UNIVERSIDAD EUROPEA DE MADRID

ESCUELA DE INGENIERÍA Y DE ARQUITECTURA

MÁSTER UNIVERSITARIO EN INGENIERÍA AERONÁUTICA

TRABAJO FIN DE MÁSTER

COMPUTER VISION FOR AUTOMATED QUALITY CONTROL: DEFECT DETECTION IN METALLURGICAL PROCESSES USING DEEP LEARNING

Author: León Enrique PRIETO BAILO
Tutor: Alan DOMÍNGUEZ MONTERO

June 2025

to obtain the degree of Master of Science in Aerospace Engineering

at the Universidad Europea de Madrid

to be defended publicly in June 2025.

“Todo ser humano, si se lo propone, puede ser escultor de su propio cerebro”

— Santiago Ramón y Cajal

Dedicatoria

A mi madre, por saber despertar en mí esa curiosidad insaciable
y alimentarla con tanto conocimiento.

A mi padre, por su esfuerzo y por proveerme de todos los medios
que han hecho posible este logro.

A mi hermano, por su inagotable originalidad y por ser,
siempre, mi compañero de vida.

Acknowledgment

I would like to express my sincere gratitude to Professor Alan for his invaluable guidance, support, and encouragement throughout the course of this work. His insights and expertise were instrumental in shaping the direction and quality of this project.

Título: Visión por Computador para el Control de Calidad Automatizado: Detección de Defectos en Procesos Metalúrgicos mediante Aprendizaje Profundo.

Autor: León Enrique Prieto Bailo

Director: Alan Domínguez Montero

Fecha: 8 de junio de 2025

Resumen

Esta tesis analiza la viabilidad de automatizar tareas de control de calidad en procesos de fabricación metalúrgica mediante técnicas de visión por computador, alineando dicho enfoque con los principios de la Industria 4.0 y la manufactura inteligente. El estudio parte de la necesidad industrial de mejorar la fiabilidad de las inspecciones, reducir los costes operativos y superar las limitaciones inherentes a la inspección visual manual, aún predominante en muchas líneas de producción convencionales. Aprovechando los avances recientes en aprendizaje profundo, este trabajo aborda la brecha tecnológica entre los enfoques tradicionales de aseguramiento de calidad y las soluciones modernas basadas en datos.

Se implementan diversas estrategias para el desarrollo de modelos, abarcando marcos de trabajo de última generación para detección y clasificación de objetos. Entre ellos se incluye el uso del framework YOLOv8 de Ultralytics, tanto para clasificación como para segmentación semántica, así como el diseño y entrenamiento de redes neuronales convolucionales personalizadas mediante las librerías Keras y TensorFlow. Además, se incorporan metodologías de entrenamiento avanzadas, como el aprendizaje por transferencia, el ajuste fino (fine-tuning) y el aprendizaje semisupervisado, con el objetivo de optimizar el rendimiento en escenarios con datos limitados o parcialmente anotados. Para facilitar el análisis del comportamiento interno de los modelos y aportar transparencia a las decisiones, se aplica el algoritmo Grad-CAM, que permite visualizar regiones relevantes dentro de las imágenes procesadas.

El trabajo experimental se desarrolla siguiendo una metodología estructurada que incluye análisis exploratorio de datos, preprocesamiento del conjunto de imágenes, entrenamiento iterativo de modelos y evaluación cuantitativa de resultados. Se exploran diversas configuraciones arquitectónicas, incluyendo soluciones híbridas que combinan propuestas de región generadas por YOLOv8 con clasificadores desarrollados en Keras, entrenados sobre segmentos recortados de las imágenes originales. Asimismo, se desarrolla un pipeline personalizado para extraer y convertir los backbones de segmentación de YOLOv8 en estructuras compatibles con Keras, permitiendo su integración con nuevas cabezas de clasificación y posterior ajuste fino.

Los resultados obtenidos se analizan utilizando múltiples métricas de evaluación, tales como precisión, exactitud, recall, F1-score y mAP. Se demuestra que los modelos basados en aprendizaje profundo son capaces de distinguir con alta fiabilidad entre piezas defectuosas y no defectuosas. En particular, los clasificadores desarrollados con Keras presentan un rendimiento destacado en varias métricas, evidenciando una notable capacidad de generalización y resistencia a la variabilidad de los datos. La aplicación de Grad-CAM mejora la interpretabilidad del sistema, generando mapas de calor que resultan visualmente coherentes con la percepción humana y fortalecen la confianza en el proceso de inspección automatizada.

La tesis concluye reafirmando la viabilidad de desplegar algoritmos explicables de visión por computador para el control de calidad en entornos industriales. El análisis comparativo entre distintas arquitecturas permite identificar ventajas y compromisos asociados a cada configuración, en términos de complejidad, tiempo de inferencia y facilidad de integración. Finalmente, se subraya la importancia de la inteligencia artificial explicable (XAI) para la validación, depuración y aceptación normativa de sistemas de inspección automatizada. En este contexto, técnicas como Grad-CAM se consolidan como herramientas esenciales para la transparencia y robustez de los modelos en aplicaciones industriales.

Title: Computer Vision for Automated Quality Control: Defect Detection in Metallurgical Processes using Deep Learning.

Author: León Enrique Prieto Bailo

Advisor: Alan Domínguez Montero

Date: June 8, 2025

Abstract

This thesis explores the feasibility of automating quality control tasks in metallurgical manufacturing processes using computer vision techniques, with a clear alignment to the principles of Industry 4.0 and smart manufacturing. The study is motivated by the industrial need to enhance inspection reliability, reduce operational costs, and mitigate the limitations of manual visual inspection, which remains dominant in many traditional production lines. By leveraging advances in deep learning, this work addresses the technological gap between conventional quality assurance approaches and modern, data-driven alternatives.

A wide range of model development strategies is implemented, covering state-of-the-art object detection and classification frameworks. These include the application of the Ultralytics YOLOv8 framework for both classification and semantic segmentation, as well as the construction and training of custom convolutional neural networks using the Keras and TensorFlow libraries. In addition to these core approaches, the thesis incorporates advanced training methodologies such as transfer learning, fine-tuning, and semi-supervised learning to optimize model performance in scenarios with limited annotated data. To further investigate model behavior and decision transparency, the Gradient-weighted Class Activation Mapping (Grad-CAM) algorithm is employed, enabling visual inspection of feature importance and model focus regions.

The experimental work is conducted using a structured methodology that begins with extensive exploratory data analysis and dataset preprocessing, followed by iterative model training and evaluation. Various architectural configurations are explored, including hybrid pipelines that combine YOLOv8-based region proposals with Keras classifiers trained on cropped segments of the input data. Furthermore, a custom pipeline is developed to extract and convert YOLOv8 segmentation backbones into Keras-compatible formats, allowing the fine-tuning of pretrained feature extractors integrated with bespoke classification heads.

Quantitative results are rigorously analyzed using multiple evaluation metrics, such as accuracy, precision, recall, F1-score, and mean Average Precision (mAP). The findings demonstrate that deep learning models can achieve high confidence levels in distinguishing between defective and non-defective parts. In particular, classification models developed with Keras consistently yield superior performance across several metrics, suggesting strong generalization capabilities and robustness to data variability. The integration of Grad-CAM enhances model interpretability, producing visually intuitive heatmaps that align well with human visual judgment, thereby increasing trust in the automated inspection system.

The thesis concludes by reaffirming the potential of deploying explainable computer vision algorithms for real-time quality control in industrial settings. The comparative evaluation of different architectures reveals key trade-offs in terms of model complexity, inference time, and deployment feasibility. Additionally, the work emphasizes the importance of explainable artificial intelligence (XAI) for the validation, debugging, and regulatory acceptance of deep learning-based inspection systems. Grad-CAM, in particular, emerges as an indispensable tool for understanding and validating model predictions, consolidating its role within industrial inspection pipelines.

Contents

1 Introduction **17**
1.1 Motivation . . . 17
1.1.1 Industrial Relevance . . . 18
1.1.2 Economic Impact . . . 19
1.1.3 Limitations of Traditional Inspection Methods . . . 20
1.1.4 Technological Gap and Research Opportunity . . . 21
1.1.5 Alignment with Industry 4.0 and Smart Manufacturing . . . 21
1.1.6 Social and Environmental Sustainability . . . 23
1.1.7 Safety and Regulatory Compliance . . . 23
1.2 State of the Art . . . 24
1.2.1 Traditional Approaches to Defect Detection . . . 24
1.2.2 Machine Learning and Early Computer Vision Techniques . . . 25
1.2.3 Deep Learning in Computer Vision . . . 26
1.2.4 Applications of Deep Learning in Defect Detection . . . 27
1.2.5 Alternative and Emerging Approaches . . . 29
1.2.6 Challenges and Limitations in Current Research . . . 30
1.3 Thesis Structure . . . 31

2 Theoretical Background: Machine Learning and Computer Vision **33**
2.1 Fundamentals of Machine Learning . . . 34
2.1.1 Definition, Scope and Taxonomy of Learning Algorithms . . . 34
2.1.2 Learning Paradigms (Supervised, Unsupervised, Semi-Supervised, Self-Supervised, Reinforcement) . . . 35
2.1.3 Bias–Variance Trade-off and Generalisation . . . 36
2.1.4 Model Capacity, Under-/Over-fitting and Regularisation . . . 37
2.1.5 Evaluation Metrics (Accuracy, Precision/Recall, F_1, mAP, IoU, Dice) . . . 38
2.2 Fundamentals of Computer Vision . . . 39
2.2.1 Image Formation, Colour Spaces and Pre-processing . . . 40
2.2.2 Classical Vision Methods (Edge Detectors, HOG, SIFT, SURF) . . . 40
2.2.3 Feature Extraction and Representation Learning . . . 41
2.2.4 Typical Computer Vision Problem Formulations . . . 42
2.3 Deep-Learning Architectures for Vision . . . 44
2.3.1 Convolutional Neural Networks (CNNs) . . . 44
2.3.2 Residual and Efficient Architectures (ResNet, EfficientNet, MobileNet) . . . 45
2.3.3 Object-Detection Frameworks (R-CNN Family, SSD, YOLO) . . . 46
2.3.4 Semantic & Instance Segmentation (FCN, U-Net, Mask R-CNN) . . . 46
2.3.5 Vision Transformers and Hybrid Models . . . 47
2.3.6 Model Architecture Components: Backbone, Neck, and Head . . . 48
2.4 Training Strategies and Data Handling . . . 49
2.4.1 Data Augmentation, Synthetic and Domain-Randomised Data . . . 49
2.4.2 Transfer Learning, Fine-Tuning and Continual Learning . . . 50
2.4.3 Hyper-parameter Optimisation and Experiment Tracking . . . 51
2.4.4 Imbalanced Data and Rare-Defect Mitigation . . . 52
2.5 Industrial Deployment Considerations . . . 52
2.5.1 Model Compression, Pruning and Quantisation . . . 53
2.5.2 Real-Time Inference on Edge/Embedded Hardware . . . 53
2.5.3 Integration with PLCs, MES and SCADA Systems . . . 54

3 Methodology and Experimental Work **57**
3.1 Software Libraries and Deep Learning Frameworks . . . 58
3.1.1 Python & Scientific Stack (NumPy, Pandas, Matplotlib) . . . 58
3.1.2 Ultralytics YOLO (v8) . . . 59
3.1.3 TensorFlow / Keras . . . 60
3.2 Data and Computational Platforms . . . 60
3.2.1 Roboflow . . . 60
3.2.2 Google Colab . . . 61
3.3 Pre-processing and Data Preparation . . . 61
3.3.1 Exploratory Data Analysis . . . 62
3.3.1.1 Directory Structure . . . 63
3.3.1.2 File Metadata . . . 63
3.3.1.3 File Content . . . 64
3.3.2 Data Cleaning and Verification . . . 68
3.3.2.1 Duplicate Element Analysis . . . 68
3.3.2.2 Greyscale Analysis . . . 70
3.3.2.3 Unique Identifier Generation . . . 70
3.3.3 Train/Validation/Test Split Strategy . . . 71
3.4 Model Development . . . 72
3.4.1 YOLOv8 Classification . . . 73
3.4.1.1 Dependencies . . . 73
3.4.1.2 Creation of Train and Val Datasets . . . 73
3.4.1.3 Model Training . . . 74
3.4.1.4 Validation on New Data . . . 74
3.4.2 YOLOv8 Segmentation . . . 75
3.4.2.1 Indexing the Dataset in Roboflow . . . 76
3.4.2.2 Labeling . . . 76
3.4.2.3 Dataset Generation . . . 78
3.4.2.4 Importing the Dataset into Google Colab . . . 78
3.4.2.5 Training the Initial Model . . . 78
3.4.2.6 Model Export and Deployment . . . 79
3.4.2.7 Annotation Verification and Inference . . . 79
3.4.2.8 Manual Filtering . . . 80
3.4.2.9 Model Training . . . 80
3.4.3 Keras/TensorFlow Classification . . . 81
3.4.3.1 Dataset Loading . . . 81
3.4.3.2 Data Augmentation . . . 81
3.4.3.3 Normalization . . . 82
3.4.3.4 Model Definition . . . 82
3.4.3.5 Class Balance . . . 83
3.4.3.6 Model Compilation . . . 84
3.4.3.7 Callbacks . . . 85
3.4.3.8 Model Training . . . 85
3.4.4 YOLOv8 Segmentation (Crop) + Keras/TensorFlow Classification . . . 85
3.4.4.1 Preprocessing . . . 86
3.4.4.2 Manual Postprocessing . . . 87
3.4.4.3 Model Training . . . 88
3.4.5 YOLOv8 Segmentation (Backbone) + Keras/TensorFlow Classification + Fine-Tuning . . . 88
3.4.5.1 Backbone Extraction . . . 89
3.4.5.2 Manual Conversion . . . 91
3.4.5.2.1 Weight Transfer . . . 92
3.4.5.2.1.1 Conversion of Convolutional (Conv2D) Weights . . . 92
3.4.5.2.1.2 Batch Normalization (BatchNorm2D) Weights . . . 93
3.4.5.2.2 Layer Conversion . . . 94
3.4.5.2.2.1 ConvModule . . . 94
3.4.5.2.2.2 Bottleneck . . . 95

3.4.5.2.2.3 C2f Block (Cross Stage Partial Layer) . . . 96
3.4.5.2.2.4 SPPF (Spatial Pyramid Pooling – Fast) . . . 98
3.4.5.3 Backbone Assembly . . . 99
3.4.5.4 Validation of the Converted Model . . . 99
3.4.5.5 Integration of the Classification Head . . . 100
3.4.5.6 Manual Dataset Adaptation . . . 101
3.4.5.7 Final Configuration and Training . . . 102
3.4.5.8 Fine-Tuning . . . 102
3.5 Grad-CAM Demonstration . . . 103
3.5.1 Pretrained Model Selection . . . 104
3.5.2 DenseNet121 Integration and Processing Pipeline . . . 105
3.5.3 Application of the Grad-CAM Algorithm . . . 106

4 Experimental Results and Analysis 109
4.1 Experimental Setup . . . 109
4.1.1 Computational Environment and Hardware Specifications . . . 110
4.1.2 Training Protocols and Hyperparameter Configuration . . . 110
4.1.3 Evaluation Metrics . . . 111
4.2 Model Results . . . 112
4.2.1 YOLOv8 Classification . . . 112
4.2.2 YOLOv8 Segmentation . . . 116
4.2.3 Keras/TensorFlow Classification . . . 121
4.2.4 YOLOv8 Segmentation (Crop) + Keras/TensorFlow Classification . . . 123
4.2.5 YOLOv8 Segmentation (Backbone) + Keras/TensorFlow Classification + Fine-Tuning . . . 125
4.2.5.1 Base Model . . . 126
4.2.5.2 Training: Stages 8–9 . . . 128
4.2.5.3 Training: Stages 6–7 . . . 129
4.2.5.4 Training: Stages 4–5 . . . 131
4.2.5.5 Training: Stages 2-3 . . . 132
4.2.5.6 Training: Stages 0-1 . . . 134
4.3 Grad-CAM Demonstration Results . . . 136

5 Conclusions 141

Bibliography 147

A Source Code for Model Training 151
A.1 YOLOv8 Classification . . . 151
A.2 YOLOv8 Segmentation (Pure) . . . 152
A.3 YOLOv8 Segmentation (Pseudo) . . . 154
A.4 Keras/TensorFlow Classification . . . 156
A.5 YOLOv8 Segmentation (Crop) + Keras/TensorFlow Classification . . . 158
A.6 YOLOv8 Segmentation (Backbone) + Keras/TensorFlow Classification + Fine-Tuning . . . 160

B Source Code and Asset Repository 167

1 Introduction

Artificial intelligence (AI) has fundamentally transformed numerous sectors, with computer vision emerging prominently as one of its most influential domains. Over the last decade, substantial advancements in deep learning methodologies, particularly convolutional neural networks (CNNs), have revolutionized machine interpretation of visual data. This evolution has been significantly propelled by increased computational capacities, the proliferation of expansive labelled datasets, and sophisticated algorithmic developments. More recent innovations, such as vision transformers, generative adversarial networks (GANs), and transfer learning techniques, have further expanded the potential of computer vision technologies. These advancements now enable machines not only to interpret visual data accurately but often to outperform human capabilities in complex tasks such as object recognition, semantic segmentation, and anomaly detection.[1]

The widespread availability of computational resources, exemplified by affordable GPUs and cloud computing infrastructure, along with the proliferation of open-source software frameworks such as TensorFlow, PyTorch, and OpenCV, have substantially democratized computer vision technology. Such accessibility has lowered the barriers to entry, allowing entities ranging from large corporations to small startups to develop and deploy sophisticated, AI-driven solutions with relative ease. Additionally, pre-trained architectures like ResNet, EfficientNet, and YOLO facilitate rapid adaptation of state-of-the-art models, significantly reducing the requirement for extensive training data and computational resources. Consequently, computer vision has transitioned from being a specialized academic pursuit into a foundational component of contemporary industrial automation.[2]

Today, computer vision is integral to critical applications spanning diverse sectors, including healthcare diagnostics, precision agriculture, autonomous driving systems, and intelligent retail solutions. Its capability to process vast amounts of visual data consistently, objectively, and tirelessly positions it as an essential tool for industries striving for scalable and error-resistant operational solutions. Furthermore, as global industries increasingly emphasize sustainable practices, computer vision's efficiency and accuracy contribute directly to reducing waste, improving resource utilization, and enhancing overall environmental sustainability.

1.1. Motivation

In industrial manufacturing, the accurate differentiation between valid and defective products remains a critical and ongoing challenge. Traditional quality assurance methods—frequently dependent on manual inspections or basic optical sensing technologies—suffer from several inherent limitations. Human inspections, although intuitive, are vulnerable to fatigue-induced errors, subjectivity biases, and restricted scalability. Rule-based optical systems often fail to adequately identify subtle, variable, or unprecedented defect types, which can result in defective products passing through quality checks undetected. In high-stakes industries such as automotive manufacturing, electronics assembly, medical device production, aerospace, and pharmaceuticals, minor undetected defects—including micro-fractures, alignment errors, or surface anomalies—can severely compromise product safety, cause costly recalls, diminish consumer trust, and negatively impact brand reputation.

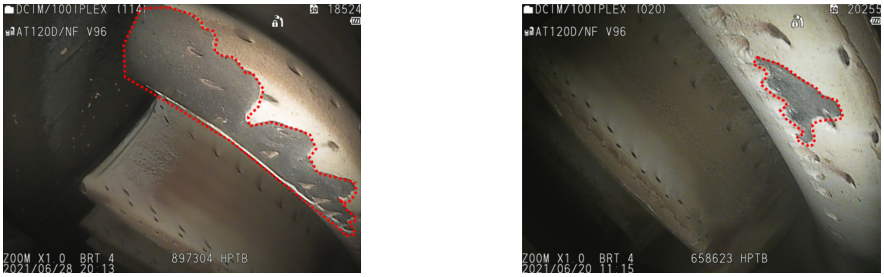


Figure 1.1: Inspection images showing areas of missing Thermal Barrier Coating (TBC) on turbine blades, with the degraded regions highlighted in red.[3]

Employing computer vision for automated defect detection directly addresses and overcomes these limitations. Advanced computer vision models, capable of analysing thousands of products hourly with meticulous pixel-level accuracy, can reliably detect and classify defects invisible or easily overlooked by human inspectors. Such automated inspection systems significantly reduce operational inefficiencies, lower costs associated with waste and rework and markedly improve overall production reliability. Moreover, automated visual inspection aligns well with contemporary global sustainability goals, minimizing resource wastage and environmental impacts by swiftly identifying and removing defective products from production lines.

Automated inspection systems powered by computer vision are especially indispensable in sectors where product integrity directly relates to safety and regulatory compliance. In industries such as aerospace and pharmaceuticals, where rigorous adherence to quality standards is non-negotiable, the deployment of robust defect detection systems becomes essential for ensuring consumer safety and meeting stringent compliance requirements. Furthermore, automation liberates human inspectors from repetitive and labour-intensive tasks, allowing their skills and expertise to be redirected towards higher-value activities, including strategic oversight, process optimization, innovation, and continuous improvement.

This thesis is fundamentally motivated by the pressing industrial necessity for efficient, reliable, and versatile defect detection systems powered by artificial intelligence. By bridging the gap between advanced AI research and tangible industrial applications, the proposed neural network model aims to enable manufacturers to realize shorter production cycles, improved product consistency, and robust compliance with evolving industry standards and quality benchmarks. Ultimately, this work seeks to contribute actively to the advancement of Industry 4.0, envisioning an integrated future where intelligent automation facilitates unprecedented levels of efficiency, sustainability, adaptability, and resilience across global manufacturing supply chains.

1.1.1. Industrial Relevance

The increasing complexity of modern manufacturing processes and the growing demand for high-quality products across global markets have elevated the importance of reliable quality assurance systems. In industrial contexts, defect detection is not merely a supportive function but a fundamental component of production workflows, directly influencing throughput, yield rates, and customer satisfaction. As manufacturing ecosystems become more integrated and data-driven, the capacity to detect, classify, and respond to product anomalies in real time has become a critical differentiator among industrial leaders.

Industries such as automotive, electronics, aerospace, and precision manufacturing operate under stringent tolerances and high output expectations. In these sectors, even minor defects—such as micro-cracks in structural components, misalignments in circuit boards, or surface irregularities in high-speed mechanical parts—can lead to catastrophic failures, product recalls, or compliance violations. These risks underscore the need for advanced inspection technologies capable of ensuring product integrity at every stage of the production cycle. Traditional inspection techniques, which often rely on manual assessments or rigid rule-based systems, are increasingly inadequate in addressing the complexity and variability of modern production environments.

Computer vision systems powered by deep learning present a scalable and adaptive solution to these in-

dustrial challenges. Their ability to learn from vast datasets and generalize to novel defect types enables them to perform with high accuracy across diverse material textures, lighting conditions, and product geometries. Such systems can be seamlessly integrated into high-speed production lines, offering continuous, non-invasive monitoring with minimal human intervention. As a result, manufacturers can significantly reduce the incidence of defective products reaching end users, while also lowering the costs associated with scrap, rework, and downtime.

