80%), validation (10–15%), and test (10–15%). The training set is used to optimize model weights and biases, the validation set supports hyperparameter tuning and model selection, and the test set provides an unbiased evaluation of final performance [99].

After splitting, the data is preprocessed to ensure they meet the model's input specifications. Without the correct format, the model will not learn properly or may train on incorrect weights, leading to lower accuracy. This involves mapping the pixel intensity values into multi-dimensional tensors, where each channel corresponds to a specific color component (e.g., RGB, grayscale). DL models, such as CNNs or ViT, operate directly on these tensor representations, learning hierarchical feature maps from raw pixel data [100]. Table 3.2 lists the most common image preprocessing techniques.

**Table 3.2: Image Pre-processing technique for DL Inputs** [101]

| Pre-processing | Description |
|---|---|
| **Resizing** | Adjusts all images to a fixed resolution to match the model's input layer size, typically using interpolation methods |
| **Normalization** | Scales pixel intensity values to a defined range (0–1) or standardizes them to have a zero mean and unit variance, thereby improving gradient stability during training. |
| **Colour space conversion** | Transforms images between colour spaces (RGB, grayscale, etc) using defined mathematical transformations |
| **Channel reordering** | Rearranges image dimensions to match DL framework requirements |
| **Noise reduction** | Applies filtering techniques (Gaussian blur, median filtering, etc.) to suppress random variations caused by sensor noise or environmental |

### 3.2.4. Model Training Process

Once the dataset has been collected and preprocessed, the next step is training the DL model. This process involves the iterative adjustment of the model's weights and biases to minimize a predefined loss function, which measures the difference between predicted and actual values. The training loop for each batch involves four principal stages as shown in Figure 3.3. These steps are repeated iteratively across all batches and epochs until the model converges.

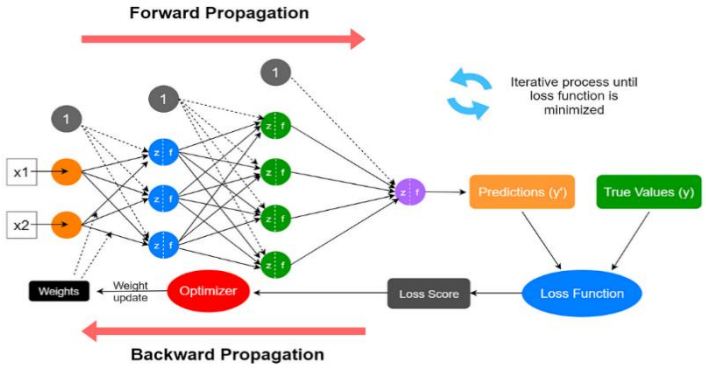



**Figure 3.3:** Forward and Backward Propagation in Neural Networks [102]

**a) Forward Propagation**

Forward propagation is the process of passing input data through the layers of a neural network to compute the predicted output. Each layer applies a linear transformation followed by a non-linear activation function, enabling to learn complex patterns [102], [103]. The common activation functions used in DL models are described in Table 3.3.

**Table 3.3:** Common Activation Functions [104]

| Activation Function | Description | Equation |
|---|---|---|
| **Sigmoid** | Map input to (0, 1), which is interpreted as probabilities; used in binary classification. | $\sigma(x) = \frac{1}{(1 + e^{-x})}$ |
| **Tanh** | Maps input to (−1, 1); zero-centered; useful for hidden layers | $\tanh(x) = \frac{(e^{x} - e^{-x})}{(e^{x} + e^{-x})}$ |
| **ReLU** | Outputs $x$ if positive, else 0; efficient and widely used in hidden layers | $ReLU(x) = max(0, x)$ |
| **GELU** | Smoothly gates inputs based on Gaussian probability; common in Transformer models | $GELU(x) = 0.5 * x * \left(1 + erf\left(\frac{x}{\sqrt{2}}\right)\right)$ |
| **Softmax** | Converts scores to probabilities summing to 1; used for multi-class outputs | $Softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}}$ |

### b) Loss Function

The loss function (cost function or objective function) measures the difference between the predicted output value and the actual value. It provides a quantitative signal that guides the optimization process during training by indicating how well or poorly the model is performing [103]. For classification tasks, common loss functions are listed in Table 3.4.

**Table 3.4:** Loss Function for Classification [103]

| Loss Function | Description | Equation |
|---|---|---|
| **Binary Cross-Entropy** | Measures the error between the predicted probability value and the actual binary value; used for binary classification | $L = -\big(y * log(p) + (1 - y) * log(1 - p)\big)$ |
| **Categorical Cross-Entropy** | Extension of binary cross-entropy for multi-class classification with one-hot encoded labels | $L = -sum_{i=1}^{K} y_i * log(p_i)$ |
| **Sparse Categorical Cross-Entropy** | Multi-class loss function using integer labels instead of one-hot encoding; more memory efficient. | $L = -log\big(p_y\big)$ |

### c) Backward Propagation

Backward propagation is the process of computing the gradients of the loss function with respect to all trainable parameters in a neural network by moving backward from the output layer to the input layer. Its goal is to reduce the difference between the model's predicted output and the actual output by adjusting the weights and biases in the network. It allows the optimizer to update these parameters in the direction that minimizes the loss [102].

### d) Optimisation

Parameter updates are performed by optimisation algorithms that use the computed gradients to minimise the loss function. The update is performed iteratively according to an update rule, which is influenced by hyperparameters such as the learning rate, momentum, or weight decay. The learning rate within this rule controls the magnitude of each update and plays a crucial role in convergence speed and stability [105]. Table 3.5 shows common optimisation algorithms used in DL, outlining their core principles and corresponding mathematical formulations.

**Table 3.5:** Optimization Algorithms in Deep Learning [106] [107] [105] [108]

| Optimizer | Description | Update Rule |
|---|---|---|
| **Stochastic Gradient Descent (SGD)** | Updates parameters in the direction opposite of the gradient of the loss function with respect to the parameters.<br><br>Uses only a subset (batch) of data to approximate the true gradient, improving computational efficiency. | $\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta_t)$ |
| **SGD with Momentum** | Extends SGD by accumulating a velocity vector to accelerate updates in consistent gradient directions, helping escape local minima and damp oscillations. | $v_t = \beta v_{t-1} + \eta \nabla_\theta J(\theta_t)$<br><br>$\theta_{t+1} = \theta_t - v_t$ |
| **AdaGrad** | Adapt and update the learning rate for each parameter based on the sum of historical squared gradients, allowing larger updates for non critical parameters and smaller updates for frequently. | $\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla_\theta J(\theta_t)$ |
| **RMSProp** | Similar to AdaGrad but uses an exponentially decaying average of squared gradients to prevent aggressive decay of learning rates. Effective for non-stationary objectives. | $g_t = \nabla_\theta J(\theta_t)$<br><br>$E[g^2]_t = \beta E[g^2]_{t-1} + (1-\beta)(\nabla_\theta J(\theta_t))^2$<br><br>$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \nabla_\theta J(\theta_t)$ |
| **Adam** | Combines momentum and RMSProp by maintaining both first (mean) and second (variance) moment estimates of gradients.<br><br>It adapts learning rates individually for each parameter. | $m_t = \beta_1 m_{t-1} + (1-\beta_1)(\nabla_\theta J(\theta_t))$<br><br>$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla_\theta J(\theta_t))^2$<br><br>$\widehat{m_t} = \frac{m_t}{1-\beta_1^t}, \quad \widehat{v_t} = \frac{v_t}{1-\beta_2^t}$<br><br>$\theta_{t+1} = \theta_t - \frac{\eta\, \widehat{m_t}}{\sqrt{\widehat{v_t}} + \epsilon}$ |
| **AdamW** | Variant of Adam that decouples weight decay from gradient-based parameter updates, improving generalization in many DL tasks. | $\theta_{t+1} = \theta_t - \eta \left( \frac{\widehat{m_t}}{\sqrt{\widehat{v_t}} + \epsilon} + \lambda \theta_t \right)$ |

### e) Iteration Across Batches and Epochs

Modern neural networks have millions of parameters, and training a model on the entire dataset at once is impractical. Instead, the data is divided into smaller subsets known as batches. Processing one batch and updating the parameters refers to one iteration, while an epoch is a complete pass through the entire training dataset. [109].

### f) Validation

Validation is performed after each epoch during model training, which evaluates performance (losses and accuracy) on a separate dataset not used for weight updates. It helps in tuning hyperparameters, early stopping to prevent overfitting and the best checkpoint [110].

### g) Fine-Tuning

Fine-tuning is the process of adapting a pre-trained model and weights to a new task on a customer dataset. Instead of training from scratch, the model starts with parameters learned from a large, generic dataset and is then updated using the customer dataset (Figure 3.4). This approach reduces training time, requires fewer data samples, and often achieves better generalisation [111]. The fine-tuning process typically involves [112]:

a) Pre-trained model that has been trained on a large, diverse dataset is loaded.

b) Initial layers, which capture basic features (e.g., edges, textures), are frozen, while later layers are updated to adjust new weights based on the new task.

c) The classification head is updated to the number of classes in the new dataset.

d) Fine-tuning often uses a smaller learning rate to avoid overwriting pre-trained knowledge while gradually adapting the model to the target domain.

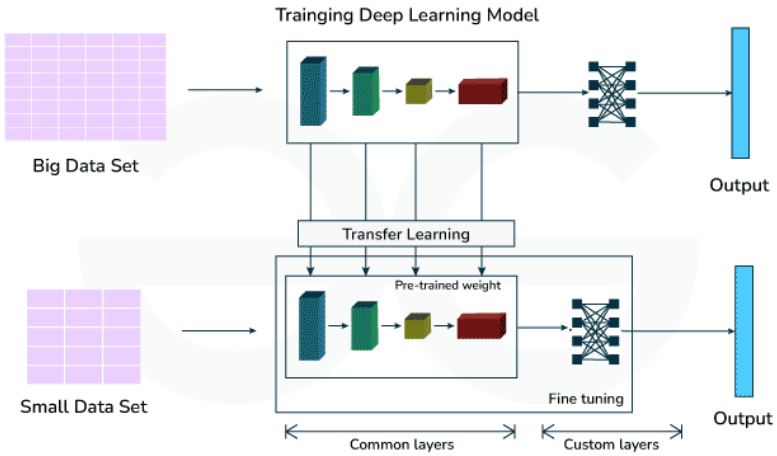



**Figure 3.4:** Fine-Tuning [113]

## 3.3. Evaluation DL Model

Evaluating the performance of a DL model involves three key steps: selecting appropriate metrics, analyzing training dynamics, and validating real-world applicability. These aspects are discussed in the following sections.

### 3.3.1. Monitoring Training Dynamics

Model training is monitored per epoch using training and validation loss, along with accuracy, to assess learning behavior [114]. Training loss indicates how well the model is learning the training data, while validation loss shows how well it generalizes to unseen data. Three common patterns are: a) Good convergence (both losses/accuracy decrease and remain close, showing good generalization). b) Overfitting (training loss decreases but validation loss rises, indicating the model is memorizing data, instead of generalizing), c) Underfitting (both losses remain high, showing poor learning)

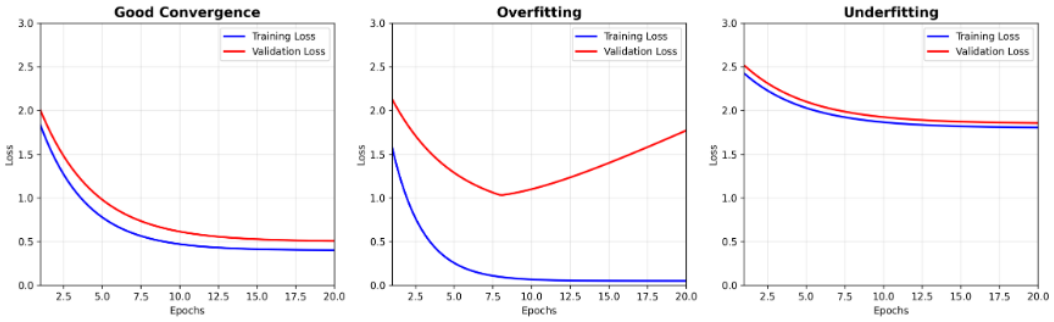



**Figure 3.5:** Different training and validation loss behaviors [114]

### 3.3.2. Performance Metrics

The performance of DL models is evaluated using the following [114] and Table 3.6;

a) A confusion matrix is a simple table used to represent the predicted results of a classification task. It shows the predictions made by the model by showing counts of true positives, true negatives, false positives, and false negatives where true if the predict the truth, false if it doesn't; positive means the model predicted the target class, and negative means the model predicted the non-target class.

b) Inference time indicates how fast a model can make predictions on new data. It is an important factor for real-time or industrial applications. A model with high accuracy but slow inference may not be suitable for time-sensitive environments.

c) t-SNE is used to visualize high-dimensional feature vectors in two or three dimensions. It helps analyze how well the model separates different classes in the feature space, especially useful in evaluating learned representations in self-supervised or embedding-based models [115].

**Table 3.6:** Evaluation Metrics: Precision, Recall, F1-Score, and Accuracy

| Metrics | Detail | Equation |
|---|---|---|
| Precision | Precision indicates how well the model has predicted favorable outcomes, and it is the ratio of correctly predicted positives to the total predicted positives | $\frac{TP}{TP + FP}$ |
| Recall | it is the ratio of correctly predicted positives to all actual positives. It shows how well the model recognize good examples. | $\frac{TP}{TP + FN}$ |
| F1- Score | It is a measure that combines recall and precision. It shows a well-rounded assessment of the model's accuracy, especially when dealing with imbalanced datasets where one class has more instances than the other. | $2 * \frac{Precision * Recall}{Precision + Recall}$ |
| Accuracy | It measures the percentage of correct predictions out of the total predictions | $\frac{TP + TN}{TP + TN + FP + R + FN}$ |

## 3.4. Network Topologies in DL for Image Classification

The following sections provide an overview of the main DL topologies applied to image classification, covering CNN, ViT, and SSL methods.

### 3.4.1. Convolutional Neural Network

CNNs are utilized in various CV tasks, including image classification, object detection, and facial recognition. CNNs are composed of multiple layers, including an input layer, several convolutional and pooling (hidden) layers, and a fully connected output layer, as shown in Figure 3.6. The network updates optimal weights and biases during training to extract features from the input data and make predictions [93].

The convolution layer is a mathematical procedure to extract the features. Sliding the filters over the input data and calculating the dot product between the filter and the input's overlapping region at each point . The convolutional layer can extract a wide

variety of features from the input by applying many filters in parallel, which hels the network build hierarchical representations of the data. Additionally, non-linearity is applied in the layer through activation functions.

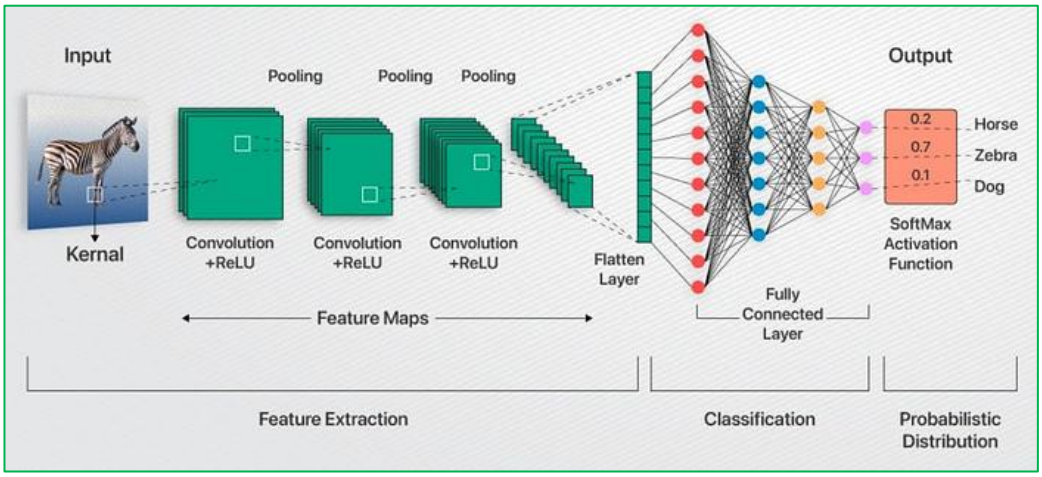



**Figure 3.6:** Convolutional Neural Network [116]

Furthermore, CNNs often include additional layers within the convolutional layer, such as batch normalization layers that normalize the activations to enhance training stability and convergence, and pooling layers that reduce the feature maps to decrease computational effort and enhance translation consistency [117].

Mathematically, the convolution for one output feature map is given by:

$$H_q = f(\sum_{p=1}^{P} X_p * W_{q.p} + b_q) \tag{3.1}$$

Max Pooling is used in CNNs to reduce the spatial dimensions (width and height) of the input feature maps while retaining the most important information. It involves sliding a two-dimensional filter over each channel of a feature map and summarizing the features within the region covered by the filter. For each feature, a max-pooling layer takes the maximum value of a feature for each subregion of the image (generally 2x2). Althrough average-pooling is possible, max pooling is more common, especially not in early stages. Pooling allows translation invariance: the same input pattern will be detected, whatever its position in the input image[118].

CNNs use fully connected layers, placed at the end of a network to perform regression, classification, or other tasks from extracted features. Each neuron in the fully connected layer connects to every neuron in the previous layer, creating a dense network to learn complex nonlinear  between features relationships. [119].

### 3.4.2. Transformers and Vision Transformers

The original transformers [120] are a type of DL architecture for natural language processing tasks. They are based on an attention mechanism to determine the importance of different parts of the input to make predictions. This design allows them to process all elements of the sequence in parallel, making them highly efficient.

Transformers are composed of an input layer, a series of encoder and decoder blocks, and an output layer, as shown in Figure A.6. Each block contains a multi-head self-attention mechanism, which enables the model to focus on different parts of the input simultaneously, and a feed-forward network that processes the attended information. Residual connections and normalization layers are also included to improve stability and training efficiency. A linear and a softmax layer are added to the final decoder output. The detailed architecture of the transformer is shown in Appendix A.6.

ViT [121] uses the same architecture for image classification by treating an image as a sequence of patches. This allows the model to capture global context across the entire image, unlike CNNs, which operate on local receptive fields. As shown in Figure 3.7, ViT includes patch embedding, positional embedding, transformer encoder blocks, and a classification head to generate final predictions.

Patch embedding converts an image into a sequence of tokens suitable for transformer processing. The image is divided into fixed-size non-overlapping patches, each patch is flattened into a vector, and then mapped to a fixed-dimensional embedding space through a trainable linear projection.

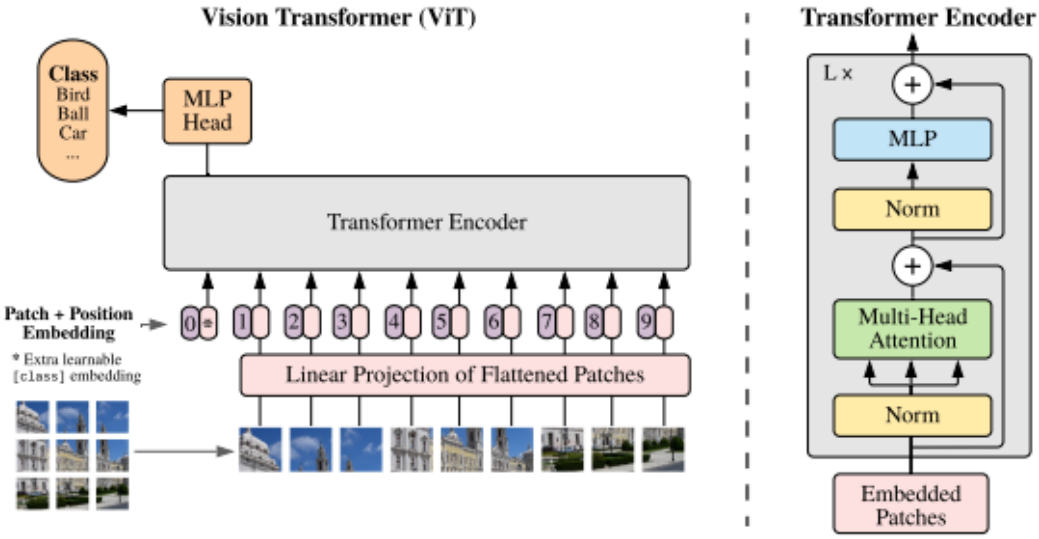



**Figure 3.7:** Vision Transformer architecture [121]

Transformers lack a sense of spatial order; positional encoding is incorporated to patch embeddings to provide information about their relative positions of patches in the image. These encodings can be fixed, using predefined sinusoidal patterns, or learned as trainable parameters. This encoding allows the model to determine the relative and absolute positions of image patches.

The encoder processes the embedded patch (token) sequence through a series of identical layers, each containing multi-head self-attention and a feed-forward network, connected via residual connections and layer normalization.

Self-attention allows each patch token to incorporate information from all other patches in the image. Each token is transformed into three vectors: Query $(Q)$, Key $(K)$, and Value $(Q)$ through learned linear projections given as;

$$Attention\ (Q, K, V)\ = softmax\ (\frac{QK^T}{\sqrt{d_k}})V \tag{3.2}$$

where $d_k$ is the dimension of the key vectors. The softmax ensures that attention weights sum to one, providing a normalized measure of importance for each token. Instead of performing attention once, ViT use a number of attention heads in parallel. Each head learns to focus on different aspects of the input.

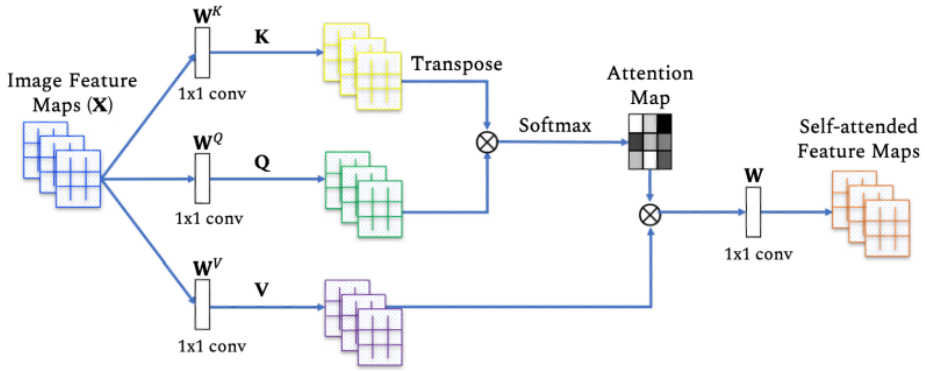



**Figure 3.8:** Self Attention Mechanism [122]

After attention, each token is processed independently by a position-wise feed-forward network using an activation function. It applies the same linear transformation with the same weights to each element in the sequence. This increases the model's capacity to learn complex transformations of the token embeddings.

Residual connections and layer normalization are included in each encoder layer to enhance stability and learning efficiency. The residual connections allow the original

token representations to bypass the sublayer, ensuring important information is preserved and gradients can flow effectively through deep networks.

Layer normalization standardizes the token features before each sublayer, keeping their scale consistent and preventing training instability. Together, these operations help the encoder maintain information across layers, support faster convergence, and improve overall model performance.

The classification head converts the high-dimensional feature representation extracted by the transformer into a format suitable for classification. After the final encoder layer, the embedding corresponding to the classification token is used as the image representation. This is passed through a multilayer perceptron head (typically one or two linear layers) to map the embedding to the number of output classes.

$$y = x_{class} W_{class} + b \tag{3.3}$$

During inference, a softmax activation is applied to produce class probabilities

### 3.4.3. Self-Supervised Learning

While CNNs and ViTs represent the main architectural structures in DL, SSL is a training approach that enables these networks to learn feature representations from unlabeled data. SSL creates supervision directly from the input data through pretext tasks. Common methods of SSL are

a) Contrastive learning methods such as SimCLR , where the model learns to bring augmented views of the same image closer while pushing apart views of different images [123].
b) Teacher–student methods such as DINO , where a student network matches the representation of a teacher network across augmented inputs [124].

These methods are applied on CNN or transformer backbones and have shown strong results in image classification.

## 3.5. Foundational DL Models for Classification

DL-based image classification has been developed using several well-established architectures that extend the general principles of CNNs, ViTs, and SSL into practical model designs. These models represent the state-of-the-art for CV tasks (section 2.4). The following sections provide an overview of key models and highlight their design concepts.

### 3.5.1. ResNet (Residual Networks)

Initial CNN-based models have a small number of layers; however, when adding more layers, vanishing gradient arises, a typical DL issue. This results in the gradient becomes either zero or overly large. Therefore, the training and test error percentage increase as the number of layers increases [125]. Figure 3.9 showes that a 20-layer CNN architecture performs better on training and testing datasets than a 56-layer CNN architecture. ResNet [126] enabling the training of extremely deep architectures without the vanishing problem that occurs when using many layers. The core concept in ResNet is residual learning, implemented through residual blocks. This approach allows the network to focus on learning the residual mapping rather than the full transformation, simplifying optimization and improving accuracy in deeper models.

The principle of residual learning assumes that directly fitting a desired mapping is more challenging than fitting the residual. ResNet formulates the mapping as;

$$y = F(x) + x \tag{3.4}$$

where $F(x)$ is the learned residual function and $x$ is the identity mapping from the input block. The operation "$+ x$" is implemented by a skip connection that performs an identity mapping to connect the input of the subnetwork with its output.

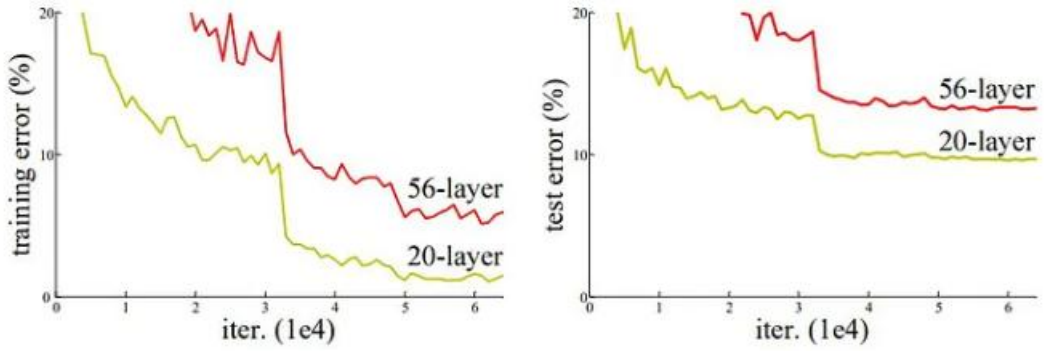



**Figure 3.9:** Comparison of 20-layer vs 56-layer architecture [126]

This connection is known as a residual connection, as shown in Figure 3.10. This addition enables gradients to propagate more effectively through deep networks, preventing vanishing gradients and allowing stable training of very deep architectures.

**ResNet Architecture**

The ResNet architecture consists of three main parts: an initial feature extraction layer, followed by several stages of residual blocks, and a final classification layer [125]. The detailed architecture is shown in Appendix A.1.

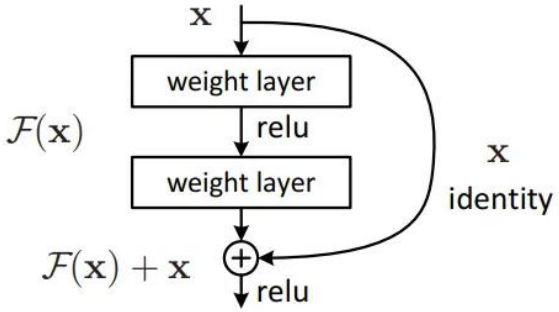



**Figure 3.10:** Residual learning block [126]

I. The network starts with a large 7 x 7 convolution layer containing 64 filters and a stride of 2, followed by batch normalization and a ReLU activation function that processes the input image and captures basic patterns (edge and textures etc) . This is followed by a pooling layer that reduces the image size and keeps the most important information.

II. After the initial layer, the network consists of multiple residual blocks. Each block has convolution layers, batch normalization, and ReLU activation, along with a skip connection that adds the input to its output. This helps the network learn more effectively and allows very deep models to be trained. There are two main types of residual blocks, either Basic Block or Bottleneck Block, each with specific convolution layers and functions for learning features, as shown in Figure 3.11.

III. At the end of the network, a Global Average Pooling (GAP) layer reduces each feature map to a single value. These values are then passed into a fully connected layer, which produces the final class scores.

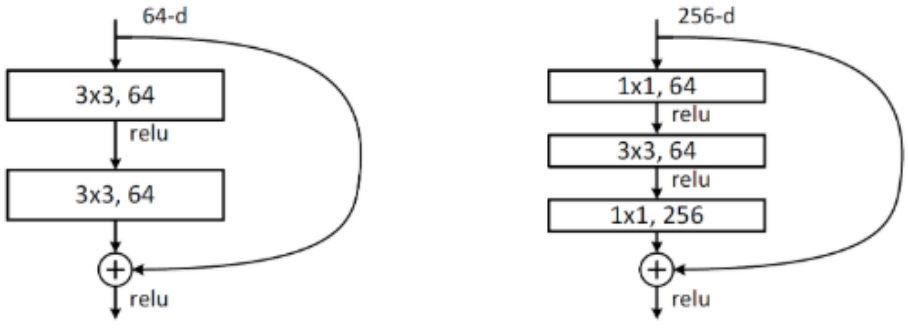



**Figure 3.11:** Basic Block and Bottleneck Block in ResNet [127]

**ResNet Versions**

ResNet models mainly differ in how deep they are and how many residual blocks they have in each stage. ResNet-18 and ResNet-34 use the Basic Block, while ResNet-50,

ResNet-101, and ResNet-152 use the Bottleneck Block. They all have the same type of first convolution layer, pooling, and final global average pooling with a fully connected layer. However, the number of residual blocks is different for each model, which lets them learn more detailed and complex features. The deeper the model, the higher the number of parameters it will learn during training [125]. The ResNet variants' detail of blocks is shown in Appendix B.1.

### 3.5.2. EfficientNet

EfficientNet[128] is a DL model developed to achieve higher accuracy while requiring fewer parameters and reduced computational resources compared to other models. EfficientNet models are based on compound scaling method that systematically and proportionally scales the network's depth, width, and input resolution. This allows the architecture to achieve better accuracy and efficiency than traditional scaling strategies, which often modify only one of these dimensions.

High-resolution images require deeper networks to capture large-scale features with more pixels. Additionally, wider networks are needed to capture the finer details present in these high-resolution images. The compound scaling method modifies a CNN by increasing the dimensions of the network depth, width, and input resolution.

Figure 3.12 shows (a) is a baseline network example; (b) to (d) are conventional scaling methods to increase one dimension of network width, depth, or resolution; (e) is the compound scaling method that uniformly scales all three dimensions with a fixed ratio, explained as below:

a) Depth scaling increases the number of layers, which allows the model to capture more complex patterns and features and generalize better. This enhances the network's capacity to model intricate patterns, but must be balanced to avoid unnecessary computational and the vanishing gradient problem
b) Width increases the number of channels in each convolutional layer, enabling the network to process more fine-grained features at each stage. However, extra-wide models are unable to capture higher-level features.
c) Resolution scaling increases the input image resolution, providing more detailed information for identifying fine-grained patterns. However, higher resolution increases the computational requirements.
d) The compound scaling coefficient method uniformly scales all three dimensions (depth, width, and resolution) in a proportional manner using a predefined compound coefficient $\phi$.

$$\text{depth} = \alpha^{\phi}, \quad \text{width} = \beta^{\phi}, \quad \text{resolution} = \gamma^{\phi} \tag{3.5}$$

$$\alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$$

$$\alpha \geq 1, \quad \beta \geq 1, \quad \gamma \geq 1$$

Here, $\alpha$, $\beta$, and $\gamma$ are constants determined through an empirical grid search or optimization process [128].

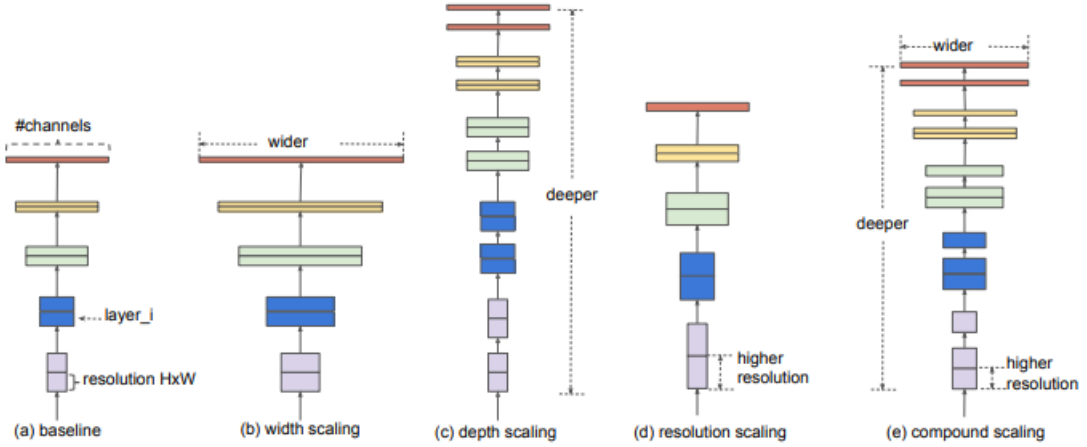



**Figure 3.12:** Model Scaling methods [128]

**EfficientNet Architecture**

The EfficientNet architecture consists of three main parts: an initial feature extraction layer, a series of MBConv (Mobile Inverted Bottleneck Convolution) stages, and a final classification layer [129]. The detailed architecture is shown in Appendix A.2.

The network begins with a 3×3 convolution layer with 32 filters and a stride of 2, followed by batch normalization and the activation function. This layer processes the input image and captures low-level features such as edges, corners, and basic textures. The resulting feature maps are passed to the first MBConv stage for further processing.

After the initial convolution layer, the network is composed of multiple MBConv blocks as shown in Figure A.2. The MBConv block is an evolved inverted residual block. Each MBConv block contains multiple layers.

- The first is the expansion layer, which is a 1×1 convolution that increases the number of channels to expand the feature representation, followed by batch normalization and a non-linear activation function.
- The second is the depthwise convolution, which is a 3×3 convolution applied to each channel, followed by batch normalization and a non-linear activation to

extract spatial features while reducing computational cost compared to standard convolutions.

- A Squeeze-and-Excitation (SE) block, which uses global average pooling to reduce the spatial dimensions of the feature map to a single channel, followed by two fully connected layers, is added after depthwise convolution (Figure A.2) .
- The last layer is the projection layer, which is a 1×1 convolution that reduces the number of channels back to the original desired output size. Batch normalization is applied, but no activation is used here.
- Skip connections are used when the input and output dimensions are the same, allowing information to bypass certain layers and improving gradient flow.

After the last MBConv stage, a convolution layer is applied to further refine the features. This is followed by a Global Average Pooling layer, which reduces each feature map to a single value. A dropout layer is applied for regularisation, and finally, a fully connected layer produces the class scores for classification.

**EfficientNet Versions**

The EfficientNet family includes eight primary models, named EfficientNetB0 through EfficientNetB7. These variants are based on a single core network architecture (B0), developed using the compound scaling method, which proportionally increases network depth, width, and input resolution in a balanced way. Appendix B.2 shows EfficientNet model versions, input resolution, parameters, and Top-1 accuracy (conventional accuracy, which measures the proportion of classes in the dataset where the model's single highest-probability prediction exactly matches the expected target label) and Floating Point Operation (FLOP) on ImageNet [130].

### 3.5.3. DeiT (Data-Efficient Image Transformer)

ViT showed good results for image classification tasks; however, it required massive labeled datasets for effective training (like Google's JFT-300M with 300 million images [131]). DeiT [132] enabling ViT based models to be trained effectively on small datasets by introducing a distillation mechanism where a teacher-student learning method incorporates a distillation token, which helps the model learn from a teacher network.

In DeiT, knowledge distillation is integrated directly into the transformer architecture through an additional distillation token. This token is processed in parallel with the standard classification token and interacts with all image patch tokens via the self-attention mechanism. At the output, two separate classification heads are used: one for the classification token and one for the distillation token. This setup enables the model

to learn both from the ground-truth labels and from a teacher model's predictions. The primary purpose of this distillation is to transfer the biases and efficient feature extraction capabilities of a high-performing CNN teacher to the transformer student. Two types of distillation are used, explained below[133];

a) Soft distillation uses the probability distribution of the teacher model as a training target for the distillation token. Instead of matching only the correct class, the model learns to match the entire output distribution from the teacher, which contains information about class similarities. It minimizes the Kullback-Leibler divergence [134] between the softmax of the student and the softmax of the teacher model. Let $Z_s$ the logits of the student model, $Z_t$ be the logits of the teacher model, $\tau$ the temperature for the distillation, λ the coefficient balancing the Kullback–Leibler divergence loss (KL), and the cross-entropy loss ($L_{CE}$) on ground truth labels (y), and ψ the softmax function. The distillation objective is:

$$L_{global} = (1-\lambda)\, L_{CE}(\psi(Z_s), y) + \lambda\, \tau^2\, \mathrm{KL}!\left(\psi!\left(\frac{Z_s}{\tau}\right),\ \psi!\left(\frac{Z_t}{\tau}\right)\right) \tag{3.6}$$

b) Hard-label distillation takes a simpler approach; it uses the teacher's top-1 predicted class as an additional training target for the student. The loss equally weights the standard ground-truth prediction and the teacher's label. Let $y_t$ be the hard decision of the teacher, the objective associated with this hard-label distillation is:

$$L_{HardDistillglobal} = \frac{1}{2}\, L_{CE}(\psi(Z_s), y)\ + \frac{1}{2}\, CE(\psi(Z_s), y_t) \tag{3.7}$$

**DeiT Architecture**

The backbone of DeiT is a vision transformer encoder, which processes image information through a sequence of self-attention and feed-forward layers. However, DeiT incorporates a distillation token in addition to the standard classification token, enabling it to simultaneously learn from dataset labels and from the knowledge of a pre-trained teacher model. The overall architecture can be divided into four main components [135]. The detailed architecture is shown in Appendix A.3.

The input image is divided into non-overlapping patches, each flattened and projected into a fixed-dimensional embedding same as in ViT. Two learnable tokens, the classification token and the distillation token, are add along with the patch sequence. Positional embeddings are added to all tokens to maintain spatial information. The classification token is used for supervised classification with dataset labels, while the

distillation token is for learning from the teacher model's predictions. Positional embeddings are added to all tokens to preserve spatial relationships before they are fed into the transformer backbone.

I. All the tokens (patches, classification, and distillation) are processed by transformer encoder layers, which are the same as the ViT encoder (explained in section 3.4.2). Both tokens interact with all patch tokens through MHSA, allowing them to gather information from the entire image.
II. A high-performing CNN-based teacher model processes the same input image in parallel. The teacher produces logits that serve as targets for the distillation token output. This supervision is implemented through either soft distillation (matching the teacher's probability distribution) or hard-label distillation (matching the teacher's predicted class). The teacher model is fixed during training to transfer knowledge consistently to the student transformer.
III. After the final encoder layer, the classification token is passed through a classification head to produce predictions based on the dataset labels. Meanwhile, the token is also passed through a separate head to produce outputs supervised by the teacher model's predictions. This dual-head output structure optimizes classification accuracy and teacher-guided knowledge distillation.

**DeiT Version**

The DeiT family includes multiple model sizes that vary in depth, embedding dimension, and number of attention heads are illutrated in Appendix B.3.

### 3.5.4. DINO (Self-Distillation with No Labels)

DINO [124] is a self-supervised learning method that uses ViT or CNNs as a backbone to learn without labeled data, based on a self-distillation architecture. A major challenge in self-supervised learning is collapse, where the network produces constant or same outputs for all inputs. DINO addresses this by introducing a self-distillation framework in which the student is trained to predict the output distribution of the teacher using different augmented views of the same image.

Self-distillation is based on a teacher–student framework, where both networks share the same architecture: ViT or CNN as backbone and a projection head; however, they have different parameters [124].

I. Student Network is updated using standard gradient backpropagation. It receives as input both global views (large crops) and local views (small crops) of

the same image. By predicting the teacher's outputs from these diverse views, the student learns to capture global semantic information as well as part-to-whole relationships.

II. Teacher network processes only the global views of the input image. The teacher parameters $\theta_t$ are updated as an exponential moving average (EMA) of the student parameters $\theta_s$:

$$\theta_t \leftarrow m \cdot \theta_t + (1 - m) \cdot \theta_s \tag{3.8}$$

Where $m$ is the momentum coefficient, progressively increased during training. This EMA update and integrating knowledge from past student states:

$$\theta_t = \sum_{k=0}^{t} (1 - m) \cdot m^k \cdot \theta_{s,t-k} \tag{3.9}$$

Additionally, to prevent collapse and maintain stability, DINO combines centering and sharpening operations on the teacher outputs [136]. A center vector $c$ is also maintained for the teacher outputs to prevent dimensional collapse:

$$c \leftarrow m_c \cdot c + (1 - m_c) \cdot \frac{1}{B} \sum_{i=1}^{B} g\theta_t\,(x_i) \tag{3.10}$$

where $m_c$ is the center momentum, $B$ is the batch size, and $y_{t,i}$ is the teacher's output for the $i$-th sample. This normalization ensures that the teacher outputs remain balanced across dimensions and do not collapse. Furthermore, sharpening is applied by using a lower temperature parameter for the teacher compared to the student to make the distribution more peaked. The probability distributions are then computed as:

$$P_s = \psi\left(\frac{Z_s}{\tau_s}\right), \quad P_t = \psi\left(\frac{Z_t - c}{\tau_t}\right) \tag{3.11}$$

where $Z_s$ and $Z_t$ are the logits of the student and teacher networks. Sharpening makes the teacher's outputs more peaked and discriminative, providing stronger supervision to the student. The DINO objective is formulated as the cross-entropy loss between the teacher and student probability distributions [137]:

$$\mathcal{L}_{\mathrm{DN}} = -\sum_{i=1}^{C} P_t(i) \log P_s\,(i) \tag{3.12}$$

The complete loss function incorporates multiple views of the same image is given as:

$$\mathcal{L}_{\mathrm{DN}} = \sum_{x \in \{x_1^g, x_2^g\}} \sum_{\substack{x' \in \mathcal{V} \\ x' \neq x}} H\big(P_t(x), P_s(x')\big) \tag{3.13}$$

Where $\mathcal{V} = \{x_1^g, x_2^g, x_1^l, \dots, x_N^l\}$, $g$ denoates global views and $l$ denotes local views

**DINO Architecture**

The architecture can be divided into the following main components [124]. The detailed architecture is shown in Appendix A.8.

I. In DINO, each training image is augmented into multiple crops. Two large global crops and several smaller local crops are generated using strong data augmentations. These crops are then converted into non overlapping patches, linearly projected into embeddings, and combined with a learnable classification token. Positional embeddings are added to maintain spatial structure.
II. The augmented crops are passed through a backbone. This backbone can be either ViTs or CNNs. The backbone is implemented within a MultiCropWrapper, which ensures that both global and local crops are processed consistently. This wrapper allows multiple views of the same image to be forwarded in a single pass, an essential element of DINO's efficiency.
III. The output of the backbone is fed into a projection head (DINO head). This head is a multi-layer perceptron with multiple layers, batch normalization, and weight normalization on the last layer. It projects the high-dimensional transformer features into a lower-dimensional space where self-distillation is applied.
IV. The teacher and student heads produce probability distributions via a softmax function with temperature scaling. The teacher's output is additionally centered and sharpened before being used as the training target for the student. The resulting features are aligned across global-to-global and local-to-global view pairs through the distillation loss.

### 3.5.5. YOLO (You Only Look Once)

YOLO [138] real-time object detection algorithm in computer vision is renowned for its simplicity and speed. These models predict bounding boxes and class probabilities directly from full images instead of dividing the image into regions and processing them separately. It tackles object detection or classification as a regression-based problem.

YOLO divides the image into an SxS grid of cells. Each grid cell predicts a single object, with the corresponding class probabilities. The division is based on the spatial layout of

the image, meaning that each grid cell represents a specific region of the image. Each bounding box is represented by a set of elements: (x, y, w, h), where (x, y) are the coordinates of the bounding box's in center point relative to the grid cell, and(w, h) represent the width and height of the bounding box relative to the whole image. For every bounding box, YOLO furthermore forecasts a confidence score that represents the model's level of assurance that the box includes an object and its degree of localization accuracy. The confidence score indicates the probability (P) that an object is inside the box as well as the precision of the box's location.

$$\text{Confidence score} = \text{P(Object)} * \text{IoU}_{\text{Pred,truth}} \tag{3.14}$$

Where $\text{IoU}_{\text{Pred,truth}}$ (Intersection over Union) measures the overlap between the predicted bounding box and the ground truth box. Ranges from 0 (no overlap) to 1 (perfect overlap). If a bounding box has a high IoU with the ground truth, it is considered a good prediction.

Additionally, in the YOLO model, each box predicts the likelihood of various object classes, assuming an object is present. These class probabilities are then combined with the bounding box confidence to determine the most likely object type within each detected region.

$$Class\ Score_i = \text{P}(Class_i) * \text{IoU}_{\text{Pred,truth}} * \text{P(Object)} \tag{3.15}$$

YOLO's final output is represented as a matrix of a specific shape, given as below,

$$S * S * (B * 5 + C) \tag{3.16}$$

where: S is the grid size. The estimated number of bounding boxes for each grid cell is B. C is the number of classes. Each grid cell predicts bounding boxes, along with their confidence scores and class probabilities as shown in Figure 3.13.

After the acquisition of the bounding boxes, confidence intervals, and class probabilities. YOLO applies non-maximum suppression (NMS) to remove duplicate detection of the same object. NMS suppresses overlapping bounding boxes by selecting the one with the highest confidence score and discarding others that have significant overlap with it.

YOLO is trained on labeled datasets using a loss function that uses sum-squared error between the predictions and the ground truth to calculate loss. It combines localization loss, confidence loss, and classification loss [139]. The classification loss

( $L_{classification}(\theta)$ ) is used to ensure each cell predicts the correct class and is the MSE between the ground truth($\hat{p}_i(c)$) and prediction($p_i(c)$).

$$L_{classification}(\theta) = \sum_{i=0}^{S^2} 1_i^{obj} \sum_{c=1}^{C} \left(p_i(c) - \hat{p}_i(c)\right)^2 \tag{3.17}$$

The localization loss ($L_{localization}(\theta)$) minimizes the MSE between the coordinates of the ground truth bounding box( $\hat{x}_i, \hat{y}_i, \hat{w}_i, \hat{h}_i$) and the predicted bounding box ($x_i, y_i, w_i, h_i$ ).

$$\begin{aligned} L_{localization}(\theta) = & \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] \\ & + \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} \left[ \left(\sqrt{w_i} - \sqrt{\hat{w}_i}\right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i}\right)^2 \right] \end{aligned} \tag{3.18}$$

Confidence loss minimizing the MSE between the predicted confidence score of the cell $C_i$, and the IoU between the ground truth bounding box and the predicted one $\hat{C}_i$.

$$L_{confidence}(\theta) = \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} \left(C_{ij} - \hat{C}_{ij}\right)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{noobj} \left(C_{ij} - \hat{C}_{ij}\right)^2 \tag{3.19}$$

Finally, the loss function is optimized using backpropagation and SGD.

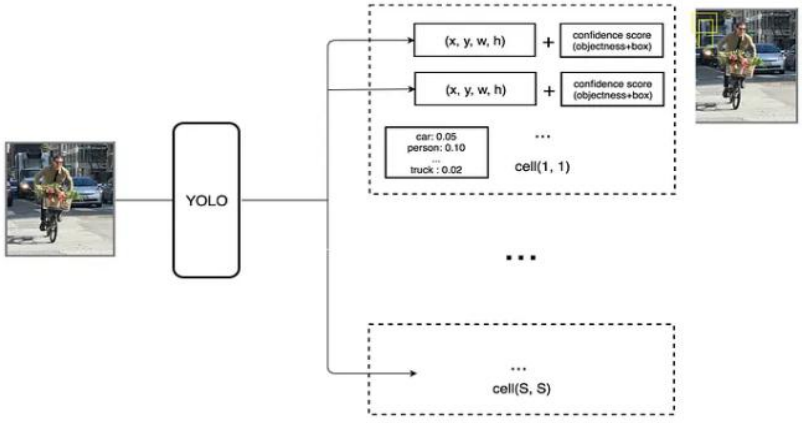



**Figure 3.13:** YOLO working explained [139]

**Yolo Architectural**

YOLO models' algorithm has backbone, neck, and head, three main components. The backbone is a CNN that has been pre-trained on a large-scale image classification dataset, responsible for feature extraction from the input image. The neck refines and combines these features, and the head makes the final predictions, including bounding boxes, objectness scores, and class probabilities[140].

YOLOv11 architecture consists of three core components: Backbone, Neck, and Head, is the latest model in YOLO variants [141]. This network includes C3k2 (Cross Stage Partial with kernel size 2) block, SPPF (Spatial Pyramid Pooling - Fast), and C2PSA (Convolutional block with Parallel Spatial Attention), along with standard convolutional blocks. The architectures of these additional blocks are shown in Figure A.5. The detailed architecture is shown in Appendix A.4

The backbone of YOLOv11 acts as the main feature extractor, capturing both low and high level semantic information from input images. It includes convolutional layers, SPPF, and C2PSA Blocks and C3K2 blocks for efficient convolutional downsampling. This improves computational efficiency by using two smaller convolutional layers instead of one large layer, as in YOLOv8[142], enabling faster processing while preserving representational quality. The SPPF module effectively captures features across various object scales of images. Additionally, the C2PSA block is added after the SPPF. This block introduces spatial attention mechanisms, allowing the network to focus more precisely on important regions within the image and thereby enhancing detection accuracy, especially for small or occluded objects. [143].

The neck is responsible for merging features from different scales and passing them to the head for final prediction. This is typically achieved through the upsampling and concatenation of feature maps from multiple layers, enabling the model to effectively capture multi-scale contextual information. It has C3K2 block to improve processing speed and C2PSA module to increase spatial attention mechanism[144].

The head of YOLOv11 is responsible for generating the final predictions in terms of object detection and classification. It processes the feature maps passed from the neck, ultimately outputting bounding boxes and class labels for objects within the image. The C3K2 blocks are integrated into multiple processing pathways within the head to efficiently refine feature maps at various depths. To further enhance feature extraction, Convolution-Batch-Normalization blocks are incorporated following the C3K2 units. These CBS blocks contribute to high-precision detection by extracting relevant features, stabilizing the learning process through batch normalization, and introducing non-linearity via the SiLU activation function. This combination improves overall model convergence and accuracy, particularly in complex visual environments [145].

Appendix B.4 shows different YOLOv11 classification models scales (nano, small, medium, large, and extra-large) to balance speed, memory usage, and accuracy on ImageNet [72].

# Chapter 4

# Methodology

This chapter describes the methodology adopted to develop AI-based classification for identifying burns on used PCBs as part of a circular manufacturing process to support sustainable practices. The overall flow is described in the flow Chart below,

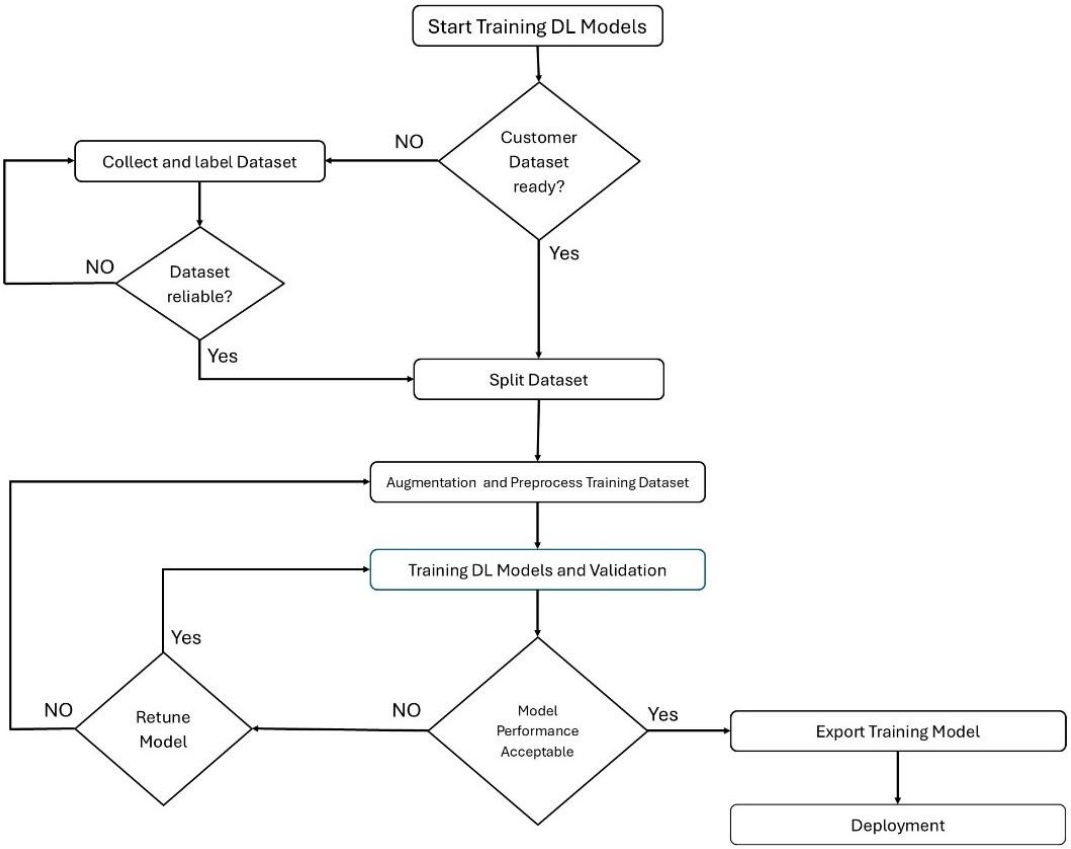



**Figure 4.1:** Methodology flow chart

## 4.1. Data Acquisition

To train the AI-based classification model, a dataset for each classification is required, with the same number of images per class, to prevent the model from being biased towards the major class [146]. The acquisition process focused on ensuring diversity of samples across all classes. Table 4.1 summarizes the overall distribution of images collected from different sources before preprocessing.

**Table 4.1:** Raw Collected Dataset Statistics by Source

| Dataset Source Type | Specific Source | Burnt_PCB | Good_PCB |
|---|---|---|---|
| Existing Datasets | Literature | 0 | 373 |
| | RoboFlow | 772 | 165 |
| Websearch Images | Google Images | 110 | 30 |
| | Flickr | 0 | 145 |
| AI-generated Dataset | ChatGPT | 30 | 10 |
| | Google Gemini | 30 | 25 |
| Self-Captured | Original | 9 | 4 |
| **Total Images** | | **951** | **752** |

The detailed description of each category is presented in the following subsections.

### 4.1.1. Existing DataSets

First, as described in the literature review, good PCB datasets are collected from published research articles during the review phase. The majority of publicly available datasets described in the section 2.3 are of defective PCBs. PCB-DSLR, MPI-PCB, FICS_PCB, and PCB-METAL datasets are initially considered, which contain PCB without defects. Images in these datasets represent samples of non-defective PCB boards that have previously been used for training AI models.

However, there are no datasets available in any publication containing Burnt PCB images. To address this gap, several online platforms containing different types of datasets for data science and ML are reviewed. However, there are no such PCBs datasets available containing burnt PCBs images. For example, Ultralytics [146] provides various datasets to facilitate CV tasks. Eleven various datasets are available for classification; however, they do not contain any burnt_PCB.

Roboflow [147] is an open-source online platform that hosts various image datasets for DL tasks. Seven different PCB-related datasets are available on Roboflow, containing a total of 772 images of only burnt_PCBs. However, these datasets have duplicate images or augmented images of the same PCB. As part of the preprocessing step (section 6.2 Data Processing), such images are removed, resulting in a final subset of 251 unique burnt PCB images used for training. In addition, one dataset comprising 165 Good PCB images is also acquired from Roboflow.

### 4.1.2. WebSearch datasets

To further expand the dataset, in addition to the literature review, good PCB images are downloaded from web-based platforms, including Google Images and Flickr [148], using relevant search queries. Flickr is reliable for DL models' image collection used in literaurre [91] . Images with high visual quality and clear labels are prioritized.

Burnt PCB images are also downloaded from Google images search to compensate for the low data set availability by searching “Burnt PCB” and visually inspecting each image. However, despite these efforts, almost all burnt PCB images could only be sourced from Google Images, resulting in a total of 110 images, which is insufficient to train a model.

### 4.1.3. AI-Generated Datasets

Although real images are collected from existing datasets and web sources, the availability of burnt PCB images remained limited and unbalanced compared to Good PCBs. To address this issue, as described in the literature review (section 4.5.2 Synthetic data generation using generative AI), AI-based tool can be used to generate images. 60 burnt_PCB images (examples are shown in ) are artificially generated using two AI tools, including a) ChatGPT (via DALL·E) and b) Google Gemini (used between April and August 2025). Burnt PCB image generation process is guided using descriptive text prompts to simulate realistic damage scenarios. Example prompts included:

- "Generate a realistic Image printed circuit board with localized burn marks on one side to train an AI classification model to classify bunt PCBs and good PCBs."

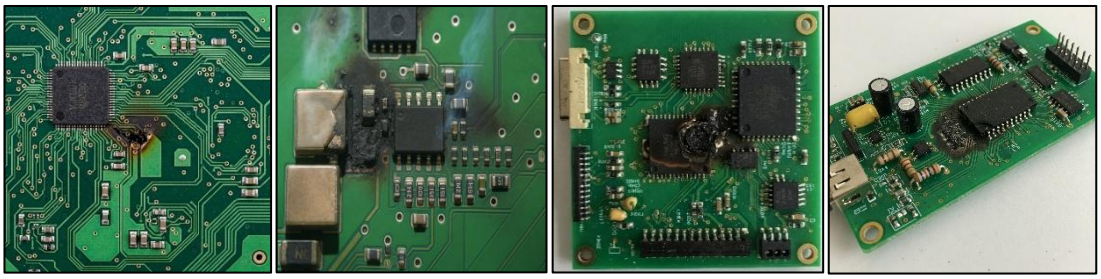


**Figure 4.2:** Examples of artificially generated burnt PCB images

Additionally, although the dataset from literature and websearch already contained a large number of Good PCB images, an additional 35 good PCB images (examples are shown in Figure 4.3) are generated using the same tools. This ensures that both classes maintain consistency in data diversity. Previous studies have shown that including synthetic data for all target classes, rather than only for the minority class, can improve model generalization and reduce class-specific bias [89]. Similar to burnt PCB image generation, the good PCB image generation process is guided using prompts such as:

- "Generate  realistic high-resolution image of a clean PCB to train AI model."

Generated images are visually inspected to ensure quality and relevance. Unrealistic outputs and duplicates are removed. The selected images are then integrated into the dataset to complement the limited real-world samples obtained from Roboflow, creating a more comprehensive training set for the classification task.

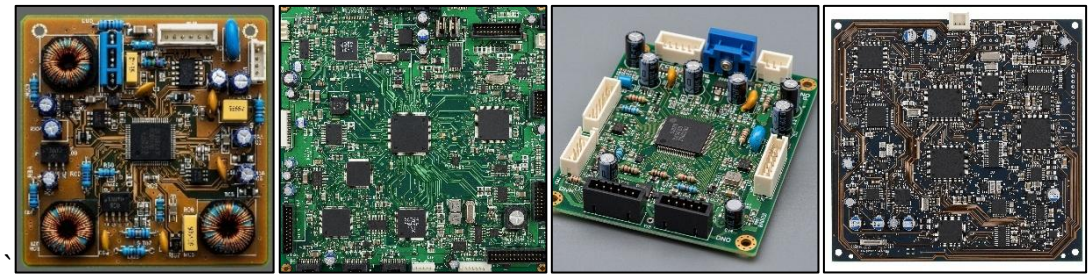


**Figure 4.3:** Examples of artificially generated good PCB images

### 4.1.4. Self Capture (Case Study PCB Dataset)

Four images of a washing machine PCB are captured from both sides of the board nd included in good PCB dataset, which serves as the basis for a case study to evaluate whether the PCB is burnt or not in future testing. Since actual burnt samples of the same PCB were not available, 9 synthetic burnt versions are generated using Photoshop to simulate these boards in a burnt condition. Examples are shown in Figure 4.4

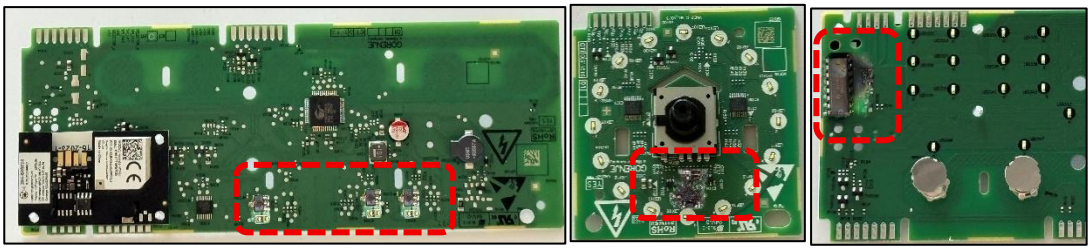


**Figure 4.4:** Synthetic burnt PCB from original PCB

## 4.2. Data Preprocessing

After collecting the data, a multi-stage data preprocessing pipeline is implemented to ensure that the dataset contains no repetitive or augmented images of the same image. Datasets downloaded from Roboflow contained augmented and duplicate images, which are necessary to remove to make the dataset unique to train the AI model. After applying the two techniques described below, 487 duplicate or augmented images are removed from the initial 772 images.

### 4.2.1. Duplicate Image Removal using Perceptual Hashing

Initially, image filtering is carried out using perceptual hashing [149]. Perceptual hashing generates a compact fingerprint of an image that reflects its overall visual structure. This detects duplicate or near-duplicate images that may only differ due to resizing, compression, or minor noise. Applying perceptual hashing ensures that exact or trivially altered copies are removed, keeping the dataset clean and preventing redundancy in training. Images with visually similar structures generate similar hash values, and then

based on these hash detects duplicates and near-duplicate images. A Python script (Appendix A.1) is developed using the imagehash and PIL libraries. It computes the pHash for each image and compares the computed hash against other images in the dataset. A similarity threshold of 2 is used to determine whether two images are similar. As described and proved by [150], threshold 2 is best to remove duplicates, higher numbers started catching false positives; images that had similar fingerprints but are too different visually. By implementing this approach, out of 772 images, 383 unique images are retained after removing duplicates.

### 4.2.2. Deep Feature-Based Filtering using Deep learning Model

Despite applying perceptual hashing, several augmented images remained in the dataset. To address this, after perceptual hashing, DL is used to enhance near-duplicate detection robustness [151] such as texture, angle, and lighting conditions. Reseach [152] has shown that ResNet50 model is highly effective for image duplication detection, achieving a precision of 0.952. In this study, DL based filtering approach is implemented using a pre-trained ResNet50 model (Python code in Appendix D.2). It extracts features from each image using a pre-trained ResNet50 and compares each image's features with a threshold of 0.95. It means that any image with morethan 95% similarity to an existing one is considered a duplicate and excluded. One study on Fitzpatrick17k (one of the largest datasets publicly available of clinical skin disease) images dataset revealed that images with similarity $\geq 0.95$ are duplicates with 98.4% precision, demonstrating reliable removal of redundant images while minimizing false positives [153] and effectively removing 133 duplicate images out of 383.

### 4.2.3. Final Dataset

After dataset preprocessing, 439 images remained. To balance the two classes, Burnt_PCB and Good_PCB, an additional 55 images from the literature are included to maintain class balance and dataset diversity, as summarized inTable 4.2.

**Table 4.2:** Summary of the Final Dataset of PCB

| Dataset Source Type | Specific Source | Burnt_PCB | Good_PCB |
|---|---|---|---|
| Existing Datasets | Literature | 0 | 55 |
| | RoboFlow | 250 | 165 |
| Websearch Images | Google Images | 110 | 25 |
| | Flickr | 0 | 145 |
| AI-generated Dataset | ChatGPT | 30 | 10 |
| | Google Gemini | 30 | 25 |
| Self-Captured | Original | 9 | 4 |
| **Total Images** | | **429** | **429** |

## 4.3. Data Splitting

One of the requirements of DL classification models is to divide the cleaned dataset into different folders: a) Training, b) Validation, and c) Test dataset. This separation ensures that the model is trained on training dataset, tuned and optimized on validation dataset, and finally evaluated on a completely unseen test dataset to provide a realistic measure of its generalization ability. In principle,

$$D = D_{train} \cup D_{val} \cup D_{test}\,, \qquad D_{train} \cap D_{val} \cap D_{test} = \emptyset$$

Investigating the impact of train/validation split ratio on the performance of pre-trained models with custom datasets shows that 70% of the dataset images for the training task give the best performance [154]. A Python code (Appendix D.3) is used to automate the splitting process, whereby 304 (~ 71%) of the images are randomly assigned to the training set, 100 (~23%) to the validation set, and the remaining 25 images to the test set. This structured and randomized data splitting process reduces the risk of overfitting and supports the fair evaluation of the trained model on unknown validation data.