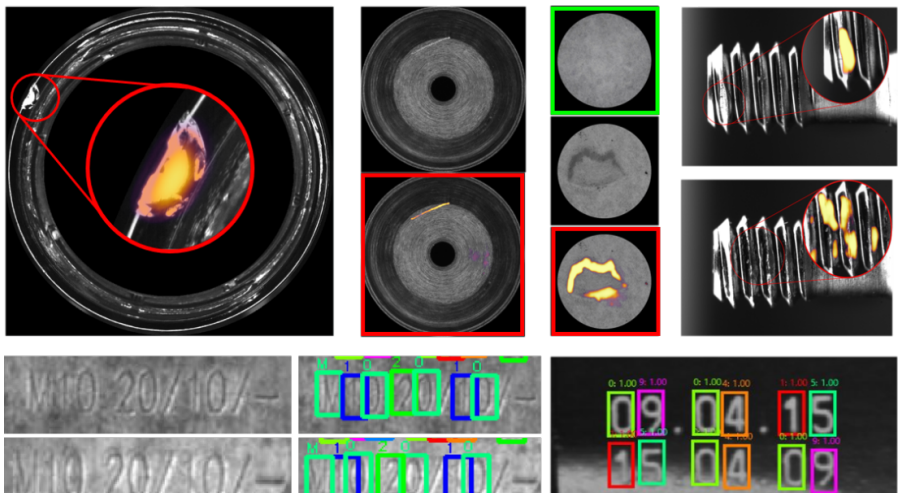


Figure 1.2: Defect detection and OCR results on mechanical parts using machine vision techniques, highlighting anomalies with bounding boxes and heatmaps. Source: H3LE[4].

Furthermore, the implementation of intelligent defect detection aligns with the principles of predictive maintenance and real-time quality control, both of which are foundational pillars of Industry 4.0. By embedding advanced inspection capabilities into manufacturing pipelines, companies can transition from reactive to proactive quality management strategies, enhancing both operational efficiency and long-term competitiveness. In this context, the development of robust, AI-driven inspection systems is not merely an enhancement but a strategic necessity for modern industrial operations. [5]

1.1.2. Economic Impact

Defect detection systems exert a substantial influence on the economic performance of manufacturing operations. Undetected defects can result in costly product recalls, customer dissatisfaction, warranty claims, and reputational damage—each of which can erode profitability and market share. In high-volume production environments, even a marginal increase in defect rates can translate into significant financial losses. Therefore, effective quality control mechanisms are not only a matter of technical precision but also of economic sustainability.

Manual inspection processes, while still common in many industries, are labour-intensive, inconsistent, and limited in scalability. They represent a recurring operational cost, particularly when considering the need for continuous staffing, training, and oversight. Moreover, the human factor introduces variability and error, especially under conditions of fatigue, time pressure, or high throughput. The economic inefficiencies associated with these traditional methods become increasingly pronounced as manufacturing scales up and product complexity increases.

Conversely, the integration of automated, AI-powered defect detection systems enables manufacturers to achieve significant cost reductions in both direct and indirect domains. Direct savings stem from lower labour costs, reduced need for rework, and decreased material waste. Indirect benefits include enhanced production line efficiency, minimized downtime, and faster time-to-market due to early fault identification and correction. Over time, these improvements contribute to higher overall equipment effectiveness (OEE) and return on investment (ROI), strengthening the economic case for intelligent automation.

Additionally, robust defect detection systems contribute to the avoidance of regulatory fines and liabilities that may arise from delivering non-compliant or unsafe products. In sectors where safety and compliance are heavily regulated—such as aerospace, automotive, and pharmaceuticals—proactive quality assurance is not only a legal requirement but also a critical factor in maintaining customer contracts and certifications. In this regard, the economic value of advanced defect detection extends beyond operational cost savings to encompass risk mitigation and long-term strategic resilience.

By investing in AI-driven quality control solutions, manufacturers position themselves to thrive in an increasingly competitive global economy, where efficiency, precision, and reliability are key determinants of success.[6]

1.1.3. Limitations of Traditional Inspection Methods

Despite their longstanding use in industrial quality control, traditional inspection methods face significant limitations that undermine their effectiveness in modern manufacturing environments. These methods, typically based on manual visual inspection or rule-based optical systems, are increasingly insufficient for meeting the precision, speed, and adaptability demanded by contemporary production lines.

Figure 1.3: A factory worker conducts a manual visual inspection of a metal component under a magnifying lamp. Source: International Institute of Non Destructive Testing[7].

Manual inspection, while intuitively appealing due to human perceptual capabilities, suffers from critical drawbacks. Human inspectors are prone to fatigue, inconsistency, and subjective judgment, particularly in high-speed or repetitive inspection settings. Studies have shown that human accuracy in visual inspection tasks can degrade rapidly over time, leading to higher rates of false positives and false negatives. This not only affects product quality but also contributes to increased rework, scrap, and overall inefficiency. Moreover, scaling manual inspection to match high production volumes often entails significant labour costs and logistical complexities, rendering it economically unsustainable.

Rule-based machine vision systems, which rely on predefined heuristics or handcrafted features, offer an automated alternative but lack the flexibility to handle variability and unforeseen defect types. These systems typically operate under tightly constrained conditions—fixed lighting, rigid object positioning, and narrowly defined defect criteria. As a result, they struggle to adapt to subtle, complex, or rare anomalies, especially when defects manifest in forms not explicitly encoded into the system. Modifying or extending these rule-based systems to accommodate new products or defect classes often requires extensive manual reconfiguration and expert intervention.

Furthermore, both manual and rule-based approaches are generally limited in their ability to capture contextual information. They may overlook defects that only become apparent when analysing spatial or temporal patterns, such as emerging trends across batches or inconsistencies related to upstream process variations. This lack of deeper analytical capacity restricts their utility in predictive maintenance, process optimization, and real-time decision-making.

In light of these limitations, there is a growing consensus across industrial and academic communities regarding the need for more intelligent, adaptive, and scalable inspection solutions. Deep learning-based computer vision systems, with their capacity to learn from data, generalize to new scenarios, and detect subtle patterns, offer a promising path forward. Their adoption not only addresses the shortcomings of traditional methods but also opens up new possibilities for integrated, data-driven manufacturing strategies aligned with the principles of Industry 4.0.[8]

1.1.4. Technological Gap and Research Opportunity

Despite the remarkable progress in computer vision and deep learning over the past decade, several critical challenges persist in the deployment of automated defect detection systems within real-world industrial contexts. While many state-of-the-art models demonstrate impressive performance under controlled laboratory conditions or on curated datasets, their applicability to dynamic, heterogeneous, and data-scarce production environments remains limited. This disparity between theoretical advancements and practical adoption defines a clear technological gap—and a compelling research opportunity.

One of the most significant challenges involves the generalization of defect detection models across diverse product types, defect morphologies, and operational conditions. In practice, defects can vary widely in size, shape, texture, and frequency, and may occur on complex or reflective surfaces that challenge conventional imaging techniques. Furthermore, the rarity of some defect classes leads to highly imbalanced datasets, which in turn hinders the training of robust models and increases the likelihood of overfitting. The development of methods capable of learning effectively from limited or imbalanced data—such as transfer learning, data augmentation, and few-shot learning—thus represents a critical area of ongoing research.

Another key limitation lies in the scarcity of high-quality, labelled datasets tailored to specific industrial applications. Many publicly available defect detection datasets are either domain-specific, simplistic, or insufficient in scale, making it difficult to benchmark new models or adapt them to varied production settings. The creation of synthetic data using techniques such as generative adversarial networks (GANs), as well as the development of semi-supervised and unsupervised learning approaches, offers promising solutions to alleviate these constraints. However, these methods are still in their infancy and require further refinement to ensure reliability, accuracy, and interpretability in safety-critical applications.

Additionally, current models often operate as "black boxes", offering little insight into the reasoning behind their predictions. In industrial environments where traceability, accountability, and compliance are essential, the lack of interpretability poses a barrier to trust and adoption. Research into explainable AI (XAI) and interpretable deep learning is therefore crucial for increasing transparency and facilitating the integration of such systems within regulated workflows.

This thesis aims to address these technological gaps by developing a deep learning-based defect detection model that is both performant and practical. By incorporating strategies such as transfer learning and customized data preprocessing, the proposed approach seeks to overcome data limitations and enhance model adaptability. Furthermore, this work contributes to the broader research agenda of translating academic innovations into scalable, interpretable, and industry-ready solutions, thereby advancing the state of intelligent quality control systems in manufacturing.[9]

1.1.5. Alignment with Industry 4.0 and Smart Manufacturing

The evolution of modern industry is increasingly shaped by the principles of Industry 4.0, a paradigm characterized by the integration of cyber-physical systems, real-time data exchange, and intelligent automation across manufacturing environments. Central to this vision is the deployment of smart systems that enable

enhanced productivity, agility, and decision-making throughout the production lifecycle. In this context, advanced computer vision technologies—particularly those powered by deep learning—play a pivotal role in realizing the full potential of intelligent manufacturing ecosystems.

Smart manufacturing emphasizes responsiveness, flexibility, and self-optimization, all of which require accurate, timely, and granular data from every stage of the production process. Automated defect detection systems contribute directly to this goal by providing continuous, high-resolution monitoring of product quality without the need for manual intervention. Their capacity to identify and classify anomalies in real time enables rapid corrective actions, reduces waste, and minimizes the risk of defective products progressing through the supply chain. This real-time feedback loop is essential for maintaining production efficiency and ensuring high standards of quality and compliance.

Moreover, deep learning-based inspection systems support predictive and prescriptive maintenance strategies by detecting patterns and trends indicative of emerging faults, not only in products but also in the machinery and processes that produce them. By embedding these capabilities within a broader digital infrastructure—including Industrial Internet of Things (IIoT) platforms, digital twins, and edge computing devices—manufacturers can achieve a holistic view of operations and continuously refine their processes based on data-driven insights.

The integration of AI-powered defect detection systems also facilitates interoperability and scalability, two cornerstones of Industry 4.0. Modular inspection solutions can be deployed across multiple lines and facilities, adapted to new products with minimal reconfiguration, and connected to centralized data systems for unified monitoring and analytics. This enhances both horizontal and vertical integration, enabling seamless communication across departments, systems, and supply chain stakeholders.

Finally, aligning defect detection with smart manufacturing initiatives promotes long-term sustainability goals. By optimizing quality control, reducing unnecessary production steps, and minimizing material consumption, intelligent inspection systems contribute to more environmentally responsible production models—an increasingly important consideration in the global industrial landscape.

This thesis aligns closely with these objectives by proposing a deep learning-based defect detection approach that not only advances technical performance but also embodies the core tenets of Industry 4.0: connectivity, automation, intelligence, and adaptability. In doing so, it aims to support the transition toward smarter, more sustainable, and resilient manufacturing systems.[10]

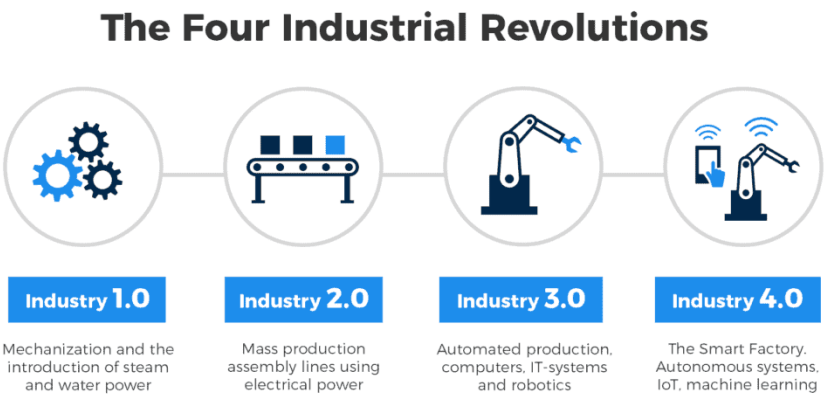


Figure 1.4: Summary of the four industrial revolutions, from mechanization to smart factories powered by IoT and AI. Source: KFactory[11].

1.1.6. Social and Environmental Sustainability

In an era of increasing ecological awareness and global social responsibility, the integration of sustainability principles into industrial practices has become both an ethical imperative and a strategic priority. Manufacturers are expected not only to optimize economic performance but also to minimize environmental impact, ensure worker well-being, and contribute positively to the broader societal context. Advanced defect detection systems powered by artificial intelligence directly support these objectives by promoting more sustainable and socially responsible manufacturing processes.

From an environmental perspective, automated defect detection significantly reduces material waste and energy consumption. By identifying defective products early in the production cycle—often before additional resources are committed—such systems prevent the unnecessary use of raw materials, reduce rework rates, and diminish the volume of defective goods that would otherwise be discarded. This efficiency contributes to a leaner, more resource-conscious manufacturing paradigm and aligns with international sustainability frameworks such as the United Nations Sustainable Development Goals (SDGs), particularly SDG 12: Responsible Consumption and Production.

Moreover, AI-driven inspection systems contribute to lowering the environmental footprint of industrial operations by enabling predictive maintenance and process optimization. By detecting anomalies in production processes in real time, manufacturers can proactively address inefficiencies that might otherwise lead to increased emissions, excessive energy use, or product non-conformity. When deployed at scale, these systems form part of a broader strategy to transition toward green manufacturing and circular economy models.

On the social dimension, automated visual inspection relieves human operators from monotonous, repetitive, and cognitively demanding tasks, thereby reducing occupational fatigue and the risk of human error. This technological shift allows workers to engage in more meaningful, creative, and value-added roles—such as process supervision, strategic analysis, or innovation—fostering a more stimulating and safer working environment. Additionally, by improving overall product quality and reliability, these systems contribute to enhanced consumer trust and safety, particularly in sectors where product integrity has direct implications for public health and well-being.

In this light, the implementation of intelligent defect detection systems represents a convergence of technological advancement with social and environmental responsibility. By contributing to waste reduction, energy efficiency, worker well-being, and product safety, these systems exemplify how artificial intelligence can serve as a catalyst for sustainable industrial transformation. The present thesis is guided by this vision, aiming to develop a solution that not only enhances operational excellence but also supports broader societal and ecological goals.[12]

1.1.7. Safety and Regulatory Compliance

In many industrial sectors, particularly those involving high-risk or safety-critical products—such as aerospace, automotive, pharmaceuticals, and medical devices—strict regulatory frameworks govern every aspect of production. Ensuring product safety and full compliance with these regulations is not only a legal obligation but a fundamental requirement for market access, consumer protection, and corporate accountability. In such environments, even minor defects can lead to catastrophic outcomes, including system failures, health risks, and loss of life. Consequently, robust and reliable defect detection mechanisms are indispensable for mitigating risk and safeguarding public safety.

Traditional inspection systems, with their inherent limitations in consistency, scalability, and adaptability, often struggle to meet the stringent quality standards imposed by regulatory bodies such as the U.S. Food and Drug Administration (FDA), the Federal Aviation Administration (FAA), the European Medicines Agency (EMA), or the International Organization for Standardization (ISO). Failure to comply with such standards can result in severe consequences, including financial penalties, forced product recalls, production shutdowns, and long-term reputational damage.

Advanced computer vision systems powered by deep learning provide a strategic advantage in this context by delivering highly accurate, repeatable, and objective inspection processes. Their capacity to operate con-

tinuously at high speed and with pixel-level granularity enables the detection of subtle and rare anomalies that may elude human inspectors or rule-based systems. Such capabilities are especially critical in scenarios where the margin for error is exceptionally small, and where regulatory audits demand comprehensive traceability and documentation of quality control procedures.

Moreover, AI-enabled inspection systems can be integrated with traceability and reporting tools to generate audit-ready records, facilitate root-cause analysis, and support the validation protocols required in regulated manufacturing environments. By enhancing transparency, accountability, and process standardization, these systems contribute to a culture of compliance and risk management that aligns with both internal governance frameworks and external regulatory expectations.

The present thesis is motivated by the need to bridge the gap between academic advancements in deep learning and their practical deployment in industrial settings governed by rigorous safety and compliance demands. Through the development of a reliable, interpretable, and adaptable defect detection system, this work aims to support manufacturers in meeting the highest standards of product integrity, regulatory conformance, and consumer protection.[13]

1.2. State of the Art

The ongoing transformation of industrial manufacturing has been deeply influenced by advances in artificial intelligence, particularly within the domain of computer vision. As industries increasingly seek to optimize quality assurance processes and minimize defects in production, the need for robust, accurate, and scalable defect detection systems has become paramount. Traditional inspection techniques—both manual and rule-based—have demonstrated substantial limitations in terms of reliability, scalability, and adaptability. In contrast, recent developments in machine learning and deep learning have paved the way for the creation of intelligent inspection systems capable of identifying subtle and complex anomalies with remarkable accuracy.

This section presents a comprehensive overview of the current state of the art in automated defect detection, with a specific focus on computer vision techniques. It begins by examining conventional approaches, including classical image processing methods and early machine learning models, highlighting their contributions and inherent constraints. Subsequently, it explores the emergence and evolution of deep learning architectures, which have significantly expanded the capabilities of visual inspection systems across a wide range of industrial contexts.

Furthermore, this review discusses how deep learning has been applied specifically to the task of defect detection, outlining the most relevant neural network architectures and their typical applications. It also considers alternative and emerging approaches—including 3D imaging, thermal sensing, and hybrid inspection systems—which complement or extend the possibilities of visual inspection. Finally, the section identifies key challenges that persist in the field, such as data scarcity, model generalization, and the need for interpretable AI, thereby motivating the research contribution of this thesis.

Through this analysis, the section aims to contextualize the proposed work within the broader research and technological landscape, providing the necessary foundation for understanding its novelty, relevance, and potential industrial impact.

1.2.1. Traditional Approaches to Defect Detection

Before the widespread adoption of artificial intelligence and deep learning techniques, defect detection in industrial environments largely relied on traditional computer vision and manual inspection strategies. These conventional approaches were typically based on handcrafted rules, classical image processing operations, and heuristic algorithms tailored to specific use cases. While such methods played a foundational role in early quality assurance systems, they exhibit several limitations that constrain their effectiveness in modern manufacturing contexts characterized by variability, complexity, and high throughput.

Classical image processing techniques—including thresholding, edge detection, blob analysis, and morphological transformations—have long been employed for identifying visual anomalies on product surfaces. These techniques operate by analysing low-level image features such as intensity gradients, colour histograms, or geometric contours. For example, thresholding methods detect regions of interest based on pixel intensity values, while edge detection algorithms like the Canny or Sobel operators aim to identify discontinuities in the image that may indicate structural defects. Although these approaches are computationally efficient and easy to implement, their performance is highly sensitive to changes in illumination, surface texture, orientation, and noise, which often occur in real-world industrial settings.

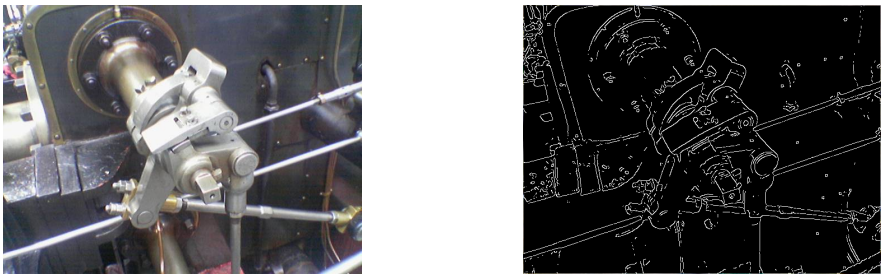


Figure 1.5: Original and Canny edge-processed images. Edge detection highlights key structures via intensity gradients. Source: Wikipedia: Canny Edge Detector. Source: Wikipedia Canny Edge Detector[14]

Another widely used category of traditional approaches involves template matching and rule-based inspection. These systems compare input images to predefined templates representing defect-free instances of a product. Deviations from the template are flagged as potential defects. However, such systems are inherently rigid: they assume consistency in product appearance and are unable to adapt to natural variability within the manufacturing process. As a result, they tend to produce high false positive rates when acceptable variations are misclassified as defects, or false negatives when subtle or novel anomalies deviate from the narrow assumptions encoded in the rules.

Manual inspection, while still prevalent in many industries, also presents significant challenges. Human inspectors are subject to fatigue, perceptual limitations, and cognitive biases, particularly when performing repetitive tasks under time constraints. Furthermore, manual inspection lacks consistency across operators and shifts, making it difficult to ensure uniform quality standards. While skilled inspectors may excel in identifying complex or context-dependent defects, the scalability of this approach is inherently limited, especially in high-volume production environments.

In summary, traditional defect detection approaches—despite their historical importance and simplicity—lack the robustness, flexibility, and scalability required for contemporary manufacturing demands. Their dependence on domain-specific knowledge, sensitivity to environmental conditions, and limited generalization capabilities underscore the need for more advanced, data-driven methods. These limitations have motivated the shift toward machine learning and, more recently, deep learning-based systems, which are capable of learning hierarchical representations directly from data and adapting to complex, dynamic production scenarios.[5]

1.2.2. Machine Learning and Early Computer Vision Techniques

The limitations of traditional rule-based and image processing methods in defect detection led to the adoption of classical machine learning techniques, which marked an important transition toward more adaptive and data-driven approaches. Unlike purely heuristic systems, these methods introduced a degree of learning from data by separating the feature extraction process from the classification or decision-making stage. Although not as autonomous or powerful as modern deep learning models, early machine learning techniques contributed significantly to enhancing the reliability and flexibility of automated visual inspection systems.

In these early approaches, feature engineering played a central role. Domain experts would design handcrafted features to capture visual characteristics of defects—such as texture, colour, shape, or frequency-

domain patterns—using descriptors like Histogram of Oriented Gradients (HOG), Local Binary Patterns (LBP), Scale-Invariant Feature Transform (SIFT), or Gray-Level Co-occurrence Matrices (GLCM). These features were then used to train classifiers such as Support Vector Machines (SVM), Decision Trees, k-Nearest Neighbors (k-NN), or ensemble methods like Random Forests. These models could learn to differentiate between defective and non-defective instances based on statistical patterns in the training data.

Support Vector Machines, in particular, gained popularity due to their robustness in high-dimensional feature spaces and their effectiveness in binary classification tasks. SVMs could model complex decision boundaries and were often used in conjunction with kernel functions to handle non-linear separability. Similarly, k-NN and Decision Trees offered interpretable and fast solutions for smaller-scale applications, albeit with limited generalization in noisy or unbalanced datasets.

Principal Component Analysis (PCA) and Linear Discriminant Analysis (LDA) were frequently employed for dimensionality reduction, especially in cases where the number of features was large compared to the number of samples. These methods aimed to retain the most discriminative information while minimizing computational cost, an important consideration in early industrial deployments where real-time constraints were critical.

Despite offering improvements over purely rule-based methods, these early machine learning systems still faced several challenges. The reliance on handcrafted features made them highly sensitive to the quality and relevance of the engineered descriptors. Performance degraded significantly when applied to new defect types or when environmental conditions—such as lighting, viewpoint, or surface variability—changed from those present in the training data. Moreover, feature engineering required substantial domain expertise and was often not scalable across different products or inspection tasks.

Furthermore, these systems lacked the capacity to extract hierarchical or contextual information from images, limiting their ability to detect subtle, compound, or localized anomalies. As manufacturing environments became more complex and diversified, the need for models capable of learning representations directly from raw data became increasingly apparent.

These limitations paved the way for the rise of deep learning, particularly convolutional neural networks (CNNs), which offered a paradigm shift by automating feature extraction and enabling end-to-end learning directly from visual inputs. The next subsection explores how these advances in deep learning have transformed the landscape of defect detection in industrial computer vision applications.[5]

1.2.3. Deep Learning in Computer Vision

The advent of deep learning has marked a transformative milestone in the field of computer vision, enabling machines to learn complex representations of visual data directly from raw input without the need for manual feature engineering. This paradigm shift has had a profound impact on industrial applications, particularly in tasks such as object detection, image classification, segmentation, and—most relevantly—automated defect detection. Deep learning methods have demonstrated superior performance, robustness, and generalization capabilities, making them the state-of-the-art choice for visual inspection systems across a wide range of sectors.

Convolutional Neural Networks (CNNs) have played a foundational role in this evolution. Their hierarchical architecture, inspired by the visual cortex of the human brain, allows them to learn spatial hierarchies of features—from simple edges and textures in early layers to complex shapes and structures in deeper layers. This ability to extract multiscale features makes CNNs particularly well-suited for analysing industrial images, where defects may appear at varying levels of granularity.