## 4.4. Data Augmentation

The primary objectives of applying data augmentation (Aug) are described in the section 3.2.2 Data Augmentation. Furthermore, in the literature review (section 2.5.1 data augmentation), multiple case studies have been mentioned where data augmentation has been applied (Table 2.4).

For the PCB dataset in this case study, six different augmentation pipelines are applied, with each pipeline generating five augmented images per original, and a seventh combined pipeline generating twenty images, resulting in 50 augmented images per original image. The Albumentations [155] library for augmentation, OpenCV for image processing, TQDM for progress visualization, and UUID for generating unique file names are implementated using Python (Appendix D.4).

a) To account for rotation and positional variation during PCB inspection, Geometrical augmentations (Rotations, flips, transposes, and perspectives) transformations are applied. These transformations simulate situations where PCBs are placed at inconsistent angles or slightly tilted due to manual handling or uneven conveyor alignment. By including these variations, the model is trained to perform classification regardless of the orientation, perspective, or placement of cameras.

b) To account for PCBs image capture while in motion, or under low-quality imaging hardware that introduces noise; Augmentation of blur, Gaussian blur, and Gaussian noise are applied. These augmentations help the model train to blur visuals caused by any movements, as well as grainy textures caused by low-qualityimages or environmental interference.

c) To account for inconsistent environmental conditions (lighting, shadows, or reflective surfaces), augmentations of brightness and contrast variation, hue-saturation shifts, RGB channel shifts, sharpening, and histogram equalization (CLAHE) are used. These adjustments train the model to detect burnt areas when lighting conditions affect the color intensity, brightness, or contrast of images.

d) In some cases, PCBs might be temporarily occluded during inspection, for example, by a worker's hand, a label, or another object, or sometimes a PCB with a slot/window design. To account for these issues, augmentation cutout (coarse dropout) and grid dropout are applied to randomly hide sections of the image. This trains the model for classifying PCBs when some parts of the board are not fully visible, or a special slot-designed PCB.

e) The collected dataset primarily consists of PCB images without any background, whereas in real-world scenarios, PCBs are captured against a background. To account for these variations and improve model robustness, augmentation techniques involving white background (for this thesis, during deployment, PCB will be captured against a white background), padding are applied. This approach ensures that the model focuses on learning discriminative PCB features rather than overfitting to specific background textures or border regions.

f) To account for the effect of compression and low-quality imaging from camera systems, augmentations of JPEG compression and image downscaling are applied. These transformations train the model to maintain accuracy when image quality is reduced due to storage constraints or camera low quality capturing.

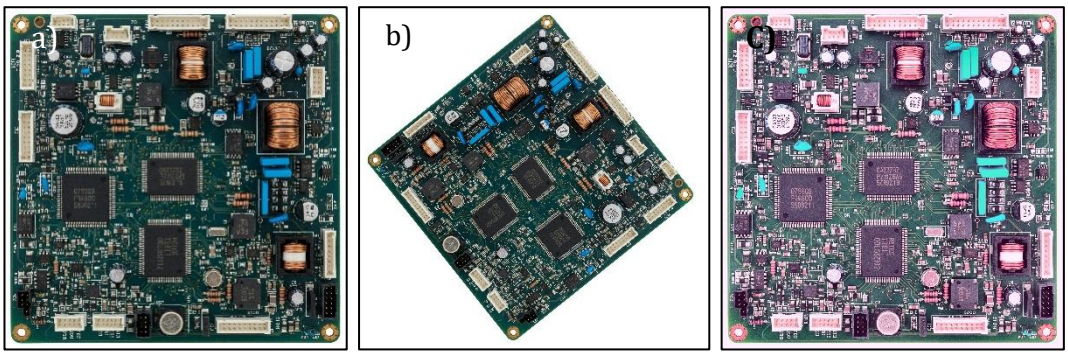



**Figure 4.5:** Aug examples a) Original b) geometrical c) Lighting effect

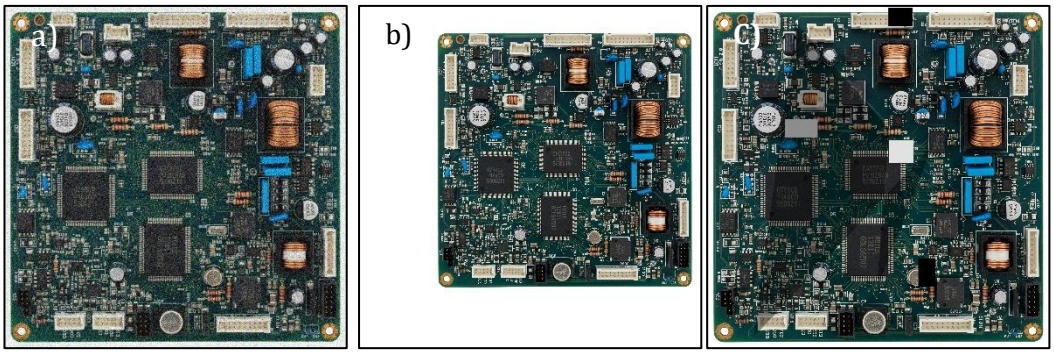



**Figure 4.6:** Aug examples a)Noise b) Background texture c) Occlusion and shadow

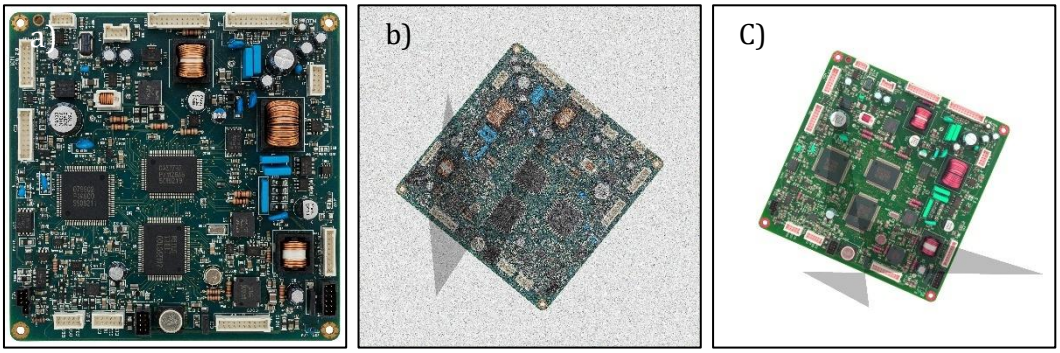



**Figure 4.7:** Aug examples a) Compression b) and c) Mixed

## 4.5. Training DL Models

The state-of-the-art models described in the section 3.5 are trained on the prepared dataset using the configurations in Table 4.3. These hyperparameters are chosen to balance convergence, stability, and computational efficiency, following common practice for pre-trained vision models and the official recipes used for Transformers. All models are initialized from ImageNet pretrained weights [156] and during training, both the best-performing weights and the final checkpoint weights are saved.

AdamW is selected because decoupled weight decay improves generalization during fine-tuning; a base learning rate of 1e-4 is the standard, stable choice for pretrained backbones. Cosine annealing allows the rate to decay smoothly, providing fast learning early on and careful refinement later. The same has been used in DL models based visual inspection of automotive parts in the remanufacturing process [82]. Epochs are set to 50 to complete the schedule without unnecessary runtime, batch sizes are set by GPU memory, and workers are chosen to keep data loading from bottlenecking training. Checkpointing stores both the best validation model and the final epoch for reproducibility. Input resolutions and transforms follow each backbone's expected recipe: ResNet and EfficientNet use their customary sizes with ImageNet statistics, DeiT

uses its patch-aligned resolution, DINO ViT uses a single-crop SSL pipeline, and DINO CaIT uses the canonical multi-crop SSL with per-iteration schedules for learning rate, teacher momentum, and weight decay.

**Table 4.3:** Model Training Configuration Summary

| Parameter | YOLOv11 x-cls | ResNet 50 | ResNet 152 | EfficientNet B3 | EfficientNet B7 | DeiT - Base | DINO (ViT-S/16) | DINO (CaIT XXS-24) |
|---|---|---|---|---|---|---|---|---|
| Input size | 224×224 | 128×128 | 128×128 | 300×300 | 300×300 | 224×224 | 224×224 | 128×128 |
| Loss | CE | CE | CE | CE | CE | CE | CE | CE |
| Base LR | Auto | 1.00E-04 | 1.00E-04 | 1.00E-04 | 1.00E-04 | 1.00E-04 | 1.00E-04 | 1.00E-04 |
| Optimizer | | AdamW | AdamW | AdamW | AdamW | AdamW | AdamW | AdamW |
| Momentum | betas=(0.9, 0.999) | betas=(0.9, 0.999) | betas=(0.9, 0.999) | betas=(0.9, 0.999) | betas=(0.9, 0.999) | betas=(0.9, 0.999) | betas=(0.9, 0.999) | betas=(0.9, 0.999) |
| LR scheduler | Internal | CosineAnnealingLR (eta_min=1e-6) | CosineAnnealingLR (eta_min=1e-6) | CosineAnnealingLR (eta_min=1e-6) | CosineAnnealingLR (eta_min=1e-6) | CosineAnnealingLR (eta_min=1e-6) | CosineAnnealingLR (eta_min=1e-6) | CosineAnnealingLR (eta_min=1e-6) |
| Epochs | 50 | 50 | 50 | 50 | 50 | 50 | 50 | 20 |
| Batch size | 8 | 32 | 32 | 32 | 8 | 32 | 32 | 0 |
| Workers | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 |
| Checkpoints | best + last | best + last | best + last | best + last | best + last | Epoch-10 + final | best + last | best + last |

Python codes used to train these models are given in Appendix D.1 to Appendix D.10. Additionally, hardwarestep nd Python libraries used to train these models are described in Appendix Appendix C and Appendix C.2.

## 4.6. Evaluation of DL Models

The models are evaluated on the test set using the best weights during training and by metrics (section 3.3) include accuracy, precision, recall, and F1 score, confusion matrix, and PR curve are computed to assess models' performance. The metrics have been reported in the literature of PCB CV visual inspection model evaluations [157].

Accuracy provides the overall percentage of correctly classified PCBs, offering a quick benchmark of model performance. Precision shows how many predictions for a class are correct, while recall shows how many actual samples of that class are detected. The F1-score combines precision and recall, making it easier to judge models when both types of errors matter. The confusion matrix visualizes misclassifications for each class, providing a clear view of where the model struggles. Finally, precision–recall curves evaluate robustness across confidence thresholds, showing how performance changes depending on the strictness of decision boundaries.

# Chapter 5

# Result

## 5.1. Training/learning behaviour of the models

The training or learning behaviour of the models is evaluated based on training loss, validation loss, and validation accuracy across different epochs. Figure 5.1, Figure 5.2 and Figure 5.3 present the respective curves for a comprehensive side-by-side comparison of all models, while the detailed graphs along with learning rate per epoch and training time for each model are provided in Appendix E.

The YOLOv11 classification model shows a rapid decrease in training loss, approaching zero from 0.25 after about 30 epochs. Validation loss fluctuates slightly in the early phase as weights adapt from the pretrained initialization but stabilizes after 20 epochs and remains constant beyond 35 epochs. Validation accuracy peaks around 99% and stabilizes after 20 epochs, indicating effective feature learning for distinguishing burnt and good PCBs without signs of overfitting.

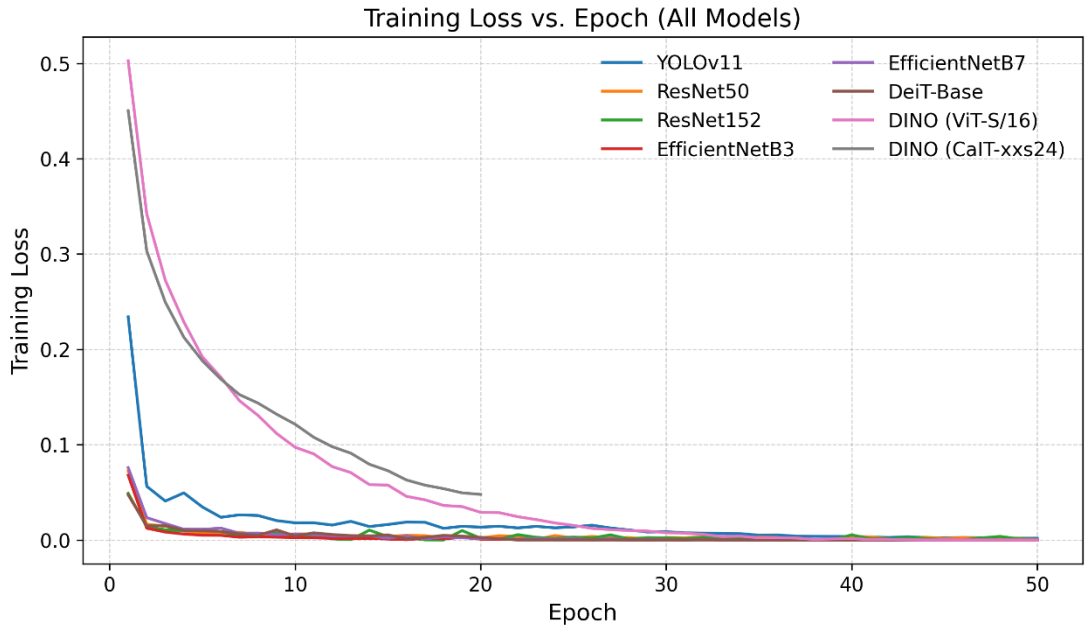



**Figure 5.1:** Training Loss per epoch of all Models

The ResNet50 model demonstrates a sharp decrease in training loss from 0.7 at the first epoch to nearly zero by epoch 30, where it remains stable. In contrast, the validation loss increases gradually until 30 epochs and fluctuates. After epoch 30, it suddenly drops to

around 0.4 and becomes relatively stable, indicating the model has generalized the features. Despite this divergence and fluctuation, the validation accuracy rises rapidly, to 99% within 15 epochs, and remains close to 99.9% until the end of training. The ResNet152 model shows a rapid decrease in training loss, dropping from 0.05 at the first epoch to nearly zero by epoch 30. Validation loss fluctuates more strongly than in ResNet50, with occasional peaks up to 0.7, but generally remains in the range of 0.1 and 0.3 after 20 epochs. Despite these variations, validation accuracy improves quickly, surpassing 99% within 10 epochs and stabilizing around 100% by the end of training.

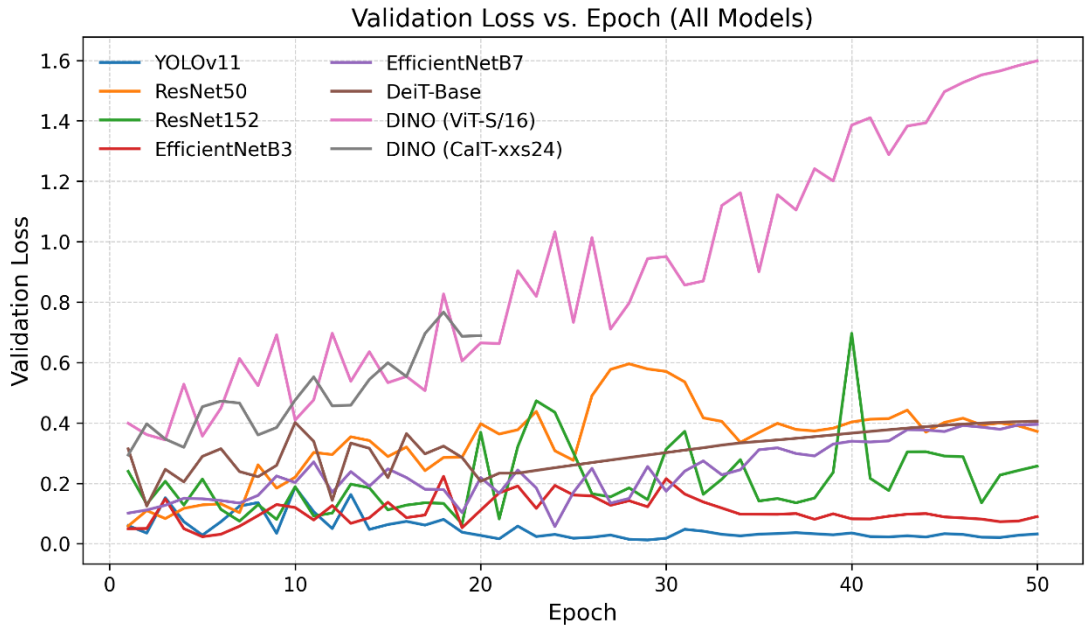



**Figure 5.2:** Validation Loss per epoch of all Models

The EfficientNetB3 model shows a steep decline in training loss, decreasing from 0.07 in the first epoch to nearly zero by epoch 20, where it remains constant. The validation loss exhibits notable fluctuations, ranging mostly between 0.07 and 0.22, with peaks during the middle training phase and gradual stabilization near 0.1 in later epochs. Validation accuracy starts around 98% and varies throughout training between 96.5% and 99%, without settling into a stable plateau. The EfficientNetB7 model records a sharp decrease in training loss, reducing from 0.07 at the first epoch to nearly zero by epoch 15, where it stabilizes. The validation loss fluctuates throughout training, ranging from 0.08 to 0.15, with a peak of approximately 0.26 observed around epoch 17. Validation accuracy starts at nearly 97% and exhibits fluctuations between 95.5% and 99% without a stable plateau until the end of training.

The DeiT model shows a sharp decrease in training loss, reducing from 0.05 at the first epoch to nearly zero by epoch 20, where it stabilizes. The validation loss displays

fluctuations in the early epochs and then increases steadily after epoch 25, reaching values of about 0.4 at epoch 50. Validation accuracy starts at 93%, varies in the first 20 epochs, and then stabilizes between 97% and 98%.

The DINO-ViT fine-tuned model shows a sharp reduction in training loss, decreasing from about 0.5 at the first epoch to nearly zero by epoch 40. Validation loss follows an opposite trend, starting around 0.35 and steadily rising across training, reaching about 1.6 at epoch 50. Validation accuracy peaks near 90% around epoch 20, before declining and stabilizing around 86% in later epochs.

The DINO-CaIT fine-tuned model shows a steady decrease in training loss, dropping from about 0.45 in the first epoch to near 0.05 by epoch 20. Validation loss starts around 0.3, fluctuates in early epochs, and then rises gradually, reaching about 0.75 by the end. Validation accuracy remains relatively stable throughout training, fluctuating between 82% and 86% without a clear upward trend.

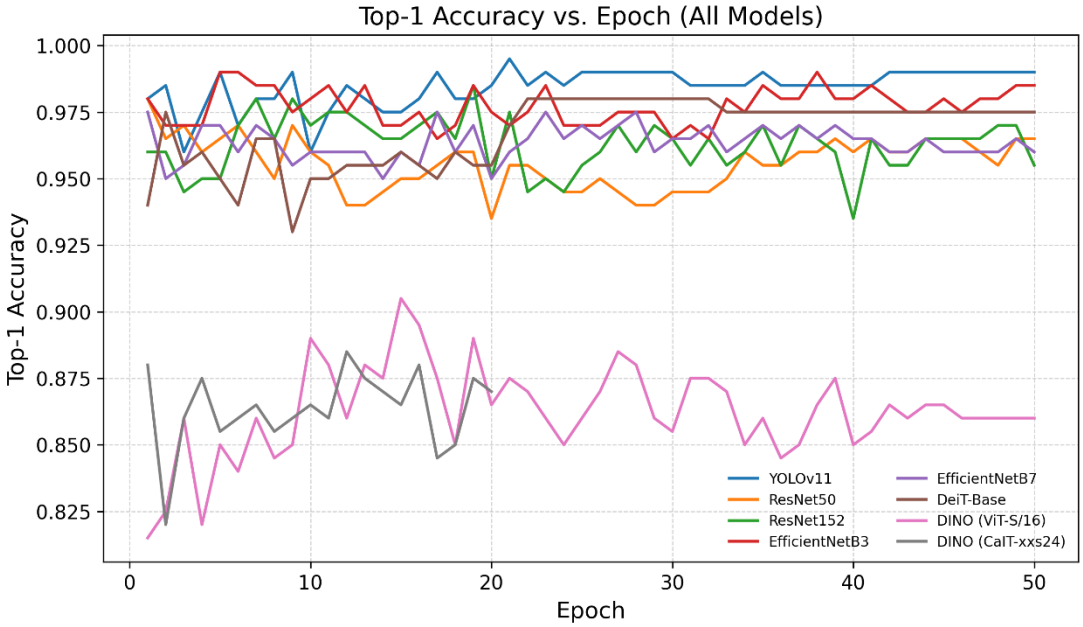



**Figure 5.3:** Top-1 Accuracy per epochs for all models

Overall, all models converge rapidly in training, with losses approaching zero within 20–30 epochs and training accuracy nearing 100%. CNN-based models show relatively stable validation trends with early accuracy saturation. The transformer-based model displays stronger fluctuations in validation loss and accuracy across epochs. SSL-based models achieve smooth convergence in training but exhibit steadily rising validation loss and less stable validation accuracy, highlighting their different training dynamics.

## 5.2. Classification Performance

To evaluate model classification performance, confusion matrices are analyzed on both the validation set (normalized percentages) and the test set (absolute counts), as shown in Figure 5.4and Figure 5.5 respectively. These confusion matrices combine the results of all trained models into one confusion matrix for easier comparison. A confusion matrix is normally a 2×2 table that shows how well a model distinguishes between two classes; in this case, Burnt PCB and Good PCB. The four large boxes represent the four possible outcomes: a) Top-left: Burnt PCBs correctly identified as burnt (true positives), b) Top-right: Burnt PCBs wrongly identified as good (false negatives), c) Bottom-left: Good PCBs wrongly identified as burnt (false positives), and Bottom-right: Good PCBs correctly identified as good (true negatives). Inside each large box, the diagonal positions contain values for each model, showing how well model performed for that particular outcome.

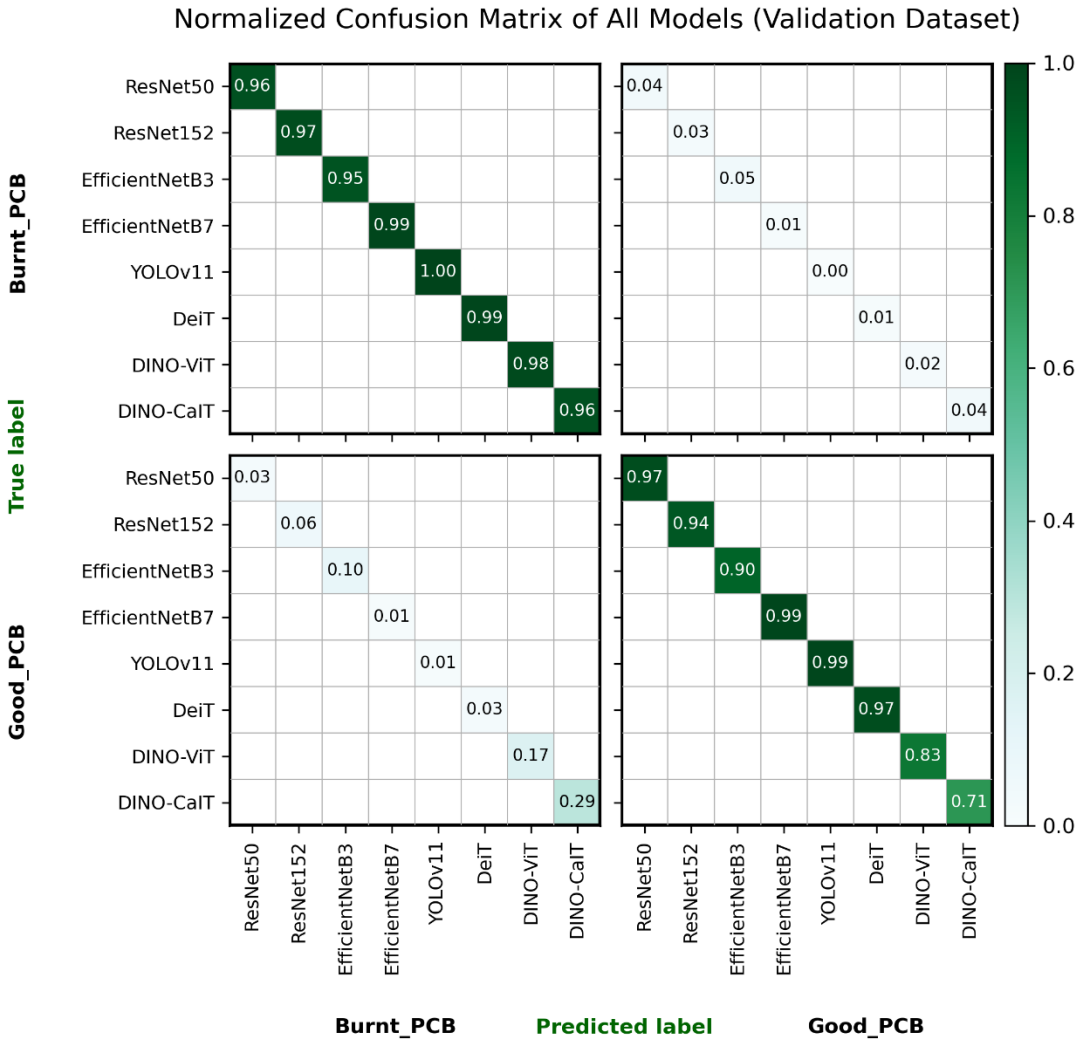



**Figure 5.4:** Normalized confusion matrix of all Models (Validation dataset)

YOLOv11 validation performance is almost perfect, with 100% accuracy on burnt PCBs and 99% on good PCBs. On the test set, it correctly classifies 23 burnt PCBs and all good PCBs, demonstrating robust generalization and balanced performance.

ResNet-50 achieves balanced validation accuracy of 96% for burnt PCBs and 97% for good PCBs. On the test set, it correctly classifies 24 burnt PCBs and all 25 good PCBs, with just one burnt board misclassified as good. This highlights ResNet-50 as one of the most reliable models, showing stable generalization across both datasets. On the other hand, the ResNet-152 model, having a higher number of trainable parameters, reaches 97% for burnt PCBs and 94% for good PCBs on validation datasets, with slightly higher errors on the good class. On the test set, it correctly classifies 24 burnt PCBs but only 15 good PCBs, indicating reduced performance on unseen data.

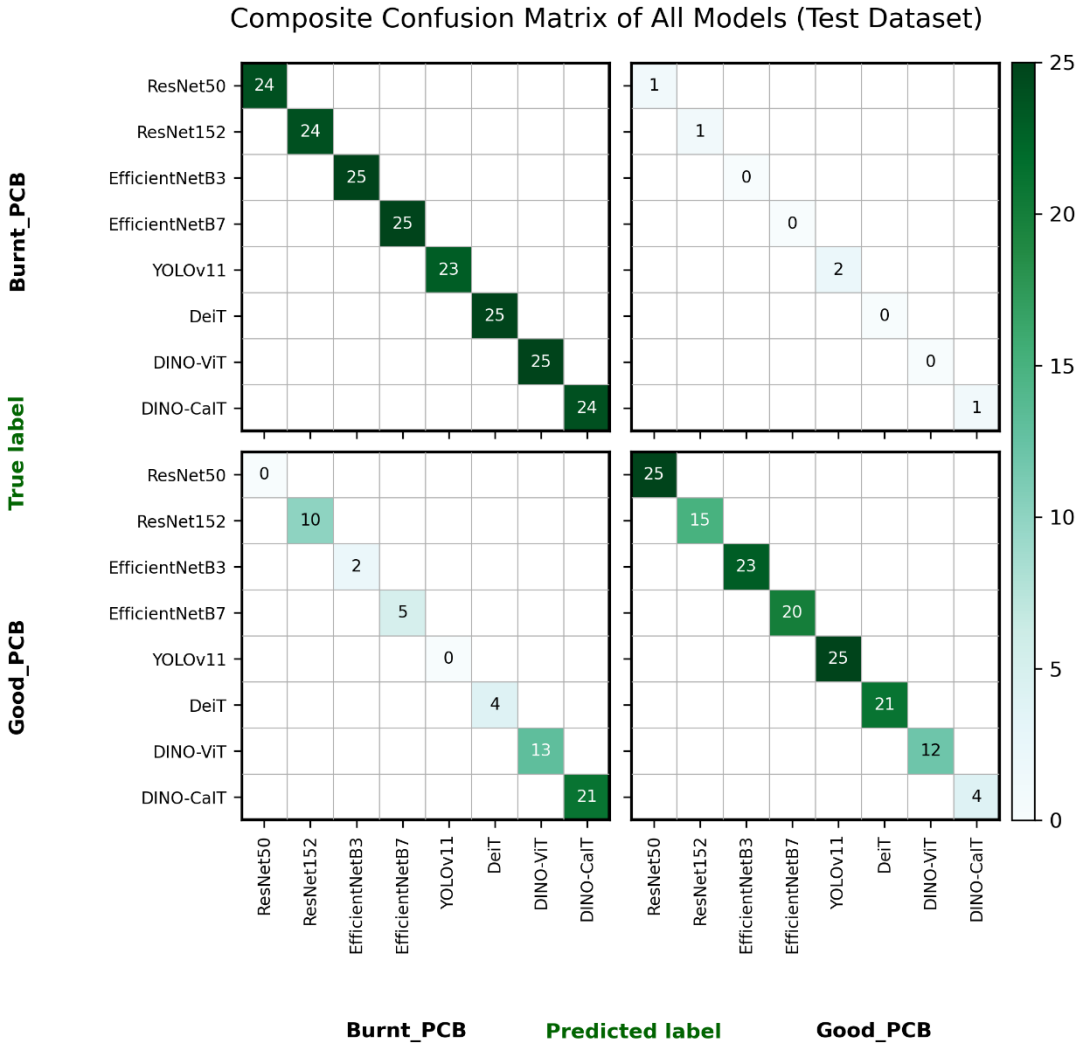



**Figure 5.5:** Confusion Matrix of all Models (Test Dataset)

On the validation set, EfficientNetB3 achieves 95% accuracy for burnt PCBs and 90% for good PCBs, misclassifying 5% and 10% respectively. On the test set, it correctly classifies 25 burnt PCBs and 23 good PCBs, with only 2 good boards misclassified as burnt. On the other hand, EfficientNetB7, validation performance is nearly perfect, with 99% accuracy for both burnt and good PCBs. On the test set, the model correctly identifies all 25 burnt PCBs, but only 20 out of 25 good PCBs, with 5 misclassified as burnt.

DeiT achieves 99% accuracy on burnt PCBs and 97% on good PCBs in validation. On the test set, it classifies all 25 burnt PCBs correctly; however, it only classifies 21 good PCBs correctly. This shows excellent detection of burnt boards, but a sensitivity for good PCBs.

DINO-ViT validation results show 98% accuracy on burnt PCBs, but only 83% on good PCBs, with 17% of good boards misclassified. On the test set, it correctly classifies all 25 burnt PCBs but only 12 good PCBs, with 13 misclassified as burnt. t-SNE before fine-tuning, burnt and good PCBs showed substantial overlap, reflecting weak class separability. After fine-tuning, two clearer clusters emerged. However, some good PCB embeddings remained mixed within the burnt cluster (Figure 5.6), reflecting a strong bias toward burnt classification and weak recognition of good PCBs.

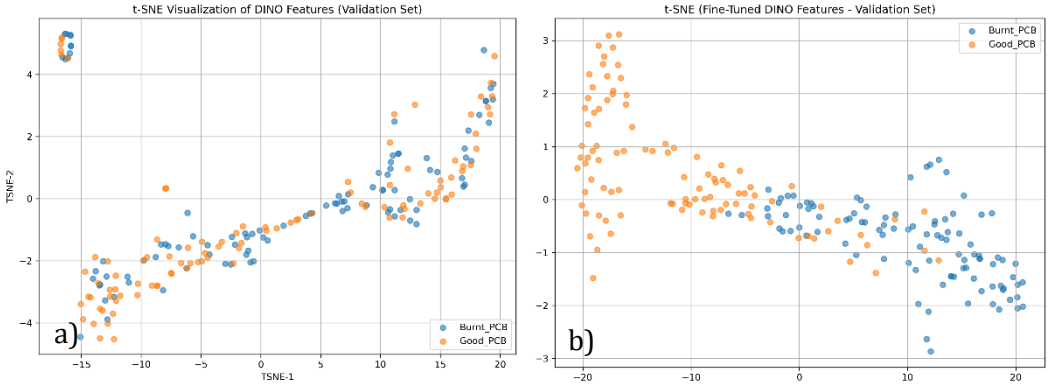



**Figure 5.6:** DINO-ViT t-SNE a) before and b) after finetuning

DINO-CaiT achieved 96% accuracy for burnt PCBs on validation but only 71% for good PCBs, with nearly one-third of good boards misclassified. Test results confirmed this imbalance: 24 burnt PCBs are correctly classified, but only 4 good PCBs were predicted correctly, while 21 are incorrectly labeled as burnt. t-SNE of DINO-CaIT before fine-tuning shows that good PCBs have a separate cluster; however, good PCBs also have substantial overlap with burnt PCBs, reflecting weak class separability. After fine-tuning, two clearer clusters emerged. However, some good PCB embeddings remained mixed within the burnt cluster (Figure 5.7), confirming unstable classification and limited generalization.

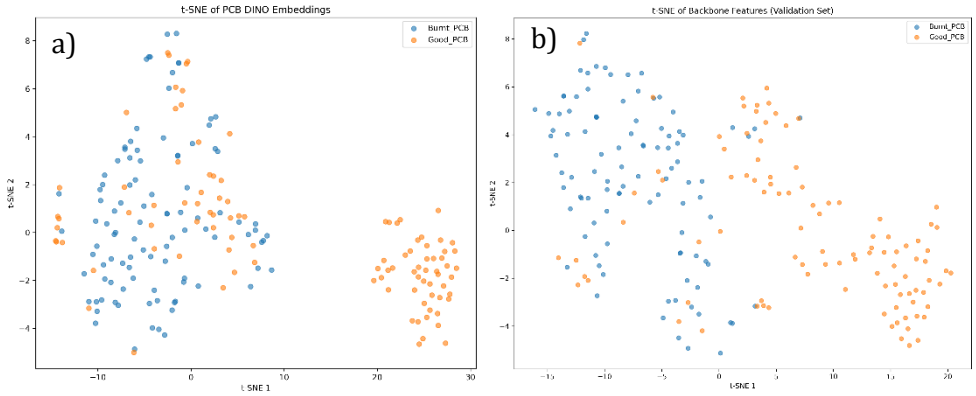



**Figure 5.7:** DINO-CaiT t-SNE a) before and b) after finetuning

## 5.3. Models Overall Performance Metrics Comparison

Table 5.1 summarizes the overall performance of all models in terms of classification performance metrics, overall accuracy, inference time per image, and frames per second.

YOLOv11 and ResNet50 both achieve near-perfect results, with precision, recall, and F1-scores of approximately 0.98 across classes. For burnt PCBs, both models maintain perfect precision (1.0) and high recall (0.96), while for good PCBs, they achieve perfect recall (1.0) and precision near 0.96. The balance between the two classes indicates that both models rarely misclassify burnt as good or vice versa, making them the most reliable for practical deployment. Their symmetry across precision and recall ensures that neither false positives nor false negatives dominate, which is particularly valuable in high-stakes industrial classification. ResNet152 exhibits clear weaknesses compared to ResNet50. While it attains a strong recall for burnt PCBs (0.96), its precision drops to 0.71. For good PCBs, precision is high (0.94), but recall falls drastically to 0.60, meaning many good boards are misclassified as burnt. WA F1-score of only 0.77, significantly lower than its shallower counterpart.

EfficientNetB3 achieves balanced and strong results, with WA of 0.96 across precision, recall, and F1 score. It identifies burnt PCBs with perfect recall (1.0) and good PCBs with perfect precision (1.0), showing a well-calibrated distribution of errors. EfficientNetB7, while still competitive, performs less consistently. It achieves perfect recall for burnt PCBs but only 0.80 recall for good PCBs, resulting in an overall F1-score of 0.90. This indicates that B7 is biased towards burnt PCB and misclassifies good boards.

The DeiT-Base transformer delivers competitive performance, with weighted precision, recall, and F1-scores around 0.92. It achieves perfect precision for good PCBs (1.0), meaning it rarely produces false positives, but recall is lower (0.84), indicating missed

detections of good boards. For burnt PCBs, recall is strong (0.91), but precision drops slightly to 0.86. These results highlight that DeiT prioritizes precision over recall, which may be preferable in contexts where false alarms are less costly than missed defects.

**Table 5.1:** Comparison of all Models' Performance Metrics (Test dataset)

| Model | Class | Precision | Recall | F1-Score | Accuracy | IT (ms) | FPS |
|---|---|---|---|---|---|---|---|
| **YOLOv11** | Burnt_PCB | **1** | 0.96 | **0.98** | **98%** | 23.3 | 43 |
| | Good_PCB | 0.96 | **1** | **0.98** | | | |
| | WA | 0.98 | 0.98 | **0.98** | | | |
| **ResNet50** | Burnt_PCB | **1** | 0.96 | **0.98** | **98%** | **3.04** | **328** |
| | Good_PCB | 0.96 | **1** | **0.98** | | | |
| | WA | 0.98 | 0.98 | **0.98** | | | |
| **ResNet152** | Burnt_PCB | 0.71 | 0.96 | 0.81 | 78% | 36.2 | 28 |
| | Good_PCB | 0.94 | 0.6 | 0.73 | | | |
| | WA | 0.82 | 0.78 | 0.77 | | | |
| **EfficientNetB3** | Burnt_PCB | 0.93 | **1** | 0.96 | 96% | 15.6 | 64 |
| | Good_PCB | **1** | 0.92 | 0.96 | | | |
| | WA | 0.96 | 0.96 | 0.96 | | | |
| **EfficientNetB7** | Burnt_PCB | 0.83 | **1** | 0.91 | 90% | 19.4 | 52 |
| | Good_PCB | **1** | 0.8 | 0.89 | | | |
| | WA | 0.92 | 0.9 | 0.9 | | | |
| **DeiT-Base** | Burnt_PCB | 0.86 | **1** | 0.86 | 92% | 13.5 | 74 |
| | Good_PCB | **1** | 0.84 | 0.91 | | | |
| | WA | 0.93 | 0.92 | 0.92 | | | |
| **DINO (ViT-S/16)** | Burnt_PCB | 0.65 | **1** | 0.80 | 74% | 7.8 | 128 |
| | Good_PCB | **1** | 0.48 | 0.64 | | | |
| | WA | 0.83 | 0.74 | 0.72 | | | |
| **DINO (CaIT-xxs/24)** | Burnt_PCB | 0.56 | 0.79 | 0.71 | 60% | 11.3 | 88 |
| | Good_PCB | 0.86 | 0.24 | 0.38 | | | |
| | WA | 0.71 | 0.6 | 0.54 | | | |

SSL DINO models perform considerably weaker. DINO-ViT achieves moderate precision (0.83) but poor recall (0.74), with particularly weak detection of good PCBs (recall 0.48). This indicates a tendency to miss many good boards while being conservative in predictions. DINO-CaIT performs worst overall, with a weighted F1-score of only 0.54. Its recall for good PCBs is especially low (0.24), meaning that nearly three-quarters of good boards are misclassified as burnt. Such instability suggests that without substantial fine-tuning or larger pretraining datasets, SSL-based models struggle to generalize reliably in this classification task.

Figure 5.8 shows Precision–Recall curves and F1-Confidence curves. YOLOv11, ResNet50, and EfficientNetB3 maintained near-perfect precision (>0.95) across the full

recall range, reflecting a strong balance between sensitivity and specificity. DeiT-Base and EfficientNetB7 also demonstrated stable performance, though with slightly lower precision at high recall values. In contrast, ResNet152, DINO-ViT, and particularly DINO-CaiT displayed weaker PR behavior. DINO-CaiT's curve dropped sharply, indicating poor balance between precision and recall and confirming its tendency to misclassify good PCBs as burnt. The F1–Confidence curves further reinforce these findings. YOLOv11, ResNet50, and EfficientNetB3 consistently exhibited high F1-scores (>0.95) across almost all confidence thresholds, demonstrating robust performance regardless of the decision boundary. EfficientNetB7 and DeiT-Base achieved slightly lower but still stable F1 values (~0.90). By contrast, ResNet152 and DINO-ViT showed a gradual decline in F1 as confidence increased, while DINO-CaiT suffered a steep drop, with F1 falling below 0.4 at moderate confidence thresholds.

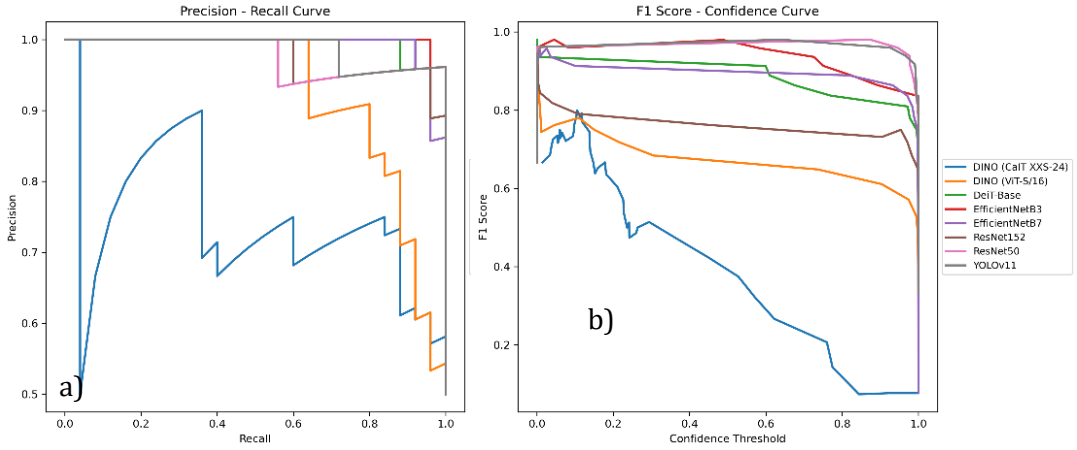



**Figure 5.8:** a) Precision- Recall Curve b) F1-Score - Confidence Curve

The results show that YOLOv11 and ResNet50 achieved the highest classification accuracy (98%), while EfficientNetB3 followed closely with 96%, and DeiT-Base reached 92%. In contrast, the deeper Reset152 and SSL models (DINO-ViT, DINO-CaiT) performed worse, with accuracies of 78%, 74%, and 60%, respectively.

In terms of computational efficiency, ResNet50 achieves the fastest inference with 3.04 ms per image, corresponding to 328 FPS, making it the most efficient model. DINO-ViT also shows strong efficiency at 7.8 ms (128 FPS), followed by DINO-CaIT at 11.3 ms (88 FPS) and DeiT at 13.5 ms (74 FPS). EfficientNetB3 and EfficientNetB7 process images at 15.6 ms (64 FPS) and 19.4 ms (52 FPS), respectively, offering real-time performance with moderate latency. YOLOv11 requires 23.3 ms per image (43 FPS), while ResNet152 is the slowest, with 36.2 ms per image (28 FPS).

# Chapter 6

# Discussion and Future Work

This thesis outlines a systematic methodology for the use of DL models for the visual inspection of used parts in remanufacturing to assess their reusability. Based on the PCB visual inspection case, its challenges have been addressed through the solution approach of this thesis, and this approach has been implemented and evaluated. In the following, the insights gained are summarized and critically assessed in relation to the research deficits (Chapter 2) derived in this thesis. These findings also provide the basis for deriving possible future research activities.

## 6.1. Suitability of Model Architectures for PCB Classification

The results demonstrate that CNN-based models are highly suitable for the PCB burn classification task, with YOLOv11 and ResNet50 achieving the same performance metrics (precision, recall, and F1-score). These outcomes confirm the effectiveness of CNNs in capturing local texture, edge patterns, and discoloration cues, which are characteristic of burnt PCB surfaces. EfficientNetB3 further demonstrated that carefully scaled CNNs can balance accuracy and computational efficiency, making them well-suited to resource-constrained environments. This is consistent with earlier studies in PCB defect detection. CNNs are widely used for PCB defect detection due to their end-to-end (from image input to final classification ) learning capability and superior accuracy in complex scenarios [71]. ResNet152 and EfficientNetB7 are supposed to perform better, as they achieved higher accuracy on ImageNet dataset described in section 2.4.1. ; however, despite their increased parameter counts and theoretical representational capacity, they underperform relative to their smaller counterparts.

While ResNet-50 offers phenomenal computational efficiency (3 ms IT, 328 FPS), YOLOv11 provides resilience through its multi-scale feature fusion, which is designed for object detection, and it can localize small and varied burnt marks better. In PCB applications, where defect scale and position vary significantly, this robustness is critical. Prior studies in PCB defect detection confirm that YOLO-based models (YOLOv11[158], CDI-YOLO [159], YOLO-MBBi [160], YOLO-HMC[161] etc.) are highly accurate (98–99%

mAP) and have real-time speed (up to 128 FPS) for PCBs visual inspection-based applications. Additionally, the majority of the DL models for automated visual inspection in manufacturing and maintenance studies are YOLO based models [162].

DeiT from ImageNet pretraining enabled the model to perform an overall 92% accuracy and perfect precision on goo_PCB. its reliance on global self-attention mechanisms made it less effective at focusing on the localized burnt regions. SSL models, including DINO-ViT and DINO-CaiT, performed even worse, reflecting their dependency on large-scale unlabeled data that is not available in this study. Thus, under the current dataset conditions, CNNs are the most suitable architectures for burnt PCB classification.

## 6.2. Performance Differences Across Architectures

Beyond the conceptual suitability of training, performance metrics provide further evidence of differences between architectures and their strengths and weaknesses.

The training and validation curves provide a comparative view of how CNNs and transformer-based architectures behave on the PCB classification task. All models reduce training loss close to zero, confirming their ability to fit the training set. However, the differences emerge when comparing validation loss and accuracy, which demonstrate generalization performance (section 4.6).

Among the CNN-based models, YOLOv11 and EfficientNetB3 achieve consistently high top-1 accuracy above 97% while maintaining low and stable validation loss across epochs. This stability highlights their ability to balance learning and generalization effectively. EfficientNetB7 also performs strongly but with slightly more fluctuations, indicating that deeper EfficientNet variants add representational capacity at the expense of stability. ResNet50 and ResNet152 display less competitive performance; although training losses converge to zero, their validation losses are higher and more variable. ResNet50 does stabilize toward the later epochs, suggesting reasonable generalization, while ResNet152 suffers from persistent instability, highlighting the limitations of deeper residual networks without stronger regularization.

For the transformer-based models, distinct behaviors are observed between supervised and self-supervised variants. DeiT-Base achieves high accuracy comparable to CNNs but its validation loss gradually increases before stabilizing at a higher level. This indicates that while DeiT effectively learns discriminative features, it is less stable than CNN counterparts and more sensitive to training dynamics. In contrast, the SSL variants (DINO) perform less favorably. DINO (ViT-S/16) shows the weakest results: although

training loss decreases steadily, validation loss rises continuously and accuracy plateaus at a much lower level, pointing to overfitting and limited generalization. DINO (CaIT-xxs24) performs slightly better but still underperforms CNNs and DeiT, with moderate accuracy and unstable validation trends.

Furthermore, the evaluation on the test dataset further supports the observations drawn from training and validation behavior. CNN-based models, both YOLOv11 and ResNet50, consistently achieved the highest performance across almost all metrics. Their training and validation curves showed stable convergence with minimal overfitting, and their confusion matrices revealed a nearly perfect balance between burnt and good PCB classification. Recall values for the good_PCB class reached 1.0, indicating that these models perfectly classified good_PCBs, meaning that none of the reusable boards are misclassified as defective. For the burnt_PCB class, the values are 0.96, very close to 1.0, resulting in weighted F1-scores of 0.98. This suggests that the models may occasionally overlook burnt PCBs. These results indicate that YOLOv11 and ResNet50 are not only accurate but also reliable under different operating conditions, making them well-suited for tasks where both sensitivity and specificity are critical.

EfficientNetB3 achieves a strong balance with moderate inference speed, confirming that its compound scaling is well-matched to the dataset. In contrast, EfficientNetB7, although deeper, shows unstable validation loss and signs of class imbalance. Both achieve perfect recall for burnt PCBs but lower recall for good PCBs, indicating an over-sensitivity toward defects. This indicates that when they classify a PCB as burnt, the prediction is highly reliable, thereby reducing the risk of defective boards being mistakenly returned to production. Thus, EfficientNet variants prioritize certainty in defect detection, in contrast to YOLOv11 and ResNet50, which maximize recall of usable boards. Both perspectives are important: one minimizes unnecessary waste, while the other ensures stringent quality assurance.

ResNet-152 underperforms among the CNN-based models. Although it reduces training loss to near zero, its validation loss fluctuates across epochs, reflecting over-parameterization and overfitting rather than stable generalization. These outcomes confirm that greater model depth does not necessarily improve performance in data-limited, domain-specific tasks.

In literature, the YOLOv11 [158] achieves 99.2% mAP@0.5, while the transformer-based model, LPViT [163] reaches 99.08% accuracy, which is higher for PCB defect detection. These figures are not directly comparable for PCB classification models, since mAP

measures detection performance and accuracy measures classification, but they both reflect state-of-the-art performance levels. In this study, however, transformer-based model (DeiT) achieves less accuracy (92%) than CNN-based models. Transformers require extensive training data to effectively utilize their global attention mechanisms [164], whereas CNNs are inherently better suited to small, domain-specific datasets due to their strong inductive bias toward local feature extraction. DeiT-Base showed yet another pattern; it achieved a perfect recall of 1.0 for Burnt_PCB, ensuring that no defective boards are missed. These results suggest that both EfficientNet and DeiT-Base prioritize defect detection, whereas YOLOv11 and ResNet50 favor maximizing recovery of reusable boards. This is consistent with the literature in PCB defect detection, where CNNs dominate due to their efficiency and robustness on limited datasets [71].

The SSL approaches, DINO-ViT and DINO-CaiT, are the weakest models in this evaluation. While both architectures misclassified a significant good PCBs as burnt, DINO-ViT performed slightly better than DINO-CaiT, achieving a weighted F1-score of 0.72 compared to 0.54. The confusion matrices and t-SNE visualizations confirmed this difference: DINO-ViT produced somewhat clearer class clusters after fine-tuning, whereas DINO-CaiT embeddings showed heavy overlap between burnt and good samples. Despite this relative difference, both models remained unsuitable for PCB burn classification in practice. Their limited success is directly linked to the absence of large-scale unlabeled PCB datasets for pretraining, which are essential for SSL methods to achieve generalizable representations [165]. The t-SNE visualizations of DINO embeddings further illustrate these limitations. While both CaIT and ViT backbones show partial separation between burnt and good boards, the clusters remain diffuse with considerable overlap. The ViT backbone exhibits somewhat clearer grouping of the Good_PCB class compared to CaIT, yet the Burnt_PCB features are still widely scattered. This dispersion confirms that, although DINO captures useful representations, its embeddings are not as discriminative as those learned by CNNs, which explains the weaker validation stability and test performance observed earlier .

The F1–confidence and Precision–Recall curves provide additional context. YOLOv11, ResNet50, and EfficientNetB3 maintained consistently high F1-scores across confidence thresholds, reflecting their robustness across different decision boundaries. EfficientNetB7 also followed this trend but with more gradual declines. DeiT-Base achieved reasonably stable F1-scores at mid-level thresholds but showed weaker curves compared to the CNN-based models. ResNet152 declined steadily as confidence increased, further illustrating its unreliability. The DINO variants, particularly CaIT-

xxs24, degraded sharply in both F1–confidence and PR curves, confirming their poor generalization in this task. Overall, the PR curves highlight the near-perfect balance of YOLOv11 and EfficientNet models, while ResNet50 remains competitive but less optimal.

## 6.3. Model Applicability for PCB Inspection in Circular Economy

For CE applications, inspection models must minimize both false negatives and false positives. A false negative, where a burnt PCB is classified as good, risks sending defective components into remanufacturing. A false positive, where a good PCB is classified as burnt, reduces reuse rates and increases electronic waste. In this study, YOLOv11 and ResNet50 maintain balanced precision and recall values close to 0.98, meeting both objectives and enabling more reliable reuse decisions. By reducing classification error, CNN models such as YOLOv11 and ResNet50 significantly improve inspection reliability. This translates into higher value recovery, as more good PCBs are correctly reused and fewer are unnecessarily discarded, aligning with CE goals of resource efficiency and waste minimization.

For integration into remanufacturing, model applicability is critical. The inspection stage requires consistent performance under varying conditions of lighting, board layout, and defect severity. Although YOLOv11 is the most promising candidate for deployment, industrial environments often demand near-zero tolerance for misclassification. To further improve reliability, a two-stage hybrid system can be considered: YOLOv11 is well-suited as a primary classifier due to its perfect precision on burnt PCBs, ensuring that unreusable PCBs are not mistakenly used. EfficientNetB3, with its perfect precision on good PCBs, can serve as a secondary verification model for good PCBs detected by YOLO models, ensuring that PCB is really a good one. Such a layered approach reduces both waste and product risk, increasing reuse rates while maintaining quality standards in remanufacturing lines.

## 6.4. Implications for Deep Learning in Circular Economy Use Cases

Manual PCB inspection has reported error rates of up to 30% [166] and is highly dependent on operator expertise. Rule-based vision algorithms, while useful for simple patterns, struggle to adapt to new or varying defect types[167]. This study demonstrates that DL can significantly enhance decision-making in the reusability of used components

in reuse and remanufacturing processes. DL models overcome these limitations by learning discriminative features directly from data, yet practical implications must be considered

The primary challenges in used PCB visual inspection stem from the complexity of industrial environments and the variability of PCB samples. It is difficult to select an optimal model between model size, inference speed, and accuracy.

- In practical industrial applications, image acquisition systems faced issues such as lighting conditions, shadows, and noise in complex environments. These issues result in poor-quality PCB images, particularly for small burnt marks. Such type of conditions make it difficult to capture focused and high-quality images, which are critical for accurate detection. Data scarcity remains the main bottleneck. Burnt PCB images are extremely difficult to obtain, requiring synthetic generation via DALL·E and Gemini. This highlights a structural challenge in CE inspection use cases: the data of interest (damaged parts) is rare by definition.
- Scalability depends on inference efficiency. ResNet50's extremely low inference time and YOLOv11's robustness confirm that CNNs are deployable in real-time inspection. In real-world applications, the environmental conditions vary; hence, model robustness is very important to consider before implementation.
- Transferability of SSL and Transformer models is currently limited in this domain, but future industrial data collaborations could unlock their potential.
- The research underscores the importance of explainability in CE adoption. Stakeholders must trust that models are making reliable decisions. While CNNs achieve strong performance, integrating interpretability techniques such as Grad-CAM [168] for CNNs and attention rollout [169] for transformers would help visualize decision pathways and build confidence in industrial adoption.

In broader CE contexts, the findings underline that AI-based inspection is not only a technical contribution but also an enabler of closed-loop resource flows, reducing reliance on manual inspection and improving throughput in remanufacturing plants.

## 6.5. Limitations of the Work

While the proposed AI-based classification method for visual inspection of burnt PCBs demonstrated strong performance, several limitations of this study must be

acknowledged. These limitations highlight areas where the present work falls short and provide directions for future improvement.

The availability of real-world burnt PCB images are limited. As noted in section 2.3, most publicly available PCB datasets focus on defects such as missing components or soldering issues, while burnt or thermally damaged boards datasets are not available. To address this scarcity, synthetic data generation and augmentation pipelines are employed (section 4.2). Although this strategy improved class balance and model robustness, it may have introduced a domain gap between synthetic and real images. However, synthetic data cannot fully replicate the complexity of real industrial burn marks. Consequently, model generalizability to highly diverse environments remains uncertain, a challenge also noted in surveys on PCB defect detection[71].

Similarly, burnt PCB datasets could also be created using conventional image-editing tools such as Photoshop, where burn marks from burnt PCB are artificially merged onto good PCB images. While such approaches are useful for rapidly expanding datasets, they still lack the realism of actual thermal damage in PCBs. Artificially generated burn patterns may fail to capture the subtle texture changes, color gradients, and material deformations that occur under real burning and high-temperature failure conditions. As a result, models trained on such data may inadvertently learn to focus on non-burnt regions or background similarities rather than truly discriminative features of burns. This limitation is particularly pronounced in transformer-based architectures, which rely on global attention mechanisms. Instead of consistently attending to localized burn areas, these models may spread attention across visually similar areas of intact boards, thereby reducing their discriminative power. In contrast, CNN-based models are somewhat more robust to this issue because their local receptive fields inherently emphasize fine-grained defect patterns. Nonetheless, reliance on synthetic editing tools underscores a critical limitation: without sufficient real burnt PCB images, model interpretability and reliability in practical industrial inspection remain uncertain [170].

The models are trained and validated on a dataset with controlled backgrounds and lighting conditions (section 4.4). In practice, factory environments introduce noise, variable illumination, and occlusions, which may degrade performance. While the augmentation pipelines simulated some of these conditions, they cannot fully capture the variability of real inspection setups. This limitation is consistent with prior findings that models often underperform in deployment environments [170].

The results showed that CNN-based architectures (YOLOv11, ResNet50, EfficientNetB3) consistently outperformed transformer-based and SSL approaches (DINO-ViT, DINO-CaiT). One reason is the relatively small dataset size, which favors CNNs due to their strong inductive bias toward local features, while transformers typically require large-scale data for stable convergence [164]. Additionally, larger architectures such as ResNet152 and EfficientNetB7 underperformed, suggesting that overparameterization reduced stability given the dataset size as described in section 3.5.1 [125].

## 6.6. Future Work

Extending the current study, future work should pursue several directions to enhance the applicability and robustness of PCB classification models in circular manufacturing.

Firstly, dataset expansion and diversification are essential. Current models are limited by the scarcity of labeled burnt-PCB. Generating or collecting varied data across different board designs, burn severity levels, lighting conditions, and manufacturing variations will improve model generalization. Researchers have proposed using semi-supervised learning with data-expanding strategies, combining small, labeled datasets with larger unlabeled ones to boost detection performance visually. This method improved mAP by 4.7% on the DeepPCB dataset and could be adapted for classification tasks [171]. Also, techniques using generative models such as Stable Diffusion combined with ControlNet have demonstrated effective automatic generation of high-quality synthetic images [172].

As this study relied partly on images produced through generative AI tools and manual image-editing pipelines, it is important to validate whether these samples accurately capture the characteristics of real burn marks. Future work should explore embedding-based evaluation methods, such as t-SNE [173] or UMAP [169] applied to deep feature representations, to compare distributions of real burnt PCBs against synthetic or edited counterparts. If synthetic samples cluster closely with real burns, this would support their reliability; if not, adjustments in the generation process would be necessary. Such validation frameworks could ensure that augmented or generated datasets truly enhance, rather than mislead, model training.

Future research should incorporate explainability methods to better understand model decision-making. For CNN-based architectures, Class Activation Mapping (Grad-CAM) [168] can be applied to visualize which regions of the PCB image contribute most to a classification decision. For transformer-based models (e.g., ViT, CaiT), attention rollout

or transformer attribution techniques [174] can serve a similar purpose, revealing how self-attention layers focus on different parts of the board. These interpretability tools would help confirm whether models attend to true burnt regions or are distracted by irrelevant background features, thereby increasing trust in real-world deployment.

Additionally, a layered hybrid inspection approach could combine the strengths of multiple models. For example, YOLOv11, noted for its near-perfect recall of burnt PCBs, can be used as a primary screener, while EfficientNetB3, with its balanced classification ability, could serve as a secondary verifier for borderline cases. Moreover, literature on PCB inspection demonstrates that ensemble methods, which merge outputs from multiple CNN models (like EfficientDet, Faster R-CNN, and YOLOv5), significantly improve detection accuracy and reliability compared to any single model [71].

Moreover, model architecture modifications inspired by detection literature can further improve performance (YOLOv11[158], CDI-YOLO [159], YOLO-MBBi [160], YOLO-HMC[161] etc.) as their performance is mentioned in section 2.3.

Finally, future research should investigate integrating DL based PCB classification with robotic inspection systems controlled by reinforcement learning. Recent work in remanufacturing demonstrates that RL agents can optimize view planning by dynamically adjusting camera poses to maximize defect coverage and classification confidence, even without prior knowledge of product geometry [82]. By combining PCB classification models studies in this thesis with RL-controlled robotic arms, inspection systems could adapt in real time to varying board layouts and defect types. This would enable fully automated inspection stations where models continuously improve through feedback loops, reducing manual supervision. Such integration would represent a significant step toward intelligent, closed-loop quality control systems in circular manufacturing.

Future research should adopt an approach to systematically evaluate which factors most influence DL performance in PCB classification i.e. design of experiment. Key parameters to consider include augmentation types (rotation, blur, color, synthetic burns), model hyperparameters (learning rate, epochs, batch size, optimizer), and architectural variables (number of layers, activation functions, depth, width, etc) [175]. This structured methodology would provide a principled alternative to trial-and-error tuning, ensuring more efficient training and optimized performance under data-limited conditions.

# Chapter 7 Conclusion

This thesis advanced the role of AI in CE value addition by developing AI-based CV Visual inspection method for used PCBs to support value recovery in remanufacturing. The first major objective is the creation of a dedicated dataset, combining collected images from literature, online sources, and manually verified results with synthetic images generated using tools like ChatGPT (DALL·E), Gemini. To ensure data uniqueness and quality, perceptual hashing and deep feature filtering using a pre-trained ResNet50 model are applied to remove any duplicates and augmented images. To further expand and diversify the dataset, six augmentation pipelines, covering geometric variation, blur and noise, lighting changes, occlusion, background shifts, and compression of images, are applied to simulate real inspection conditions.

For model training, state-of-the-art DL architectures are fine-tuned on this dataset using transfer learning. Pre-trained weights from ImageNet dataset are used to classify burnt and good PCBs for these DL models. CNN-based models, particularly YOLOv11 and ResNet50, achieved the best performance with 98% accuracy and perfect precision for burnt PCB. EfficientNetB3 and EfficientNetB7 follow with perfect precision for good PCBs; however, EfficientNetB3 achieved better overall accuracy 96% than EfficientNetB7. DeiT reached competitive but lower performance (F1-score 0.92), while ResNet152 and DINO variants performed poorly, especially in detecting good PCBs, with weighted F1-scores below 0.78. These results confirm that for domain-specific and limited datasets, lightweight supervised CNN architectures remain more effective than deeper networks or Transformer-based self-supervised methods.

The final recommendation is to deploy YOLOv11 as the primary classifier for PCB visual inspection, and EfficientNetB3 as a verification step for industrial deployment to minimize misclassifications. This solution offers high accuracy, robust generalization, and real-time inspection capability for automated systems.

In conclusion, this AI-driven approach adds value to the CE by reducing dependence on manual inspection and following strict algorithm-based rules, increasing throughput, and enabling resource-efficient reuse decisions in remanufacturing. Future work should focus on expanding PCB datasets with greater diversity and exploring advanced DL architectures, explainable and trusworthy DL, integation of DL visual inspection with automated system and DOE techiniques to select optimal hyperparameters.

# Appendix A

# State-of-the-Art Models' Architectures

The detailed architectures of the DL models described in *Chapter 3 State-of-the-art* are presented here for reference.