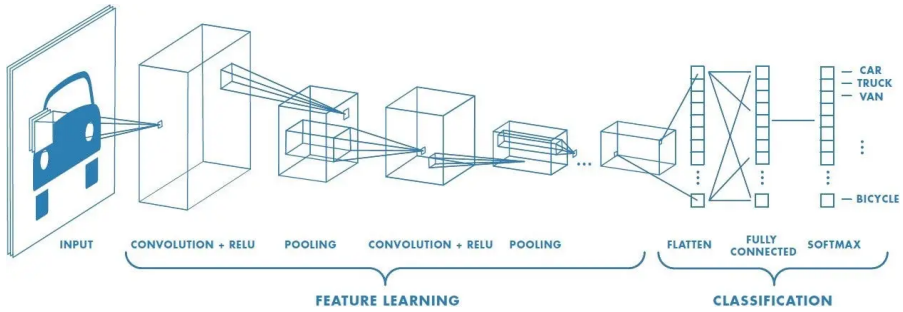


Figure 1.6: Overview of a CNN architecture showing feature extraction through convolution and pooling, followed by classification via fully connected and softmax layers. Source: SaturnCloud.[15]

The breakthrough of AlexNet in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012 marked a turning point in deep learning's dominance in computer vision. AlexNet demonstrated that deep architectures, when combined with GPU acceleration and large labelled datasets, could achieve unprecedented accuracy in visual recognition tasks. Subsequent architectures such as VGGNet, GoogLeNet (Inception), and ResNet further improved performance through deeper networks, parallel filter structures, and residual connections, respectively. ResNet, in particular, introduced identity mappings that allowed training of networks with over 100 layers without degradation, setting a new benchmark for performance and training stability.

In parallel, models like EfficientNet and DenseNet introduced novel ways to optimize model size and accuracy, making deep learning more accessible for industrial applications where computational resources may be limited. EfficientNet, for example, uses a compound scaling method that uniformly scales network depth, width, and resolution to achieve better efficiency and performance trade-offs.

More recently, the emergence of Vision Transformers (ViT) has challenged the CNN-centric paradigm by applying self-attention mechanisms—originally developed for natural language processing—to visual tasks. Vision Transformers demonstrate competitive performance in image classification and are increasingly being explored for defect detection due to their ability to capture long-range dependencies and contextual relationships within images.

In addition to discriminative models, Generative Adversarial Networks (GANs) have also gained traction in the context of industrial inspection, particularly for tasks like data augmentation, synthetic defect generation, and unsupervised anomaly detection. GANs enable the creation of realistic defect samples, which can be used to enrich training datasets and address the challenge of data imbalance, especially when real-world defective samples are rare or diverse.

One of the major advantages of deep learning in defect detection is its ability to generalize to previously unseen patterns, materials, and lighting conditions—an essential requirement for real-world deployment. Moreover, with the availability of large pre-trained models and transfer learning techniques, it is now possible to fine-tune high-performance networks on relatively small, domain-specific datasets, reducing the need for extensive data collection and annotation efforts.

In summary, deep learning has revolutionized computer vision by offering scalable, accurate, and adaptable solutions for complex visual inspection tasks. Its application to defect detection has not only surpassed the limitations of earlier approaches but has also opened new research avenues for developing intelligent, interpretable, and autonomous inspection systems capable of operating in highly variable industrial environments. The next subsection explores how these deep learning models are specifically applied to the task of defect detection and the architectural choices most commonly adopted in industrial settings.[16]

1.2.4. Applications of Deep Learning in Defect Detection

The application of deep learning techniques in industrial defect detection has significantly advanced the capabilities of automated visual inspection systems. Unlike traditional rule-based or shallow machine learning approaches, deep learning models are capable of learning hierarchical and abstract representations of data,

enabling the detection of subtle, irregular, or previously unseen defect patterns with remarkable accuracy. These models have been successfully implemented across a wide range of manufacturing domains, demonstrating adaptability, precision, and scalability in complex inspection tasks.

In practice, Convolutional Neural Networks (CNNs) are the most commonly employed architecture for defect detection due to their ability to extract spatial features from images. These models have been applied to classify and localize surface defects in materials such as steel, wood, leather, plastic, textiles, glass, and semiconductors. In classification tasks, CNNs distinguish between defective and non-defective items, while in object detection scenarios, they can also localize the defect using bounding boxes. Models like YOLO (You Only Look Once), SSD (Single Shot Detector), and Faster R-CNN have proven particularly effective for real-time detection and localization of defects on fast-moving production lines.

For applications that require pixel-level accuracy, such as identifying fine cracks, micro-scratches, or corrosion, segmentation-based networks are often preferred. Architectures such as U-Net, Mask R-CNN, and DeepLab enable semantic or instance segmentation, allowing precise delineation of defect regions. These models are particularly useful in scenarios where the size, shape, and position of the defect are critical for subsequent processing or decision-making.

Transfer learning plays a crucial role in industrial contexts where annotated datasets are limited. Pre-trained models, typically trained on large-scale datasets like ImageNet or MS COCO, can be fine-tuned on small domain-specific datasets, significantly reducing the need for extensive training data and computational resources. This has made it feasible for manufacturers to adopt deep learning models even in highly specialized or low-volume production settings.

In more advanced settings, unsupervised and semi-supervised learning approaches have gained traction, especially for anomaly detection tasks. These methods typically model the distribution of defect-free samples and identify anomalies as deviations from this learned distribution. Autoencoders, Variational Autoencoders (VAEs), and Generative Adversarial Networks (GANs) are often used to reconstruct normal images and measure reconstruction errors to detect anomalies. This is particularly advantageous in cases where defective examples are scarce, imbalanced, or difficult to enumerate exhaustively.

Additionally, some systems integrate multi-modal data (e.g., combining RGB images with thermal, depth, or X-ray imaging) to enhance detection robustness. These hybrid approaches allow for the identification of defects that may not be visually apparent under standard lighting or in RGB images alone.

Real-world implementations of these techniques span various industries:

- In automotive manufacturing, CNNs detect surface imperfections in painted car bodies or structural components.
- In electronics and PCB inspection, deep learning is used to identify soldering faults, missing components, or alignment issues.
- In textile production, fabric defects such as holes, stains, and misweaves are reliably detected through segmentation networks.
- In the steel and metal industry, models are used to identify dents, scratches, and cracks in high-speed rolling processes.
- In semiconductor manufacturing, ultra-fine defect detection is critical for wafer quality assurance, often requiring high-resolution input and multi-scale analysis.

These applications demonstrate the growing maturity and versatility of deep learning in defect detection. The ability of deep models to generalize, adapt to new defect types, and operate at production scale makes them integral to modern quality assurance pipelines. Nevertheless, challenges such as explainability, data imbalance, and model robustness remain, motivating continued research and development in this field.[17]

1.2.5. Alternative and Emerging Approaches

While deep learning—particularly convolutional neural networks—has become the dominant paradigm in automated defect detection, ongoing research continues to explore alternative and emerging approaches that either complement or offer new perspectives beyond standard 2D image-based methods. These approaches address some of the inherent limitations of traditional deep learning models, such as the need for large labelled datasets, lack of interpretability, or challenges in detecting non-visual or subsurface defects.

One important direction is the use of three-dimensional (3D) imaging and depth-based vision systems. Techniques such as structured light scanning, stereo vision, and time-of-flight cameras provide depth information that can be critical in detecting geometric deformations, dents, or assembly misalignments that may not be clearly visible in 2D images. In applications such as automotive part inspection or additive manufacturing quality control, the integration of depth data enables more comprehensive and accurate assessments of part geometry and surface topology.

Thermal and infrared imaging is another complementary modality, particularly effective in detecting defects that are associated with temperature variations or subsurface anomalies. For example, in the inspection of electronic circuits or composite materials, infrared thermography can reveal overheating components, internal voids, or delaminations that are invisible to the human eye or standard RGB cameras. These methods are often used in tandem with deep learning models, where thermal images are processed either separately or as additional input channels to enhance detection sensitivity.

Beyond visual and infrared modalities, acoustic, ultrasonic, and vibration-based inspection techniques offer valuable insights for detecting internal defects, especially in metals, welds, and composite structures. These approaches are particularly relevant in aerospace and civil engineering applications, where structural integrity must be guaranteed and surface-level inspection is insufficient. Machine learning models are increasingly being applied to analyze signal patterns from these sensors, enabling non-destructive evaluation (NDE) in automated settings.

In terms of algorithmic innovations, self-supervised and unsupervised learning methods have gained traction in scenarios where labelled defect data is scarce or unavailable. These approaches allow models to learn useful feature representations from unlabelled data, which can then be used for anomaly detection or fine-tuned for supervised tasks. Notable examples include contrastive learning, clustering-based pretraining, and reconstruction-based techniques using autoencoders or GANs. These methods are especially valuable for industrial environments where collecting defective samples is costly, time-consuming, or practically infeasible.

Another emerging trend is the incorporation of explainable artificial intelligence (XAI) techniques into defect detection pipelines. As AI systems become more prevalent in regulated and safety-critical industries, the ability to interpret and justify model predictions becomes essential. Techniques such as saliency maps, Grad-CAM, Layer-wise Relevance Propagation (LRP), and SHAP are being used to visualize and understand which features or regions of an image contribute most to a model's decision, thus increasing transparency and trust.

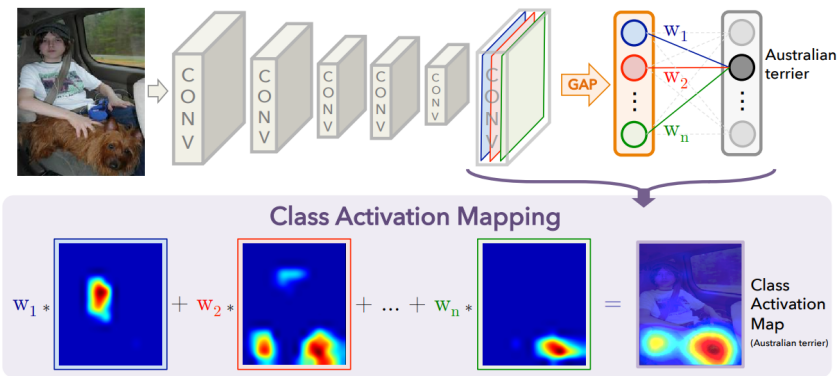


Figure 1.7: Visualization of Class Activation Mapping (CAM), highlighting image regions most relevant to a CNN's classification decision.[18]

Lastly, edge computing is becoming increasingly relevant as manufacturers aim to deploy inspection systems directly on production lines with real-time processing requirements. By implementing lightweight, optimized models on embedded devices or industrial controllers, it becomes possible to perform defect detection with minimal latency and without relying on cloud-based infrastructures. This reduces bandwidth usage, enhances data privacy, and enables immediate corrective actions during production.

These alternative and emerging approaches do not aim to replace deep learning in defect detection but rather to extend its reach and robustness by incorporating additional sensing modalities, learning paradigms, and deployment strategies. The continued integration of these technologies is shaping the future of smart manufacturing, pushing the boundaries of what is possible in automated quality control.[19]

1.2.6. Challenges and Limitations in Current Research

Despite the significant progress made in applying deep learning and computer vision to defect detection, several challenges and limitations persist in current research. These limitations constrain the scalability, reliability, and real-world applicability of existing models, particularly in industrial settings characterized by variability, data scarcity, and strict performance requirements. Identifying and addressing these issues is essential for advancing the development of practical and trustworthy automated inspection systems.

One of the most prominent challenges is the scarcity of labelled data, especially for rare or highly specific defect types. Deep learning models typically require large, diverse, and well-annotated datasets to achieve high performance and generalization. However, in many industrial domains, collecting a sufficient number of defective samples is difficult, time-consuming, or even impossible due to low defect rates or safety concerns. This data imbalance often leads to biased models that underperform in detecting rare anomalies, prompting the need for alternative strategies such as data augmentation, synthetic data generation, or semi-supervised learning.

Closely related to this is the issue of generalization and domain transfer. Models trained on a particular set of products, defect types, or imaging conditions often struggle to perform well when applied to new environments with different lighting, materials, or product geometries. Domain shift can severely degrade model performance, limiting the portability and robustness of inspection systems across manufacturing lines or facilities. Ongoing research in domain adaptation and transfer learning aims to address this, but current solutions are still limited in scope and often require non-trivial fine-tuning efforts.

Another critical concern is the interpretability and transparency of deep learning models. Most state-of-the-art models operate as black boxes, providing predictions without clear explanations of the underlying reasoning. In high-stakes industrial sectors—such as aerospace, pharmaceuticals, or medical devices—this lack of explainability hinders user trust and complicates regulatory compliance. Although explainable AI (XAI) techniques are emerging, they are not yet mature enough to be widely adopted in production environments where traceability and accountability are essential.

Real-time performance and computational efficiency also remain practical constraints, particularly for high-speed production lines where rapid decision-making is required. Deep neural networks, especially those with large parameter counts or complex architectures, can be computationally intensive and difficult to deploy on resource-limited hardware. While techniques such as model pruning, quantization, and knowledge distillation have been developed to reduce computational burden, achieving an optimal balance between speed and accuracy is still an active area of research.

Additionally, there is a lack of standardized benchmarks and evaluation protocols in the field of industrial defect detection. Unlike generic vision tasks, where datasets like ImageNet or COCO provide common grounds for comparison, industrial inspection datasets are often proprietary, domain-specific, and inaccessible to the broader research community. This limits the reproducibility and comparability of results, making it difficult to assess the true progress of competing approaches.

Finally, there are integration and deployment challenges associated with embedding AI models into existing manufacturing systems. This includes handling heterogeneous sensor inputs, ensuring seamless com-

munication with industrial control units (PLCs), managing data pipelines, and maintaining long-term model performance in non-stationary environments where process conditions evolve over time. These issues underscore the importance of interdisciplinary collaboration between machine learning researchers, domain experts, and industrial engineers.

In summary, while deep learning has dramatically advanced the field of defect detection, current research continues to grapple with challenges related to data limitations, generalization, interpretability, computational efficiency, and system integration. Addressing these open problems is essential for translating academic innovations into reliable, industry-ready solutions. The present thesis contributes to this ongoing effort by proposing a neural network-based defect detection system that aims to mitigate some of these challenges through methodological and architectural design choices.[20]

1.3. Thesis Structure

This thesis is structured to provide a coherent and progressive account of the research and development process behind the implementation of deep-learning models for automated defect detection in metallurgical manufacturing. The structure reflects the chronological and methodological steps followed throughout the project, from theoretical grounding to experimental validation and interpretability analysis.

Chapter 1 introduces the motivation behind the work, highlighting its industrial relevance, economic impact, and alignment with contemporary paradigms such as Industry 4.0. It also outlines the limitations of traditional visual inspection methods and identifies the technological gap that this research aims to address. Additionally, the chapter surveys the current state of the art in defect detection, covering both classical and modern approaches, and concludes with a summary of the thesis structure.

Chapter 2 provides the theoretical foundations necessary to contextualize the project. It introduces key concepts in machine learning and computer vision, including different learning paradigms, evaluation metrics, and architectural components of modern deep-learning systems. Emphasis is placed on convolutional neural networks (CNNs), object detection frameworks such as YOLO, and relevant training strategies, including data augmentation, transfer learning, and techniques to mitigate class imbalance. The chapter also touches on industrial deployment considerations, including model optimisation and real-time inference constraints.

Chapter 3 describes the complete methodological pipeline of the project. It begins with a detailed overview of the software ecosystem and computational platforms that support model development. The chapter then outlines the processes of data acquisition, cleaning, and preparation, including exploratory data analysis and the construction of train/validation/test splits. The core of the chapter is devoted to model development, covering several experimental configurations using YOLOv8 and TensorFlow/Keras architectures for both classification and segmentation tasks. Furthermore, the chapter presents a hybrid approach combining segmentation backbones with classification heads. The final section introduces the application of the Grad-CAM algorithm as a tool for visual interpretability, offering insight into the inner workings of the trained models.

Chapter 4 presents the experimental results and analysis. It begins by describing the training protocols, computational environment, and evaluation criteria. Quantitative results are reported for each model configuration, followed by an interpretability study based on Grad-CAM visualizations. These analyses are used to assess the models' capacity to detect and explain visual patterns associated with defects.

Finally, Chapter 5 concludes the thesis by summarising the main findings, discussing limitations, and proposing directions for future research. It reflects on the practical applicability of the developed models in real industrial settings and outlines steps towards further integration and refinement.

This structured progression from motivation and theory through implementation and evaluation aims to guide the reader through the technical and conceptual landscape of the project while ensuring methodological transparency and scientific rigour.

2

Theoretical Background: Machine Learning and Computer Vision

This chapter provides the theoretical foundations necessary to understand the subsequent methodological choices and experimental work carried out in this thesis. As the project focuses on the development of a deep learning system for defect detection in industrial parts, it is crucial to establish a clear conceptual framework encompassing the key principles of machine learning and computer vision.

First, the chapter introduces the fundamentals of machine learning, discussing its definitions, paradigms, and the trade-offs that influence model development and performance. Understanding these core concepts is essential for appreciating the challenges and design decisions faced when training neural networks for real-world applications. Particular emphasis is placed on the evaluation metrics commonly used in classification and segmentation tasks, ensuring a rigorous interpretation of experimental results.

Secondly, the chapter delves into the basics of computer vision, presenting traditional image-processing techniques and the transition towards learned feature representations. This historical perspective highlights the evolution from hand-crafted methods to modern deep-learning approaches, underscoring the motivations for adopting deep models in defect detection tasks.

Following this, an overview of deep-learning architectures specifically tailored to computer vision is provided. The discussion spans from classical Convolutional Neural Networks (CNNs) to more recent innovations such as Vision Transformers, detailing their structure, strengths, and applicability to industrial defect detection problems.

The chapter then addresses practical aspects of model training and data handling, including data augmentation, transfer learning, and strategies to tackle class imbalance—factors that critically impact the effectiveness and generalisation ability of defect detection models.

Finally, considerations for deploying models in industrial environments are examined. These include methods for model optimisation, the challenges of real-time inference on edge devices, and integration strategies with existing manufacturing systems. This industrial perspective ensures that the research conducted is not only scientifically robust but also aligned with the operational realities of smart manufacturing contexts.

By building a comprehensive theoretical foundation, this chapter aims to equip the reader with the necessary knowledge to critically assess the methodology, results, and conclusions presented in the following sections of the thesis.

2.1. Fundamentals of Machine Learning

Machine learning (ML) has emerged as a transformative technology capable of solving complex problems across a wide array of domains, including industrial defect detection. At its core, machine learning is concerned with the development of algorithms that can infer patterns from data and make decisions or predictions without being explicitly programmed for specific tasks. This paradigm shift from rule-based programming to data-driven model construction has opened new avenues for automation, particularly in contexts where traditional approaches struggle with variability, ambiguity, or high-dimensional information.

In this section, the fundamental concepts underlying machine learning are explored to establish a solid theoretical basis for the models developed in this thesis. Definitions and taxonomies of learning algorithms are first presented, clarifying the distinction between different types of learning based on the nature of the supervision signal and the learning objectives. Subsequently, the primary learning paradigms—supervised, unsupervised, semi-supervised, self-supervised, and reinforcement learning—are introduced, each offering distinct strategies for knowledge extraction and adaptation.

Furthermore, critical notions such as the bias–variance trade-off and generalisation are discussed, providing insight into how models achieve balance between underfitting and overfitting. Understanding these phenomena is crucial for designing robust systems capable of performing reliably in real-world, unseen conditions.

The section also examines concepts of model capacity, regularisation techniques, and the challenges associated with under- and over-fitting, all of which play a pivotal role in shaping the performance and stability of machine learning models.

Finally, an overview of evaluation metrics commonly employed in classification and segmentation tasks is provided. Metrics such as accuracy, precision, recall, F_1-score, mean Average Precision (mAP), Intersection over Union (IoU), and the Dice coefficient are described, as they serve as essential tools for the quantitative assessment of the models' capabilities in detecting defects with high confidence.

By systematically addressing these fundamental principles, this section prepares the ground for understanding the motivations behind the methodological decisions made in the later stages of model development and evaluation.[21][22]

2.1.1. Definition, Scope and Taxonomy of Learning Algorithms

Machine learning can be defined as the scientific field concerned with the design of algorithms that enable systems to improve their performance at a given task through experience. Rather than relying on explicitly programmed instructions, machine-learning models autonomously identify patterns and relationships within data, enabling predictive or decision-making capabilities that generalise to new, unseen examples.

The scope of machine learning is broad, encompassing a variety of applications across domains such as healthcare, finance, autonomous systems, and industrial manufacturing. Within the context of this thesis, machine learning serves as the foundational approach for developing automated defect detection systems capable of operating reliably under the variability inherent in industrial environments.

To organise the diverse methods and techniques within machine learning, various taxonomies have been proposed. One common classification is based on the type of supervision provided during training:

- Supervised Learning: The model learns from labelled data, aiming to predict outputs based on input–output pairs.
- Unsupervised Learning: The model identifies patterns or structures in unlabelled data, such as clustering or dimensionality reduction.
- Semi-Supervised Learning: Combines small amounts of labelled data with large quantities of unlabelled data to improve learning performance.

- Self-Supervised Learning: Generates supervision signals from the data itself, often through pretext tasks, to learn useful feature representations without manual labels.

- Reinforcement Learning: The model learns through interaction with an environment, receiving rewards or penalties to guide behaviour towards achieving long-term objectives.

This taxonomy provides a conceptual framework for understanding how different machine-learning approaches operate, and it serves as the basis for selecting suitable methodologies depending on the specific challenges and constraints of the defect detection task addressed in this work.[23]

2.1.2. Learning Paradigms (Supervised, Unsupervised, Semi-Supervised, Self-Supervised, Reinforcement)

Machine learning encompasses a variety of paradigms, each characterised by the nature of the learning process and the availability of supervision signals during training. A clear understanding of these paradigms is essential for selecting appropriate methods for different industrial applications, including defect detection tasks where data availability and labelling costs can vary significantly.

Supervised Learning is the most widely used paradigm in industrial contexts. In this setting, the model is trained on a labelled dataset, where each input is associated with a known output. The objective is to learn a mapping from inputs to outputs that generalises well to new, unseen data. Common supervised tasks include classification, regression, and segmentation, all of which are fundamental for detecting and categorising defects.

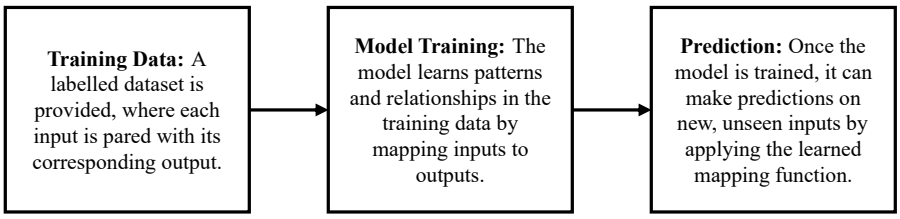


Figure 2.1: Basic workflow of supervised machine learning, including training data preparation, model training, and prediction.

Unsupervised Learning, in contrast, deals with unlabelled data. The model seeks to discover intrinsic structures, patterns, or distributions within the input data without external guidance. Clustering and dimensionality reduction are typical examples. In defect detection, unsupervised methods can be employed for anomaly detection, where defects represent deviations from learned normal patterns.

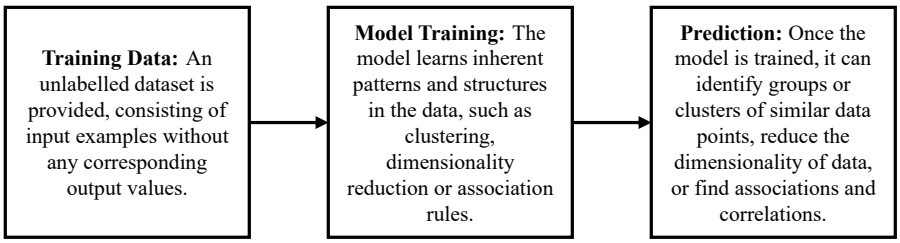


Figure 2.2: Basic workflow of unsupervised machine learning, including training data preparation, model training, and prediction.