## A.1 ResNet Architecture

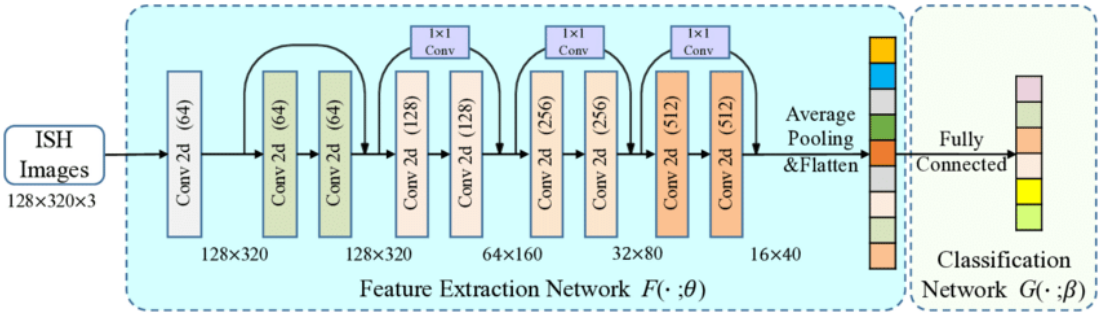



**Figure A.1:** ResNet general architecture [125]

## A.2 EfficientNet Architecture

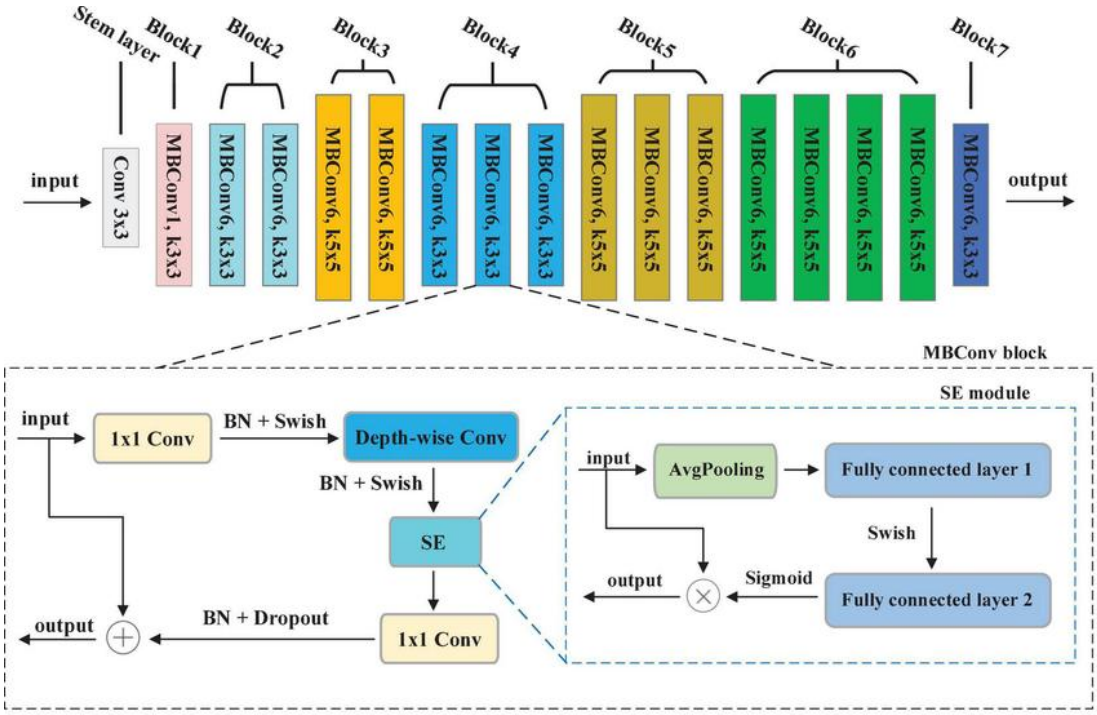



**Figure A.2:** Architecture of EfficientNetB0, MBCConv, and SE Blocks [129]

## A.3 DeiT Architecture

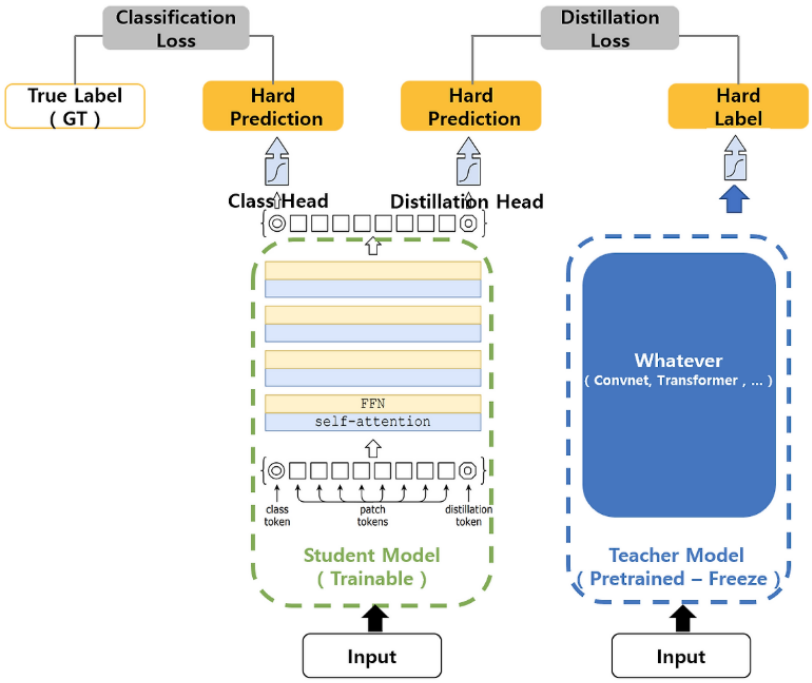



**Figure A.3:** DeiT Architecture [135]

## A.4 YOLOv11 Architecture

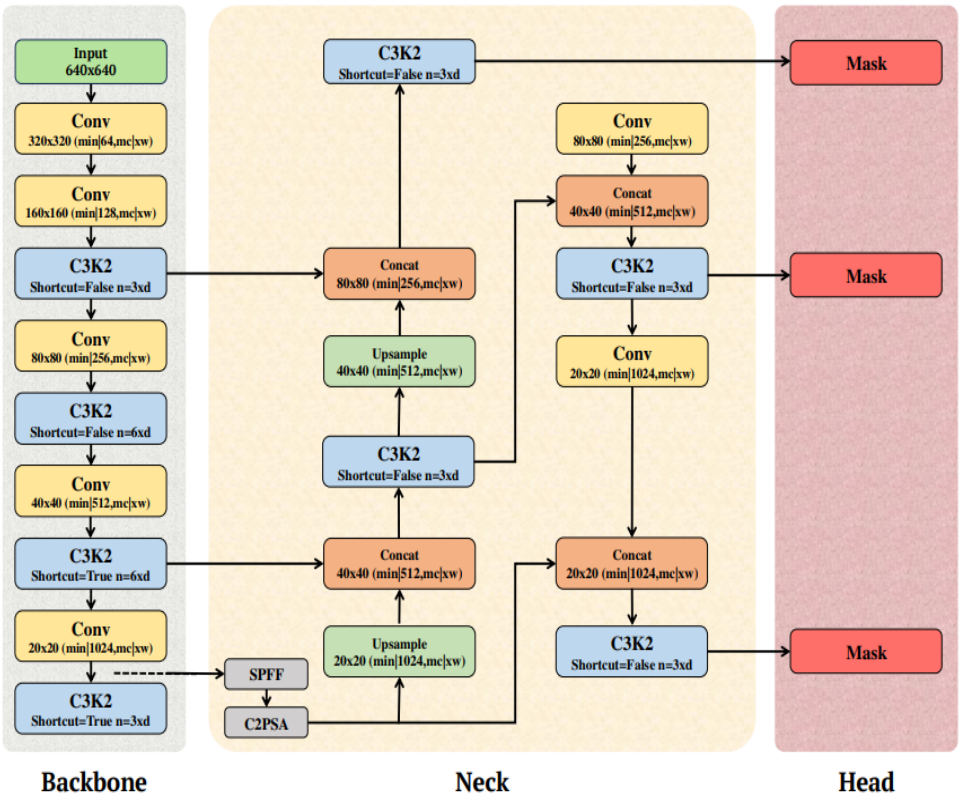



**Figure A.4:** YOLOv11 Architecture [176]

YOLOv11 architecture has SPPF, C2PSA, and C3k2 blocks. The architecture of these blocks is shown as below;

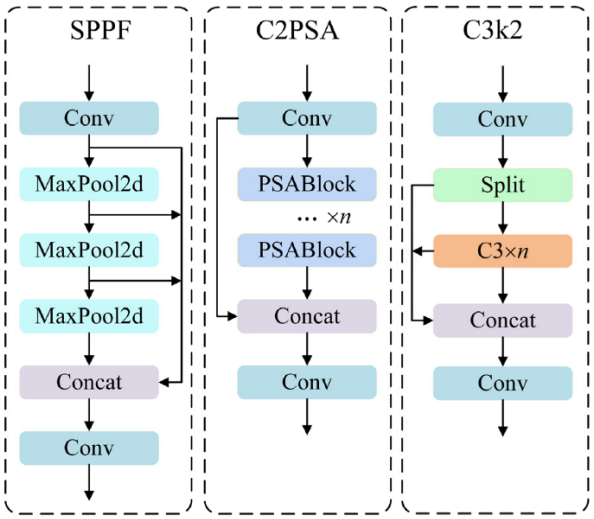



**Figure A.5:** SPPF, C2PSA and C3K2 blocks architecture [177]

## A.5 Transformer Architecture

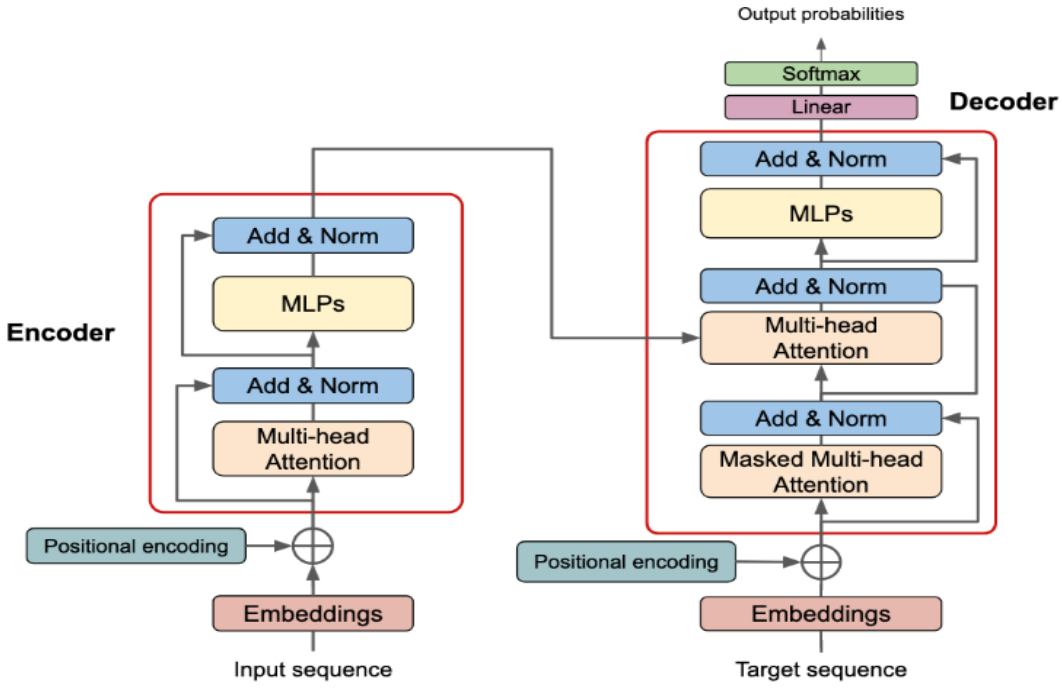



**Figure A.6:** The Transformer architecture [120]

## A.6 ViT Architecture

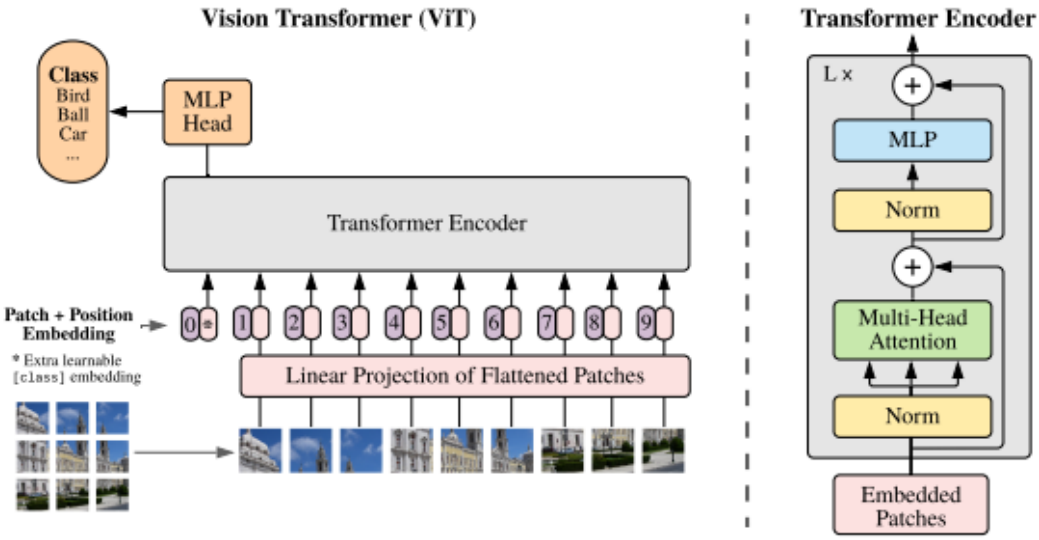
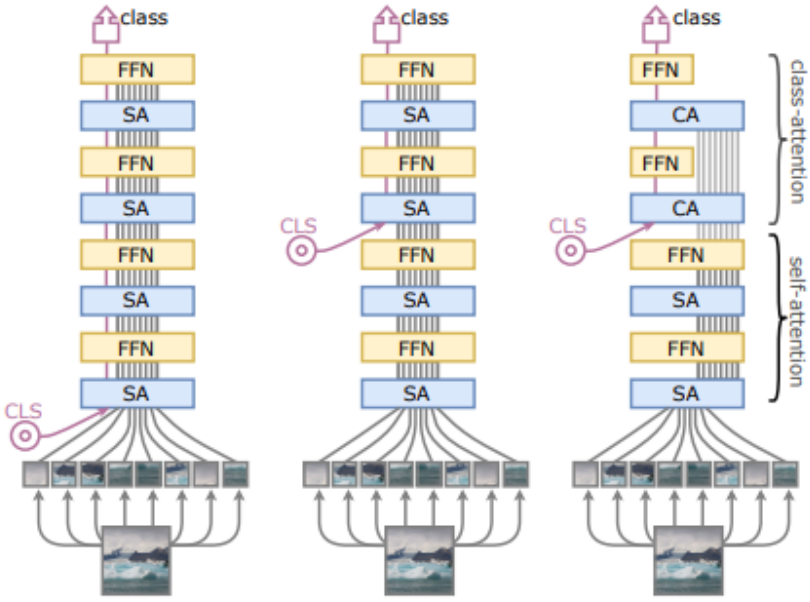



**Figure A.7:** Vision Transformer architecture [121]

## A.7 CaIT Architecture




In the ViT transformer (*left*), the class embedding (CLS) is inserted along with the patch embeddings. This choice is detrimental, as the same weights are used for two different purposes: helping the attention process, and preparing the vector to be fed to the classifier. We put this problem in evidence by showing that inserting CLS later improves performance (*middle*). In the **CaiT** architecture (*right*), we further propose to freeze the patch embeddings when inserting CLS to save compute, so that the last part of the network (typically 2 layers) is fully devoted to summarizing the information to be fed to the linear classifier.

**Figure A.8:** CaiT Architecture [178]

## A.8 DINO Architecture

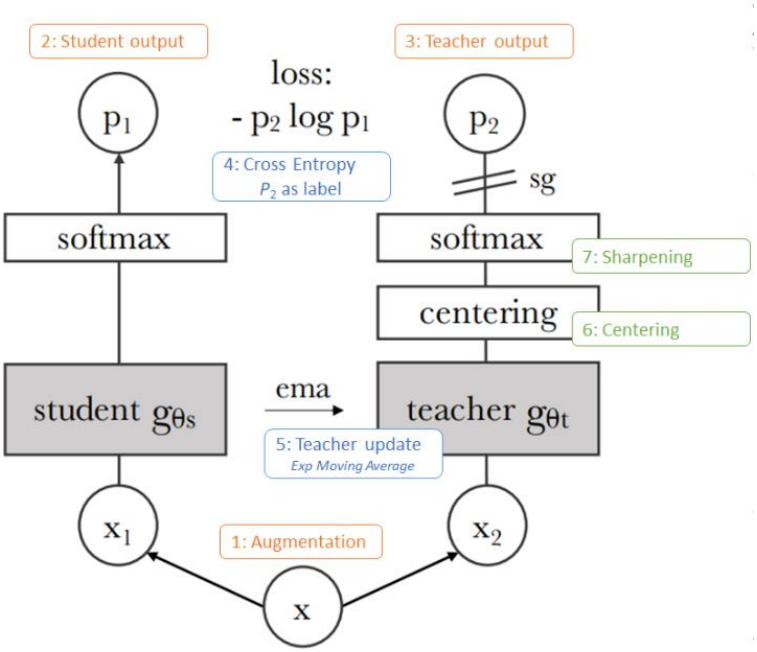



**Figure A.9:** Self-distillation (Student-Teacher framework) [179]

# Appendix B

# State-of-the-Art DL Models' Variants

Each of the models described in Chapter 3 State of the Art, and their model variants/version detail tables are described here.

## B.1 ResNet Variants

**Table B.1:** Layer Composition of Different ResNet Variants [126]

| layer name | output size | 18-layer | 34-layer | 50-layer | 101-layer | 152-layer |
|---|---|---|---|---|---|---|
| conv1 | 112×112 | 7×7, 64, stride 2 | | | | |
| | | 3×3 max pool, stride 2 | | | | |
| conv2_x | 56×56 | [3×3, 64; 3×3, 64] ×2 | [3×3, 64; 3×3, 64] ×3 | [1×1, 64; 3×3, 64; 1×1, 256] ×3 | [1×1, 64; 3×3, 64; 1×1, 256] ×3 | [1×1, 64; 3×3, 64; 1×1, 256] ×3 |
| conv3_x | 28×28 | [3×3, 128; 3×3, 128] ×2 | [3×3, 128; 3×3, 128] ×4 | [1×1, 128; 3×3, 128; 1×1, 512] ×4 | [1×1, 128; 3×3, 128; 1×1, 512] ×4 | [1×1, 128; 3×3, 128; 1×1, 512] ×8 |
| conv4_x | 14×14 | [3×3, 256; 3×3, 256] ×2 | [3×3, 256; 3×3, 256] ×6 | [1×1, 256; 3×3, 256; 1×1, 1024] ×6 | [1×1, 256; 3×3, 256; 1×1, 1024] ×23 | [1×1, 256; 3×3, 256; 1×1, 1024] ×36 |
| conv5_x | 7×7 | [3×3, 512; 3×3, 512] ×2 | [3×3, 512; 3×3, 512] ×3 | [1×1, 512; 3×3, 512; 1×1, 2048] ×3 | [1×1, 512; 3×3, 512; 1×1, 2048] ×3 | [1×1, 512; 3×3, 512; 1×1, 2048] ×3 |
| | 1×1 | average pool, 1000-d fc, softmax | | | | |
| FLOPs | | $1.8\times10^9$ | $3.6\times10^9$ | $3.8\times10^9$ | $7.6\times10^9$ | $11.3\times10^9$ |

## B.2 EfficientNet Variants

**Table B.2:** Comparison of EfficientNet Variants [180]

| Features | Top-1 Acc | Top-5 Acc | Params | FLOPs |
|---|---|---|---|---|
| EfficientNet-B0 | 77.1% | 93.3% | 5.3M | 0.39B |
| EfficientNet-B1 | 79.1% | 94.4% | 7.8M | 0.70B |
| EfficientNet-B2 | 80.1% | 94.9% | 9.2M | 1.0B |
| EfficientNet-B3 | 81.6% | 95.7% | 12M | 1.8B |
| EfficientNet-B4 | 82.9% | 96.4% | 19M | 4.2B |
| EfficientNet-B5 | 83.6% | 96.7% | 30M | 9.9B |
| EfficientNet-B6 | 84.0% | 96.8% | 43M | 19B |
| EfficientNet-B7 | 84.3% | 97.0% | 66M | 37B |

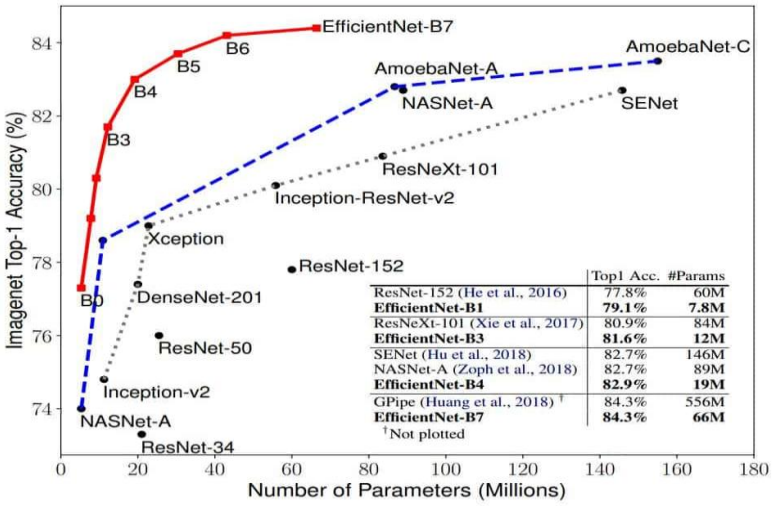



**Figure B.1:** Comparison of EfficientNet-based DL Models [181]

## B.3 DeiT Variants

**Table B.3:** Variants of DeiT architecture [132]

| Model | embedding dimension | #heads | #layers | #params | training resolution | throughput (im/sec) |
|---|---|---|---|---|---|---|
| DeiT-Ti | 192 | 3 | 12 | 5M | 224 | 2536 |
| DeiT-S | 384 | 6 | 12 | 22M | 224 | 940 |
| DeiT-B | 768 | 12 | 12 | 86M | 224 | 292 |

## B.4 YOLOv11 Variants

**Table B.4:** YOLOv11 variants comparisons[141]

| Model | size (pixels) | acc top1 | acc top5 | Speed CPU ONNX (ms) | Speed T4 TensorRT10 (ms) | params (M) | FLOPs (B) at 224 |
|---|---|---|---|---|---|---|---|
| YOLO11n-cls | 224 | 70.0 | 89.4 | 5.0 ± 0.3 | 1.1 ± 0.0 | 1.6 | 0.5 |
| YOLO11s-cls | 224 | 75.4 | 92.7 | 7.9 ± 0.2 | 1.3 ± 0.0 | 5.5 | 1.6 |
| YOLO11m-cls | 224 | 77.3 | 93.9 | 17.2 ± 0.4 | 2.0 ± 0.0 | 10.4 | 5.0 |
| YOLO11l-cls | 224 | 78.3 | 94.3 | 23.2 ± 0.3 | 2.8 ± 0.0 | 12.9 | 6.2 |
| YOLO11x-cls | 224 | 79.5 | 94.9 | 41.4 ± 0.9 | 3.8 ± 0.0 | 28.4 | 13.7 |

## B.5 SSL Methods (with CNN and ViT Backbone)

**Table B.5:** SSL Models and their backbone

| Method | Arch. | Param. | im/s | Linear | $k$-NN |
|---|---|---|---|---|---|
| Supervised | RN50 | 23 | 1237 | 79.3 | 79.3 |
| SCLR [12] | RN50 | 23 | 1237 | 69.1 | 60.7 |
| MoCov2 [15] | RN50 | 23 | 1237 | 71.1 | 61.9 |
| InfoMin [67] | RN50 | 23 | 1237 | 73.0 | 65.3 |
| BarlowT [81] | RN50 | 23 | 1237 | 73.2 | 66.0 |
| OBoW [27] | RN50 | 23 | 1237 | 73.8 | 61.9 |
| BYOL [30] | RN50 | 23 | 1237 | 74.4 | 64.8 |
| DCv2 [10] | RN50 | 23 | 1237 | 75.2 | 67.1 |
| SwAV [10] | RN50 | 23 | 1237 | **75.3** | 65.7 |
| DINO | RN50 | 23 | 1237 | **75.3** | **67.5** |
| Supervised | ViT-S | 21 | 1007 | 79.8 | 79.8 |
| BYOL* [30] | ViT-S | 21 | 1007 | 71.4 | 66.6 |
| MoCov2* [15] | ViT-S | 21 | 1007 | 72.7 | 64.4 |
| SwAV* [10] | ViT-S | 21 | 1007 | 73.5 | 66.3 |
| DINO | ViT-S | 21 | 1007 | **77.0** | **74.5** |
| *Comparison across architectures* | | | | | |
| SCLR [12] | RN50w4 | 375 | 117 | 76.8 | 69.3 |
| SwAV [10] | RN50w2 | 93 | 384 | 77.3 | 67.3 |
| BYOL [30] | RN50w2 | 93 | 384 | 77.4 | – |
| DINO | ViT-B/16 | 85 | 312 | 78.2 | 76.1 |
| SwAV [10] | RN50w5 | 586 | 76 | 78.5 | 67.1 |
| BYOL [30] | RN50w4 | 375 | 117 | 78.6 | – |
| BYOL [30] | RN200w2 | 250 | 123 | 79.6 | 73.9 |
| DINO | ViT-S/8 | 21 | 180 | 79.7 | **78.3** |
| SCLRv2 [13] | RN152w3+SK | 794 | 46 | 79.8 | 73.1 |
| DINO | ViT-B/8 | 85 | 63 | **80.1** | 77.4 |

# Appendix C

# Hardware Setup and Python Libraries

## C.1 Hardware & Software Environment

**Table C.1:** Hardware and software specifications used for model training

| Component | Specification |
|---|---|
| **CPU** | Intel(R) Core(TM) i9-14900KF, 24 cores, 32 Logical Processors, 3.2 GHz |
| **RAM** | 128 GB (130,915 MB) |
| **GPU** | 2 × NVIDIA GeForce RTX 4080 SUPER (16 GB VRAM each) |
| **PyTorch + CUDA** | torch 2.7.1+cu128 (CUDA 12.8) |
| **Python Version** | 3.12.7 |
| **Operating System** | Microsoft Windows 11 Enterprise LTSC |

## C.2 Python Libraries

**Table C.2:** Python libraries with version

| Step | Libraries (with Versions) |
|---|---|
| **Data Processing & Pre-Filtering** | numpy 1.26.4, pandas 2.2.3, Pillow 11.1.0, opencv-python 4.12.0.88, ImageHash 4.3., tensorflow 2.19.0, keras 3.10.0, scikit-learn 1.6.1, os (stdlib), shutil (stdlib) |
| **Data Augmentation** | albumentations 2.0.8, opencv-python 4.12.0.88, tqdm 4.67.1, torchvision 0.22.1+cu128, uuid (stdlib) |
| **Dataset Splitting & Loading** | torch 2.7.1+cu128, torchvision 0.22.1+cu128, scikit-learn 1.6.1, os (stdlib), random (stdlib), shutil (stdlib) |
| **Model Training** | torch 2.7.1+cu128, torchvision 0.22.1+cu128, timm 1.0.17, ultralytics 8.3.171, tqdm 4.67.1, time (stdlib), copy (stdlib), math (stdlib) |
| **Evaluation & Metrics** | scikit-learn 1.6.1, torchmetrics 1.7.4, matplotlib 3.10.0, seaborn 0.13.2, pandas 2.2.3 |
| **Visualization** | matplotlib 3.10.0, seaborn 0.13.2 |
| **Logging & Export** | pandas 2.2.3, openpyxl 3.1.5, csv (stdlib), time (stdlib), os (stdlib) |

# Appendix D

# Python Codes

## D.1 Duplicate Image Filtering using Imageha

**Code 4.2.1: Duplicate Image Filtering using Imageha** python

```python
import os
import shutil
from PIL import Image
import imagehash

# Input and output folders
image_folder = 'Roboflow All'
output_folder = 'Roboflow All Removed Duplicate'

# Create the output folder if it doesn't exist
os.makedirs(output_folder, exist_ok=True)

# Dictionary to store unique image hashes
hash_dict = {}

# Set a similarity threshold (lower = stricter)
similarity_threshold = 2

# Loop through all images
for filename in os.listdir(image_folder):
    if filename.lower().endswith(('.png', '.jpg', '.jpeg', '.bmp', '.gif')):
        filepath = os.path.join(image_folder, filename)
        try:
            img = Image.open(filepath)
            hash_value = imagehash.phash(img)

            is_duplicate = False
            for existing_hash in hash_dict:
                distance = hash_value - existing_hash
                if distance <= similarity_threshold:
                    is_duplicate = True
                    print(f"Skipping duplicate: {filename} (similar to {hash_dict[existing_hash]})")
                    break
            if not is_duplicate:
                hash_dict[hash_value] = filename
                shutil.copy(filepath, os.path.join(output_folder, filename))
                print(f"Saved unique: {filename}")
        except Exception as e:
            print(f"Error processing {filename}: {e}")
```

## D.2 Duplicate Image Filtering using ResNeT50

**Code 4.2.2: Duplicate Image Filtering using ResNeT50** python

```python
import os
import cv2
import numpy as np
import shutil
from tensorflow.keras.applications.resnet50 import ResNet50, preprocess_input
from tensorflow.keras.preprocessing import image
from sklearn.metrics.pairwise import cosine_similarity

# Load pre-trained ResNet50 model (without top layer)
model = ResNet50(weights='imagenet', include_top=False, pooling='avg')

# Set your folders (adjust names if needed)
input_folder = 'Roboflow All_Removed Duplicate'
output_folder = 'Roboflow Filtered Unique Image'

# Create output folder if it doesn't exist
os.makedirs(output_folder, exist_ok=True)

# Helper: extract feature vector using ResNet
def get_feature_vector(img_path):
    try:
        img = image.load_img(img_path, target_size=(224, 224))
        x = image.img_to_array(img)
        x = np.expand_dims(x, axis=0)
        x = preprocess_input(x)
        features = model.predict(x, verbose=0)
        return features.flatten()
    except Exception as e:
        print(f"Error loading {img_path}: {e}")
        return None

# Track saved image features
image_vectors = []

# Set similarity threshold (0.95 = very similar)
similarity_threshold = 0.95

#  Loop through all images
for filename in os.listdir(input_folder):
    if filename.lower().endswith(('.jpg', '.jpeg', '.png', '.bmp')):
        filepath = os.path.join(input_folder, filename)
        vec = get_feature_vector(filepath)

        if vec is None:
            continue
        is_duplicate = False
```

```
    # Compare with all previously saved vectors
    for existing_vec in image_vectors:
        similarity = cosine_similarity([vec], [existing_vec])[0][0]
        if similarity >= similarity_threshold:
            print(f"Skipped duplicate: {filename} (similarity: {similarity:.3f})")
            is_duplicate = True
            break

    # If not duplicate, save it
    if not is_dupliate:
        image_vectors.append(vec)

        # To ensure filename is safe
        safe_filename = os.path.basename(filename)
        dst_path = os.path.join(output_folder, safe_filename)

        try:
            shutil.copy(filepath, dst_path)
            print(f"Saved unique: {safe_filename}")
        except Exception as e:
            print(f"Error copying {safe_filename}: {e}")
```

## D.3 Dataset Splitting for Model Training

**Code 4.3.1: Dataset Splitting for Model Training** python

```
import os
import random
import shutil

src_folder = 'PCB_Burnt'      # Source folder with all images
val_folder = 'val_PCB_Burnt'      # Folder to move 25% randomly selected images

# Make sure val folder exists
os.makedirs(val_folder, exist_ok=True)

# List all images and randomly pick 100 images
all_images = os.listdir(src_folder)
# val_images = random.sample(all_images, 100)

# Move selected images to val folder
for img in val_images:
    shutil.move(os.path.join(src_folder, img), os.path.join(val_folder, img))

print(f"Moved {len(val_images)} images to {val_folder}")
```