Semi-Supervised Learning lies between supervised and unsupervised learning. It leverages a small amount of labelled data together with a larger pool of unlabelled data to improve learning performance. This approach is particularly valuable in industrial scenarios where obtaining labelled defect data is expensive or time-consuming, but unlabelled samples are abundant.

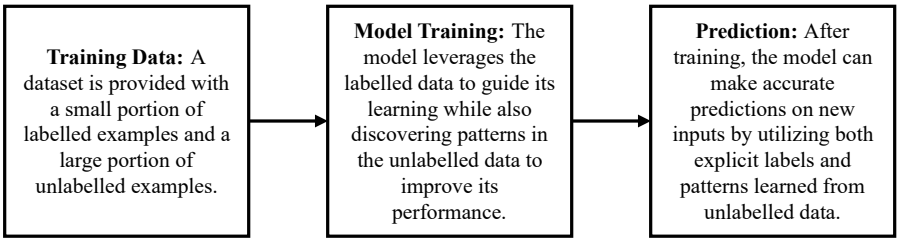


Figure 2.3: Basic workflow of semi-supervised machine learning, including training data preparation, model training, and prediction.

Self-Supervised Learning has recently gained prominence, especially in vision applications. In this paradigm, models are trained using automatically generated labels derived from the data itself through carefully designed pretext tasks (e.g., predicting missing parts of an image). Self-supervised techniques enable the learning of rich feature representations without manual annotation, offering significant potential for industrial applications with limited labelled datasets.

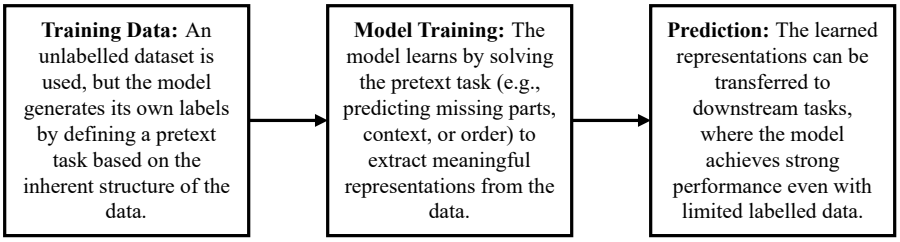


Figure 2.4: Basic workflow of self-supervised machine learning, including training data preparation, model training, and prediction.

Reinforcement Learning represents a distinct approach where an agent learns to make decisions by interacting with an environment, receiving feedback in the form of rewards or penalties. Although less common in traditional defect detection pipelines, reinforcement learning holds promise for adaptive quality-control systems or robotic inspection tasks that require sequential decision-making under uncertainty.

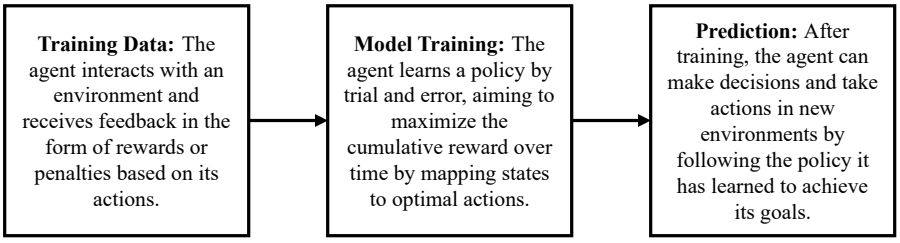


Figure 2.5: Basic workflow of reinforced machine learning, including training data preparation, model training, and prediction.

Understanding these paradigms provides the theoretical foundation for designing machine-learning systems tailored to the specific data characteristics, operational constraints, and performance objectives of industrial defect detection.[23]

2.1.3. Bias–Variance Trade-off and Generalisation

One of the fundamental challenges in machine learning is achieving a model that generalises well to new, unseen data. Generalisation refers to the model's ability to perform accurately not only on the training dataset but also on real-world inputs that it has not encountered before. A central concept in understanding generalisation is the bias–variance trade-off, which describes the balance between two types of errors that can arise during model training.

Bias refers to errors due to overly simplistic assumptions in the learning algorithm. A model with high bias may underfit the data, failing to capture the underlying patterns and thus performing poorly on both the training and testing datasets. This often occurs when the model is not complex enough to represent the true

relationships in the data.

Variance, on the other hand, refers to errors due to the model's sensitivity to small fluctuations in the training data. A model with high variance may overfit, capturing noise and specific peculiarities of the training set instead of the general trends. Consequently, while such a model may perform exceptionally well on the training data, its performance degrades significantly on new data.

The bias–variance trade-off is about finding the right balance: models must be flexible enough to learn the relevant structures in the data (low bias) but not so complex that they memorise noise (low variance). Achieving this balance is critical for building robust machine-learning systems capable of operating reliably in industrial environments, where variability in conditions is inevitable.

In the context of defect detection, poor generalisation could lead to the dangerous misclassification of defective parts as acceptable or the rejection of acceptable parts, both of which carry significant operational and economic consequences. Therefore, careful model selection, regularisation techniques, and validation strategies are essential to manage the bias–variance trade-off effectively.

Understanding these concepts lays the groundwork for making informed choices about model complexity, training strategies, and evaluation methods, all of which directly impact the reliability and industrial viability of the deep-learning solutions developed in this thesis.[21]

2.1.4. Model Capacity, Under-/Over-fitting and Regularisation

The concept of model capacity is central to understanding a machine-learning model's ability to fit complex patterns in data. Model capacity refers to the range and complexity of functions that a model can represent. Models with low capacity are limited to learning simple patterns, whereas high-capacity models can capture intricate, highly non-linear relationships.

However, higher capacity comes with increased risk. If a model's capacity exceeds what is necessary to capture the true underlying structure of the data, it may begin to fit noise and idiosyncrasies present in the training set—a phenomenon known as overfitting. Overfitting leads to excellent performance on the training data but poor generalisation to new inputs, undermining the model's practical utility in real-world scenarios.

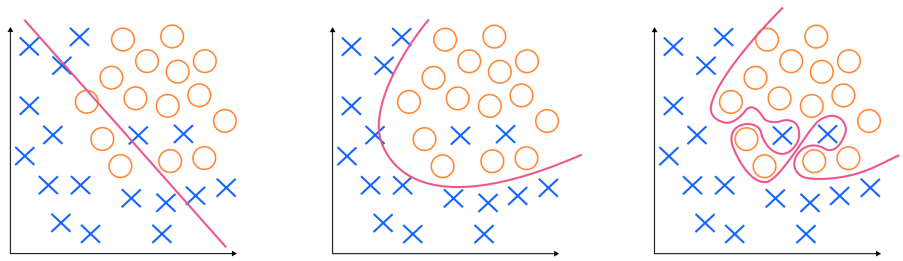


Figure 2.6: Illustration of underfitting (left), good generalization (center), and overfitting (right) in a classification problem. Source: IBM.[24]

Conversely, when a model has insufficient capacity to capture the essential structure of the data, it suffers from underfitting. In this case, the model exhibits high bias, failing to achieve even good performance on the training set, and thus cannot be expected to perform well on unseen examples either.

To mitigate the risks of overfitting and underfitting, various regularisation techniques are employed. Regularisation involves introducing additional information or constraints during training to encourage the model to learn simpler, more generalisable patterns. Common regularisation methods include:

- L1 & L2 regularization: Adding penalty terms to the loss function to discourage overly large weights, thereby promoting simpler models.
- Dropout: Randomly deactivating a fraction of neurons during each training iteration to prevent co-adaptation and improve robustness.

- Early stopping: Monitoring model performance on a validation set and halting training when performance ceases to improve, thereby preventing overfitting.

- Data augmentation: Expanding the training dataset with artificially modified samples to expose the model to a broader range of scenarios and reduce overfitting.

Balancing model capacity with appropriate regularisation is essential to develop machine-learning systems that achieve strong performance without compromising reliability. In industrial defect detection, where generalisation to unseen defects or varying operational conditions is critical, these considerations are particularly important to ensure consistent, trustworthy results.[25]

2.1.5. Evaluation Metrics (Accuracy, Precision/Recall, F_1, mAP, IoU, Dice)

The evaluation of machine-learning models requires the use of appropriate metrics that accurately reflect their performance, particularly in the context of the specific task at hand. In industrial defect detection, where misclassification of defective or non-defective parts can have serious operational, economic, and safety implications, selecting and interpreting the right metrics is critical.

Accuracy is one of the most intuitive metrics, measuring the proportion of correctly classified instances over the total number of instances. While useful for balanced datasets, accuracy can be misleading in cases of class imbalance, where the defective class may represent only a small fraction of the data.

In the evaluation of classification models, the confusion matrix is a fundamental tool that provides a detailed breakdown of the model's predictions. It is a tabular representation that contrasts the predicted class labels against the true class labels, thereby offering insight into the types and frequencies of errors made by the model.

A typical confusion matrix for binary classification consists of four elements: true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). This allows practitioners to see not only how many samples were classified correctly, but also the specific ways in which misclassifications occur. For example, in the context of industrial defect detection, the confusion matrix makes it immediately apparent whether the model tends to overlook defects (false negatives) or wrongly classify good parts as defective (false positives).

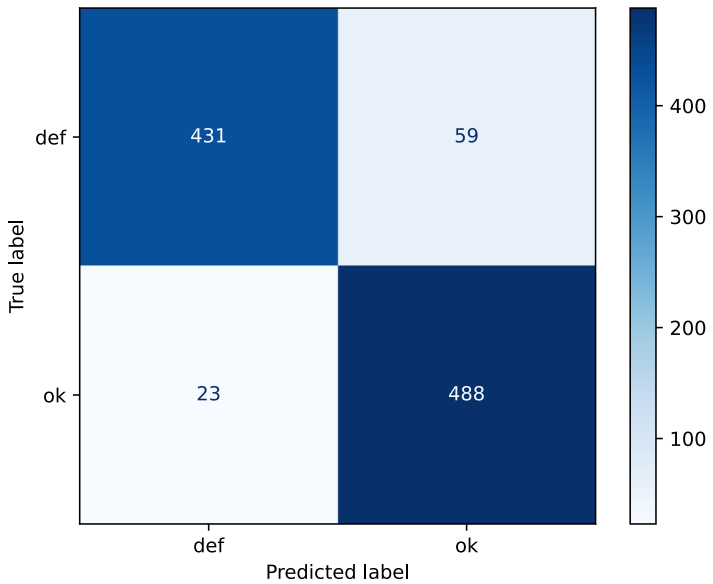


Figure 2.7: Sample classification matrix illustrating the distribution of predicted versus actual class labels.

The confusion matrix serves as the foundation for key evaluation metrics such as accuracy, precision, recall, and F1-score. Its visual nature helps to quickly assess model strengths and weaknesses, guiding further refinement and enabling targeted improvements. Figure 2.7 illustrates a representative confusion matrix for a binary defect classification scenario.

To address this, precision and recall are often employed:

- Precision measures the proportion of true positive predictions among all positive predictions made by the model. High precision indicates that when the model predicts a defect, it is usually correct.
- Recall (also known as sensitivity) measures the proportion of actual defects that are correctly identified by the model. High recall ensures that most defects are detected, even at the risk of some false positives.

The F_1-score is the harmonic mean of precision and recall, providing a single metric that balances the trade-off between them. It is particularly useful when the costs of false positives and false negatives are both significant.

For object detection and segmentation tasks, where locating and delineating defects is required, additional metrics are crucial:

- mean Average Precision (mAP) summarises the precision–recall curve by averaging precision scores across multiple recall levels and, when applicable, across multiple classes. It is a standard benchmark for object-detection models.
- Intersection over Union (IoU) quantifies the overlap between the predicted and ground-truth bounding boxes or segmentation masks. A higher IoU indicates better localisation and more accurate predictions.
- Dice coefficient (or Dice Similarity Coefficient) is widely used in segmentation tasks. It measures the similarity between the predicted and ground-truth regions, offering an alternative to IoU that is especially sensitive to the overlap quality in small or irregularly shaped objects.

Selecting and properly interpreting these metrics allows for a nuanced and comprehensive assessment of model performance, ensuring that defect detection systems meet the high reliability standards demanded by industrial environments.[26]

2.2. Fundamentals of Computer Vision

Computer vision is a field of artificial intelligence focused on enabling machines to interpret and process visual information from the world, emulating the remarkable capabilities of human vision. In industrial applications, computer vision techniques are critical for tasks such as quality inspection, anomaly detection, and process monitoring, where the accurate and efficient analysis of visual data can lead to significant operational improvements.

This section presents the fundamental concepts of computer vision, providing a historical and technical perspective on the evolution of the field. It begins with an overview of the physical principles of image formation, including the role of lighting, optics, and sensors, as well as a discussion on various colour spaces and preprocessing techniques. These foundational elements are essential for understanding how raw visual data can be effectively prepared for further analysis.

Following this, classical computer vision methods are introduced. Techniques such as edge detection, Histogram of Oriented Gradients (HOG), Scale-Invariant Feature Transform (SIFT), and Speeded-Up Robust Features (SURF) are reviewed, illustrating the traditional reliance on hand-engineered features to capture relevant information from images. Although powerful, these methods often struggle with variability in lighting, orientation, and scale, highlighting their limitations when faced with complex industrial environments.

The discussion then transitions to the concept of feature extraction and representation learning, a key step toward the adoption of machine learning techniques in vision tasks. By learning feature representations directly from data, modern approaches have overcome many of the constraints associated with hand-crafted

descriptors, enabling more flexible and powerful models.

Through this exploration of computer vision fundamentals, the section lays the groundwork for understanding how traditional techniques have influenced and been surpassed by deep learning methodologies, which now dominate the field, particularly in demanding applications such as defect detection.

2.2.1. Image Formation, Colour Spaces and Pre-processing

Understanding the basics of image formation is essential for any computer vision system, as the quality, structure, and characteristics of the input images directly influence model performance. In industrial applications, images are typically acquired using cameras or specialised sensors, and several factors—such as lighting conditions, lens properties, and sensor noise—can affect the resulting visual data. A solid grasp of how images are formed enables better control of the acquisition process and informs the design of effective pre-processing pipelines.

An image can be understood as a two-dimensional projection of a three-dimensional scene, capturing the spatial distribution of light intensities across a surface. Most digital images are represented as grids of pixels, each encoding information about light intensity (in grayscale images) or colour (in colour images). The fidelity of this representation depends on the resolution, dynamic range, and quality of the imaging system.

Colour spaces play a fundamental role in how images are processed and interpreted. The most common colour space is RGB (Red, Green, Blue), where colours are expressed as combinations of three primary colour intensities. However, alternative colour spaces such as HSV (Hue, Saturation, Value), LAB, and YCbCr can offer advantages for specific tasks. For example, HSV separates colour information from intensity, making it more robust to lighting variations—a critical factor in industrial environments where illumination conditions may change.

Pre-processing steps are crucial for enhancing image quality, reducing noise, and normalising inputs to ensure that the vision system focuses on relevant features. Typical pre-processing techniques include:

- Normalisation: Scaling pixel values to a standard range (e.g., [0,1] or [-1,1]) to facilitate model training.
- Histogram equalisation: Improving contrast in images by redistributing intensity values, especially useful for detecting subtle defects.
- Filtering: Applying smoothing (e.g., Gaussian blur) or sharpening filters to remove noise or highlight edges.
- Resizing and cropping: Standardising image dimensions and focusing on regions of interest, which improves computational efficiency and model focus.

Effective pre-processing not only improves model performance but also enhances robustness by mitigating the impact of acquisition artefacts and environmental variability. In defect detection systems, where precision is critical, attention to these fundamental aspects of image formation and preparation is a decisive factor in achieving reliable results.[27]

2.2.2. Classical Vision Methods (Edge Detectors, HOG, SIFT, SURF)

Before the widespread adoption of deep learning, classical computer vision methods formed the backbone of automated visual analysis systems. These approaches typically relied on manually designed algorithms to extract meaningful features from images, enabling tasks such as object detection, recognition, and defect identification. Understanding these traditional methods provides valuable context for appreciating the advances brought by deep-learning techniques.

Edge detection is one of the most fundamental operations in classical vision. Edges represent boundaries between regions of different intensity or texture, often corresponding to important structural features in the scene. Algorithms such as the Sobel, Prewitt, and Canny edge detectors are widely used to identify these discontinuities. The Canny detector, in particular, introduced a multi-stage process combining noise reduction,

gradient calculation, non-maximum suppression, and hysteresis thresholding, achieving robust and precise edge localisation.

Histogram of Oriented Gradients (HOG) is a feature descriptor designed to capture the distribution of edge orientations within localised portions of an image. By computing gradients and aggregating them into orientation histograms, HOG provides a compact and informative representation that is particularly effective for object detection tasks, such as pedestrian or vehicle recognition. Its success lies in its invariance to small deformations and robustness to illumination changes.

Scale-Invariant Feature Transform (SIFT) revolutionised local feature detection by providing a method to identify and describe distinctive image keypoints invariant to scale, rotation, and minor affine transformations. SIFT extracts keypoints at multiple scales and orientations, describing each one with a highly distinctive descriptor based on local gradients. This capability made SIFT extremely powerful for matching and recognition tasks, even under significant changes in viewpoint or illumination.

Speeded-Up Robust Features (SURF) built upon the ideas of SIFT, offering a faster alternative with comparable performance. SURF uses integral images and approximations of the Hessian matrix to detect keypoints rapidly, making it more suitable for real-time applications where computational efficiency is critical.

Although classical methods have largely been supplanted by deep learning in recent years, they still offer valuable insights and, in some cases, remain viable solutions when computational resources are limited or when training data is scarce. Moreover, understanding these methods helps bridge the conceptual gap between handcrafted feature engineering and modern, data-driven feature learning.[28]

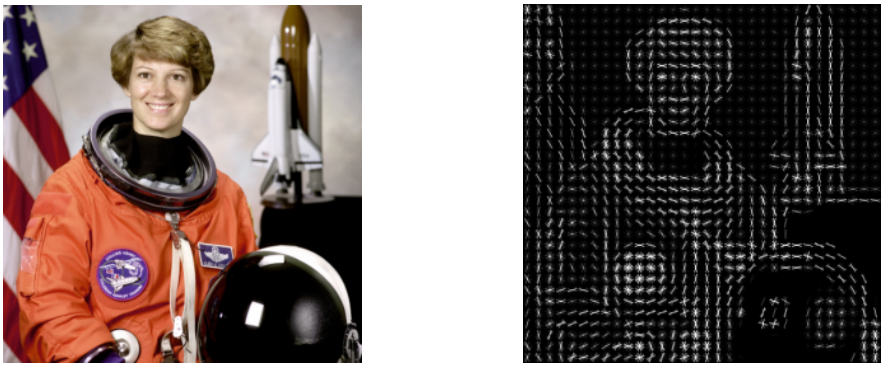


Figure 2.8: Original image and its corresponding HOG (Histogram of Oriented Gradients) visualization, highlighting the edge and gradient structure used for feature extraction in image recognition tasks. Source: Scikit Image[29]

2.2.3. Feature Extraction and Representation Learning

Feature extraction lies at the heart of computer vision systems, serving as the process by which raw image data is transformed into a structured representation that captures the relevant characteristics of the visual input. Effective feature extraction enables models to distinguish between different classes or detect anomalies, making it a critical step in tasks such as defect detection.

In traditional computer vision, feature extraction relied heavily on manual engineering. Techniques like edge detection, HOG, SIFT, and SURF exemplified the design of algorithms that could detect salient points, textures, or shapes within images. These features were then used as inputs to machine-learning classifiers such as Support Vector Machines (SVMs) or Random Forests. While effective in controlled scenarios, handcrafted features often struggled to generalise to new environments with varying lighting, orientations, and object appearances.

The advent of representation learning marked a paradigm shift. Rather than manually specifying the types of features to extract, representation learning allows models to automatically discover the most useful features

directly from data. Deep-learning models, particularly Convolutional Neural Networks (CNNs), are capable of learning hierarchical feature representations: lower layers capture simple structures like edges and textures, while deeper layers encode more abstract concepts such as object parts or even entire object categories.

This automatic discovery process has several advantages:

- Adaptivity: The model tailors its features to the specific task and data distribution, leading to better performance.
- Robustness: Learned representations are often more invariant to changes in scale, rotation, illumination, and occlusion.
- Reduced human bias: By eliminating manual feature selection, representation learning can capture unexpected patterns or correlations that might be overlooked by human designers.

In the context of industrial defect detection, where defects can vary widely in appearance and subtle anomalies must be reliably identified, the ability of deep models to learn rich, multi-scale feature representations has proven to be a decisive advantage. As such, representation learning forms a foundational component of the deep-learning approaches employed in this thesis.[30]

2.2.4. Typical Computer Vision Problem Formulations

The field of computer vision encompasses a range of canonical problem formulations, each defined by the nature of the desired output and the specific requirements of the application domain. In the context of automated defect detection within metallurgical manufacturing, the selection of the appropriate problem formulation is crucial, as it determines the granularity and utility of the information extracted from visual data. The following subsections outline the most prominent computer vision tasks relevant to industrial inspection scenarios.

Image Classification

Image classification represents the foundational problem in computer vision, in which the model assigns a single label to an entire image based on its global visual characteristics. In industrial contexts, this task is commonly applied to determine whether a component is defective or not, or to identify the general type of defect present. The simplicity of this approach offers high throughput but limited spatial detail, as no information is provided regarding the location or extent of defects within the image.

Figure 2.9: Sample image classification example showing how an input image is assigned to a predicted category by the model. Source: Ultralytics.[31]

Object Detection

Object detection extends the capabilities of image classification by not only recognizing the presence of specific classes but also localizing their instances within the image using bounding boxes. This formulation enables the identification of multiple defects within a single sample and provides spatial information that is critical for downstream processes such as quality control, part sorting, or targeted rework. Popular object detection frameworks, such as YOLO and SSD, have achieved significant success in industrial inspection pipelines due to their balance of accuracy and inference speed.

Figure 2.10: Sample object detection result illustrating identified objects in an image with corresponding bounding boxes. Source: Ultralytics.[31]

Segmentation

Semantic segmentation addresses the problem of assigning a class label to each pixel in the image, thereby producing a dense, per-pixel classification map. This approach is particularly valuable in scenarios where the precise shape and extent of defects are of interest, such as the delineation of cracks, corrosion, or surface anomalies on metal parts. Semantic segmentation provides a detailed understanding of the spatial distribution of defects, facilitating more informed decisions in quality assessment and process optimization.

While semantic segmentation distinguishes between classes at the pixel level, instance segmentation further separates individual object instances within the same class. This distinction is essential in cases where multiple defects of the same type occur in close proximity, as it enables the quantification and characterization of each defect independently. Techniques such as Mask R-CNN have demonstrated state-of-the-art performance in instance-level analysis, enabling advanced inspection tasks that require fine-grained defect tracking.

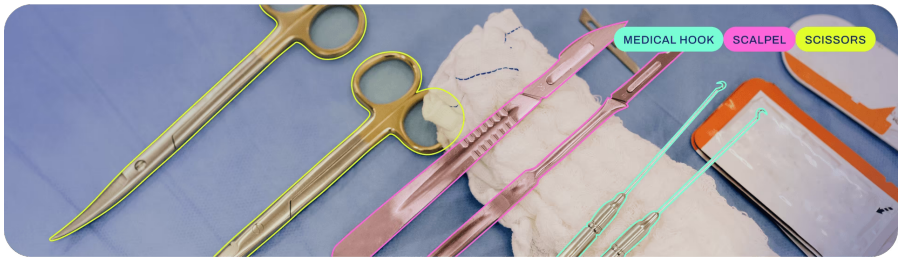


Figure 2.11: Sample image segmentation output highlighting different regions or objects within the image using distinct colors or masks. Source: Ultralytics.[31]

Keypoint Detection

Keypoint detection, also referred to as landmark detection, focuses on locating specific, predefined points of interest within an image. In the manufacturing domain, keypoint detection can be leveraged to identify structural features, reference markers, or salient points on defects (e.g., crack tips or edge discontinuities). The extracted keypoints may serve as the basis for geometric measurements, deformation analysis, or as auxiliary information for other inspection algorithms.

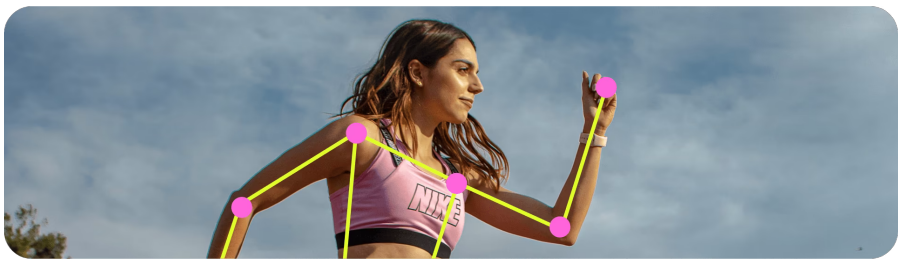


Figure 2.12: Sample keypoint detection result showing identified keypoints or landmarks on objects within the image. Source: Ultralytics.[31]

Additional Problem Formulations

Beyond the core tasks described above, additional problem formulations such as panoptic segmentation (which unifies semantic and instance segmentation), object tracking (for sequential analysis), and image captioning (for automated report generation) are of growing interest in both research and industrial applications. However, their adoption within metallurgical defect inspection remains limited, primarily due to domain-specific constraints and the unique requirements of manufacturing workflows.

2.3. Deep-Learning Architectures for Vision

The advent of deep learning has revolutionised the field of computer vision, enabling unprecedented levels of performance across a wide range of tasks, from image classification to object detection and semantic segmentation. Unlike traditional computer vision methods that relied heavily on manual feature engineering, deep-learning models automatically learn hierarchical representations of visual data, extracting increasingly abstract and informative features directly from raw inputs.

This section explores the principal deep-learning architectures that underpin modern vision systems, with particular emphasis on their design principles, strengths, and relevance to defect detection in industrial settings. It begins by introducing Convolutional Neural Networks (CNNs), the foundational architecture that sparked the deep-learning revolution in vision tasks. The mechanisms of convolutional layers, pooling, and activation functions are outlined, highlighting how these components enable CNNs to efficiently capture spatial hierarchies in image data.

Building upon this foundation, the section reviews advancements in network design that have addressed key challenges such as vanishing gradients, computational inefficiency, and limited model capacity. Architectures like ResNet, EfficientNet, and MobileNet are discussed, showcasing how innovations such as residual connections and compound scaling have expanded the applicability of deep models to more complex and resource-constrained environments.

Furthermore, the section presents an overview of object-detection frameworks, including the R-CNN family, Single Shot Multibox Detector (SSD), and You Only Look Once (YOLO) models, which are pivotal for locating and classifying defects within images. These methods combine classification and localisation tasks into unified systems capable of real-time inference, a crucial capability for industrial applications.

Finally, emerging approaches in semantic and instance segmentation, such as Fully Convolutional Networks (FCNs), U-Net, and Mask R-CNN, are introduced, alongside a discussion of the recent rise of Vision Transformers and hybrid architectures that integrate convolutional and attention mechanisms.

Through this examination of deep-learning architectures, the section provides essential insights into the design choices and technological advancements that inform the model selection and development strategies employed in this thesis.

2.3.1. Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) have established themselves as a cornerstone of modern computer vision, offering a highly effective and scalable framework for learning hierarchical feature representations directly from raw image data. Their development marked a significant leap forward, enabling automatic extraction of complex visual features and driving major advances in areas such as image classification, object detection, and semantic segmentation.

The fundamental operation at the heart of CNNs is the convolution, whereby learnable filters, or kernels, are systematically applied across the spatial dimensions of an input image. Each filter acts as a detector for specific local patterns, such as edges, corners, or textures. As layers of convolutions are stacked, the network progressively constructs more abstract and complex representations, moving from simple visual primitives in early layers to high-level semantic concepts in deeper ones.

A typical CNN architecture is composed of several essential building blocks:

- Convolutional layers, which apply spatial filters to detect meaningful patterns and generate feature maps.
- Activation functions, such as the Rectified Linear Unit (ReLU), which introduce non-linearity and allow the network to model complex relationships.
- Pooling layers, commonly using max pooling operations, which reduce the spatial resolution of feature maps, promoting translational invariance and lowering computational cost.
- Fully connected layers, located towards the end of the network, which consolidate the learned features and perform final tasks such as classification or regression.

Beyond their architectural components, CNNs possess several intrinsic properties that contribute to their success. Parameter sharing across different spatial locations greatly reduces the number of learnable parameters compared to fully connected networks, enhancing computational efficiency and mitigating overfitting. Local connectivity between neurons and input regions encourages the network to focus on spatially coherent structures, enabling it to learn meaningful patterns with relatively few resources. Moreover, the hierarchical nature of feature learning allows CNNs to gradually build complex abstractions from simple inputs, leading to strong generalisation capabilities across diverse visual tasks.

In the industrial domain, these characteristics are particularly advantageous. CNNs demonstrate remarkable robustness to variations in lighting, scale, orientation, and noise, all of which are common in real-world manufacturing environments. Their ability to capture subtle differences in surface textures, contours, and material properties makes them ideally suited for defect detection applications, where fine-grained discrimination between acceptable and defective parts is often required.

In this thesis, CNNs form the foundational architecture for both classification and segmentation models, providing the basis for the development of accurate and reliable deep-learning systems aimed at improving industrial quality-control processes.[32]

2.3.2. Residual and Efficient Architectures (ResNet, EfficientNet, MobileNet)

While early Convolutional Neural Networks achieved remarkable success, deeper architectures soon revealed critical challenges, particularly related to training stability and computational demands. As networks grew in depth and complexity, issues such as vanishing gradients, excessive training times, and deployment constraints on edge devices became increasingly apparent. To address these challenges, a new generation of architectures introduced innovative design principles focused on enhancing both performance and efficiency.

One of the most influential advancements was the introduction of Residual Networks (ResNet). ResNet architectures proposed the concept of residual learning, where shortcut connections allow the gradient to flow more easily through deep networks. Instead of learning an entire transformation, each layer learns a residual function relative to its input. This simple yet powerful idea effectively mitigates the vanishing gradient problem and enables the training of extremely deep networks with hundreds or even thousands of layers, achieving superior performance across a wide range of computer vision tasks.

Building upon the need for more efficient models, EfficientNet introduced a systematic approach to scaling neural networks. Rather than arbitrarily increasing network depth, width, or resolution, EfficientNet proposed a compound scaling method that balances all three dimensions in a principled manner. This strategy leads to models that are not only more accurate but also significantly more computationally efficient, making them highly attractive for industrial applications where resource constraints are a consideration.

Complementing this trend towards lightweight architectures, MobileNet was specifically designed for mobile and embedded vision applications. MobileNet models employ depthwise separable convolutions, a technique that factorises a standard convolution into a depthwise convolution followed by a pointwise convolution. This reduces computational complexity and model size dramatically while maintaining competitive

accuracy levels. As a result, MobileNet and its successive versions have become popular choices for real-time inference on resource-constrained devices.

These architectural innovations—residual connections for deeper and more stable learning, compound scaling for balanced model growth, and separable convolutions for efficient deployment—have profoundly influenced the design of modern computer vision systems. In the context of defect detection, where models must often operate in real-time and within strict hardware limitations, leveraging these advanced architectures is crucial for developing solutions that are not only accurate but also practical for industrial deployment.[33]

2.3.3. Object-Detection Frameworks (R-CNN Family, SSD, YOLO)

Object detection is a fundamental task in computer vision that combines localisation and classification, aiming not only to identify the presence of objects within an image but also to precisely locate them through bounding boxes. In the context of industrial defect detection, object-detection frameworks are particularly valuable, as they enable the precise identification of defective regions within manufactured parts, facilitating automated quality-control processes.

The evolution of object-detection frameworks has been marked by significant innovations aimed at improving both accuracy and inference speed. One of the earliest influential approaches was the R-CNN family. The original R-CNN (Regions with CNN features) proposed a two-stage process: first, generating region proposals using traditional computer vision methods, and then classifying each proposed region with a CNN. Although R-CNN achieved high accuracy, its computational inefficiency led to the development of faster variants such as Fast R-CNN and Faster R-CNN. Fast R-CNN improved efficiency by sharing convolutional computations across proposals, while Faster R-CNN introduced a Region Proposal Network (RPN) that integrated the proposal generation process into the network itself, further accelerating the detection pipeline.[34]

Seeking even greater speed, the Single Shot MultiBox Detector (SSD) framework adopted a single-stage approach. SSD eliminates the need for a separate proposal generation step by directly predicting class probabilities and bounding box offsets from multiple feature maps at different scales. This design allows SSD to perform object detection in a single forward pass, striking a balance between accuracy and real-time performance, which is crucial for applications requiring immediate defect identification.

Pushing the boundaries of real-time detection even further, the You Only Look Once (YOLO) series of models reframed object detection as a regression problem.[35] YOLO models divide the image into a grid and simultaneously predict bounding boxes and class probabilities for each cell. This radical redesign leads to exceptionally fast inference times, making YOLO a preferred choice for industrial scenarios where high-speed visual inspection is required. Subsequent iterations of YOLO, including the latest YOLOv8, have progressively improved accuracy while maintaining real-time capabilities.

The choice of object-detection framework significantly impacts the trade-offs between detection precision, computational demands, and latency. In industrial defect detection systems, where both accuracy and operational speed are vital, selecting an appropriate framework is critical to ensuring effective and scalable deployment.

2.3.4. Semantic & Instance Segmentation (FCN, U-Net, Mask R-CNN)

Whereas object-detection frameworks enclose regions of interest in coarse bounding boxes, segmentation methods assign a label to every pixel, producing far richer spatial detail. Two principal flavours exist. Semantic segmentation classifies each pixel according to category, merging all objects of the same class into a single mask; instance segmentation goes a step further by distinguishing between individual occurrences of the same category. In industrial quality inspection, pixel-accurate masks are essential for quantifying defect size, shape and location, enabling automated downstream decisions such as selective re-work, precise material removal, or targeted coating.

The first deep architecture to demonstrate end-to-end semantic segmentation was the Fully Convolutional Network (FCN). By replacing fully connected layers with convolutions and introducing learnable up-sampling

through transposed convolutions, FCNs converted classification backbones into dense predictors that output resolution-matched score maps. Although influential, early FCNs produced relatively coarse masks because successive pooling operations gradually eliminated fine-grained spatial cues.[34]

Addressing this limitation, U-Net introduced an encoder–decoder topology with long skip connections that directly shuttle high-resolution features from the contracting path to the corresponding layers of the expanding path. These skip connections restore spatial detail lost during down-sampling, yielding sharp boundaries even when trained on modest datasets—a frequent scenario in industrial contexts where acquiring pixel-level annotations is costly.[36]

For instance segmentation, Mask R-CNN extended Faster R-CNN by appending a lightweight mask branch to each detected region of interest. This multi-task design simultaneously predicts object class, refines bounding boxes, and generates a binary mask for every proposal. By decoupling mask prediction from class prediction, Mask R-CNN delivers high-quality instance masks with only a modest increase in computation, making it practical for near-real-time inspection when paired with hardware acceleration.

Collectively, these architectures illustrate the progression from coarse region proposals to fine, per-pixel understanding. Their capacity to delineate defects with surgical precision underpins many of the experimental designs presented in later chapters, where segmentation masks serve not only as ground truth for training but also as actionable outputs for intelligent manufacturing systems.

2.3.5. Vision Transformers and Hybrid Models

Recent developments in deep-learning architectures for vision tasks have seen a substantial shift toward attention-based methods, most notably Vision Transformers (ViTs). Initially inspired by the success of Transformers in natural language processing, ViTs represent images as sequences of flattened patches, replacing traditional convolutional operations with self-attention mechanisms. Through this approach, ViTs effectively capture global context and long-range spatial relationships within visual data, overcoming inherent limitations of purely convolutional architectures that are primarily biased towards local patterns.

Unlike CNNs, which leverage spatial locality through convolutional filters, Vision Transformers learn visual patterns by explicitly modelling interactions among all image regions simultaneously. Although computationally intensive at scale, ViTs have demonstrated superior capabilities in tasks that require reasoning over large image areas or capturing subtle contextual dependencies. Their ability to generalize effectively has made them particularly attractive in scenarios characterized by limited data or high variability, typical conditions encountered in industrial defect detection.[37]

However, the adoption of pure Transformer models in practical vision tasks still poses challenges, especially related to computational demands, training stability, and the need for extensive datasets. To mitigate these limitations, hybrid models combining the strengths of CNNs and Transformers have emerged. These hybrid architectures integrate convolutional layers in early stages to capture low-level spatial features efficiently, followed by Transformer layers that provide powerful global context modelling capabilities. By blending local feature extraction with global attention, hybrid models balance computational efficiency with the superior representational power of attention mechanisms. [38]

In the context of industrial inspection, this hybrid approach presents clear advantages. The local spatial sensitivity of CNNs remains highly effective for detecting fine-grained texture anomalies or small-scale defects, while the global context afforded by Transformers ensures robust interpretation of complex or ambiguous defect patterns. Recent studies suggest that these architectures can outperform traditional CNN-based methods, particularly when defects exhibit irregular patterns or subtle global dependencies that simpler models might overlook.

In summary, Vision Transformers and hybrid models represent a promising direction for advancing defect detection systems. Their capacity to integrate local precision with global context modelling positions them as strong candidates for future research and practical deployment, underpinning some of the methodological explorations and recommendations presented later in this thesis.

2.3.6. Model Architecture Components: Backbone, Neck, and Head

State-of-the-art deep learning architectures for computer vision commonly adopt a modular structure, comprising three principal components: the backbone, the neck, and the head.[39][40] This design paradigm provides a high degree of flexibility and scalability, supporting the integration of specialized modules and facilitating targeted optimizations for specific vision tasks, such as image classification, object detection, or semantic segmentation.

Backbone: Feature Extraction

The backbone constitutes the foundational component of the network and is responsible for extracting hierarchical feature representations from the input image. Typically, the backbone is implemented as a deep convolutional neural network (CNN), leveraging architectures such as ResNet, VGG, EfficientNet, or CSP-Darknet, each with distinct trade-offs in terms of representational power, parameter efficiency, and inference speed. The backbone progressively transforms raw pixel data into increasingly abstract feature maps through stacked convolutional, normalization, and non-linear activation layers. The selection of the backbone is critical, as it directly influences the model's ability to capture relevant visual patterns and its suitability for deployment under hardware or real-time constraints. In industrial settings, backbones may be further optimized or pruned to enable real-time inference on edge devices, while maintaining sufficient feature richness to support downstream tasks.

Neck: Feature Aggregation and Enhancement

The neck functions as an intermediate module that processes and aggregates the multi-scale features generated by the backbone. This stage is particularly important in tasks involving objects or defects of varying sizes, where the effective fusion of low-level and high-level features is essential for robust performance. Popular neck architectures include the Feature Pyramid Network (FPN), which constructs a top-down pathway to combine semantic-rich features at different resolutions, the Path Aggregation Network (PAN), which further refines spatial information through bidirectional paths, and the Spatial Pyramid Pooling (SPP) module, which pools features at multiple spatial scales to increase the receptive field. By integrating these or similar blocks, the neck enhances the expressivity and context-awareness of the feature maps, providing the subsequent head with a more comprehensive representation of the input image. In recent models, the neck also serves as a site for attention mechanisms or additional normalization, further improving feature selectivity and model stability.

Head: Task-Specific Prediction

The head constitutes the terminal segment of the architecture and is responsible for producing the final outputs corresponding to the specific computer vision task at hand. In object detection frameworks, the head typically consists of multiple parallel branches predicting bounding box coordinates, class probabilities, and objectness scores for each candidate region. For semantic or instance segmentation, the head often includes upsampling layers and produces dense, pixel-wise classification masks. The architectural design of the head is closely aligned with the target application: for classification, it may be a series of fully connected layers; for detection and segmentation, it may incorporate anchor generation, non-maximum suppression, or custom loss functions. Additionally, the head may be adapted to address particular challenges such as class imbalance, rare defect detection, or the need for real-time output.

Advantages of Modular Architecture

This modular decomposition provides significant practical benefits. It enables the reuse of pre-trained backbones, expediting transfer learning and reducing the required amount of task-specific labelled data. Furthermore, it allows independent replacement or fine-tuning of each component, supporting experimentation with novel architectures or rapid adaptation to new operational requirements. In industrial defect detection workflows, this flexibility is crucial for accommodating evolving manufacturing processes, novel defect types, or changing hardware constraints, thereby supporting continuous improvement and maintainability of deployed vision systems.

2.4. Training Strategies and Data Handling

The success of a deep-learning model depends not only on its architecture but also critically on the strategies employed during training and the quality of the data used. In industrial defect detection, where data can be scarce, imbalanced, or highly variable, careful management of the training process and dataset preparation is essential to achieve robust and generalisable models.

This section explores the key methodologies and best practices related to training strategies and data handling. It begins with a discussion on data augmentation techniques, which artificially expand the training set through transformations such as rotations, flips, scaling, and colour adjustments. These techniques are crucial for improving model generalisation, particularly when annotated data is limited.

The use of synthetic and domain-randomised data is also examined, highlighting how artificially generated samples can help bridge gaps in real-world datasets and expose models to a wider range of variations. Such approaches are especially valuable when dealing with rare defects or when collecting real-world samples is impractical or costly.

Further, the section addresses transfer learning, fine-tuning, and continual learning strategies. These methods enable models to leverage pre-existing knowledge from related domains, reducing training time, improving performance, and facilitating adaptation to evolving production environments.

Attention is also given to hyper-parameter optimisation and systematic experiment tracking. Careful tuning of parameters such as learning rates, batch sizes, and regularisation terms, combined with rigorous tracking of experimental configurations and outcomes, ensures reproducibility and facilitates iterative model improvement.

Finally, the section delves into challenges associated with imbalanced data and rare-defect mitigation. Special techniques, such as resampling, class weighting, and anomaly detection approaches, are discussed as essential tools for ensuring that models perform well even when certain defect categories are underrepresented.

By outlining these critical aspects of training and data management, this section equips the reader with a comprehensive understanding of the practical steps necessary to develop high-performing, reliable deep-learning systems for industrial defect detection.

2.4.1. Data Augmentation, Synthetic and Domain-Randomised Data

The availability and quality of training data critically influence the performance of deep-learning models. Particularly in industrial defect detection, the acquisition of large, diverse, and accurately annotated datasets is challenging, often costly, and sometimes impractical due to the rarity of specific defect types or constraints on production-line interruptions. To overcome these limitations and improve generalisation, various data-handling strategies such as data augmentation, synthetic data generation, and domain randomisation have emerged as essential techniques.

Data augmentation encompasses a series of transformations applied to existing images to artificially expand the size and diversity of training datasets. Common augmentation techniques include rotations, translations, scaling, cropping, colour jittering, and flipping. These transformations help models become invariant to minor variations that naturally occur in real-world conditions, effectively enhancing robustness and generalisation without incurring significant annotation costs.

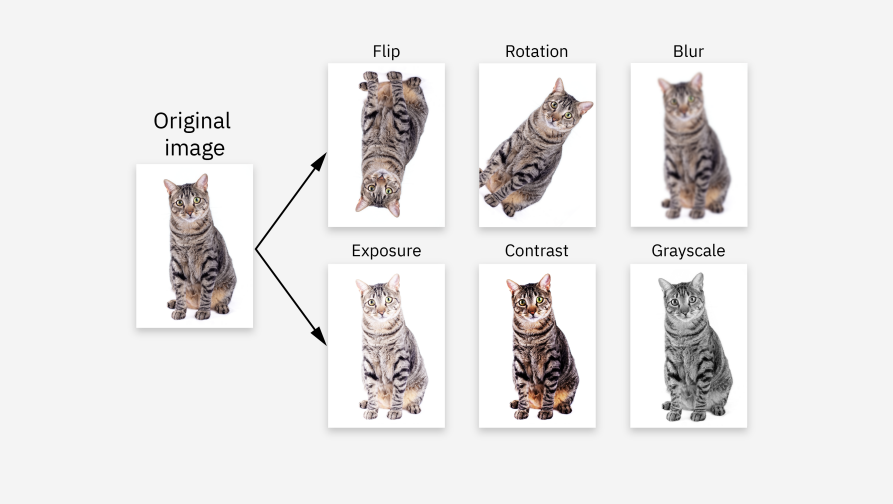


Figure 2.13: Illustration of data augmentation techniques applied to an original image, including flipping, rotation, blur, exposure, contrast adjustment, and grayscale conversion to enhance model robustness.[41]

When real-world data is scarce or insufficient, synthetic data generation presents a viable alternative. Synthetic images, generated using computer graphics, simulations, or generative models, can be produced in large volumes with exact control over defect characteristics and environmental conditions. This control facilitates systematic training scenarios that might be difficult or impossible to replicate through conventional data collection. However, synthetic images may exhibit domain discrepancies compared to real images, necessitating additional techniques such as domain adaptation to bridge the resulting gap.

A closely related approach, domain randomisation, involves intentionally varying aspects of simulated environments or image characteristics such as textures, backgrounds, lighting, and camera angles during synthetic data generation. Rather than creating highly realistic synthetic images, domain randomisation aims to expose models to a broad range of plausible variations, thereby promoting resilience to domain shifts. This method encourages the learned representations to focus on essential features rather than superficial patterns specific to particular conditions, significantly improving transferability from synthetic environments to real-world applications.

These methodologies collectively provide effective tools for addressing dataset limitations inherent in industrial settings. By integrating data augmentation, synthetic data generation, and domain randomisation into the training pipeline, defect-detection models can achieve greater accuracy, robustness, and adaptability, making them better suited to the dynamic and varied conditions of industrial production environments.[42]

2.4.2. Transfer Learning, Fine-Tuning and Continual Learning

One of the most prominent challenges faced in industrial defect detection scenarios is the limited availability of annotated data, often making it impractical to train sophisticated deep-learning models entirely from scratch. To address this issue, strategies such as transfer learning, fine-tuning, and continual learning have been developed, allowing models to leverage existing knowledge, adapt effectively to new tasks, and progressively improve over time.

Transfer learning involves applying knowledge acquired from a previously trained model to a new but related task. Typically, models are pre-trained on large-scale, general-purpose datasets such as ImageNet, acquiring valuable representations of visual patterns and structures. When applying these models to the specific domain of industrial defect detection, only minor adaptations, usually limited to the final layers of the network, are required. This strategy significantly reduces both the amount of data needed for effective training and the computational cost, enabling rapid development of robust detection systems even with modest-sized datasets.

Fine-tuning, closely related to transfer learning, extends this approach by adjusting not only the final layers

but also selectively refining deeper layers of the pre-trained model. By carefully allowing the model to adapt its feature representations to the specific characteristics of the new domain, fine-tuning often results in improved performance, especially when domain-specific details differ significantly from those in the original pre-training data. However, careful management of learning rates and regularisation techniques is necessary to avoid overfitting, particularly given the limited datasets common in industrial applications.

Continual learning addresses another critical requirement in industrial contexts: the capacity of models to adapt to evolving conditions and incremental changes without the need for complete retraining. Production environments are dynamic, and defect patterns or manufacturing conditions may shift over time. Continual learning methods enable models to integrate new knowledge progressively while retaining previously acquired skills, mitigating the problem of catastrophic forgetting—a common pitfall where models lose performance on previously learned tasks after being trained on new data.