## D.4 Image Augmentation

**Code 4.4.1: Image Augmentation** python

```
import os, cv2, albumentations as A
from tqdm import tqdm
import uuid

INPUT_FOLDER = "Final Dataset/train/Good_PCB"
OUTPUT_FOLDER = "Final Dataset Aug/train/Good_PCB"
NUM_AUGMENTED_COPIES = 5
os.makedirs(OUTPUT_FOLDER, exist_ok=True)

image_files = [f for f in os.listdir(INPUT_FOLDER) if f.lower().endswith(('.png', '.jpg', '.jpeg'))]

NUM_IMAGES = len(image_files)
NUM_PIPELINES = 7
COPIES_PER_PIPELINE = {"aug1": NUM_AUGMENTED_COPIES,"aug2":
NUM_AUGMENTED_COPIES,"aug3": NUM_AUGMENTED_COPIES,"aug4":
NUM_AUGMENTED_COPIES,"aug5": NUM_AUGMENTED_COPIES,"aug6":
NUM_AUGMENTED_COPIES,"aug7": 20}
TOTAL_STEPS = NUM_IMAGES * sum(COPIES_PER_PIPELINE.values())

progress_bar = tqdm(total=TOTAL_STEPS, desc="Applying all augmentations")

for filename in image_files:
   image_path = os.path.join(INPUT_FOLDER, filename)
   image = cv2.cvtColor(cv2.imread(image_path), cv2.COLOR_BGR2RGB)
   h, w, _ = image.shape

# Geometric transformation
   transforms_1 = A.Compose([
            A.PadIfNeeded(min_height=int(h*1.7), min_width=int(w*1.7),
border_mode=cv2.BORDER_CONSTANT, fill=[255,255,255], p=1.0),
            A.Rotate(limit=[-90,90], border_mode=cv2.BORDER_CONSTANT, fill=[255,255,255],
p=1),
            A.HorizontalFlip(p=0.5),
            A.VerticalFlip(p=0.5),
            A.Transpose(p=0.5)])

# Blur and Noise Augmentation
    transforms_2 = A.Compose([
            A.Blur(blur_limit=[3,7], p=0.5),
            A.GaussianBlur(blur_limit=(3,7), p=0.5),
            A.GaussNoise(std_range=(0.1,0.2), noise_scale_factor=1, p=0.5)])

# Colour Transformation
transforms_3 = A.Compose([A.RandomBrightnessContrast(p=0.7),
```

```
A.HueSaturationValue(p=0.7), A.CLAHE(p=0.3),  A.RGBShift(p=0.7), A.Sharpen(alpha=(0.5,0.9),
lightness=(0.5,1.0), method="kernel", kernel_size=5, sigma=1, p=0.5)])

# Occlusion & Shadows
 transforms_4 = A.Compose([
            A.CoarseDropout(num_holes_range=(1,2), hole_height_range=(int(h*0.05),
int(h*0.1)), hole_width_range=(int(w*0.05), int(w*0.1)), fill=225, p=0.5),
            A.CoarseDropout(num_holes_range=(1,2), hole_height_range=(int(h*0.05),
int(h*0.1)), hole_width_range=(int(w*0.05), int(w*0.1)), p=0.5),
            A.RandomShadow(shadow_roi=[0,0,1,1], num_shadows_limit=[1,3],
shadow_intensity_range=[0.2,0.4], p=1)])

# Background Variation
  transforms_5 = A.Compose([
            A.PadIfNeeded(min_height=int(h*1.3), min_width=int(w*1.3), position="random",
border_mode=cv2.BORDER_CONSTANT, fill=[255,255,255], p=1.0)])

# Compression & Sensor downscaling
   transforms_6 = A.Compose([
            A.ImageCompression(quality_range=(20,50), p=1),
            A.Downscale(scale_range=(0.2,0.5), p=1)])

# Mixed Augmentation Pipeline
   transforms_7 = A.Compose([
            A.PadIfNeeded(min_height=int(h*1.7), min_width=int(w*1.7),
border_mode=cv2.BORDER_CONSTANT, fill=[255,255,255], p=1.0),
            A.Rotate(limit=[-90,90], border_mode=cv2.BORDER_CONSTANT, fill=[255,255,255],
p=1),
            A.HorizontalFlip(p=0.5), A.Blur(blur_limit=[3,7], p=0.5),
            A.GaussianBlur(blur_limit=(3,7), p=0.5),
            A.GaussNoise(std_range=(0.1,0.2), noise_scale_factor=1, p=0.5),
            A.RandomBrightnessContrast(p=0.5),
            A.HueSaturationValue(p=0.5),
            A.Sharpen(alpha=(0.2,0.5), lightness=(0.5,1.0), p=0.5),
            A.RandomShadow(shadow_roi=[0,0,1,1], num_shadows_limit=[1,2],
shadow_intensity_range=[0.2,0.3], p=0.5),
            A.ImageCompression(quality_range=(20,50), p=0.5)])

   transformers = {"aug1": transforms_1,"aug2": transforms_2,"aug3": transforms_3,"aug4":
transforms_4,"aug5": transforms_5,"aug6": transforms_6,"aug7": transforms_7}

   for key, transform in transformers.items():
      for i in range(COPIES_PER_PIPELINE[key]):
         augmented = transform(image=image)
         aug_image_bgr = cv2.cvtColor(augmented["image"], cv2.COLOR_RGB2BGR)

         unique_id = uuid.uuid4().hex[:8]
         new_filename = f"{os.path.splitext(filename)[0]}_{key}_{i}_{unique_id}.jpg"
         cv2.imwrite(os.path.join(OUTPUT_FOLDER, new_filename), aug_image_bgr)
         progress_bar.update(1)
```

## D.5 Model Training Yolov11

**Code 4.5.1: Model Training Overview** python

```
!pip install ultralytics
from ultralytics import YOLO

# Load a YOLOv11 classification model
model = YOLO('yolo11x-cls.pt')

# Train the model
results = model.train(
    data='Training AI Model Aug',
    epochs=50,
    imgsz=224,        # imgsz 224 YOLOv11 standard for classification
    batch=8,
    workers=0
)
```

## D.6 Model Training ResNet50/ResNet152

For ResNet152, the same training pipeline as that used for ResNet50 as below. The only difference lies in model initialization: resnet50 is replaced by resnet152, along with the corresponding pretrained weights (ResNet152_Weights.DEFAULT)

**Code 4.5.2: Model Training ResNet50/ResNet152** python

```
import os
import time
import torch
import torch.nn as nn
import pandas as pd
import matplotlib.pyplot as plt
from collections import Counter
from torchvision import datasets, transforms
from torchvision.models import resnet50, ResNet50_Weights    # For Resnet152, replace it
from torch.utils.data import DataLoader
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, classification_report,
accuracy_score
from tqdm import tqdm
from torch.optim.lr_scheduler import CosineAnnealingLR

# imagenet normalization
mean = [0.485, 0.456, 0.406]
std  = [0.229, 0.224, 0.225]

# transforms for train, val, test
transform = { 'train': transforms.Compose([ transforms.Resize((128, 128)),
transforms.ToTensor(), transforms.Normalize(mean=mean, std=std)]),

   'val': transforms.Compose([ transforms.Resize((128, 128)), transforms.ToTensor(),
transforms.Normalize(mean=mean, std=std)]),

   'test': transforms.Compose( transforms.Resize((128, 128)), transforms.ToTensor(),
transforms.Normalize(mean=mean, std=std)])}

# load datasets
data_dir = "Final Dataset Aug"
image_datasets = { x: datasets.ImageFolder(os.path.join(data_dir, x), transform[x])
   for x in ['train', 'val', 'test']}
dataloaders = { x: DataLoader(image_datasets[x], batch_size=32, shuffle=True,
num_workers=8)
   for x in ['train', 'val', 'test']}
# class names
class_names = image_datasets['train'].classes
print("Classes:", class_names)
```

```
# count images
for split in ['train', 'val', 'test']:
    class_counts = Counter()
    for _, label in image_datasets[split].imgs:
        class_counts[class_names[label]] += 1
    print(f"\n{split.upper()} Set Image Count:")
    for cls in class_names:
        print(f"  {cls}: {class_counts[cls]}")

# load resnet50 and modify last layer
weights = ResNet50_Weights.DEFAULT                    # For Resnet152, replace it ResNet152
model = resnet50(weights=weights)                     # For Resnet152, replace it ResNet152
model = resnet50(weights=weights)
num_ftrs = model.fc.in_features
model.fc = nn.Linear(num_ftrs, 2)
model = model.to(device)

# loss, optimizer, scheduler
criterion = nn.CrossEntropyLoss()
num_epochs = 50
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-4)
scheduler = CosineAnnealingLR(optimizer, T_max=num_epochs, eta_min=1e-6)

# top-k accuracy function
def top_k_accuracy(output, target, k=1):
    with torch.no_grad():
        _, pred = output.topk(k, 1, True, True)
        pred = pred.t()
        correct = pred.eq(target.view(1, -1).expand_as(pred))
        return correct[:k].reshape(-1).float().sum(0) / target.size(0)

# training function
def train_model(model, criterion, optimizer, scheduler=None, num_epochs=50):
    records = []
    best_val_acc = -1.0
    total_start = time.time()

    for epoch in range(num_epochs):
        epoch_start = time.time()
        print(f"\nEpoch {epoch + 1}/{num_epochs}")

        for phase in ['train', 'val']:
            model.train() if phase == 'train' else model.eval()
            running_loss = 0.0
            running_corrects = 0
            running_top2_sum = 0.0

            for inputs, labels in tqdm(dataloaders[phase], desc=f"{phase.capitalize()} Phase",
leave=False):
                inputs = inputs.to(device)
                labels = labels.to(device)
                optimizer.zero_grad()
```

```
 with torch.set_grad_enabled(phase == 'train'):
                outputs = model(inputs)
                _, preds = torch.max(outputs, 1)
                loss = criterion(outputs, labels)
                if phase == 'train':
                    loss.backward()
                    optimizer.step()

            batch_size = inputs.size(0)
            running_loss += loss.item() * batch_size
            running_corrects += torch.sum(preds == labels.data).item()
            running_top2_sum += (top_k_accuracy(outputs, labels, k=2).item() * batch_size)
            epoch_loss = running_loss / len(image_datasets[phase])
            top1_acc = running_corrects / len(image_datasets[phase])
            top2_acc = running_top2_sum / len(image_datasets[phase])

        print(f"{phase.capitalize()} Loss: {epoch_loss:.4f} | Top-1 Acc: {top1_acc:.4f} | Top-2
Acc: {top2_acc:.4f}")
        records.append({ 'Epoch': epoch + 1, 'Phase': phase, 'Loss': epoch_loss, 'Top1_Accuracy':
top1_acc, 'Top2_Accuracy': top2_acc, 'Learning_Rate': optimizer.param_groups[0]['lr']})

# save best on validation accuracy
        if phase == 'val':
            val_acc = float(top1_acc)
            if val_acc > best_val_acc:
                best_val_acc = val_acc
                torch.save(model.state_dict(), best_model_path)
                print(f"Saved best model to {best_model_path} (val acc {best_val_acc:.4f})")

    epoch_time = time.time() - epoch_start
    print(f"Epoch Time: {epoch_time:.2f}s")
    for i in [-1, -2]:
        records[i]['Epoch_Time_sec'] = epoch_time

    if torch.cuda.is_available():
        mem = torch.cuda.memory_allocated(device) / 1024**2
        peak = torch.cuda.max_memory_allocated(device) / 1024**2
        print(f"GPU Mem Used: {mem:.1f} MB | Peak: {peak:.1f} MB")
        torch.cuda.reset_peak_memory_stats(device)

    if scheduler is not None:
        scheduler.step()
  total_time = time.time() - total_start
  print(f"\nTotal Training Time: {total_time:.2f}s ({total_time/60:.2f} min)")
  df = pd.DataFrame(records)
  df.to_excel("training_results.xlsx", index=False)
  print("Saved: training_results.xlsx")
  torch.save(model.state_dict(), "resnet50_pcb_model.pth")
  print("Model saved as resnet50_pcb_model.pth")
  return model, df

# Train the model
model, df = train_model(model, criterion, optimizer, scheduler=scheduler,
num_epochs=num_epochs)
```

## D.7 Model Training EfficientNetB3/EfficientNetB7

For EfficientNetB7, the same training pipeline as that used for EfficientNetB7 is as follows. The only difference lies in model initialization: EfficientNetB3 is replaced by EfficientNetB7, along with the corresponding pretrained weights.

**Code 4.5.3: Model Training EfficientNetB3/EfficientB7** python

```python
import os
import time
import torch
import torch.nn as nn
import pandas as pd
from tqdm import tqdm
import torchvision.transforms as transforms
from torchvision import datasets
from torchvision.models import efficientnet_b3, EfficientNet_B3_Weights
from torch.optim.lr_scheduler import CosineAnnealingLR

# dataset paths
data_dir = "Final Dataset Aug"

# transforms (EfficientNet-B3 input size 300x300)
transform = transforms.Compose([
    transforms.Resize((300, 300)),
    transforms.ToTensor(),
])

# datasets
train_data = datasets.ImageFolder(root=os.path.join(data_dir, "train"), transform=transform)
val_data   = datasets.ImageFolder(root=os.path.join(data_dir, "val"),   transform=transform)
test_data  = datasets.ImageFolder(root=os.path.join(data_dir, "test"),  transform=transform)

# dataloaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=32, shuffle=True,
num_workers=8)
val_loader   = torch.utils.data.DataLoader(val_data,   batch_size=32, shuffle=False,
num_workers=8)
test_loader  = torch.utils.data.DataLoader(test_data,  batch_size=32, shuffle=False,
num_workers=8)

# model init with pretrained weights and replaceclssifier hed for 2 classes
weights = EfficientNet_B3_Weights.IMAGENET1K_V1
model = efficientnet_b3(weights=weights)
num_ftrs = model.classifier[1].in_features
model.classifier[1] = nn.Linear(num_ftrs, 2)
```

```python
# training loop with cosine annealing lr
def train_model(model, criterion, optimizer, num_epochs=50):
    scheduler = CosineAnnealingLR(optimizer, T_max=num_epochs, eta_min=1e-6)
    results = []
    best_val_acc = 0.0
    best_model_wts = None
    total_start = time.time()

    for epoch in range(num_epochs):
        epoch_start = time.time()
        model.train()
        running_loss, correct, total = 0, 0, 0

        print(f"\nEpoch {epoch+1}/{num_epochs}")
        train_loop = tqdm(train_loader, desc="Training", leave=False)

        for images, labels in train_loop:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            running_loss += loss.item() * images.size(0)
            _, preds = torch.max(outputs, 1)
            correct += (preds == labels).sum().item()
            total += labels.size(0)

            train_acc = correct / total
            train_loss = running_loss / total
            train_loop.set_postfix(loss=train_loss, acc=train_acc)

        train_loss = running_loss / len(train_loader.dataset)
        train_acc = correct / total

        # validation
        model.eval()
        val_loss, val_correct, val_total = 0, 0, 0
        val_loop = tqdm(val_loader, desc="Validation", leave=False)

        with torch.no_grad():
            for images, labels in val_loop:
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                loss = criterion(outputs, labels)

                val_loss += loss.item() * images.size(0)
                _, preds = torch.max(outputs, 1)
                val_correct += (preds == labels).sum().item()
                val_total += labels.size(0)
```

```
            val_acc = val_correct / val_total
            val_avg_loss = val_loss / val_total
            val_loop.set_postfix(loss=val_avg_loss, acc=val_acc)

        val_loss = val_loss / len(val_loader.dataset)
        val_acc = val_correct / val_total
        epoch_time = time.time() - epoch_start
        current_lr = optimizer.param_groups[0]['lr']

        print(f"Epoch {epoch+1}/{num_epochs} | Train Loss: {train_loss:.4f} | Val Loss:
{val_loss:.4f} | " f"Train Acc: {train_acc:.4f} | Val Acc: {val_acc:.4f} | LR: {current_lr:.8f}")

        # keep best weights by val accuracy
        if val_acc > best_val_acc:
            best_val_acc = val_acc
            best_model_wts = model.state_dict()

        # record logs
        results.append({
            "epoch": epoch + 1,
            "train_loss": train_loss,
            "val_loss": val_loss,
            "train_acc": train_acc,
            "val_acc": val_acc,
            "learning_rate": current_lr,
            "epoch_time_sec": epoch_time })
        scheduler.step()

    total_time = time.time() - total_start
    print(f"\nTotal Training Time: {total_time:.2f}s ({total_time / 60:.2f} min)")

    # save last and best models
    torch.save(model.state_dict(), "efficientnet_b3_last_model.pth")
    print("Saved last model: efficientnet_b3_last_model.pth")
    if best_model_wts:
        torch.save(best_model_wts, "efficientnet_b3_best_model.pth")
        print("Saved best model (by val acc): efficientnet_b3_best_model.pth")

    # save training logs
    df = pd.DataFrame(results)
    df.to_excel("efficientnet_b3_training(LR)_results.xlsx", index=False)
    print("Saved: efficientnet_b3_training(LR)_results.xlsx")

    return model, df

# loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-4)

# train
model, results_df = train_model(model, criterion, optimizer, num_epochs=50)
```

## D.8 Model Training DeiT

**Code 4.5.4: Model Training DeiT** python

```python
import os
import torch
import torch.nn as nn
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import timm
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report, ConfusionMatrixDisplay
import numpy as np
from torch.optim.lr_scheduler import CosineAnnealingLR

# transforms
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.5], [0.5])])

# data
data_dir =  "Final Dataset Aug"
train_dataset = datasets.ImageFolder(os.path.join(data_dir, 'train'), transform=transform)
val_dataset   = datasets.ImageFolder(os.path.join(data_dir, 'val'),   transform=transform)
test_dataset  = datasets.ImageFolder(os.path.join(data_dir, 'test'),  transform=transform)

# loaders
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True,  num_workers=8)
val_loader   = DataLoader(val_dataset,   batch_size=32, shuffle=False, num_workers=8)
test_loader  = DataLoader(test_dataset,  batch_size=32, shuffle=False, num_workers=8)

# model
model = timm.create_model('deit_base_patch16_224', pretrained=True)
model.head = nn.Linear(model.head.in_features, 2)
model.to(device)

# epochs, loss, optimizer, scheduler
num_epochs = 50
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-4)
scheduler = CosineAnnealingLR(optimizer, T_max=num_epochs, eta_min=1e-6)

# tracking
train_losses, val_losses = [], []
train_accs, val_accs = [], []
lr_values = []
```

```
# train
from tqdm import tqdm

for epoch in range(num_epochs):
    model.train()
    train_loss, correct_train, total_train = 0.0, 0, 0
    train_loop = tqdm(train_loader, desc=f"Training Epoch {epoch+1}/{num_epochs}",
leave=True)

    for inputs, labels in train_loop:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
        loss = criterion(outputs, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        train_loss += loss.item() * inputs.size(0)
        _, preds = torch.max(outputs, 1)
        correct_train += (preds == labels).sum().item()
        total_train += labels.size(0)

        train_loop.set_postfix(loss=loss.item())

    epoch_train_loss = train_loss / len(train_loader.dataset)
    epoch_train_acc  = correct_train / total_train
    train_losses.append(epoch_train_loss)
    train_accs.append(epoch_train_acc)

# validation
    model.eval()
    val_loss, correct_val, total_val = 0.0, 0, 0
    val_loop = tqdm(val_loader, desc=f"Validation Epoch {epoch+1}/{num_epochs}",
leave=True)

    with torch.no_grad():
        for inputs, labels in val_loop:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            val_loss += loss.item() * inputs.size(0)
            _, preds = torch.max(outputs, 1)
            correct_val += (preds == labels).sum().item()
            total_val += labels.size(0)

            val_loop.set_postfix(loss=loss.item())

    epoch_val_loss = val_loss / len(val_loader.dataset)
    epoch_val_acc  = correct_val / total_val
    val_losses.append(epoch_val_loss)
    val_accs.append(epoch_val_acc)
```

```
# lr track and print
    current_lr = optimizer.param_groups[0]['lr']
    lr_values.append(current_lr)
    print(f"Epoch {epoch+1}/{num_epochs}, Train Loss: {epoch_train_loss:.4f}, Val Loss:
{epoch_val_loss:.4f}, "
          f"Train Acc: {epoch_train_acc:.4f}, Val Acc: {epoch_val_acc:.4f}, LR: {current_lr:.8f}")

    # checkpoints
    if (epoch + 1) % 10 == 0:
        model_path = f"DeiT_Base_Results/deit_epoch_{epoch+1}.pth"
        torch.save(model.state_dict(), model_path)
        print(f"Model saved at epoch {epoch+1}: {model_path}")

    if (epoch + 1) == num_epochs:
        final_model_path = "DeiT_Base_Results/deit_model_final.pth"
        torch.save(model.state_dict(), final_model_path)
        print(f"Final model saved at: {final_model_path}")

    # step scheduler
    scheduler.step()

# save results to Excel
results_df = pd.DataFrame({
    "Epoch": list(range(1, len(train_losses)+1)),
    "Train_Loss": train_losses,
    "Val_Loss": val_losses,
    "Train_Acc": train_accs,
    "Val_Acc": val_accs,
    "Learning_Rate": lr_values
})

# To ensure output dir exists before saving
os.makedirs("Results", exist_ok=True)
results_df.to_excel("Results/Deit_results.xlsx", index=False)
print("Training log saved to: Results/Deit_results.xlsx")
```

## D.9 Model Training DINO-ViT

**Code 4.5.5: Model Training DeiT** python

```python
import os
import time
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision.transforms as transforms
from torchvision import datasets
from torch.utils.data import DataLoader
import timm
from tqdm import tqdm
import pandas as pd

# DINO-style SSL pretraining
# strong augmentations
ssl_transform = transforms.Compose([
    transforms.RandomResizedCrop(224, scale=(0.4, 1.0)),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(0.4, 0.4, 0.4, 0.1),
    transforms.RandomGrayscale(p=0.2),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))])

# unlabeled training data path
train_path_ssl = "Final Dataset Aug/train"
ssl_dataset = datasets.ImageFolder(train_path_ssl, transform=ssl_transform)
ssl_loader = DataLoader(ssl_dataset, batch_size=64, shuffle=True, num_workers=4,
drop_last=True)

# DINO projector head
class DINOHead(nn.Module):
    def __init__(self, in_dim, out_dim, nlayers=3, hidden_dim=2048, bottleneck_dim=256):
        super().__init__()
        layers = []
        for i in range(nlayers):
            dim_in = in_dim if i == 0 else hidden_dim
            dim_out = bottleneck_dim if i == nlayers - 1 else hidden_dim
            layers.append(nn.Linear(dim_in, dim_out))
            if i < nlayers - 1:
                layers.append(nn.BatchNorm1d(dim_out))
                layers.append(nn.GELU())
        self.mlp = nn.Sequential(*layers)
        self.last_layer = nn.utils.weight_norm(nn.Linear(bottleneck_dim, out_dim, bias=False))
        self.last_layer.weight_g.data.fill_(1.0)
        self.last_layer.weight_g.requires_grad = False
```

```
    def forward(self, x):
        x = self.mlp(x)
        x = F.normalize(x, dim=-1)
        return self.last_layer(x)

# DINO loss
class DINOLoss(nn.Module):
    def __init__(self, out_dim, teacher_temp=0.04, student_temp=0.1,
center_momentum=0.9):
        super().__init__()
        self.teacher_temp = teacher_temp
        self.student_temp = student_temp
        self.center_momentum = center_momentum
        self.register_buffer("center", torch.zeros(1, out_dim))

    def forward(self, student_output, teacher_output):
        student_out = student_output / self.student_temp
        centered_teacher = (teacher_output - self.center.to(teacher_output.device)) /
self.teacher_temp
        teacher_out = F.softmax(centered_teacher, dim=-1).detach()
        loss = -torch.sum(teacher_out * F.log_softmax(student_out, dim=-1), dim=-1).mean()
        self.center = self.center.to(teacher_output.device)
        self.center = self.center * self.center_momentum + (1 - self.center_momentum) *
teacher_output.mean(dim=0, keepdim=True)
        return loss

# DiNO Backbone (ViT)
def get_vit_backbone():
    model = timm.create_model('vit_small_patch16_224', pretrained=True)
    return model

# DINO student-teacher wrapper
class DINOModel(nn.Module):
    def __init__(self, out_dim=65536):
        super().__init__()
        self.backbone = get_vit_backbone()
        self.teacher_backbone = get_vit_backbone()
        self.student_head = DINOHead(384, out_dim)
        self.teacher_head = DINOHead(384, out_dim)
        for p in self.teacher_backbone.parameters():
            p.requires_grad = False

    def forward_student(self, x):
        tokens = self.backbone.forward_features(x)
        cls_token = tokens[:, 0]
        return self.student_head(cls_token)

    def forward_teacher(self, x):
        with torch.no_grad():
            tokens = self.teacher_backbone.forward_features(x)
            cls_token = tokens[:, 0]
            return self.teacher_head(cls_token)
```

```
# init model, loss, optimizer
model = DINOModel().to(device)
criterion = DINOLoss(out_dim=65536)
optimizer = torch.optim.AdamW(model.backbone.parameters(), lr=3e-4)
epochs_ssl = 50

# cosine-like momentum schedule for EMA teacher update
momentum_schedule = lambda epoch: 0.996 + (1 - 0.996) * (1 + torch.cos(torch.tensor(epoch
* 3.14 / epochs_ssl))) / 2

# logs
records = []
total_start_time = time.time()

# train SSL
for epoch in range(epochs_ssl):
    epoch_start_time = time.time()
    model.train()
    running_loss = 0.0

    for images, _ in tqdm(ssl_loader, desc=f"Epoch {epoch+1}/{epochs_ssl}"):
        images = images.to(device)
        student_out = model.forward_student(images)
        teacher_out = model.forward_teacher(images)
        loss = criterion(student_out, teacher_out)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        model.update_teacher(momentum_schedule(epoch).item())
        running_loss += loss.item()

    avg_loss = running_loss / len(ssl_loader)
    epoch_duration = time.time() - epoch_start_time
    print(f"Epoch {epoch+1}/{epochs_ssl} | Loss: {avg_loss:.4f} | Time: {epoch_duration:.2f}
sec")
    records.append({'Epoch': epoch + 1, 'Loss': avg_loss, 'Time (sec)': epoch_duration})

total_duration = time.time() - total_start_time
print(f"Total SSL Training Time: {total_duration/60:.2f} minutes")

# save logs and SSL weights
df_ssl = pd.DataFrame(records)
df_ssl.to_excel("dino_training_log.xlsx", index=False)
print("Training log saved as 'dino_training_log.xlsx'")
torch.save(model.state_dict(), "dino_vit_small_ssl.pth")
print("SSL weights saved as 'dino_vit_small_ssl.pth'")
```

```
# Supervised fine-tuning
# classifier that reuses DINO backbone
class DINOClassifier(nn.Module):
    def __init__(self, dino_model, num_classes=2):
        super().__init__()
        self.backbone = dino_model.backbone
        self.classifier = nn.Linear(384, num_classes)

    def forward(self, x):
        tokens = self.backbone.forward_features(x)
        cls_token = tokens[:, 0]
        return self.classifier(cls_token)

# supervised transforms and loaders
transform_train = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))])

transform_eval = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))])

train_path = "Final Dataset Aug/train"
val_path   = "Final Dataset Aug/val"
test_path  = "Final Dataset Aug/test"

train_data = datasets.ImageFolder(train_path, transform=transform_train)
val_data   = datasets.ImageFolder(val_path,   transform=transform_eval)
test_data  = datasets.ImageFolder(test_path,  transform=transform_eval)

train_loader = DataLoader(train_data, batch_size=32, shuffle=True,  num_workers=4)
val_loader   = DataLoader(val_data,   batch_size=32, shuffle=False, num_workers=4)
test_loader  = DataLoader(test_data,  batch_size=32, shuffle=False, num_workers=4)

print("Classes:", train_data.classes)
print("Train size:", len(train_data))
print("Val size:", len(val_data))
print("Test size:", len(test_data))

# build fine-tune model from saved SSL
dino_model = DINOModel().to(device)
dino_model.load_state_dict(torch.load("dino_vit_small_ssl.pth", map_location=device))
model_ft = DINOClassifier(dino_model).to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(model_ft.parameters(), lr=1e-4)
from torch.optim.lr_scheduler import CosineAnnealingLR
num_epochs = 50
scheduler = CosineAnnealingLR(optimizer, T_max=num_epochs, eta_min=1e-6)
best_val_acc = 0.0
history = []
start_time = time.time()
```

```
for epoch in range(num_epochs):
    model_ft.train()
    train_loss_sum, train_correct = 0.0, 0
    for imgs, labels in tqdm(train_loader, desc=f"Epoch {epoch+1}/{num_epochs} - Training"):
        imgs, labels = imgs.to(device), labels.to(device)
        outputs = model_ft(imgs)
        loss = criterion(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        train_loss_sum += loss.item() * imgs.size(0)
        train_correct += (outputs.argmax(1) == labels).sum().item()
    train_loss_avg = train_loss_sum / len(train_data)
    train_acc = train_correct / len(train_data)

    model_ft.eval()
    val_loss_sum, val_correct = 0.0, 0
    with torch.no_grad():
        for imgs, labels in tqdm(val_loader, desc=f"Epoch {epoch+1}/{num_epochs} - Validation"):
            imgs, labels = imgs.to(device), labels.to(device)
            outputs = model_ft(imgs)
            loss = criterion(outputs, labels)
            val_loss_sum += loss.item() * imgs.size(0)
            val_correct += (outputs.argmax(1) == labels).sum().item()
    val_loss_avg = val_loss_sum / len(val_data)
    val_acc = val_correct / len(val_data)

    scheduler.step()
    current_lr = optimizer.param_groups[0]['lr']
    print(f"Epoch {epoch+1}: Train_Loss={train_loss_avg:.4f}, Val_Loss={val_loss_avg:.4f}, "
          f"Train_Acc={train_acc:.3f}, Val_Acc={val_acc:.3f}, LR={current_lr:.6f}")

    if val_acc > best_val_acc:
        best_val_acc = val_acc
        torch.save(model_ft.state_dict(), "dino_finetuned_classifier_best.pth")
        print("Best model saved.")

    if (epoch + 1) % 10 == 0:
        ckpt_path = f"dino_checkpoint_epoch{epoch+1}.pth"
        torch.save(model_ft.state_dict(), ckpt_path)
        print(f"Checkpoint saved: {ckpt_path}")

    history.append({"Epoch": epoch + 1, "Train_Loss": train_loss_avg,"Val_Loss": val_loss_avg,
"Train_Acc": train_acc,"Val_Acc": val_acc, "Learning_Rate": current_lr})

end_time = time.time()
print(f"Total fine-tuning time: {(end_time - start_time)/60:.2f} minutes")

torch.save(model_ft.state_dict(), "dino_finetuned_classifier_final.pth")
print("Final fine-tuned model saved as: dino_finetuned_classifier_final.pth")

df_ft = pd.DataFrame(history)
df_ft.to_excel("dino_finetune_metrics.xlsx", index=False)
print("Saved fine-tuning metrics to: dino_finetune_metrics.xlsx")
```