Collectively, these strategies facilitate the practical and efficient implementation of deep-learning systems in real-world industrial scenarios, where data limitations and environmental variability are the norm rather than the exception. By harnessing transfer learning, fine-tuning, and continual learning, defect-detection models become more robust, adaptable, and cost-effective, ensuring sustained accuracy and reliability over time.[43]

2.4.3. Hyper-parameter Optimisation and Experiment Tracking

The performance of machine-learning models is highly sensitive to the choice of hyper-parameters, which govern aspects such as learning rate, batch size, network depth, regularisation strength, and data augmentation intensity. Unlike model parameters, which are learned during training, hyper-parameters must be set before training begins and can significantly influence convergence speed, generalisation ability, and overall model robustness. Consequently, systematic hyper-parameter optimisation is a critical component of building high-performing defect-detection systems.

Traditional approaches to hyper-parameter selection, such as manual tuning based on intuition or grid search over predefined ranges, are often time-consuming and inefficient, particularly when dealing with high dimensional hyper-parameter spaces. More sophisticated techniques, such as random search, Bayesian optimisation, and evolutionary algorithms, offer more efficient alternatives by intelligently exploring the space of possibilities. Bayesian optimisation, in particular, models the performance landscape using probabilistic surrogates, enabling more informed and efficient exploration compared to naive exhaustive methods.[44]

However, optimising hyper-parameters in isolation is not sufficient; careful and systematic experiment tracking is essential to ensure reproducibility, facilitate model comparison, and enable informed decision-making. Modern deep-learning workflows increasingly rely on dedicated experiment-tracking tools that record configurations, metrics, model artefacts, and training logs in a structured manner. Platforms such as MLflow, Weights & Biases, and TensorBoard provide capabilities for tracking experiments over time, visualising learning curves, and performing detailed comparisons across different model versions.

Experiment tracking also supports better collaboration and transparency, both of which are crucial in industrial environments where models must be validated, audited, and maintained over long operational periods. Moreover, by preserving a detailed record of experiments, it becomes feasible to trace the evolution of a model from early prototypes to production-ready deployments, identify factors contributing to performance improvements or degradations, and rapidly recover previous best-performing configurations when needed.

Altogether, hyper-parameter optimisation and rigorous experiment tracking are not merely auxiliary practices but central pillars of a robust and efficient deep-learning development process. In the context of this thesis, they underpin the methodology used to identify optimal model configurations and ensure that the presented results are reproducible, verifiable, and systematically obtained.

2.4.4. Imbalanced Data and Rare-Defect Mitigation

In industrial defect detection, the distribution of defects is often heavily imbalanced. Most manufactured parts are defect-free, while actual defective cases constitute only a small fraction of the total production. This natural imbalance poses significant challenges for machine-learning models, which tend to be biased towards the majority class during training, leading to poor sensitivity in detecting rare, but critical, defects.

When trained on imbalanced datasets, standard models can achieve deceptively high accuracy by predominantly predicting the majority class, effectively neglecting the minority classes that are often of greatest operational importance. Therefore, addressing data imbalance is essential to ensure that defect-detection systems maintain high recall and precision rates on rare-defect categories.

Several strategies can be employed to mitigate the effects of class imbalance. Data-level methods such as oversampling the minority class or undersampling the majority class are common approaches. Oversampling involves replicating or synthetically generating new examples of rare defects, with techniques like SMOTE (Synthetic Minority Over-sampling Technique) providing more sophisticated alternatives to simple duplication. Undersampling reduces the size of the majority class to balance the dataset but must be applied carefully to avoid losing valuable information.

At the algorithmic level, cost-sensitive learning introduces class-weighted loss functions that penalise the misclassification of minority classes more heavily. This encourages the model to pay greater attention to rare defects during training, improving their detection rates without requiring explicit manipulation of the dataset distribution. Additionally, focal loss, originally proposed for dense object detection tasks, dynamically down-weights the contribution of easy examples and focuses training on hard, minority-class examples, proving particularly effective in highly imbalanced scenarios.

Another important approach is anomaly detection, where models are trained primarily on normal examples and tasked with identifying deviations. In highly skewed datasets, this framing can be more natural and effective than attempting to learn from a limited set of rare-defect examples directly.

Effectively addressing imbalance and rare-defect mitigation is crucial for building trustworthy industrial inspection systems. Ensuring that models are sensitive to all defect types, regardless of their frequency, directly impacts product quality, operational safety, and customer satisfaction. For this reason, imbalance management strategies form an integral part of the data preparation and model development process described in this thesis.[45]

2.5. Industrial Deployment Considerations

While achieving high performance in controlled experimental settings is important, the true value of a machine-learning system lies in its successful deployment within real industrial environments. Industrial settings impose unique challenges that extend beyond model accuracy, including strict constraints on computational resources, real-time inference requirements, integration with existing manufacturing infrastructures, and the need for robust, maintainable solutions.

This section addresses the critical considerations involved in transitioning deep-learning models from development to operational deployment. It begins by examining model compression techniques such as pruning, quantisation, and knowledge distillation, which are essential for reducing model size and computational complexity without significant loss of accuracy. These methods enable the deployment of sophisticated models on resource-limited edge devices commonly used in industrial environments.

Real-time inference capabilities are also explored, with particular focus on deploying models to embedded hardware platforms such as NVIDIA Jetson devices and Intel Neural Processing Units (NPUs). Achieving low-latency, high-throughput performance is a key requirement for applications like real-time defect detection on production lines, where delays can compromise operational efficiency.

The section further discusses the integration of deep-learning models with industrial control systems, in-

cluding Programmable Logic Controllers (PLCs), Manufacturing Execution Systems (MES), and Supervisory Control and Data Acquisition (SCADA) systems. Seamless interoperability with these established infrastructures is vital to ensure that AI solutions enhance, rather than disrupt, existing manufacturing workflows.

Finally, considerations regarding monitoring, maintenance, and update strategies are reviewed. In dynamic industrial environments, models must be continuously monitored for performance degradation, updated to adapt to new defect types, and managed to ensure compliance with safety and operational standards.

By outlining these deployment considerations, this section highlights the importance of designing machine-learning systems with practical industrial constraints in mind, ensuring that the research conducted can have tangible, sustainable impact in real-world manufacturing contexts.

2.5.1. Model Compression, Pruning and Quantisation

Deploying deep-learning models in industrial environments often requires balancing predictive performance with computational efficiency. Many production settings impose strict constraints on model size, memory consumption, inference latency, and energy usage, especially when deployment targets embedded systems, edge devices, or real-time inspection lines. In this context, model compression techniques, including pruning and quantisation, play a pivotal role in enabling the practical deployment of defect-detection systems without significant sacrifices in accuracy.

Model compression broadly refers to a set of techniques aimed at reducing the computational and memory footprint of neural networks. One of the most widely used approaches is pruning, which involves removing redundant or less important weights, neurons, or even entire filters from the network. Pruning can be applied in a structured or unstructured manner; structured pruning removes entire units such as channels or layers, which better aligns with hardware optimisation, while unstructured pruning eliminates individual connections, offering finer granularity but sometimes complicating efficient implementation. Properly applied, pruning can substantially reduce model complexity while maintaining, or even occasionally improving, generalisation by eliminating overfitting-prone parameters.[46]

Quantisation complements pruning by reducing the numerical precision of model weights and activations. Instead of using 32-bit floating-point representations, quantised models employ lower-bit representations, such as 16-bit, 8-bit, or even binary formats. Quantisation drastically reduces model size and accelerates inference, particularly on hardware accelerators that support low-precision operations. While aggressive quantisation can lead to slight performance degradation, careful calibration and quantisation-aware training can mitigate this impact and preserve the model's accuracy.

Beyond these core methods, other compression strategies such as knowledge distillation, where a compact "student" model learns to mimic a larger "teacher" model, can further enhance deployment efficiency without requiring extensive retraining.

The integration of model compression, pruning, and quantisation is essential for achieving scalable, cost-effective, and reliable industrial AI solutions. In the specific case of defect detection, where rapid and accurate decisions must often be made in resource-constrained settings, these techniques enable the real-world applicability of models initially developed in high-performance research environments.

2.5.2. Real-Time Inference on Edge/Embedded Hardware

In industrial environments, particularly within production lines and quality-control systems, real-time inference is often a non-negotiable requirement. The ability to analyse visual data and make immediate decisions without introducing delays is critical for maintaining operational efficiency, ensuring product quality, and reducing downtime. Deploying deep-learning models on edge or embedded hardware platforms offers a promising solution, providing the necessary computational capabilities directly at the point of data acquisition without reliance on remote servers or cloud infrastructures.

Edge devices, such as NVIDIA Jetson platforms, Intel Neural Compute Sticks, or custom FPGA and ASIC so-

lutions, are designed to deliver high-throughput, low-latency inference while operating under constraints of power consumption, size, and environmental robustness. These characteristics are particularly suited to industrial settings where reliability, compactness, and energy efficiency are vital.

However, achieving real-time performance on edge devices presents significant challenges. Models must often be carefully optimised through techniques such as pruning, quantisation, and architecture selection, ensuring that computational demands align with the hardware's capabilities. Additionally, memory management becomes a critical consideration, as embedded systems typically offer much less RAM and storage compared to server-grade GPUs.

Another key factor is the use of optimised inference engines such as TensorRT, OpenVINO, or ONNX Runtime, which compile models into highly efficient, hardware-specific binaries. These engines apply graph optimisations, operator fusion, and precision lowering techniques to maximise throughput while maintaining acceptable accuracy levels.

Beyond technical optimisation, deploying models on edge hardware also enhances system resilience by reducing dependency on external networks. In industrial facilities where network interruptions are possible or where data privacy and security are paramount, on-device inference ensures continuous, autonomous operation with minimal external dependencies.

Ultimately, enabling real-time inference on edge and embedded platforms is not merely an exercise in technical adaptation; it represents a fundamental enabler for scalable, robust, and intelligent manufacturing systems. Throughout this thesis, careful consideration is given to deployment constraints, with model design and optimisation strategies explicitly geared towards fulfilling the stringent real-time and resource-efficiency requirements of industrial applications.[47]

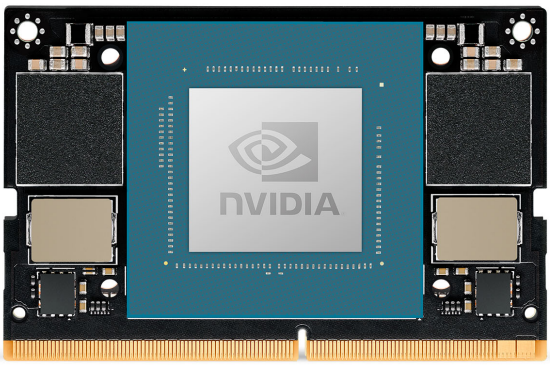


Figure 2.14: NVIDIA Jetson Orin module — a compact, high-performance AI computing platform designed for edge and embedded systems. Source: NVIDIA [48]

2.5.3. Integration with PLCs, MES and SCADA Systems

For a machine-learning-based defect-detection system to deliver tangible value in industrial environments, it must not operate in isolation but instead integrate seamlessly into the broader manufacturing ecosystem. This integration ensures that the outputs of the AI models can drive actionable decisions, interact with existing automation infrastructure, and contribute to the overall optimisation of production processes. Three critical layers commonly involved in such integration are Programmable Logic Controllers (PLCs), Manufacturing Execution Systems (MES), and Supervisory Control and Data Acquisition (SCADA) systems.

PLCs are the backbone of industrial automation, responsible for the direct control of machinery, sensors, and actuators. Integrating deep-learning models with PLCs allows defect-detection outputs to trigger immediate control actions, such as rejecting defective products, adjusting process parameters, or halting production lines for inspection. Achieving reliable and low-latency communication between inference engines and PLCs typically involves the use of industrial communication protocols such as OPC UA, Modbus, or Ethernet/IP, and demands strict adherence to real-time and safety requirements.

At a higher level, MES systems manage the execution of manufacturing operations on the shop floor, providing scheduling, tracking, and performance analysis functionalities. Connecting AI-driven defect detection with MES platforms enables richer data aggregation and contextualisation. For instance, defect rates can be correlated with specific production batches, operator shifts, or raw material lots, facilitating root-cause analysis and continuous improvement initiatives.

SCADA systems oversee and coordinate large-scale industrial processes, offering real-time monitoring, alarm management, and historical data logging. Integrating defect-detection outputs into SCADA dashboards allows operators and supervisors to visualise inspection results in real time, receive alerts about quality deviations, and interactively explore inspection histories. Such integration enhances transparency and operational responsiveness, contributing to higher product quality and production efficiency.

Successfully embedding AI models within these industrial layers requires careful system design to ensure reliability, fault tolerance, scalability, and compliance with industry standards. It also demands consideration of cybersecurity practices to protect critical manufacturing assets from potential vulnerabilities introduced by new digital components.

In this thesis, deployment strategies are conceived with these integration requirements in mind, aiming not just for model accuracy, but for practical, operationally viable solutions that can be fully incorporated into existing industrial infrastructures to support smarter and more adaptive manufacturing environments.[49]

3

Methodology and Experimental Work

This chapter details the comprehensive methodology underpinning the development, training, evaluation, and deployment of deep-learning models for the automated detection of defects in metallurgical manufacturing components. The experimental framework is designed to address the challenges of industrial computer vision applications, emphasizing scientific rigor, reproducibility, and practical viability.

The chapter opens with an overview of the software technologies and platforms that constitute the technological backbone of this research. Key tools include Python's scientific stack (NumPy, Pandas, Matplotlib) for data manipulation and visualization, Ultralytics YOLOv8 for state-of-the-art object detection and segmentation, and TensorFlow/Keras for the construction, training, and fine-tuning of deep-learning models. The integration of Roboflow streamlines dataset management and annotation, while Google Colab provides the necessary computational resources and scalability for conducting large-scale experiments. Throughout the process, robust experiment tracking and version control practices ensure transparency and reproducibility.

A major focus of this chapter is the pre-processing and data preparation workflow, which is foundational to the success of any machine learning system. Here, a systematic approach to exploratory data analysis (EDA) is outlined, including the examination of directory structures, file metadata, and image content. Rigorous data cleaning procedures—such as duplicate detection, greyscale analysis, and unique identifier generation—are implemented to enhance dataset quality. Attention is also given to the critical process of partitioning data into training, validation, and test subsets, taking into account best practices to avoid data leakage and ensure robust model evaluation. Special consideration is devoted to addressing challenges like class imbalance, the scarcity of defective samples, and the definition of a precise defect taxonomy, all of which are common in industrial inspection tasks.

The model development section provides a step-by-step account of constructing and training a variety of deep-learning architectures tailored for defect detection. This includes both classification and segmentation models, leveraging the latest advances in YOLOv8 as well as custom models implemented in Keras and TensorFlow. The methodology encompasses hyperparameter optimization strategies, regularization techniques, and data augmentation practices aimed at improving generalization and mitigating overfitting. For segmentation tasks, detailed workflows are presented covering dataset indexing, manual annotation, export, and verification, as well as integration with automated pipelines. Hybrid approaches—such as cascading segmentation with classification or the integration of YOLOv8 backbones into custom Keras/TensorFlow networks—are explored to maximize model performance across different scenarios.

Further, the chapter discusses the adaptation of model architectures to industrial deployment requirements, including considerations for model size, inference speed, and compatibility with production hardware. Techniques such as transfer learning, fine-tuning, and manual layer conversion are described in the context of leveraging pre-trained weights for specialized defect detection tasks. Emphasis is placed on experiment tracking and model validation, ensuring that each experimental result can be reliably reproduced and compared.

The final section is dedicated to the interpretability of the trained models, with a focus on the application of the Grad-CAM algorithm for visual explanation of model decisions. The rationale behind model selection for Grad-CAM, the integration and preprocessing pipeline, and the practical steps for generating and analysing activation heatmaps are all presented in detail. These interpretability experiments provide valuable insight into the inner workings of deep neural networks, helping to validate their reliability in critical industrial applications.

By providing a thorough and structured overview of the experimental methodology, this chapter establishes a solid foundation for understanding and interpreting the results, analyses, and conclusions that follow in subsequent sections of the thesis.

3.1. Software Libraries and Deep Learning Frameworks

The development and evaluation of deep-learning models for industrial defect detection rely on a carefully selected set of software libraries and frameworks. The technological stack adopted in this thesis has been chosen to balance ease of development, model interpretability, and industrial applicability, while ensuring reproducibility and robustness throughout the experimental pipeline.

Python serves as the primary programming language due to its rich ecosystem of scientific computing libraries, intuitive syntax, and widespread adoption in machine learning research and industry. Essential libraries such as NumPy and Pandas support data manipulation and numerical processing, while Matplotlib provides reliable tools for visualisation and exploratory data analysis.

For deep-learning model construction and training, TensorFlow with its Keras API is employed as the central framework. Keras offers a high-level, modular interface for defining, compiling, and training neural networks, significantly reducing the complexity of implementation while maintaining full compatibility with TensorFlow's deployment and optimisation capabilities. This framework is particularly well-suited for model lifecycle management, including training, evaluation, and exporting models to optimised formats such as TensorFlow Lite.

In addition to custom models built with TensorFlow/Keras, this work integrates Ultralytics' implementation of YOLOv8—a high-performance architecture originally designed for object detection and segmentation. Although YOLOv8 is primarily associated with detection tasks, its classification capabilities are leveraged here by fine-tuning its pre-trained backbone for binary classification of defective versus non-defective parts. The YOLOv8 framework offers built-in support for mixed-precision training, data augmentation, and exportable model formats, making it an effective tool for rapid experimentation.

Finally, Roboflow is employed extensively to manage the dataset lifecycle, from annotation and augmentation to version control and export. While other image processing libraries such as OpenCV were used sparingly for basic preprocessing tasks like image normalization, the central role in data preparation is held by Roboflow due to its integration with modern computer vision workflows and compatibility with the selected deep-learning frameworks.

This software stack provides the foundational infrastructure for the experimental work presented in this thesis. Its components enable structured experimentation, support modern training workflows, and facilitate the practical deployment of models in real-world industrial scenarios.

3.1.1. Python & Scientific Stack (NumPy, Pandas, Matplotlib)

Python[50] has been the programming language of choice for the development of the entire experimental framework due to its versatility, extensive library ecosystem, and strong support within the machine-learning and computer-vision communities. Its simplicity and readability facilitate rapid prototyping, experimentation, and collaboration, which are essential characteristics for both academic research and industrial innovation.

At the core of the Python scientific computing ecosystem lie NumPy and Pandas, two fundamental libraries

that underpin most data manipulation and analysis tasks. NumPy provides efficient array structures and a comprehensive suite of mathematical operations, enabling fast numerical computation essential for preprocessing large-scale image datasets and managing model inputs and outputs. Its highly optimised operations on multi-dimensional arrays (tensors) are critical for implementing custom data transformations and augmentations during the training phase.

Complementing NumPy, Pandas offers powerful tools for structured data handling, making it particularly well-suited for managing metadata associated with images, such as defect classifications, labelling information, and cross-validation splits. The library's DataFrame structure simplifies the organisation, querying, and summarisation of tabular data, which facilitates systematic experiment tracking and evaluation.

For visualisation and exploratory data analysis, Matplotlib has been extensively employed. It enables the creation of detailed plots, charts, and heatmaps, which are crucial for understanding dataset distributions, monitoring training dynamics, and interpreting model performance. Visual analysis using Matplotlib provides immediate, interpretable feedback during model development, assisting in early detection of issues such as class imbalance, overfitting, or convergence problems.

The combined use of these libraries forms the scientific backbone of the project's data preparation and analysis phases. Their reliability, performance, and widespread community support ensure a solid foundation for the methodological rigor demanded by this thesis and for the robustness required in industrial applications.

3.1.2. Ultralytics YOLO (v8)

Ultralytics' implementation of YOLOv8 represents one of the latest advancements in the YOLO (You Only Look Once) family of deep-learning models. YOLO has historically been renowned for its real-time object detection capabilities, offering a highly efficient trade-off between accuracy and speed. YOLOv8 extends this tradition with a fully re-engineered architecture that brings significant improvements in flexibility, performance, and usability across a range of computer-vision tasks, including detection, segmentation, and classification.

From a technical perspective, YOLOv8 introduces a new backbone and neck design aimed at enhancing feature extraction while maintaining computational efficiency. Its architecture moves towards an anchor-free detection paradigm, which simplifies the model's learning process and improves training stability. Additionally, it supports dynamic label assignment and optimised multi-scale feature aggregation, allowing the model to adapt more effectively to varied object sizes and complex visual patterns.

Although YOLOv8 is inherently designed for detection and segmentation tasks, it also provides robust capabilities for pure image classification, enabling the use of its pre-trained backbones in transfer learning scenarios. In the context of this thesis, YOLOv8 is leveraged as a pre-trained model to evaluate its effectiveness in classifying industrial parts based on visual features. By fine-tuning the YOLOv8 architecture specifically for classification purposes, the project aims to assess its suitability compared to other standard classification models and explore its potential advantages in terms of generalisation, robustness, and computational efficiency.

The high modularity and user-friendly API provided by Ultralytics facilitate the integration of YOLOv8 into the experimental framework, supporting tasks such as model fine-tuning, evaluation, and export for deployment scenarios. Its support for mixed-precision training, on-the-fly augmentation, and streamlined model export formats makes it a highly attractive technology for rapid experimentation and benchmarking within industrial machine-learning pipelines.

Thus, YOLOv8 serves not as a detection engine in this work, but rather as a foundation for classification studies, allowing a rigorous exploration of its capabilities in the specific context of defect identification and part classification.

3.1.3. TensorFlow / Keras

TensorFlow, developed by Google Brain, is one of the most widely used deep-learning frameworks in both research and industrial applications. Designed for large-scale machine-learning tasks, TensorFlow provides a flexible and efficient platform for building, training, and deploying neural networks. Its architecture supports a wide variety of computational backends, including CPUs, GPUs, and specialised accelerators such as TPUs (Tensor Processing Units), allowing models to scale seamlessly from research prototypes to production environments.

Keras, originally an independent project and now fully integrated into TensorFlow as its high-level API, offers a user-friendly and modular interface for deep-learning model development. Through Keras, TensorFlow simplifies the creation and training of complex neural networks, providing an intuitive structure for defining models, managing training workflows, and conducting evaluation and inference tasks. Its abstraction layers allow researchers and developers to focus on design and experimentation without being burdened by low-level computational details.

TensorFlow/Keras supports a rich ecosystem of tools essential for modern machine learning, including automatic differentiation, model checkpointing, early stopping, mixed-precision training, and a wide array of built-in layers, loss functions, and optimisation algorithms. Additionally, the TensorFlow Extended (TFX) ecosystem and TensorFlow Lite facilitate model deployment across diverse platforms, ranging from server environments to mobile and embedded devices.

In the context of this thesis, TensorFlow/Keras has been employed to develop, train, and fine-tune classification models aimed at detecting defects in industrial parts. The framework's capabilities in model management, hyper-parameter tuning, and streamlined export to optimised inference formats play a key role in ensuring that the resulting models are not only accurate but also suitable for deployment in resource-constrained industrial environments.

TensorFlow/Keras thus forms a core technological foundation for the experimental work presented, offering a powerful and flexible platform to explore various modelling strategies and validate their performance in the specific context of binary classification of defective versus non-defective parts.

3.2. Data and Computational Platforms

This section describes the platforms employed for two fundamental components of the deep-learning pipeline: data preparation and computational execution. While the success of deep-learning models is often attributed to architectural design and training strategies, the infrastructure used to manage datasets and execute computations plays an equally critical role. Roboflow and Google Colab have been selected in this work for their capacity to address the practical demands of computer vision projects in industrial contexts. Roboflow facilitates structured dataset management and robust augmentation procedures, while Google Colab provides scalable cloud-based computation for training and evaluating the proposed models. Together, these platforms support a reproducible, efficient, and modular experimental workflow.

3.2.1. Roboflow

Roboflow[51] is a cloud-based platform designed to streamline the entire data pipeline for computer-vision projects, offering solutions for dataset management, annotation, augmentation, and export. It addresses many of the practical challenges associated with preparing image data for machine-learning workflows, particularly in projects where high-quality, well-structured datasets are crucial for model performance.

One of Roboflow's core strengths lies in its ability to centralise dataset organisation and automate pre-processing tasks. Users can easily upload images, apply a range of augmentations—such as rotations, flips, contrast adjustments, or noise injection—and generate multiple dataset versions optimised for different training strategies. This flexibility facilitates systematic experimentation, allowing controlled studies of the effects of augmentation on model generalisation and robustness.

Roboflow also provides powerful annotation management tools. Through its web-based interface, bounding boxes, segmentation masks, and classification labels can be created and edited collaboratively. In projects such as defect detection, where accurate labelling is fundamental for model success, this functionality ensures consistent, high-quality annotations across large datasets.

Another critical advantage of Roboflow is its support for exporting datasets in various formats compatible with popular machine-learning frameworks, including TensorFlow, YOLO, and others. This capability simplifies the integration of data into existing training pipelines, reducing manual conversion efforts and minimising the risk of errors during data preparation.

In the context of this thesis, Roboflow has been employed to organise and augment the datasets used for training and evaluating classification models. Its role in standardising data pre-processing and managing dataset versions has been essential for maintaining the reproducibility and consistency of experimental results, while also accelerating the iterative development process.

By facilitating efficient and reliable data pipeline management, Roboflow contributes significantly to the methodological rigor and operational efficiency required for deploying deep-learning systems in industrial defect-detection applications.

3.2.2. Google Colab

Google Colaboratory (Colab)[52] is a cloud-based platform that provides an accessible and flexible environment for developing, training, and testing machine-learning models. Built on top of Jupyter Notebooks, Google Colab offers a fully interactive workspace where users can write and execute Python code directly from their browsers, without the need for complex local configurations or specialised hardware.

One of the main advantages of Google Colab is its provision of free access to computational resources, including GPUs and TPUs, which significantly accelerates the training of deep-learning models. For projects focused on computer vision, where training times can be substantial due to the high dimensionality of image data, this access enables rapid prototyping and iterative experimentation that would otherwise require dedicated hardware infrastructure.

Google Colab also integrates seamlessly with major machine-learning libraries such as TensorFlow, Keras, and PyTorch, and provides a straightforward mechanism for connecting to external data sources like Google Drive. This facilitates efficient data handling, model checkpointing, and experiment management within a single unified environment.

In the context of this thesis, Google Colab has been employed as a primary development and experimentation platform, supporting the training and evaluation of classification models for defect detection tasks. Its collaborative features, including the ability to share notebooks and maintain version histories, have further enhanced the reproducibility and transparency of the research process.

By providing an easily accessible, powerful, and flexible computational environment, Google Colab has played a central role in enabling the rapid exploration and validation of deep-learning models under the constraints and objectives of this industrial application.

3.3. Pre-processing and Data Preparation

The preparation of data constitutes a fundamental pillar in the development of any machine-learning system, particularly in domains such as industrial defect detection, where small imperfections in texture, shape, or material composition can have critical implications for product quality. The effectiveness of a model does not solely depend on its architecture or optimisation techniques; it is heavily influenced by the cleanliness, structure, and representativeness of the data it learns from. In industrial scenarios, where datasets may contain imbalances, inconsistencies, or artefacts introduced during acquisition, a rigorous pre-processing and preparation phase becomes indispensable.

This section details the processes undertaken to transform raw image datasets into structured, high-quality inputs suitable for deep-learning model training and evaluation. The methodology begins with exploratory data analysis (EDA) to gain a comprehensive understanding of the dataset's composition, distribution of defect types, and potential sources of bias. Through systematic analysis, key issues such as class imbalance, outlier presence, and annotation inconsistencies are identified early in the workflow.

Following the insights gained from EDA, targeted data cleaning procedures are applied. These include the removal of corrupted files, correction of mislabelled examples, and standardisation of image properties such as resolution and colour profiles. Normalisation strategies are then employed to ensure that input images conform to the requirements of modern deep-learning architectures, facilitating more stable and efficient model convergence.

A vital component of the data preparation process is the establishment of a defect taxonomy. Clear, consistent definitions of defect categories are critical for training models capable of making meaningful distinctions between acceptable and defective parts. Without a well-defined taxonomy, even sophisticated models risk underperforming due to ambiguous or inconsistent labels.

The dataset is subsequently annotated and organised into appropriate structures for both classification and segmentation tasks, depending on the experiment. Particular care is taken to design robust train, validation, and test splits that preserve the statistical properties of the overall dataset while ensuring that evaluation results reflect true model generalisation rather than memorisation. Strategies such as stratified sampling are considered to mitigate the effects of imbalance across splits.

In addition to these core steps, specific attention is given to addressing the challenges posed by imbalanced data and rare-defect representation. Techniques such as data augmentation, class-weighting, and synthetic sample generation are explored as means to enhance the diversity and representativeness of the training data, ultimately contributing to the development of models that perform reliably across a wide range of defect types and conditions.

This structured and methodical approach to pre-processing and data preparation ensures that the subsequent phases of model development rest on a foundation of high-quality, consistent, and meaningful data, aligning both with research rigor and with the operational demands of industrial deployment.

3.3.1. Exploratory Data Analysis

Exploratory Data Analysis (EDA) represents a foundational stage in the development pipeline of computer vision models. Its primary goal is to gain a comprehensive understanding of the dataset prior to model training, allowing us to identify potential issues, validate assumptions, and inform subsequent design decisions. In the context of image-based machine learning, this process extends beyond a superficial review of the visual data and encompasses a systematic inspection of metadata, file naming conventions, annotation formats, and directory structures.

A well-executed EDA serves multiple purposes. First, it enables a thorough visual examination of the sample images, helping to detect variations in resolution, aspect ratios, lighting conditions, or defects that may affect model performance. Second, it facilitates the validation of labels and annotations, ensuring their consistency, completeness, and alignment with the intended taxonomy of defects. In addition, it is essential to identify and remove duplicate or corrupted files, as their presence could introduce bias or degrade generalisation capabilities.

Moreover, this phase often involves the application of initial preprocessing steps, such as image normalization, format unification, and label cleaning, which contribute to establishing a high-quality dataset. Special attention is also given to class distributions, as imbalanced datasets can severely impact the performance of classification and segmentation models. Visualisations such as histogram plots, scatter matrices, and image grids are frequently used to support the analysis and interpretation of the data.

In this section, we systematically carry out the exploratory analysis of the dataset that will later serve as the

foundation for model training. This includes verifying the dataset structure, inspecting the integrity and representativeness of the samples, and preparing the data for subsequent stages such as augmentation, splitting, and labelling. Through this process, we aim to establish a clean, coherent, and well-understood dataset tailored to the specific requirements of defect detection in industrial components.

3.3.1.1. Directory Structure

The first step in conducting an effective exploratory data analysis involves examining the dataset's directory structure. Understanding how the data is organised on disk can offer valuable insights into its internal composition, class distribution, and annotation logic. This is particularly relevant in computer vision tasks, where directory hierarchies often encode semantic information related to the problem formulation.

The dataset under study is structured as follows:

```
+-- casting_512x512
|   +-- casting_512x512
|       +-- def_front
|       \-- ok_front
\-- casting_data
    +-- casting_data
        +-- test
        |   +-- def_front
        |   \-- ok_front
        \-- train
            +-- def_front
            \-- ok_front
```

Listing 3.1: Hierarchical structure of the dataset employed for training and evaluating a casting defect classification model, showing the organization into separate folders for defective and non-defective image samples within the training and test sets.

As shown, there are two main top-level directories: `casting_512x512` and `casting_data`. Each of them follows a similar internal structure, ultimately containing two folders: `ok_front` and `def_front`. These folders represent a binary classification task, where `ok_front` contains images of non-defective components and `def_front` contains samples exhibiting visible defects on the front surface.

The `casting_data` directory introduces an additional level of organisation, explicitly separating the dataset into train and test subsets. This split suggests that the dataset is pre-partitioned for supervised learning, with one portion intended for model training and the other reserved for performance evaluation. In contrast, `casting_512x512` does not follow this convention; instead, it consolidates all samples into a single directory without subdivision. This may indicate that it serves a different purpose—potentially as an augmented dataset or a resized variant for compatibility testing.

To better understand the dataset composition, a count of image files in each directory is performed:

```
casting_512x512/casting_512x512/def_front   | 781   items
casting_512x512/casting_512x512/ok_front    | 519   items
casting_data/casting_data/train/def_front   | 3,758 items
casting_data/casting_data/train/ok_front    | 2,875 items
casting_data/casting_data/test/def_front    | 453   items
casting_data/casting_data/test/ok_front     | 262   items
```

Listing 3.2: Number of images in each category and split of the casting dataset, showing the distribution of defective and non-defective samples across training and test sets.

These figures reveal that `casting_data` contains significantly more samples than `casting_512x512`, confirming that the former is the primary dataset intended for training and evaluation purposes. Moreover, an evident class imbalance is present in both the training and test subsets, with defective parts consistently outnumbering non-defective ones. This imbalance must be taken into consideration during model development, particularly when defining loss functions, evaluation metrics, and sampling strategies, in order to prevent biased learning.

3.3.1.2. File Metadata

After gaining an understanding of the directory structure, the next step consists of analysing the dataset files. This section focuses on the inspection of file-level metadata and the characteristics that define the image

data used for training and evaluation.

The files consist primarily of images, each accompanied by naming conventions and resolutions that offer insight into their provenance and intended purpose. Below, the main characteristics of the file metadata are detailed:

- Format: All images are stored in JPEG format. This is a lossy compression format widely used in computer vision due to its efficiency and compatibility. JPEG images are encoded using three colour channels (typically RGB), which is important for defining the input structure of neural networks. Unlike PNG, JPEG does not support an alpha channel, hence transparency information is absent. This limitation, however, does not affect the defect detection task.

- Filename Convention: Two distinct naming patterns are observed within the dataset:

  - `cast_(def|ok)_0_\d{1,4}`: This is the predominant format and appears across all directories. It encodes both the class label (def for defective or ok for non-defective) and an identifier number. This structure facilitates automatic parsing and label extraction for supervised learning tasks.

  - `new__0_\d{1,4}`: This secondary format is less frequent and only appears in directories corresponding to more recent additions. These files are intended for extended testing scenarios or validation beyond the original dataset split, often used to evaluate model generalisation on unseen data.

- Resolution: The dataset is composed of two distinct image resolutions:

  - 512×512 pixels (aspect ratio 1:1): Found in the `dataset/casting_512x512` directory. These high-resolution images are useful for segmentation tasks or for architectures that benefit from finer spatial detail.

  - 300×300 pixels (aspect ratio 1:1): Located in the `dataset/casting_data` directory. This resolution is commonly used for lightweight models or early experimentation, given its lower memory and computational footprint.

Overall, no additional directories or filenames deviate from these conventions. The consistency in image format, resolution, and naming facilitates reproducibility and simplifies the design of automated pipelines for training and evaluation. This standardisation is essential in industrial inspection scenarios where data quality and labelling precision directly impact the effectiveness of the machine learning models.

3.3.1.3. File Content

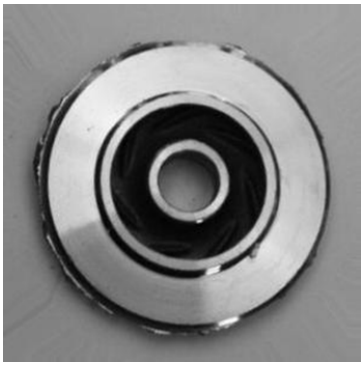
The dataset consists of grayscale images depicting a circular metallic component with a hollow centre. This part, composed of concentric rings, exhibits a predominant steel-grey colour, with a distinct black section located in the intermediate ring. According to the dataset author, this component corresponds to the upper section of an impeller used in submersible hydraulic pumps, a critical element in industrial fluid management systems.[53]

Figure 3.1: Submersible hydraulic pump impeller.

In order to guide the model development process, a preliminary analysis of relevant visual and structural characteristics is conducted. The aim is to identify features that influence defect classification and segmentation performance.

- Component Identity: The object depicted in each image is a metallic part from the upper region of a submersible pump impeller. Its recurring geometric and material structure provides a level of visual consistency, which is beneficial for feature extraction tasks.
- Defect Typology: The dataset includes a wide variety of defect types commonly found in metal casting and machining processes. These include:
  - Blow holes: gas pockets formed during solidification.
  - Pinholes: smaller, often less visible gas-induced cavities.
  - Burrs: small protrusions or residual material along edges.
  - Shrinkage defects: voids resulting from non-uniform cooling.
  - Mould material defects: contamination or structural inconsistency in the mould.
  - Pouring metal defects: issues arising from improper metal flow.
  - Metallurgical defects: internal inconsistencies such as segregation or improper grain structure.

(a) Blow hole defect.



(b) Burr defect.

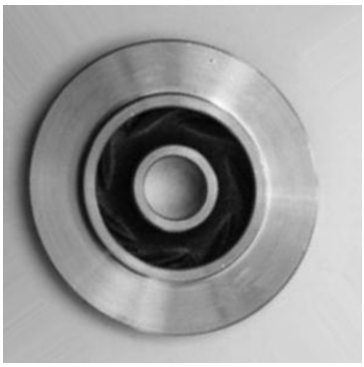
Figure 3.2: Examples of impellers exhibiting manufacturing defects.

In certain cases, multiple defect types co-occur within a single piece, further increasing the complexity of the classification task.

Figure 3.3: Defective impeller showing blow holes, burrs, and pinholes.

Additionally, the dataset contains ambiguous samples to the human eye in which the distinction between defective and non-defective parts is not immediately clear. This introduces a degree of label uncertainty, which may affect model performance and may require special attention during training and evaluation.

(a) Impeller labelled as defective.



(b) Impeller labelled as valid.

Figure 3.4: Samples with ambiguous visual characteristics.

- Colour Information: Although the dataset is confirmed to be in grayscale, there is notable variation in the tonal palette across images. These discrepancies appear to be caused by inconsistencies in lighting conditions, surface reflections, or differences in the imaging background. In some instances, the tonal range of the background is inverted relative to that of the part, potentially leading to confusion during segmentation unless adequately addressed through preprocessing.

Figure 3.5: Illustration of colour variation between two samples.

- Geometric Consistency: While the part is inherently circular, certain images exhibit deformations that suggest artificial modifications, such as geometric contraction or expansion. These may have been introduced during image pre-processing or as a result of compression artifacts. Such alterations can disrupt the uniformity of the part's circular shape and introduce noise into shape-based feature extraction methods.

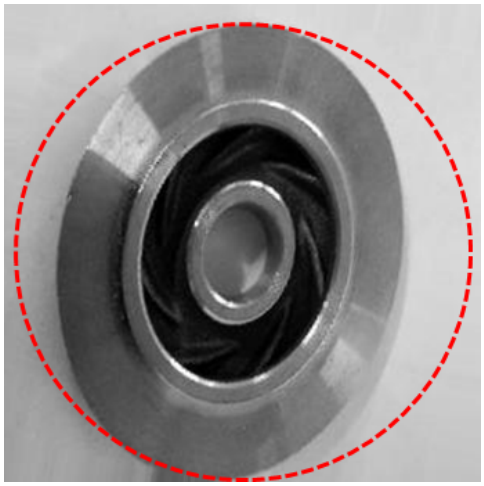


Figure 3.6: Visualization of a squashed impeller enclosed by a red circle.

- Camera Perspective: The dataset reveals inconsistencies in camera angle and orientation. Not all images are captured from a top-down, perpendicular view. This variation is likely due to the use of non-professional imaging equipment or manual acquisition methods. The lack of viewpoint standardisation may pose challenges for convolutional models that rely on translational invariance, and could potentially benefit from data augmentation techniques aimed at rotational normalisation.

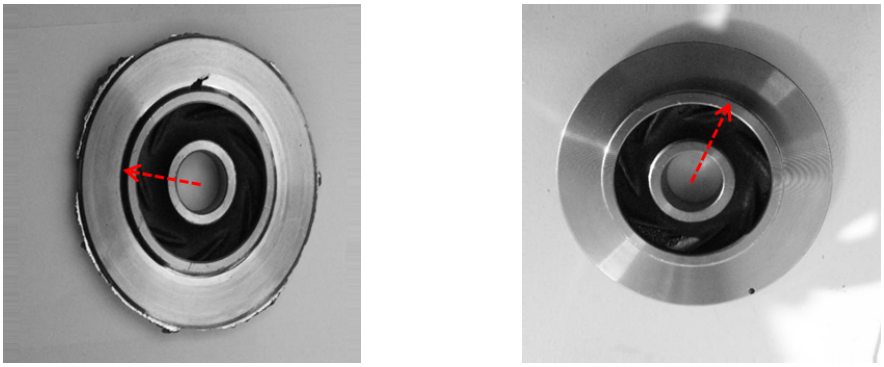


Figure 3.7: Comparison of two images captured from different perspectives.

While the dataset offers a rich and varied representation of real-world defects in industrial components, it also presents several challenges related to variability in imaging conditions, geometric distortions, and class ambiguity. These factors must be carefully considered when designing pre-processing pipelines and selecting suitable model architectures.

3.3.2. Data Cleaning and Verification

Before proceeding with model development, it is essential to ensure that the dataset meets the expected structural and semantic requirements. Although the dataset used in this project is already pre-normalised in terms of image format and dimensions, a thorough verification process remains crucial to guarantee data integrity and consistency.

Data cleaning and verification serve as a preliminary quality control stage. This involves checking for corrupted or unreadable files, ensuring that all images are correctly labelled, and verifying that the dataset adheres to the predefined taxonomy of defects. It also includes the identification and removal of duplicate samples, as well as the validation of file naming conventions and directory structures, especially when using automated data loaders or integration with tools such as Roboflow.

Integrity checks are conducted to confirm that the dataset split (training, validation, test) does not introduce data leakage or class imbalance that might skew performance metrics. Where relevant inspection is performed on a representative sample of images to validate label accuracy and visual relevance of the data.

This section documents the verification procedures applied to the dataset prior to model training, ensuring that the data pipeline rests on a solid and trustworthy foundation. Careful attention to these aspects minimises the risk of downstream issues and enhances the reproducibility and reliability of experimental results.

3.3.2.1. Duplicate Element Analysis

One critical aspect of dataset verification involves the detection and handling of duplicate samples. In image-based machine learning tasks, duplicated elements can introduce bias into the model, causing it to overfit or learn redundant patterns that do not generalise to new data. This problem becomes particularly acute when duplicates are present across different dataset splits—such as between training and validation sets—potentially leading to data leakage.

Data leakage, whether intentional or inadvertent, occurs when information from the validation or test set is accessible during training, thus inflating performance metrics and compromising the credibility of the results. Preventing such leakage is essential to ensure that model evaluation reflects true generalisation performance and is not artificially boosted by memorisation of repeated content.

For this reason, detecting and removing duplicated images, either exact copies or near-identical variants,

is a necessary preprocessing step in any robust computer vision pipeline.

To carry out the duplicate element analysis, a Python script that checks duplicated file contents, has been developed:

```python
import os

dir = "/home/leprieto/tfm/resources/dataset/casting_data"

def find_duplicate_files(directory):
    """Find and delete duplicate files based on byte-by-byte comparison."""
    file_hashes = {}
    files_to_remove = []

    for root, dirs, files in os.walk(directory):
        for file in files:
            file_path = os.path.join(root, file)

            with open(file_path, 'rb') as f:
                file_content = f.read()

            file_hash = hash(file_content)
            if file_hash in file_hashes:
                relpath1 = os.path.relpath(file_hashes[file_hash], dir)
                relpath2 = os.path.relpath(file_path, dir)
                files_to_remove.append(relpath2)
                print(f"Duplicated: {relpath1} <===> {relpath2}")
            else:
                file_hashes[file_hash] = file_path

    return files_to_remove

def delete_files(files):
    """Delete the duplicate files."""
    count = 0
    for file in files:
        os.remove(file)
        print(f"Removed: {file}")
        count += 1
    return count

files_to_remove = find_duplicate_files(dir)
count = delete_files(files_to_remove)

print(f"Files removed: {count}")
```

Listing 3.3: Python script for detecting and removing exact duplicate files across the dataset.

By executing this script, it becomes possible to automatically identify and eliminate all files that are byte-wise identical. During this process, several duplicated samples were detected within the dataset. The presence of such redundancies is particularly concerning, as they can compromise the training process by inflating performance metrics or introducing unintended data leakage across subsets.

```
Duplicated: casting_data\test\ok_front\cast_ok_0_1001.jpeg <===> casting_data\train\ok_front\cast_ok_0_1001.jpeg
Duplicated: casting_data\test\ok_front\cast_ok_0_1002.jpeg <===> casting_data\train\ok_front\cast_ok_0_1002.jpeg
Duplicated: casting_data\test\ok_front\cast_ok_0_1003.jpeg <===> casting_data\train\ok_front\cast_ok_0_1003.jpeg
...
Removed: casting_data\train\ok_front\cast_ok_0_1001.jpeg
Removed: casting_data\train\ok_front\cast_ok_0_1002.jpeg
Removed: casting_data\train\ok_front\cast_ok_0_1003.jpeg
...
Files removed: 64
```

Listing 3.4: Script output displaying multiple duplicated files.

All duplicated files identified were located across the `train` and `test` directories. To minimise the impact on the overall class distribution and preserve the integrity of the evaluation subset, the duplicated samples were removed exclusively from the train folder.

3.3.2.2. Greyscale Analysis

Another characteristic stated by the dataset provider is that all images are encoded in grayscale. In digital image representation, a true grayscale image is one in which all three RGB channels (Red, Green, Blue) hold identical intensity values for every pixel. This uniformity ensures that the image does not carry any unintended colour information, which could otherwise introduce noise or inconsistency into the model's feature extraction process.

Verifying this property across the dataset is important, especially when relying on external sources or automated preprocessing pipelines, as certain tools may preserve the three-channel format while embedding colour variations. Such discrepancies, if undetected, may affect the reproducibility and accuracy of training results.

A simple verification procedure can be implemented programmatically to ensure that each image conforms strictly to grayscale encoding. The following code snippet performs this verification:

```
import os
from PIL import Image
import numpy as np

dir = "/home/leprieto/tfm/resources/dataset/casting_data"

greyscale = True

def is_greyscale(image_path):
    image = Image.open(image_path)
    image = image.convert('RGB')
    img_array = np.array(image)
    r, g, b = img_array[:,:,0], img_array[:,:,1], img_array[:,:,2]
    if np.array_equal(r, g) and np.array_equal(g, b):
        return True
    else:
        return False

for root1, dirs1, files1 in os.walk(dir):
        for file1 in files1:
            greyscale &= is_greyscale(os.path.join(root1, file1))

if is_greyscale:
    print("All images are greyscale.")
else:
    print("Non greyscale images found.")
```

Listing 3.5: Script for verifying that all images conform to grayscale format by checking RGB channel uniformity.

Upon executing the script, the following output is obtained:

```
All images are greyscale.
```

Listing 3.6: Script output indicating whether each image satisfies the grayscale condition based on RGB channel equality.

3.3.2.3. Unique Identifier Generation

As observed in the previous verification step, several files within the dataset share identical filenames despite residing in different directories. This naming collision can lead to confusion or unintended overwriting when processing the dataset with external tools that do not preserve the original directory hierarchy.

To mitigate this risk and ensure unambiguous identification of each sample, a strategy was implemented to assign a unique identifier (UID) to every file. These UIDs can serve as a reliable reference when integrating the dataset into pipelines where file names are used as primary keys, such as in database indexing, metadata generation, or batch-processing workflows.

This approach becomes particularly advantageous when working with third-party tools or platforms—such as Roboflow or cloud-based annotation systems—that may alter file paths, flatten directory structures, or rename files during export.