## D.10 Model Training DINO-CaiT

**Code 4.5.6: Model Training DINO-CaiT** python

```
import os, sys, time, math, copy, csv, numpy as np, torch, torch.nn as nn, torch.nn.functional
as F, timm, pandas as pd
from PIL import Image, ImageFilter, ImageOps
from torch.utils.data import Dataset, DataLoader
from torchvision.datasets import ImageFolder
import torchvision.transforms.v2 as v2
from tqdm import tqdm
from torch.optim.lr_scheduler import CosineAnnealingLR

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Device:", device.type)

# config
cfg_ssl = dict(model='cait_xxs24_224', img=128, batch=32, epochs=25, lr=5e-4, wd=0.04,
mom_t=0.996,
               out_dim=2048, local=2, wup_t=0.04, tea_t=0.07, wup_t_ep=10, stu_t=0.1,
center_m=0.9)
cfg_ft  = dict(model='cait_xxs24_224', img=128, batch=32, classes=2, epochs=50,
               train='../02. Dataset and Augmentation/01. Final Dataset Aug/train',
               val  ='../02. Dataset and Augmentation/01. Final Dataset Aug/val',
               ckpt_dir='checkpoints')

# utils
def set_seed(s=42):
   torch.manual_seed(s); np.random.seed(s); torch.cuda.manual_seed_all(s)
set_seed(42)

class GaussianBlur:
   def __init__(self, p=0.5, rmin=0.1, rmax=2.0): self.p, self.rmin, self.rmax = p, rmin, rmax
   def __call__(self, img):
      if torch.rand(1).item() <= self.p: img =
img.filter(ImageFilter.GaussianBlur(radius=float(torch.empty(1).uniform_(self.rmin,
self.rmax))))
      return img

class Solarization:
   def __init__(self, p=0.2): self.p = p
   def __call__(self, img): return ImageOps.solarize(img) if torch.rand(1).item() <= self.p else
img

class DataAugmentationDINO:
   def __init__(self, g_scale, l_scale, l_num, size=128):
      norm = [0.485, 0.456, 0.406], [0.229, 0.224, 0.225]
      cj = v2.ColorJitter(0.4, 0.4, 0.2, 0.1)
```

```
base = [v2.RandomHorizontalFlip(p=0.5), v2.RandomApply([cj], p=0.8),
v2.RandomGrayscale(p=0.2)]
        tail = [v2.ToImage(), v2.ToDtype(torch.float32, scale=True), v2.Normalize(*norm)]
        self.g1 = v2.Compose([v2.RandomResizedCrop(size, scale=g_scale), *base,
GaussianBlur(1.0), *tail])
        self.g2 = v2.Compose([v2.RandomResizedCrop(size, scale=g_scale), *base,
GaussianBlur(0.1), Solarization(0.2), *tail])
        self.l  = v2.Compose([v2.RandomResizedCrop(size, scale=l_scale), *base,
GaussianBlur(0.5), *tail])
        self.ln = l_num
    def __call__(self, img):
        crops = [self.g1(img), self.g2(img)]
        crops.extend(self.l(img) for _ in range(self.ln))
        return crops

class PCBUnlabeledDataset(Dataset):
    def __init__(self, root, transform): self.ds, self.t = ImageFolder(root), transform
    def __len__(self): return len(self.ds)
    def __getitem__(self, i):
        p, _ = self.ds.imgs[i]
        try: return self.t(Image.open(p).convert("RGB")), 0
        except Exception as e:
            print("Bad image:", p, e)
            dummy = torch.zeros(3, cfg_ssl['img'], cfg_ssl['img'])
            return [dummy]*(2+cfg_ssl['local']), 0

def cosine_sched(base, final, epochs, iters_per_ep, warmup_ep=0):
    warm = warmup_ep * iters_per_ep
    sched = list(np.linspace(0, base, warm)) if warm > 0 else []
    rest = epochs * iters_per_ep - len(sched)
    if rest > 0:
        t = np.arange(rest)
        sched += list(final + 0.5*(base-final)*(1 + np.cos(np.pi*t/rest)))
    return sched

class DINOHead(nn.Module):
    def __init__(self, in_dim, out_dim, use_bn=False, norm_last=True):
        super().__init__()
        h = in_dim
        self.mlp = nn.Sequential(nn.Linear(in_dim, h), nn.GELU(), nn.BatchNorm1d(h) if use_bn
else nn.Identity(),
                                 nn.Linear(h, h), nn.GELU(), nn.BatchNorm1d(h) if use_bn else nn.Identity())
        self.last = nn.Linear(h, out_dim, bias=False); self.norm_last = norm_last
    def forward(self, x):
        x = self.mlp(x)
        return F.linear(x, F.normalize(self.last.weight, dim=1)) if self.norm_last else self.last(x)

class MultiCropWrapper(nn.Module):
    def __init__(self, backbone, head):
        super().__init__(); backbone.head = nn.Identity(); self.backbone, self.head = backbone,
head
    def forward(self, x):
        if not isinstance(x, list): x = [x]
```

```
sizes = torch.tensor([inp.shape[-1] for inp in x]); idx =
torch.cumsum(torch.unique_consecutive(sizes, return_counts=True)[1], 0)
      out, s = [], 0
      for e in idx:
        y = self.backbone(torch.cat(x[s:e]))
        y = y[0] if isinstance(y, tuple) else y
        out.append(y); s = e
      return self.head(torch.cat(out))

class DINOLoss(nn.Module):
  def __init__(self, out_dim, ncrops, wup_t, tea_t, wup_ep, nepochs, stu_t, center_m):
    super().__init__()
    self.stu_t, self.ncrops, self.center_m = stu_t, ncrops, center_m
    self.register_buffer("center", torch.zeros(1, out_dim))
    self.tea_ts = np.concatenate((np.linspace(wup_t, tea_t, wup_ep), np.ones(nepochs -
wup_ep)*tea_t))
  def forward(self, s_out, t_out, epoch):
    s = (s_out / self.stu_t).chunk(self.ncrops); t = F.softmax((t_out -
self.center)/self.tea_ts[epoch], -1).detach().chunk(2)
    loss, n = 0, 0
    for i, q in enumerate(t):
      for v in range(len(s)):
        if v == i: continue
        loss += torch.sum(-q*F.log_softmax(s[v], -1), -1).mean(); n += 1
    self.update_center(t_out); return loss / n
  @torch.no_grad()
  def update_center(self, t_out):
    self.center = self.center*self.center_m + (1-self.center_m)*t_out.mean(0, keepdim=True)

# SSL data
ssl_dir = "../02. Dataset and Augmentation/01. Final Dataset Aug/train"
ssl_tf  = DataAugmentationDINO((0.4,1.0), (0.05,0.4), cfg_ssl['local'], cfg_ssl['img'])
ssl_ds  = PCBUnlabeledDataset(ssl_dir, ssl_tf)
ssl_dl  = DataLoader(ssl_ds, batch_size=cfg_ssl['batch'], shuffle=True, num_workers=0,
pin_memory=True)
print("SSL images:", len(ssl_ds))

# student and teacher
student_backbone = timm.create_model(cfg_ssl['model'], pretrained=False,
img_size=cfg_ssl['img'], num_classes=0)
emb = student_backbone.embed_dim
student = MultiCropWrapper(student_backbone, DINOHead(emb, cfg_ssl['out_dim'],
use_bn=False, norm_last=True)).to(device)
teacher = copy.deepcopy(student).to(device)
for p in teacher.parameters(): p.requires_grad = False

# loss and optimizer
crit = DINOLoss(cfg_ssl['out_dim'], 2+cfg_ssl['local'], cfg_ssl['wup_t'], cfg_ssl['tea_t'],
cfg_ssl['wup_t_ep'],
            cfg_ssl['epochs'], cfg_ssl['stu_t'], cfg_ssl['center_m']).to(device)
opt  = torch.optim.AdamW(student.parameters(), lr=cfg_ssl['lr']*cfg_ssl['batch']/256,
weight_decay=cfg_ssl['wd'])
```

```
# logging and checkpoints
os.makedirs(cfg_ft['ckpt_dir'], exist_ok=True)
with open('training_log.csv', 'w', newline='') as f: csv.writer(f).writerow(['Epoch','Loss','LR'])

print("Starting SSL pretraining...")
for ep in range(cfg_ssl['epochs']):
    student.train(); epoch_loss = 0
    for i, (imgs, _) in enumerate(tqdm(ssl_dl, desc=f"SSL {ep+1}/{cfg_ssl['epochs']}")):
        it = ep*niter + i
        for g in opt.param_groups:
            g['lr'] = lr_sched[it]
            if i == 0: g['weight_decay'] = wd_sched[it]
        imgs = [im.to(device, non_blocking=True) for im in imgs]
        with torch.no_grad(): t_out = teacher(imgs[:2])
        s_out = student(imgs)
        loss = crit(s_out, t_out, ep)
        if not math.isfinite(loss.item()): print("Non finite loss"); sys.exit(1)
        opt.zero_grad(); loss.backward(); torch.nn.utils.clip_grad_norm_(student.parameters(),
3.0); opt.step()
        m = mom_sched[it]
        with torch.no_grad():
            for q, k in zip(student.parameters(), teacher.parameters()): k.data.mul_(m).add_((1-
m)*q.data)
        epoch_loss += loss.item()
    avg = epoch_loss / niter
    with open('training_log.csv', 'a', newline='') as f: csv.writer(f).writerow([ep+1, avg,
opt.param_groups[0]['lr']])
    print(f"SSL epoch {ep+1}: loss {avg:.4f}, lr {opt.param_groups[0]['lr']:.6f}")
    if (ep+1) % 20 == 0 or ep+1 == cfg_ssl['epochs']:
        torch.save({'student': student.state_dict(), 'teacher': teacher.state_dict(), 'epoch': ep+1,
'cfg': cfg_ssl},
                   os.path.join(cfg_ft['ckpt_dir'], f'dino_checkpoint_{ep+1:04d}.pth'))

# fine-tuning setup
ckpt_path = os.path.join(cfg_ft['ckpt_dir'], f'dino_checkpoint_{cfg_ssl["epochs"]:04d}.pth')
backbone = timm.create_model(cfg_ft['model'], pretrained=False, img_size=cfg_ft['img'],
num_classes=0); backbone.head = nn.Identity()
tmp_student = MultiCropWrapper(timm.create_model(cfg_ft['model'], pretrained=False,
img_size=cfg_ft['img'], num_classes=0),
                               DINOHead(emb, cfg_ssl['out_dim'], use_bn=False, norm_last=True))
tmp_student.load_state_dict(torch.load(ckpt_path, map_location=device)['student'],
strict=False)
backbone.load_state_dict(tmp_student.backbone.state_dict(), strict=False)

class Classifier(nn.Module):
    def __init__(self, bb, dim, ncls): super().__init__(); self.bb, self.fc = bb, nn.Linear(dim, ncls)
    def forward(self, x):
        y = self.bb(x); y = y[0] if isinstance(y, tuple) else y
        return self.fc(y)
model = Classifier(backbone, emb, cfg_ft['classes']).to(device)
# transforms and loaders for fine-tuning
def tf_train_val(sz):
    tr = v2.Compose([v2.Resize((sz, sz)), v2.RandomHorizontalFlip(0.5), v2.ToImage(),
v2.ToDtype(torch.float32, scale=True),
                     v2.Normalize([0.485,0.456,0.406],[0.229,0.224,0.225])])
```

```
    va = v2.Compose([v2.Resize((sz, sz)), v2.ToImage(), v2.ToDtype(torch.float32, scale=True),
                     v2.Normalize([0.485,0.456,0.406],[0.229,0.224,0.225])])

return tr, va
tr_tf, va_tf = tf_train_val(cfg_ft['img'])
tr_ds, va_ds = ImageFolder(cfg_ft['train'], tr_tf), ImageFolder(cfg_ft['val'], va_tf)
tr_dl = DataLoader(tr_ds, batch_size=cfg_ft['batch'], shuffle=True)
va_dl = DataLoader(va_ds, batch_size=cfg_ft['batch'], shuffle=False)
print("Classes:", tr_ds.classes, "Train:", len(tr_ds), "Val:", len(va_ds))

crit_ft = nn.CrossEntropyLoss()
opt_ft  = torch.optim.AdamW(model.parameters(), lr=1e-4)
sch     = CosineAnnealingLR(opt_ft, T_max=cfg_ft['epochs'], eta_min=1e-6)

best_acc, hist = 0.0, []
for ep in range(cfg_ft['epochs']):
    model.train(); tr_loss, tr_cor = 0.0, 0
    for x, y in tqdm(tr_dl, desc=f"FT {ep+1}/{cfg_ft['epochs']} T"):
        x, y = x.to(device), y.to(device)
        out = model(x); loss = crit_ft(out, y)
        opt_ft.zero_grad(); loss.backward(); opt_ft.step()
        tr_loss += loss.item() * x.size(0); tr_cor += (out.argmax(1) == y).sum().item()
    tr_loss /= len(tr_ds); tr_acc = tr_cor / len(tr_ds)

    model.eval(); va_loss, va_cor = 0.0, 0
    with torch.no_grad():
        for x, y in tqdm(va_dl, desc=f"FT {ep+1}/{cfg_ft['epochs']} V"):
            x, y = x.to(device), y.to(device)
            out = model(x); loss = crit_ft(out, y)
            va_loss += loss.item() * x.size(0); va_cor += (out.argmax(1) == y).sum().item()
    va_loss /= len(va_ds); va_acc = va_cor / len(va_ds)

    sch.step(); lr = opt_ft.param_groups[0]['lr']
    print(f"FT epoch {ep+1}: train_loss {tr_loss:.4f} val_loss {va_loss:.4f} train_acc {tr_acc:.3f}
val_acc {va_acc:.3f} lr {lr:.6f}")

    if va_acc > best_acc:
        best_acc = va_acc
        torch.save(model.state_dict(), "finetuned_dino_classifier_best.pth")
        print("Saved best finetuned model")

    if (ep+1) % 10 == 0:
        torch.save(model.state_dict(), f"checkpoint_epoch_{ep+1}.pth")

    hist.append(dict(Epoch=ep+1, Train_Loss=tr_loss, Val_Loss=va_loss, Train_Acc=tr_acc,
Val_Acc=va_acc, LR=lr))

torch.save(model.state_dict(), "finetuned_dino_classifier_final.pth")
pd.DataFrame(hist).to_excel("finetune_dino_cait_logs.xlsx", index=False)
print("Done. Final and logs saved.")
```

# Appendix E

# State-of-the-Art Models' Training Behaviour

Training time taken to train the models as per the configuration mentioned in Table 4.3 and hardware in Appendix Appendix C is shown below Figure E.1.

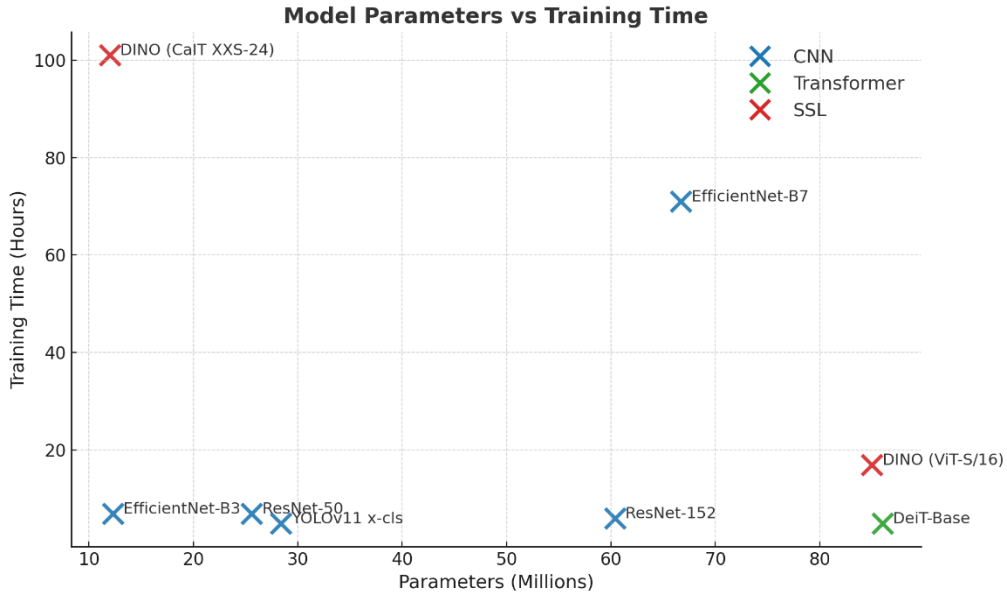



**Figure E.1:** Training Time vs Models' Parameters

The graphs Figure E.2 to Figure E.9 show the detailed training behavior of each model.

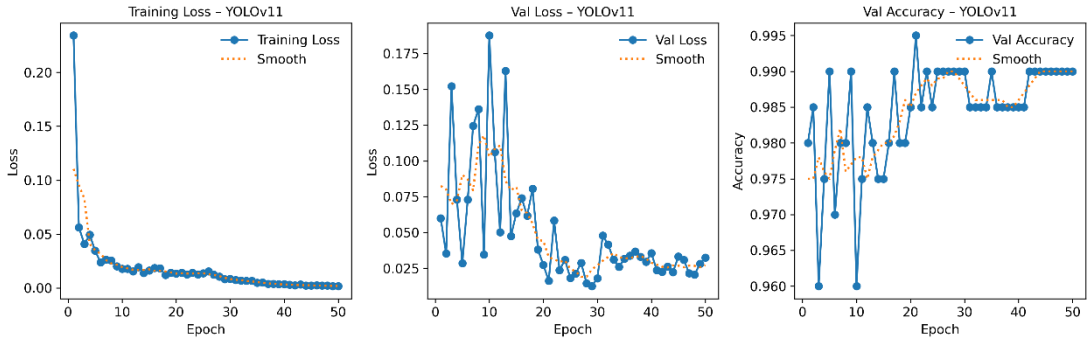



**Figure E.2:** YOLOv11 Training loss, val losses, and val accuracy vs Epochs

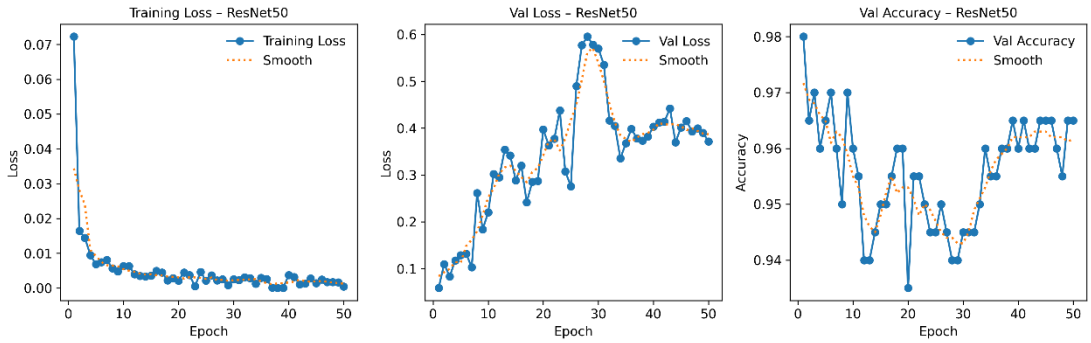



**Figure E.3:** ResNet50 Training loss, val losses, and val accuracy vs Epochs

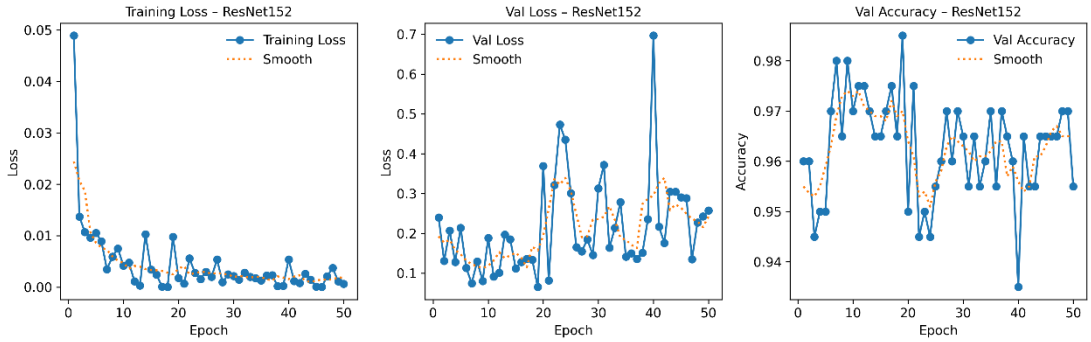



**Figure E.4:** ResNet152 Training loss, val losses, and val accuracy vs Epochs

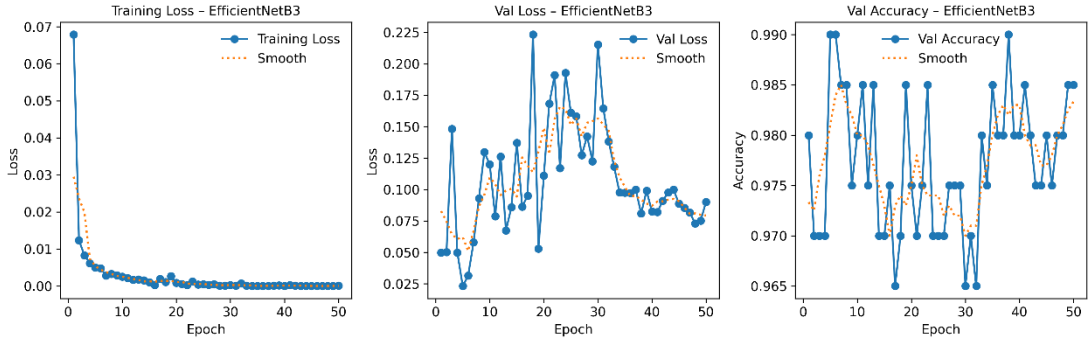



**Figure E.5:** EfficientNetB3 Training loss, val losses, and val accuracy vs Epochs

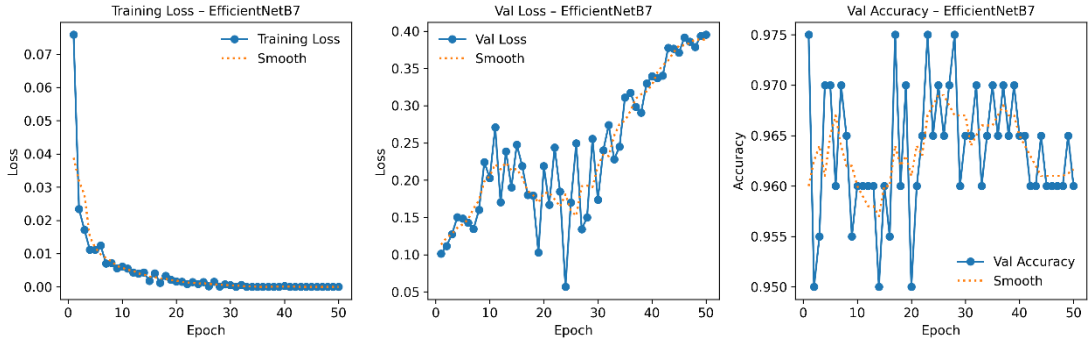



**Figure E.6:** EfficientNetB7 Training loss, val losses, and val accuracy vs Epochs

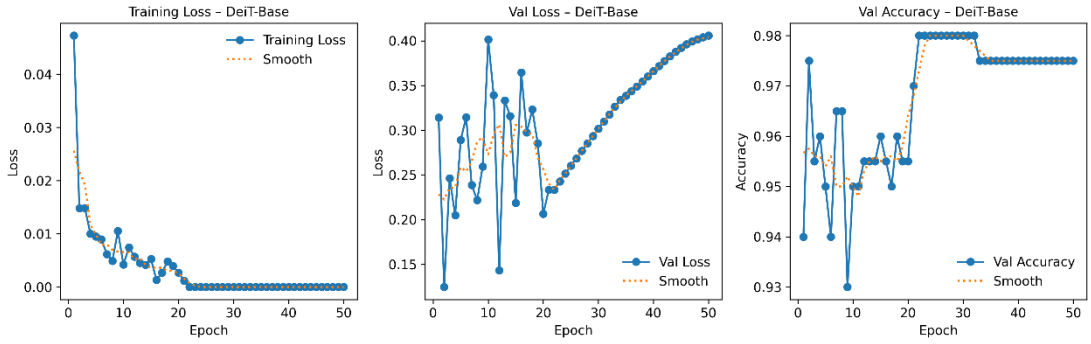



**Figure E.7:** DeiT-Base Training loss, val losses, and val accuracy vs Epochs

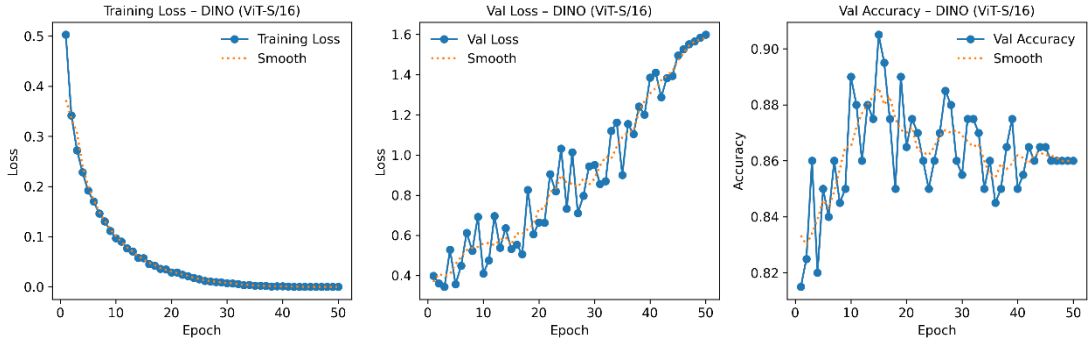



**Figure E.8:** DINO (ViT-S/16) Training loss, val losses, and val accuracy vs Epochs

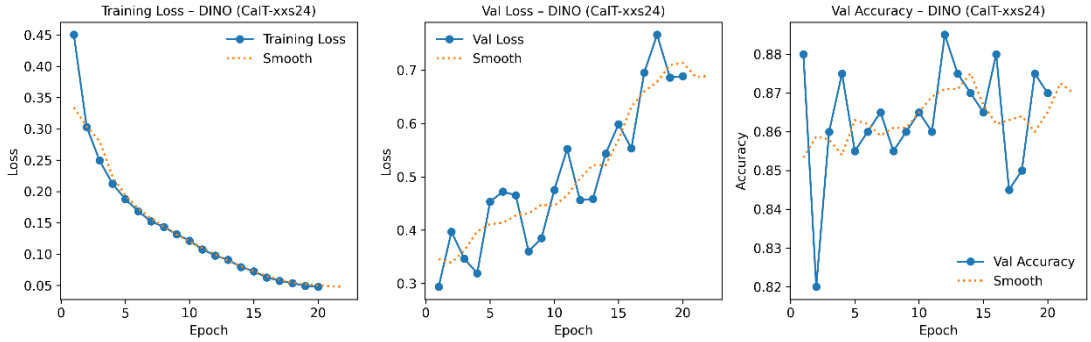



**Figure E.9:** DINO (CaIT-xx24) Training loss, val losses, and val accuracy vs Epochs

For YOLOv11, the learning rate is automatically initialized and decays over time. For all other models, the same learning rate configuration (as described in section 4.5) is applied uniformly, with identical initialization, optimizer, and scheduling strategy.

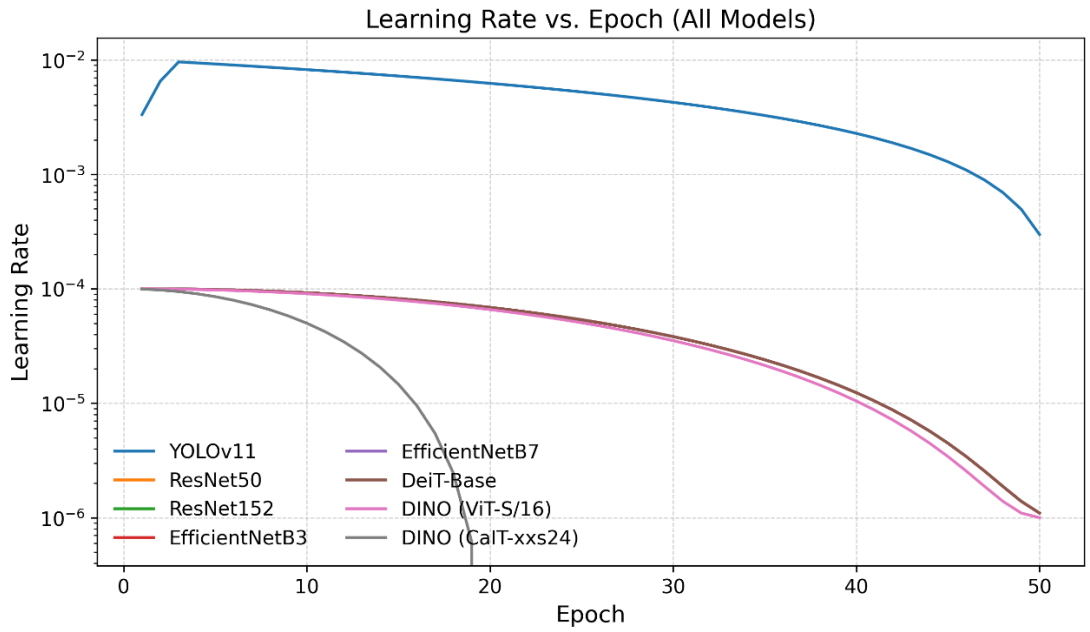



**Figure E.10:** Learning rate per epoch for all Models

# Bibliography


[1] "The Global E-waste Monitor 2024 - E-Waste Monitor." Accessed: Aug. 18, 2025. [Online]. Available: https://ewastemonitor.info/the-global-e-waste-monitor-2024/

[2] "Plattform Industrie 4.0 - Artificial Intelligence for Circular Economy." Accessed: Jul. 17, 2025. [Online]. Available: https://www.plattform-i40.de/IP/Redaktion/EN/Use-Cases/536-circular-economy/article-circular-economy.html

[3] X. Sun, H. Yu, and W. D. Solvang, "Towards the smart and sustainable transformation of Reverse Logistics 4.0: a conceptualization and research agenda," *Environmental Science and Pollution Research*, vol. 29, no. 46, pp. 69275–69293, 2022, doi: 10.1007/s11356-022-22473-3.

[4] S. Raut, N. U. I. Hossain, M. Kouhizadeh, and S. A. Fazio, "Application of Artificial Intelligence in Circular Economy: A Critical Analysis of the Current Research," *Sustainable Futures*, p. 100784, 2025.

[5] D. Parker, K. Riley, S. Robinson, H. Symington, J. Tewson, and O. Hollins, "Remanufacturing Market Study," p. 67, 2015, Accessed: Jul. 17, 2025. [Online]. Available: https://cris.vtt.fi/en/publications/remanufacturing-market-study

[6] M. A. Seitz and K. Peattie, "Meeting the closed-loop challenge: the case of remanufacturing," *Calif Manage Rev*, vol. 46, no. 2, pp. 74–89, 2004.

[7] M. Schlüter *et al.*, "AI-enhanced identification, inspection and sorting for reverse logistics in remanufacturing," *Procedia CIRP*, vol. 98, pp. 300–305, 2021.

[8] T. Tolio *et al.*, "Design, management and control of demanufacturing and remanufacturing systems," *CIRP Annals*, vol. 66, no. 2, pp. 585–609, 2017.

[9] M. Errington and S. J. Childe, "A business process model of inspection in remanufacturing," *Journal of Remanufacturing*, vol. 3, no. 1, p. 7, 2013.

[10] P. Lundmark, E. Sundin, and M. Björkman, "Industrial challenges within the remanufacturing system," in *3rd swedish production symposium 2009, göteborg*, 2009, pp. 132–138.

[11] J. Kirchherr, D. Reike, and M. Hekkert, “Conceptualizing the circular economy: An analysis of 114 definitions,” *Resour Conserv Recycl*, vol. 127, pp. 221–232, 2017.

[12] J. Kirchherr, N.-H. N. Yang, F. Schulze-Spüntrup, M. J. Heerink, and K. Hartley, “Conceptualizing the circular economy (revisited): an analysis of 221 definitions,” *Resour Conserv Recycl*, vol. 194, p. 107001, 2023.

[13] A. Mayer, W. Haas, D. Wiedenhofer, F. Krausmann, P. Nuss, and G. A. Blengini, “Measuring Progress towards a Circular Economy: A Monitoring Framework for Economy-wide Material Loop Closing in the EU28,” *J Ind Ecol*, vol. 23, no. 1, pp. 62–76, Feb. 2019, doi: 10.1111/JIEC.12809.

[14] “Circular material use rate in Europe | European Environment Agency’s home page.” Accessed: Aug. 18, 2025. [Online]. Available: https://www.eea.europa.eu/en/analysis/indicators/circular-material-use-rate-in-europe

[15] “Circular economy flow diagrams.” Accessed: Aug. 17, 2025. [Online]. Available: https://ec.europa.eu/eurostat/cache/sankey/circular_economy/sankey.html?geos=EU27_2020&unit=G_T&materials=TOTAL&material=TOTAL&highlight=&nodeDisagg=0101000000&flowDisagg=false&language=EN#

[16] S. Peng, J. Ping, T. Li, F. Wang, H. Zhang, and C. Liu, “Environmental benefits of remanufacturing mechanical products: a harmonized meta-analysis of comparative life cycle assessment studies,” *J Environ Manage*, vol. 306, p. 114479, 2022.

[17] S. Quinto, N. Law, C. Fletcher, J. Le, S. Antony Jose, and P. L. Menezes, “Exploring the E-Waste Crisis: Strategies for Sustainable Recycling and Circular Economy Integration,” *Recycling*, vol. 10, no. 2, p. 72, 2025.

[18] S. Nazir and F. Doni, “Nexus of circular economy R0 to R9 principles in integrated reporting: Insights from a multiple case study comparison,” *Bus Strategy Environ*, vol. 33, no. 5, pp. 4058–4085, 2024.

[19] “CATEGORISATION SYSTEM FOR THE CIRCULAR ECONOMY A sector-agnostic approach for activities contributing to the circular economy Independent Expert Report”, doi: 10.2777/172128.

[20] J. Potting, M. P. Hekkert, E. Worrell, and A. Hanemaaijer, “Circular economy: measuring innovation in the product chain,” 2017.

[21] T. Tolio *et al.*, “Design, management and control of demanufacturing and remanufacturing systems,” *CIRP Annals*, vol. 66, no. 2, pp. 585–609, 2017, doi: 10.1016/j.cirp.2017.05.001.

[22] M. MatsumotoDr. and W. IjomahDr., “Remanufacturing,” in *Handbook of Sustainable Engineering*, J. Kauffman and K.-M. Lee, Eds., Dordrecht: Springer Netherlands, 2013, pp. 389–408. doi: 10.1007/978-1-4020-8939-8_93.

[23] British Standards Institute, “BS 8887-2:2009: Design for manufacture,assembly, disassembly and end-of-life processing (MADE),” *Part 2: Terms and definitions*, 2009, Accessed: Aug. 18, 2025. [Online]. Available: https://bsol.bsigroup.com/en/Bsol-Item-Detail-Page/?pid=000000000030182997

[24] “Revising RIC001.2-2021: Specifications for the Process of Remanufacturing | Reman Standard.” Accessed: Aug. 18, 2025. [Online]. Available: https://remanstandard.us/revising-ric001-1-2016-specifications-for-the-process-of-remanufacturing-2/

[25] “DIN SPEC 91472:2023-06, Remanufacturing (Reman)_- Qualitätsklassifizierung für zirkuläre Prozesse,” Jun. 2023, doi: 10.31030/3434252.

[26] J. Zhao, Z. Xue, T. Li, J. Ping, and S. Peng, “An energy and time prediction model for remanufacturing process using graphical evaluation and review technique (GERT) with multivariant uncertainties,” *Environmental Science and Pollution Research*, pp. 1–13, 2021.

[27] M. Andrew-Munot, R. N. Ibrahim, and E. Junaidi, “An overview of used-products remanufacturing,” *Mechanical Engineering Research*, vol. 5, no. 1, p. 12, 2015.

[28] B. Salah *et al.*, “A qualitative and quantitative analysis of remanufacturing research,” *Processes*, vol. 9, no. 10, p. 1766, 2021, doi: 10.3390/pr9101766.

[29] H. N. W. Gunasekara, J. R. Gamage, and H. K. G. Punchihewa, “Remanufacture for sustainability: a comprehensive business model for automotive parts remanufacturing,” *International Journal of Sustainable Engineering*, vol. 14, no. 6, pp. 1386–1395, Nov. 2021, doi: 10.1080/19397038.2021.1990437.

[30] D. P. José *et al.*, “Remanufacturing in Developing Countries–A Case Study in Automotive Sector in Ecuador,” *Procedia CIRP*, vol. 116, pp. 534–539, 2023, doi: 10.1016/j.procir.2023.02.090.

[31] H. Gunasekara, J. Gamage, and H. Punchihewa, “Remanufacture for Sustainability: A review of the barriers and the solutions to promote remanufacturing,” in *2018 International Conference on Production and Operations Management Society (POMS)*, 2018, pp. 1–7. doi: 10.1109/POMS.2018.8629474.

[32] H. Geist and F. Balle, “A circularity engineering focused empirical status quo analysis of automotive remanufacturing processes,” *Resour Conserv Recycl*, vol. 201, p. 107328, 2024, doi: https://doi.org/10.1016/j.resconrec.2023.107328.

[33] B. Salah *et al.*, “A qualitative and quantitative analysis of remanufacturing research,” *Processes*, vol. 9, no. 10, p. 1766, 2021, doi: 10.3390/pr9101766.

[34] A. Alkouh, K. A. Keddar, and S. Alatefi, “Remanufacturing of industrial electronics: A case study from the GCC region,” *Electronics (Basel)*, vol. 12, no. 9, p. 1960, 2023.

[35] E. Sundin, B. Backman, K. Johansen, M. Hochwallner, S. Landscheidt, and S. Shahbazi, “Automation Potential in the Remanufacturing of Electric and Electronic Equipment (EEE),” in *SPS2020*, IOS Press, 2020, pp. 285–296. doi: 10.3233/ATDE200166.

[36] F. Claudia, P. Emilio, and R. Chiara, “Scaling up a circular business model for remanufacturing: A case study of a sustainable value creation strategy for the white goods industry,” *Bus Strategy Environ*, vol. 33, no. 7, pp. 7479–7510, 2024.

[37] E. Sundin, “An economical and technical analysis of a household appliance remanufacturing process,” in *Proceedings Second International Symposium on Environmentally Conscious Design and Inverse Manufacturing*, IEEE, 2001, pp. 536–541.

[38] M. Hoffmann, A. Krini, A. Mueller, and S. Knorn, “Remanufacturing production planning and control: Conceptual framework for requirement definition,” *Journal of Remanufacturing*, vol. 15, no. 1, pp. 97–126, 2025, doi: 10.1007/s13243-025-00149-8.

[39] “DIN 31051:2019-06, Grundlagen der Instandhaltung,” Jun. 2019, doi: 10.31030/3048531.

[40] R. T. Lund, “Remanufacturing: the experience of the United States and implications for developing countries,” 1984.

[41] C. Liu *et al.*, "A review on remanufacturing assembly management and technology," *The International Journal of Advanced Manufacturing Technology*, vol. 105, no. 11, pp. 4797–4808, 2019, doi: 10.1007/s00170-019-04617-x.

[42] R. Steinhilper, "Remanufacturing," 1998.

[43] M. MatsumotoDr. and W. IjomahDr., "Remanufacturing," in *Handbook of Sustainable Engineering*, J. Kauffman and K.-M. Lee, Eds., Dordrecht: Springer Netherlands, 2013, pp. 389–408. doi: 10.1007/978-1-4020-8939-8_93.

[44] "Bosch Core acceptance criteria for starter motors," 2021.

[45] M. Schlüter *et al.*, "AI-enhanced Identification, Inspection and Sorting for Reverse Logistics in Remanufacturing," *Procedia CIRP*, vol. 98, pp. 300–305, 2021, doi: https://doi.org/10.1016/j.procir.2021.01.107.

[46] J. E. See, C. G. Drury, A. Speed, A. Williams, and N. Khalandi, "The role of visual inspection in the 21st century," in *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, SAGE Publications Sage CA: Los Angeles, CA, 2017, pp. 262–266. doi: 10.1177/154193121360154.

[47] Z. Ren, F. Fang, N. Yan, and Y. Wu, "State of the Art in Defect Detection Based on Machine Vision," *International Journal of Precision Engineering and Manufacturing-Green Technology*, vol. 9, no. 2, pp. 661–691, 2022, doi: 10.1007/s40684-021-00343-6.

[48] Z. Ren, F. Fang, N. Yan, and Y. Wu, "State of the Art in Defect Detection Based on Machine Vision," *International Journal of Precision Engineering and Manufacturing-Green Technology*, vol. 9, no. 2, pp. 661–691, 2022, doi: 10.1007/s40684-021-00343-6.

[49] T. S. Newman and A. K. Jain, "A Survey of Automated Visual Inspection," *Computer Vision and Image Understanding*, vol. 61, no. 2, pp. 231–262, 1995, doi: https://doi.org/10.1006/cviu.1995.1017.

[50] S.-H. Huang and Y.-C. Pan, "Automated visual inspection in the semiconductor industry: A survey," *Comput Ind*, vol. 66, pp. 1–10, 2015, doi: https://doi.org/10.1016/j.compind.2014.10.006.

[51] C. E. Nwankpa, W. Ijomah, and A. Gachagan, "Design for automated inspection in remanufacturing: A discrete event simulation for process improvement," *Clean*

*Eng Technol*, vol. 4, p. 100199, 2021, doi: https://doi.org/10.1016/j.clet.2021.100199.

[52] C. Nwankpa, S. Eze, W. Ijomah, A. Gachagan, and S. Marshall, "Achieving remanufacturing inspection using deep learning," *Journal of Remanufacturing*, vol. 11, no. 2, pp. 89–105, 2021, doi: 10.1007/s13243-020-00093-9.

[53] C. Pramerdorfer and M. Kampel, "A dataset for computer-vision-based PCB analysis," in *2015 14th IAPR international conference on machine vision applications (MVA)*, IEEE, 2015, pp. 378–381.

[54] C. Pramerdorfer, "Feature-based PCB Recognition for recycling purposes," in *International Conference on Computer Vision Theory and Applications, 2015*, 2015.

[55] W. Li, S. Neullens, M. Breier, M. Bosling, T. Pretz, and D. Merhof, "Text recognition for information retrieval in images of printed circuit boards," in *IECON 2014-40th Annual Conference of the IEEE Industrial Electronics Society*, IEEE, 2014, pp. 3487–3493.

[56] G. Mahalingam, K. M. Gay, and K. Ricanek, "Pcb-metal: A pcb image dataset for advanced computer vision machine learning component analysis," in *2019 16th International Conference on Machine Vision Applications (MVA)*, IEEE, 2019, pp. 1–5.

[57] W. Huang and P. Wei, "A PCB dataset for defects detection and classification," *arXiv preprint arXiv:1901.08204*, 2019.

[58] S. Tang, F. He, X. Huang, and J. Yang, "Online PCB defect detector on a new PCB defect dataset," *arXiv preprint arXiv:1902.06197*, 2019.

[59] H. Lu, D. Mehta, O. Paradis, N. Asadizanjani, M. Tehranipoor, and D. L. Woodard, "Fics-pcb: A multi-modal image dataset for automated printed circuit board visual inspection," *Cryptology ePrint Archive*, 2020.

[60] E. Arbash, M. Fuchs, B. Rasti, S. Lorenz, P. Ghamisi, and R. Gloaguen, "PCB-vision: A multiscene rgb-hyperspectral benchmark dataset of printed circuit boards," *IEEE Sens J*, vol. 24, no. 10, pp. 17140–17158, 2024.

[61] D. Candido de Oliveira, B. T. Nassu, and M. A. Wehrmeister, "Image-based detection of modifications in assembled pcbs with deep convolutional autoencoders," *Sensors*, vol. 23, no. 3, p. 1353, 2023.

[62] J. Yang, Z. Liu, W. Du, and S. Zhang, “A PCB defect detector based on coordinate feature refinement,” *IEEE Trans Instrum Meas*, vol. 72, pp. 1–10, 2023.

[63] M. Yuan, Y. Zhou, X. Ren, H. Zhi, J. Zhang, and H. Chen, “YOLO-HMC: An Improved Method for PCB Surface Defect Detection,” *IEEE Trans Instrum Meas*, vol. 73, pp. 1–11, 2024, doi: 10.1109/TIM.2024.3351241.

[64] B. Du, F. Wan, G. Lei, L. Xu, C. Xu, and Y. Xiong, “YOLO-MBBi: PCB surface defect detection method based on enhanced YOLOv5,” *Electronics (Basel)*, vol. 12, no. 13, p. 2821, 2023.

[65] Z. Xie and X. Zou, “MFAD-RTDETR: A multi-frequency aggregate diffusion feature flow composite model for printed circuit board defect detection,” *Electronics (Basel)*, vol. 13, no. 17, p. 3557, 2024.

[66] D. Li, C. Jiang, and T. Liang, “REDef-DETR: real-time and efficient DETR for industrial surface defect detection,” *Meas Sci Technol*, vol. 35, no. 10, p. 105411, 2024.

[67] J. Jin *et al.*, “Defect detection of printed circuit boards using efficientdet,” in *2021 ieee 6th international conference on signal and image processing (icsip)*, IEEE, 2021, pp. 287–293.

[68] H. Zhang, L. Jiang, and C. Li, “CS-ResNet: Cost-sensitive residual convolutional neural network for PCB cosmetic defect detection,” *Expert Syst Appl*, vol. 185, p. 115673, 2021, doi: https://doi.org/10.1016/j.eswa.2021.115673.

[69] J. Zhang, X. Shi, D. Qu, H. Xu, and Z. Chang, “PCB defect recognition by image analysis using deep convolutional neural network,” *Journal of Electronic Testing*, vol. 40, no. 5, pp. 657–667, 2024.

[70] T. Luo *et al.*, “A lightweight defect detection transformer for printed circuit boards combining image feature augmentation and refined cross-scale feature fusion,” *Eng Appl Artif Intell*, vol. 155, p. 111128, 2025, doi: https://doi.org/10.1016/j.engappai.2025.111128.

[71] Z. He, Y. Lian, Y. Wang, and Z. Lu, “A comprehensive review of research on surface defect detection of PCBs based on machine vision,” *Results in Engineering*, vol. 27, p. 106437, 2025, doi: https://doi.org/10.1016/j.rineng.2025.106437.

[72] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, Ieee, 2009, pp. 248–255.

[73] "Models Supported by Ultralytics - Ultralytics YOLO Docs." Accessed: Jul. 20, 2025. [Online]. Available: https://docs.ultralytics.com/models/

[74] "COCO - Common Objects in Context." Accessed: Jul. 21, 2025. [Online]. Available: https://cocodataset.org/#home

[75] "The History Of Neural Networks - Dataconomy." Accessed: Aug. 20, 2025. [Online]. Available: https://dataconomy.com/2017/04/19/history-neural-networks/

[76] A. Dosovitskiy *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[77] H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. Jégou, "Training data-efficient image transformers & distillation through attention," in *International conference on machine learning*, PMLR, 2021, pp. 10347–10357.

[78] Z. Liu *et al.*, "Swin transformer: Hierarchical vision transformer using shifted windows," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2021, pp. 10012–10022.

[79] Z. Liu *et al.*, "Swin transformer v2: Scaling up capacity and resolution," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 12009–12019.

[80] H. Touvron, M. Cord, A. Sablayrolles, G. Synnaeve, and H. Jégou, "Going deeper with image transformers," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2021, pp. 32–42.

[81] C. Shorten and T. M. Khoshgoftaar, "A survey on Image Data Augmentation for Deep Learning," *J Big Data*, vol. 6, no. 1, p. 60, 2019, doi: 10.1186/s40537-019-0197-0.

[82] J.-P. Kaiser, *Autonomous View Planning using Reinforcement Learning-Modeling and Application for Visual Inspection in Remanufacturing*. 2025.

[83] J. Nine, A. Shoukat, and W. Hardt, "Towards a Situation-Aware Cloud-Based Autonomous Driving System using Object Detection and Rule-Based Techniques,"

in *2024 International Symposium ELMAR*, 2024, pp. 295–299. doi: 10.1109/ELMAR62909.2024.10694231.

[84] L. Perez and J. Wang, "The effectiveness of data augmentation in image classification using deep learning," *arXiv preprint arXiv:1712.04621*, 2017.

[85] P. Rajpurkar *et al.*, "Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning," *arXiv preprint arXiv:1711.05225*, 2017.

[86] L. Zhang *et al.*, "Generalizing Deep Learning for Medical Image Segmentation to Unseen Domains via Deep Stacked Transformation," *IEEE Trans Med Imaging*, vol. 39, no. 7, pp. 2531–2540, 2020, doi: 10.1109/TMI.2020.2973595.

[87] E. D. Cubuk, B. Zoph, D. Mane, V. Vasudevan, and Q. V Le, "Autoaugment: Learning augmentation policies from data," *arXiv preprint arXiv:1805.09501*, 2018.

[88] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, Ieee, 2009, pp. 248–255.

[89] Z. Yang, F. Zhan, K. Liu, M. Xu, and S. Lu, "Ai-generated images as data source: The dawn of synthetic era," *arXiv preprint arXiv:2310.01830*, 2023.

[90] Y. Tian, L. Fan, P. Isola, H. Chang, and D. Krishnan, "Stablerep: Synthetic images from text-to-image models make strong visual representation learners," *Adv Neural Inf Process Syst*, vol. 36, pp. 48382–48402, 2023.

[91] S.-Y. Wang, O. Wang, A. Owens, R. Zhang, and A. A. Efros, "Detecting photoshopped faces by scripting photoshop," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 10072–10081.

[92] J. Kim, J. Ko, H. Choi, and H. Kim, "Printed circuit board defect detection using deep learning via a skip-connected convolutional autoencoder," *Sensors*, vol. 21, no. 15, p. 4968, 2021.

[93] J. Kaur, M. A. Khan, M. Iftikhar, M. Imran, and Q. E. U. Haq, "Machine learning techniques for 5G and beyond," *IEEE Access*, vol. 9, pp. 23472–23488, 2021.

[94] "7 stages of ML model development | Steps in machine learning life cycle | ML lifecycle guide | Lumenalta." Accessed: Aug. 31, 2025. [Online]. Available: https://lumenalta.com/insights/7-stages-of-ml-model-development

[95] Y. Roh, G. Heo, and S. E. Whang, "A survey on data collection for machine learning: a big data-ai integration perspective," *IEEE Trans Knowl Data Eng*, vol. 33, no. 4, pp. 1328–1347, 2019.

[96] R. Szeliski, *Computer vision: algorithms and applications*. Springer Nature, 2022.

[97] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *J Big Data*, vol. 6, no. 1, pp. 1–48, 2019.

[98] J. Ma, C. Hu, P. Zhou, F. Jin, X. Wang, and H. Huang, "Review of image augmentation used in deep learning-based material microscopic image segmentation," *Applied Sciences*, vol. 13, no. 11, p. 6478, 2023.

[99] H. Bichri, A. Chergui, and M. Hain, "Investigating the Impact of Train/Test Split Ratio on the Performance of Pre-Trained Models with Custom Datasets.," *International Journal of Advanced Computer Science & Applications*, vol. 15, no. 2, 2024.

[100] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[101] R. C. Gonzalez, *Digital image processing*. Pearson education india, 2009.

[102] "Understanding Forward and Backward Propagation in Neural Networks | LinkedIn." Accessed: Aug. 12, 2025. [Online]. Available: https://www.linkedin.com/pulse/understanding-forward-backward-propagation-neural-suresh-beekhani-e0rkf/

[103] K. Janocha and W. M. Czarnecki, "On loss functions for deep neural networks in classification," *arXiv preprint arXiv:1702.05659*, 2017.

[104] S. R. Dubey, S. K. Singh, and B. B. Chaudhuri, "Activation functions in deep learning: A comprehensive survey and benchmark," *Neurocomputing*, vol. 503, pp. 92–108, 2022.

[105] "Optimizers in Deep Learning. What is an optimizer? | by Musstafa | Medium." Accessed: Aug. 12, 2025. [Online]. Available: https://musstafa0804.medium.com/optimizers-in-deep-learning-7bf81fed78a0

[106] D. Choi, C. J. Shallue, Z. Nado, J. Lee, C. J. Maddison, and G. E. Dahl, "On empirical comparisons of optimizers for deep learning," *arXiv preprint arXiv:1910.05446*, 2019.

[107] R.-Y. Sun, "Optimization for deep learning: An overview," *Journal of the Operations Research Society of China*, vol. 8, no. 2, pp. 249–294, 2020.

[108] "Optimization Rule in Deep Neural Networks - GeeksforGeeks." Accessed: Aug. 12, 2025. [Online]. Available: https://www.geeksforgeeks.org/deep-learning/optimization-rule-in-deep-neural-networks/

[109] "What is the Difference Between 'Epoch' and 'Iteration' in Training Neural Networks - GeeksforGeeks." Accessed: Aug. 31, 2025. [Online]. Available: https://www.geeksforgeeks.org/data-science/what-is-the-difference-between-epoch-and-iteration-in-training-neural-networks/

[110] E. Lopez, J. Etxebarria-Elezgarai, J. M. Amigo, and A. Seifert, "The importance of choosing a proper validation strategy in predictive models. A tutorial with real examples," *Anal Chim Acta*, vol. 1275, p. 341532, 2023, doi: https://doi.org/10.1016/j.aca.2023.341532.

[111] Z. Han, C. Gao, J. Liu, J. Zhang, and S. Q. Zhang, "Parameter-efficient fine-tuning for large models: A comprehensive survey," *arXiv preprint arXiv:2403.14608*, 2024.

[112] "What is Fine-Tuning? - GeeksforGeeks." Accessed: Aug. 12, 2025. [Online]. Available: https://www.geeksforgeeks.org/deep-learning/what-is-fine-tuning/

[113] "What is Fine-Tuning? - GeeksforGeeks." Accessed: Aug. 30, 2025. [Online]. Available: https://www.geeksforgeeks.org/deep-learning/what-is-fine-tuning/

[114] "DL_for_practitioners/03_Evaluation_Datasets/03_1_Metrics_and_Evaluations.ipynb." Accessed: Aug. 02, 2025. [Online]. Available: https://github.com/hamkerlab/DL_for_practitioners/blob/main/03_Evaluation_Datasets/03_1_Metrics_and_Evaluations.ipynb

[115] S. Arora, W. Hu, and P. K. Kothari, "An analysis of the t-sne algorithm for data visualization," in *Conference on learning theory*, PMLR, 2018, pp. 1455–1462.

[116] "Decoding CNNs: A Beginner's Guide to Convolutional Neural Networks and their Applications | by Ravjot Singh | Medium." Accessed: Jul. 16, 2025. [Online]. Available: https://ravjot03.medium.com/decoding-cnns-a-beginners-guide-to-convolutional-neural-networks-and-their-applications-1a8806cbf536

[117] B. P. Sowmya and M. C. Supriya, “Convolutional neural network (cnn) fundamental operational survey,” in *International Conference on Innovative Computing and Cutting-edge Technologies*, Springer, 2020, pp. 245–258.

[118] “CS231n Deep Learning for Computer Vision.” Accessed: Jul. 16, 2025. [Online]. Available: https://cs231n.github.io/convolutional-networks/

[119] “What is Fully Connected Layer in Deep Learning? - GeeksforGeeks.” Accessed: Jul. 16, 2025. [Online]. Available: https://www.geeksforgeeks.org/deep-learning/what-is-fully-connected-layer-in-deep-learning/

[120] A. Vaswani *et al.*, “Attention is all you need,” *Adv Neural Inf Process Syst*, vol. 30, 2017.

[121] A. Dosovitskiy *et al.*, “An image is worth 16x16 words: Transformers for image recognition at scale,” *arXiv preprint arXiv:2010.11929*, 2020.

[122] S. Khan, M. Naseer, M. Hayat, S. W. Zamir, F. S. Khan, and M. Shah, “Transformers in vision: A survey,” *ACM computing surveys (CSUR)*, vol. 54, no. 10s, pp. 1–41, 2022.

[123] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, “A simple framework for contrastive learning of visual representations,” in *International conference on machine learning*, PmLR, 2020, pp. 1597–1607.

[124] M. Caron *et al.*, “Emerging properties in self-supervised vision transformers,” in *Proceedings of the IEEE/CVF international conference on computer vision*, 2021, pp. 9650–9660.

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

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

[127] “How do bottleneck architectures work in neural networks? - Cross Validated.” Accessed: Aug. 13, 2025. [Online]. Available: https://stats.stackexchange.com/questions/205150/how-do-bottleneck-architectures-work-in-neural-networks

[128] M. Tan and Q. Le, “Efficientnet: Rethinking model scaling for convolutional neural networks,” in *International conference on machine learning*, PMLR, 2019, pp. 6105–6114.

[129] W. Zhou, J. Ji, Y. Jiang, J. Wang, Q. Qi, and Y. Yi, “EARDS: EfficientNet and attention-based residual depth-wise separable convolution for joint OD and OC segmentation,” *Front Neurosci*, vol. 17, p. 1139181, 2023.

[130] “ImageNet.” Accessed: Aug. 14, 2025. [Online]. Available: https://www.image-net.org/

[131] C. Sun, A. Shrivastava, S. Singh, and A. Gupta, “Revisiting unreasonable effectiveness of data in deep learning era,” in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 843–852.

[132] H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. Jégou, “Training data-efficient image transformers & distillation through attention,” in *International conference on machine learning*, PMLR, 2021, pp. 10347–10357.

[133] “Review: Data Efficient Image Transformer (DeiT) | by Sik-Ho Tsang | Medium.” Accessed: Sep. 01, 2025. [Online]. Available: https://sh-tsang.medium.com/review-deit-data-efficient-image-transformer-b5b6ee5357d0

[134] G. Hinton, O. Vinyals, and J. Dean, “Distilling the knowledge in a neural network,” *arXiv preprint arXiv:1503.02531*, 2015.

[135] “DeiT: Training data-efficient image transformers & distillation through attention.” Accessed: Aug. 16, 2025. [Online]. Available: https://hyoseok-personality.tistory.com/entry/Paper-Review-DeiT-Training-data-efficient-image-transformers-distillation-through-attention

[136] “DINO: Emerging Properties in Self-Supervised Vision Transformers | dino – Weights & Biases.” Accessed: Sep. 01, 2025. [Online]. Available: https://wandb.ai/self-supervised-learning/dino/reports/DINO-Emerging-Properties-in-Self-Supervised-Vision-Transformers--VmlldzoxMzM2MTAz

[137] “Tutorial 6.3: Self-distillation with no labels (DINO).” Accessed: Sep. 01, 2025. [Online]. Available: https://github.com/hamkerlab/DL_for_practitioners/blob/main/06_3_SSL_DiNO/06_3_SSL_DINO.ipynb

[138] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[139] “Real-time Object Detection with YOLO, YOLOv2 and now YOLOv3 | by Jonathan Hui | Medium.” Accessed: Jul. 20, 2025. [Online]. Available: https://jonathan-hui.medium.com/real-time-object-detection-with-yolo-yolov2-28b1b93e2088

[140] “Know your Neural Network architecture more by understanding these terms | by Megha Shroff | Medium.” Accessed: Jul. 20, 2025. [Online]. Available: https://medium.com/@shroffmegha6695/know-your-neural-network-architecture-more-by-understanding-these-terms-67faf4ea0efb

[141] “Ultralytics YOLO11 - Ultralytics YOLO Docs.” Accessed: Jul. 21, 2025. [Online]. Available: https://docs.ultralytics.com/models/yolo11/

[142] “Explore Ultralytics YOLOv8 - Ultralytics YOLO Docs.” Accessed: Jul. 21, 2025. [Online]. Available: https://docs.ultralytics.com/models/yolov8/

[143] R. Khanam and M. Hussain, “Yolov11: An overview of the key architectural enhancements. arXiv 2024,” *arXiv preprint arXiv:2410.17725*, 2024.

[144] “Mastering All YOLO Models from YOLOv1 to YOLOv12.” Accessed: Jul. 21, 2025. [Online]. Available: https://learnopencv.com/mastering-all-yolo-models/

[145] J. Huang, K. Wang, Y. Hou, and J. Wang, “LW-YOLO11: a lightweight arbitrary-oriented ship detection method based on improved YOLO11,” *Sensors*, vol. 25, no. 1, p. 65, 2024.

[146] “Datasets Overview - Ultralytics YOLO Docs.” Accessed: Aug. 28, 2025. [Online]. Available: https://docs.ultralytics.com/datasets/

[147] “Roboflow Universe: Computer Vision Datasets.” Accessed: Aug. 28, 2025. [Online]. Available: https://universe.roboflow.com/

[148] “Flickr | The best place to be a photographer online.” Accessed: Aug. 28, 2025. [Online]. Available: https://www.flickr.com/

[149] “How to Create a Duplicate Image Detection System | by Matt Podolak | TDS Archive | Medium.” Accessed: Aug. 28, 2025. [Online]. Available: https://medium.com/data-science/how-to-create-a-duplicate-image-detection-system-a30f1b68a2e3

[150] “Duplicate image detection with perceptual hashing in Python.” Accessed: Aug. 28, 2025. [Online]. Available: https://benhoyt.com/writings/duplicate-image-detection/

[151] Y. Jakhar and M. D. Borah, “Effective near-duplicate image detection using perceptual hashing and deep learning,” *Inf Process Manag*, vol. 62, no. 4, p. 104086, 2025.

[152] M. H. Mohiuddin and L. Tamilselvan, “IDedupNet: A MobileNetV3-Based Deep Learning Framework for Efficient Image Deduplication in Cloud Computing Environments,” *Informatica*, vol. 49, no. 13, 2025.

[153] K. Abhishek, A. Jain, and G. Hamarneh, “Investigating the quality of dermamnist and fitzpatrick17k dermatological image datasets,” *Sci Data*, vol. 12, no. 1, p. 196, 2025.

[154] H. Bichri, A. Chergui, and M. Hain, “Investigating the Impact of Train/Test Split Ratio on the Performance of Pre-Trained Models with Custom Datasets.,” *International Journal of Advanced Computer Science & Applications*, vol. 15, no. 2, 2024.

[155] “Albumentations: fast and flexible image augmentations.” Accessed: Aug. 28, 2025. [Online]. Available: https://albumentations.ai/

[156] “Models and pre-trained weights — Torchvision 0.23 documentation.” Accessed: Sep. 03, 2025. [Online]. Available: https://docs.pytorch.org/vision/stable/models.html

[157] Z. He, Y. Lian, Y. Wang, and Z. Lu, “A comprehensive review of research on surface defect detection of PCBs based on machine vision,” *Results in Engineering*, vol. 27, p. 106437, 2025, doi: https://doi.org/10.1016/j.rineng.2025.106437.

[158] M. Dayıoğlu, A. K. Eyüboğlu, and R. Unal, “Performance Analysis of YOLO11 Models in PCB Defect Detection Tasks,” *Kuzey Ege Teknik Bilimler ve Teknoloji Dergisi*, vol. 2, no. 1, pp. 33–50, 2025.

[159] G. Xiao, S. Hou, and H. Zhou, “PCB defect detection algorithm based on CDI-YOLO,” *Sci Rep*, vol. 14, no. 1, p. 7351, 2024, doi: 10.1038/s41598-024-57491-3.

[160] B. Du, F. Wan, G. Lei, L. Xu, C. Xu, and Y. Xiong, “YOLO-MBBi: PCB surface defect detection method based on enhanced YOLOv5,” *Electronics (Basel)*, vol. 12, no. 13, p. 2821, 2023.

[161] M. Yuan, Y. Zhou, X. Ren, H. Zhi, J. Zhang, and H. Chen, "YOLO-HMC: An Improved Method for PCB Surface Defect Detection," *IEEE Trans Instrum Meas*, vol. 73, pp. 1–11, 2024, doi: 10.1109/TIM.2024.3351241.

[162] N. Hütten, M. Alves Gomes, F. Hölken, K. Andricevic, R. Meyes, and T. Meisen, "Deep learning for automated visual inspection in manufacturing and maintenance: a survey of open-access papers," *Applied System Innovation*, vol. 7, no. 1, p. 11, 2024.

[163] K. An and Y. Zhang, "LPViT: a transformer based model for PCB image classification and defect detection," *Ieee Access*, vol. 10, pp. 42542–42553, 2022.

[164] A. Dosovitskiy *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[165] A. El-Nouby, G. Izacard, H. Touvron, I. Laptev, H. Jegou, and E. Grave, "Are large-scale datasets necessary for self-supervised pre-training?," *arXiv preprint arXiv:2112.10740*, 2021.

[166] J. E. See, C. G. Drury, A. Speed, A. Williams, and N. Khalandi, "The role of visual inspection in the 21st century," in *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, SAGE Publications Sage CA: Los Angeles, CA, 2017, pp. 262–266. doi: 10.1177/154193121360154.

[167] C. E. Nwankpa, W. Ijomah, and A. Gachagan, "Design for automated inspection in remanufacturing: A discrete event simulation for process improvement," *Clean Eng Technol*, vol. 4, p. 100199, 2021, doi: https://doi.org/10.1016/j.clet.2021.100199.

[168] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-cam: Visual explanations from deep networks via gradient-based localization," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 618–626.

[169] L. McInnes, J. Healy, and J. Melville, "Umap: Uniform manifold approximation and projection for dimension reduction," *arXiv preprint arXiv:1802.03426*, 2018.

[170] Y. Ma, J. Yin, F. Huang, and Q. Li, "Surface defect inspection of industrial products with object detection deep networks: a systematic review," *Artif Intell Rev*, vol. 57, no. 12, p. 333, 2024, doi: 10.1007/s10462-024-10956-3.

[171] Y. Wan, L. Gao, X. Li, and Y. Gao, “Semi-supervised defect detection method with data-expanding strategy for PCB quality inspection,” *Sensors*, vol. 22, no. 20, p. 7971, 2022.

[172] Y. Xu, H. Wu, Y. Liu, and X. Liu, “Printed Circuit Board Sample Expansion and Automatic Defect Detection Based on Diffusion Models and ConvNeXt,” *Micromachines (Basel)*, vol. 16, no. 3, p. 261, 2025.

[173] L. van der Maaten and G. Hinton, “Visualizing data using t-SNE,” *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[174] S. Abnar and W. Zuidema, “Quantifying attention flow in transformers,” *arXiv preprint arXiv:2005.00928*, 2020.

[175] R. Arboretti, R. Ceccato, L. Pegoraro, and L. Salmaso, “Design of Experiment-based Configuration of Hyperparameters Of An Artificial Neural Network,” in *Proceedings of the 2020 JSM-Joint Statistical Meetings*, 2020, pp. 1735–1743.

[176] A. T. Khan and S. M. Jensen, “LEAF-Net: A unified framework for leaf extraction and analysis in multi-crop phenotyping using YOLOv11,” *Agriculture*, vol. 15, no. 2, p. 196, 2025.

[177] J. Huang, K. Wang, Y. Hou, and J. Wang, “LW-YOLO11: a lightweight arbitrary-oriented ship detection method based on improved YOLO11,” *Sensors*, vol. 25, no. 1, p. 65, 2024.

[178] H. Touvron, M. Cord, A. Sablayrolles, G. Synnaeve, and H. Jégou, “Going deeper with image transformers,” in *Proceedings of the IEEE/CVF international conference on computer vision*, 2021, pp. 32–42.

[179] M. Caron *et al.*, “Emerging properties in self-supervised vision transformers,” in *Proceedings of the IEEE/CVF international conference on computer vision*, 2021, pp. 9650–9660.

[180] M. Tan and Q. Le, “Efficientnet: Rethinking model scaling for convolutional neural networks,” in *International conference on machine learning*, PMLR, 2019, pp. 6105–6114.

[181] “EfficientNet: Improving Accuracy and Efficiency through AutoML and Model Scaling.” Accessed: Aug. 20, 2025. [Online]. Available: https://research.google/blog/efficientnet-improving-accuracy-and-efficiency-through-automl-and-model-scaling/