The following Python script was used to generate and assign unique identifiers to the dataset files:

```python
import os
import shutil
dir = "/home/leprieto/tfm/resources/dataset/casting_data"

for root, dirs, files in os.walk(dir):
    for file in files:
        vector = root.split("/")[-3:-1]
        uid_prefix = "_".join(vector) + "_"
        shutil.move(os.path.join(root, file), os.path.join(root, uid_prefix + file))
```

Listing 3.7: Script for renaming files with unique identifiers to ensure consistent referencing and enable directory structure reconstruction.

This procedure transforms the filenames to encode directory structure information directly into the file names themselves. By doing so, each image receives a unique, self-descriptive identifier that preserves key metadata such as its original split (e.g., train, test) and class label (e.g., OK, DEF), regardless of its subsequent position in the filesystem.

The renaming strategy results in a filename format that reflects the full original path. For example:

```
casting_data/train/ok_front/ok_0_1.jpeg -> casting_data_train_ok_front_ok_0_1.jpeg
```

Listing 3.8: Mapping example showing the transformation from original file path to uniquely encoded filename.

This naming strategy ensures that, even if the files are later shuffled, merged into a single directory, or processed by tools that disrupt the original folder hierarchy, the semantic information embedded in the filename allows for accurate recovery of both the original dataset split (TRAIN, TEST) and the class label (OK, DEF). This guarantees traceability, facilitates reproducibility, and preserves the structural integrity of the dataset throughout the preprocessing and training pipeline.

3.3.3. Train/Validation/Test Split Strategy

A fundamental aspect of any supervised learning pipeline is the division of the dataset into three distinct subsets: Training, Validation, and Test. Each subset plays a specific role in the development, tuning, and final evaluation of the model. A clear and consistent separation between them is essential to ensure that performance metrics are both meaningful and generalisable.

- Training Set: This is the portion of the dataset used to fit the model's parameters. It typically represents the largest fraction of the data and serves as the foundation upon which the model learns patterns, features, and class distinctions.
- Validation Set: A separate subset used during the training process to monitor performance, fine-tune hyperparameters, and prevent overfitting. Unlike the training set, the model does not directly learn from this data, making it suitable for guiding decisions such as early stopping or model selection.
- Test Set: This final subset remains entirely unseen during both training and validation. It is used exclusively to evaluate the generalisation ability of the final model. The test set provides an estimate of real-world performance, simulating how the model would behave when deployed in production.

In the case of the dataset used in this project, the original structure provides only a split between train and test. To introduce a validation subset, a portion of the contents of the train directory is reallocated using a 4:1 ratio, where 80% of the samples are retained for training and the remaining 20% are assigned to validation. The contents of the original test folder are preserved for final model evaluation.

It is critical that this partitioning remains strictly enforced throughout the entire project lifecycle. Any form of cross-contamination—such as duplicate or near-identical samples appearing in both training and test sets—would compromise the objectivity of the evaluation process. Such leakage could lead to artificially inflated performance metrics and undermine the reliability of the model in real-world scenarios. Therefore, maintaining a clean and non-overlapping separation between subsets is essential to uphold the validity of experimental results.

3.4. Model Development

In this section, we will explore the different models proposed to solve the problem at hand. The primary goal of this thesis is not only to develop models capable of distinguishing between the functional state of two pieces but also to explore multiple solutions to this problem. This multifaceted approach is crucial for assessing various methodologies that could potentially offer superior accuracy or efficiency.

The development of several models stems from the idea that no single model can always provide the optimal solution. Therefore, the investigation of different approaches enables us to compare their performance, generalizability, and suitability for this specific task. Below, we outline the various models utilized in this study and the strategies behind their design and implementation.

- **YOLOv8 Classification:** The YOLOv8 Classification model is a pre-trained classification model that employs the widely recognized YOLO (You Only Look Once) framework in computer vision. YOLO is renowned for its ability to perform real-time object detection with high accuracy and efficiency. In this approach, we utilize a pre-trained YOLOv8 model and fine-tune it on a dataset of piece images. This enables the model to generalize and learn the distinguishing features of the piece's functional state. The advantage of using a pre-trained model lies in leveraging previously learned features, significantly reducing the computational burden and accelerating the training process.

- **YOLOv8 Segmentation:** YOLOv8 Segmentation involves the use of a pre-trained segmentation model to label pixels within an image. Unlike classification, segmentation requires the model to assign a label to every individual pixel in the image, helping to distinguish different regions. In this specific case, segmentation is used to differentiate between the part of the image representing the piece and the surrounding background. The model will be trained using a semi-supervised learning technique, which is essential due to the lack of fully labelled data. This approach helps the model learn useful features from a smaller set of labelled data while leveraging the vast amounts of unlabelled data available.

- **Keras/TensorFlow Classification:** This model involves the creation of a classification model using the Keras framework, with the backend powered by TensorFlow. Unlike the YOLO-based models, this model will not be pre-trained and will be built from scratch. While the complexity is considerably lower compared to YOLO, this approach allows for a more customizable architecture tailored specifically to the problem. The trade-off, however, is the greater computational cost in terms of training time and resource consumption, especially when dealing with larger datasets.

- **YOLOv8 Segmentation (Crop) + Keras/TensorFlow Classification:** This approach combines the segmentation capabilities of YOLOv8 with the classification power of Keras/TensorFlow. The idea is to use the YOLOv8 segmentation model to isolate the region of interest—the piece—by removing background elements. Once the segmentation model identifies and crops the relevant part of the image, the Keras/TensorFlow classification model is then tasked with focusing on the remaining image, which now solely contains the piece. This dual-step process enhances the accuracy of the classification model by eliminating irrelevant information and ensuring that the model is focused on the key features of the piece.

- **YOLOv8 Segmentation (Backbone) + Keras/TensorFlow Classification + Fine-Tuning:** In this more advanced approach, the backbone of the YOLOv8 segmentation model is extracted and used as a feature extractor for classification tasks. The backbone is then combined with a classification head built in Keras/TensorFlow. This method applies transfer learning, capitalizing on the YOLOv8 model's pre-trained ability to focus on relevant features of the piece. Once this transfer learning is applied, fine-tuning is conducted to optimize the model for the specific dataset and task at hand. Fine-tuning allows the model to adjust its weights in a way that improves its performance on the task of classifying the functional state of the piece.

Throughout this section, we will not only examine the development and training processes of these models but also review key theoretical concepts, such as the semi-supervised learning paradigm. Additionally, we will explore important elements relevant to this stage, including data preprocessing, augmentation, and the challenges inherent in training models with limited labelled data. This holistic approach will provide a comprehensive understanding of the model development process and its theoretical underpinnings.

3.4.1. YOLOv8 Classification

This initial model involves utilizing the You Only Look Once (YOLO) framework to train a pre-trained classification model. Employing a well-established framework like YOLO provides several benefits, including the ability to rapidly develop a functional model with relatively low development overhead while still achieving impressive performance. The primary advantage of using such frameworks lies in their built-in class balancing and data augmentation techniques during training. This significantly reduces the need for manual preprocessing, resulting in simpler, more efficient solutions with reduced complexity.

One of the key strengths of YOLO is its default application of data augmentation strategies, which help improve the model's generalization capabilities. These augmentations include techniques such as Mosaic, HSV shift, random flipping, scaling, rotation, and cropping. These transformations allow the model to learn more robust features and handle variations in input data without requiring extensive customization of preprocessing pipelines.

Below is a detailed explanation of the code and the specific functionalities applied in this approach.

3.4.1.1. Dependencies

For this experiment, we will include the following dependencies:

- Ultralytics: This library allows us to access the YOLO framework, providing a straightforward interface for model training and inference. It simplifies the implementation process by offering a ready-made environment for training YOLO models on custom datasets.
- Scikit-learn: A powerful library that provides a range of useful tools for evaluating the performance of machine learning models. In particular, Scikit-learn will be utilized for validating the results of our YOLO-based model using metrics such as accuracy, precision, recall, and F1 score.

By leveraging these libraries, we can focus on fine-tuning the YOLOv8 model for our specific task, rather than spending time on the complexities of custom data augmentation or model evaluation. This streamlined approach enables a more efficient model development process while maintaining high accuracy and reliability in classification tasks.

3.4.1.2. Creation of Train and Val Datasets

In this step, we define a function that takes our preprocessed dataset and splits the "train" directory into two subsets: "Train" and "Val", following the 4:1 ratio discussed earlier. This ratio is crucial for ensuring a proper distribution of data for both training and validation, providing the model with sufficient data to learn from while also maintaining a separate validation set to evaluate its performance.

Once the datasets have been split, the resulting subsets are stored in the "split" folder located at the root of our Google Colab environment. This organization allows for easy access and management of the datasets during training and validation phases. By separating the training and validation data, we can ensure that the model is evaluated on data it has not seen during training, giving us a better understanding of its generalization capabilities.

The process of splitting the dataset is automated through the function, which helps maintain consistency across different runs and ensures that the datasets are divided accurately. This approach minimizes human error and allows for a more streamlined and reproducible experiment setup.

```
def split_train_val(base_dir, val_size=0.2):
    classes = [folder for folder in os.listdir(base_dir) if os.path.isdir(os.path.join(base_dir, folder))]

    train_images = []
    val_images = []

    for class_name in classes:
        class_folder = os.path.join(base_dir, class_name)
```

```
        images = [os.path.join(class_folder, f) for f in os.listdir(class_folder) if f.endswith('.jpeg')]

        if len(images) == 0:
            continue

        train, val = train_test_split(images, test_size=val_size, random_state=42)
        train_images.extend([(img, class_name) for img in train])
        val_images.extend([(img, class_name) for img in val])

    if len(train_images) == 0:
        raise ValueError("No images were selected for the training set.")
    if len(val_images) == 0:
        raise ValueError("No images were selected for the validation set.")

    train_dir = "/content/split/train"
    val_dir = "/content/split/val"

    for class_name in classes:
        os.makedirs(os.path.join(train_dir, class_name), exist_ok=True)
        os.makedirs(os.path.join(val_dir, class_name), exist_ok=True)
    for img, class_name in train_images:
        shutil.move(img, os.path.join(train_dir, class_name, os.path.basename(img)))
    for img, class_name in val_images:
        shutil.move(img, os.path.join(val_dir, class_name, os.path.basename(img)))

    return train_dir, val_dir
```

Listing 3.9: Function for splitting images into "Train" and "Val" datasets based on a predefined ratio.

3.4.1.3. Model Training

The next step is to define the training process for the model. To achieve this, we provide the dataset that was previously split into "Train" and "Val" subsets. The input images are initially 300 x 300 px, but the YOLO engine automatically adjusts the dimensions by adding a margin to ensure they conform to the nearest multiple of 32. As a result, the final image size for training becomes 320 x 320 px.

In this process, we also select the available GPU on Google Colab, specifying the device to ensure that the model training leverages the hardware acceleration for faster computation. This significantly reduces training time, enabling more extensive experimentation and quicker iteration.

Furthermore, the number of epochs and batch size can be adjusted based on the model's training progress and the computational resources at our disposal. Increasing the number of epochs allows the model to train longer, improving its ability to learn intricate patterns, while adjusting the batch size can help optimize memory usage and training stability. These hyperparameters are fine-tuned throughout the training process to find the optimal configuration for the given dataset and available hardware.

```
def train_yolo(epochs=100, batch_size=32):
    model = YOLO("yolov8n-cls.pt")
    model.train(data="/content/split", epochs=epochs, batch=batch_size, imgsz = 300, device="0")
    return model
```

Listing 3.10: Function to train the YOLO model using specified parameters, including the number of epochs, batch size, and image size, while utilizing GPU device "0" for acceleration.

3.4.1.4. Validation on New Data

In this step, we perform inference on the "Test" dataset to evaluate the model's performance on unseen data. We then generate a confusion matrix, a powerful tool that helps visualize how well the model is classifying the images by comparing the predicted labels with the true labels. This matrix provides valuable insights into the model's ability to correctly classify the different classes and identify any potential misclassifications. The confusion matrix serves as a key metric for understanding the model's strengths and weaknesses, especially in terms of false positives and false negatives, which are crucial when dealing with real-world data.

```python
def test_and_generate_matrix(model, test_dir):
    results = model.predict(os.path.join(test_dir, "ok_front")) + model.predict(os.path
    .join(test_dir, "def_front"))

    y_true = []
    y_pred = []
    inference_time = 0
    i = 0
    for result in results:

        true_label = result.path.split('/')[-2]
        y_true.append(true_label)

        predicted_class = result.probs.top1
        predicted_label = model.names[predicted_class]
        y_pred.append(predicted_label)
        if true_label != predicted_label:
            result.show()
            result.save(f"missclassification_{i}.png")

        inference_time += result.speed['inference']
        i += 1

    inference_time = np.mean(inference_time)
    print(f"Inference time: {inference_time:.2f} ms")

    cm = confusion_matrix(y_true, y_pred, labels=['def_front', 'ok_front'])
    cm_display = ConfusionMatrixDisplay(cm, display_labels=model.names.values())

    cm_display.plot(cmap=plt.cm.Blues)
    plt.show()

    report = classification_report(y_true, y_pred, target_names=model.names.values())
    print(report)
```

Listing 3.11: Function to perform inference on the "Test" dataset and generate a confusion matrix to evaluate the model's performance on unseen data, visualizing the true versus predicted labels.

3.4.2. YOLOv8 Segmentation

This second model, although not directly responsible for the final classification of the piece's state, plays a crucial intermediate role for future models. Specifically, this model will be used again in the models discussed in sections 3.4.4 and 3.4.5.

The segmentation model is tasked with labeling the pixels of the piece so that it can distinguish between the region of the image corresponding to the piece and the background. For pixel labeling, we will use the Roboflow tool, which simplifies the process of generating annotated datasets. Pixel-level labeling is a computationally expensive task and beyond the scope of this thesis if done manually, especially considering the potential size of the dataset. Therefore, an alternative method is required that allows us to train the model with a dataset that is not too large but sufficient to make full manual labeling impractical.

To address this challenge, we opt for a semi-supervised training approach. This method enables us to leverage a large volume of unlabelled data for training, which is a significant advantage when working with limited labelled datasets. Semi-supervised learning allows the model to generalize effectively without requiring large volumes of labelled data, which is often a limitation in real-world scenarios.

The workflow for creating the segmentation model under this semi-supervised paradigm is as follows. The first step will involve manual labeling of a small subset of the data to obtain an initial model, trained in a supervised fashion with these few labelled samples. This model will then serve as the foundation for further semi-supervised training, allowing it to learn from the vast amounts of unlabelled data available.

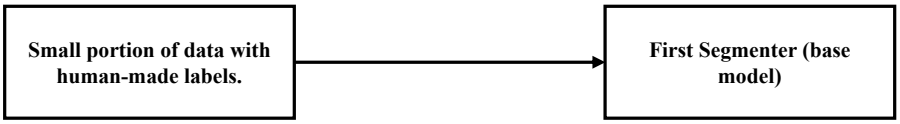


Figure 3.8: Semi-supervised workflow for training the initial segmentation model, starting with a small amount of labelled data under a supervised paradigm.

For the second step, the remaining unlabelled data will be processed by the model to perform inference and generate pseudo-labels. These pseudo-labels will act as semi-automatically generated annotations, enabling the model to continue learning from the unlabelled data. This approach allows the model to expand its understanding of the dataset without requiring manual labeling, making the most of both labelled and unlabelled data to improve its performance and generalization capabilities.

Figure 3.9: Generation of pseudo-labels from unlabelled data by applying inference with the initial classifier during the semi-supervised training process.

Finally, a manual review of the pseudo-labelled data will be conducted, and the poorest-quality labels will be discarded. The remaining high-confidence pseudo-labelled data, combined with the manually labelled data, will be used to generate a refined dataset. This dataset will then be employed to train a second segmentation model, following a supervised learning paradigm. This second model, now trained on both high-quality labelled data and pseudo-labels, will serve as the final, refined segmentation model, expected to perform with higher accuracy and reliability.

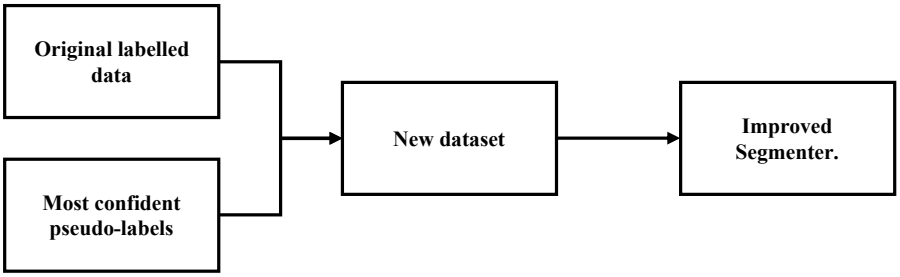


Figure 3.10: Creation of a refined training dataset by combining manually labelled data with high-confidence pseudo-labels, followed by supervised training of an improved segmentation model.

3.4.2.1. Indexing the Dataset in Roboflow

The first step involves indexing the dataset using Roboflow, a third-party tool designed to simplify the process of dataset preparation, especially for image-based tasks. To begin, we need to create an account on Roboflow, set up a new project, and upload our preprocessed training dataset. Once the dataset is uploaded, Roboflow indexes the images, allowing us to efficiently manage and track our data. This step serves as the foundation for the labeling process, enabling easy access to the images and facilitating the next stage of manual annotation.

3.4.2.2. Labeling

Once the dataset is successfully indexed, the next step is to begin the labeling process. Roboflow provides a powerful tool called the "Smart Polygon," which simplifies the task of labeling by allowing us to define polygons around the objects of interest. This tool is particularly useful for tasks like segmentation, where precise delineation of object boundaries is crucial. The Smart Polygon tool automatically adjusts the polygon's shape to fit the contours of the object, making the labeling process faster and more accurate.

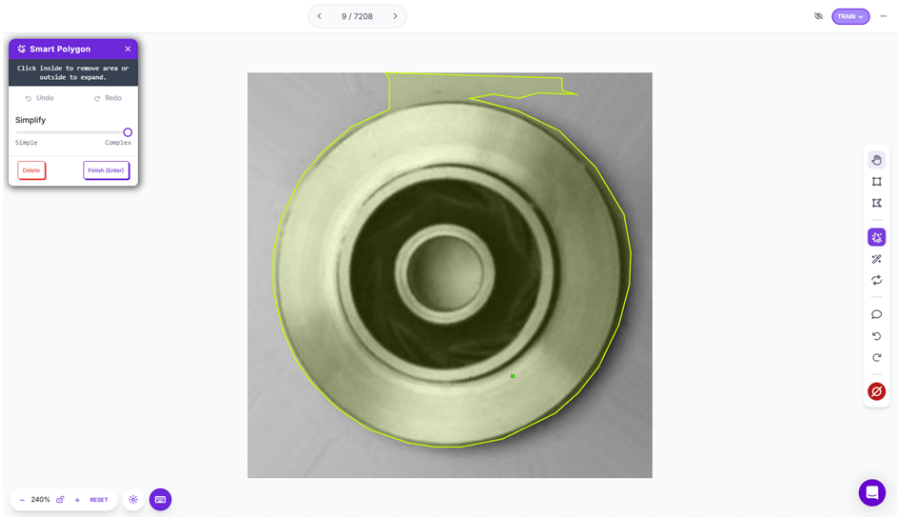


Figure 3.11: Screenshot of the Roboflow annotation interface utilizing the Smart Polygon tool for image segmentation labeling.

We can then modify the generated polygon to better align with the contours of our piece, ensuring a more accurate representation of its boundaries.

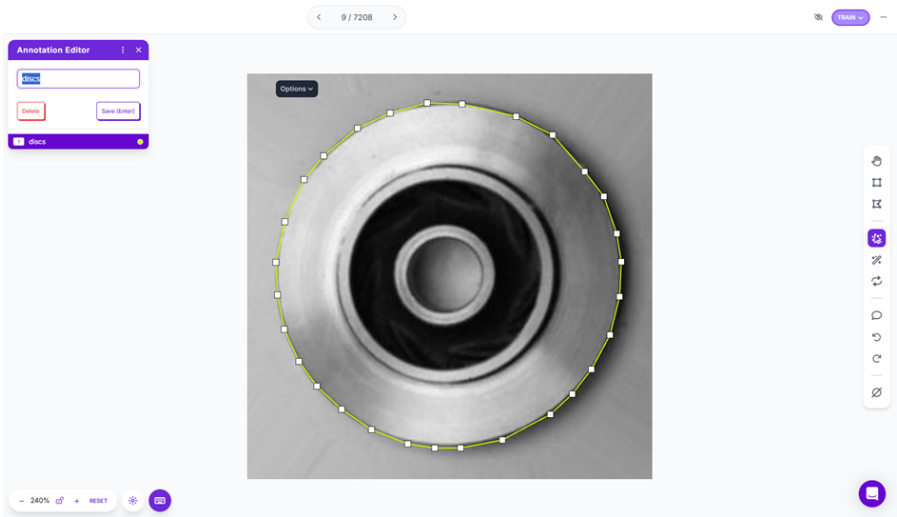


Figure 3.12: Screenshot of the Roboflow annotation interface with a segmentation mask manually refined.

Once the annotation process is completed, the labeling is confirmed and the dataset is saved within the platform.

For this initial stage, a total of 100 images are manually annotated. These annotations serve as the foundational ground truth for training the first instance of the segmentation model. The manual labeling process ensures high-quality and accurate masks, which are essential for the model to learn to differentiate between defective and non-defective regions within the metallurgical components.

This preliminary dataset, although limited in size, allows for the first experimental evaluation of the segmentation pipeline. The objective at this stage is not to achieve optimal performance, but rather to validate the feasibility of the model, identify early training dynamics, and verify that the labelled samples reflect the relevant defect patterns with sufficient precision and consistency.

As the project progresses, this dataset will be incrementally expanded and refined based on model feedback and error analysis, enabling progressive improvement through iterative annotation and re-training cycles.

3.4.2.3. Dataset Generation

Once the image labeling process is completed, the corresponding tool is used to generate the dataset. The labelled data is randomly split into the standard subsets: train, validation, and test. This partitioning ensures a representative distribution of the classes across each subset, thus enabling a robust evaluation of the model's performance.

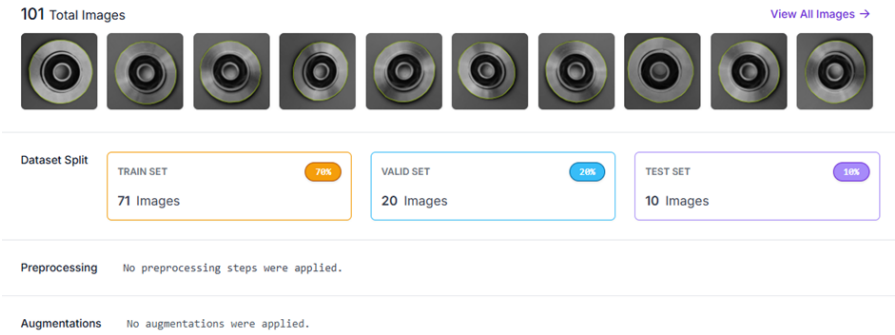


Figure 3.13: Screenshot of Roboflow interface illustrating the process of splitting and generating the dataset.

In this case, no pre-processing or data augmentation tasks are performed, as these operations are delegated to the YOLO framework, which will be employed in subsequent stages of the workflow.

3.4.2.4. Importing the Dataset into Google Colab

Using the Roboflow API, the previously created dataset is downloaded and prepared for use in the training process. This integration streamlines the workflow and facilitates direct interaction between the dataset and the model development environment within Google Colab.

```
rf = Roboflow(api_key=${SECRET})
project = rf.workspace("tfm-ks6ji").project("disc-detection-v2-zqrln")
version = project.version(1)
dataset = version.download("yolov8")
```

Listing 3.12: Roboflow API integration for programmatic dataset download using Python.

3.4.2.5. Training the Initial Model

With the dataset successfully downloaded and prepared, the training process for the first model is initiated. This step follows the same procedural framework described in the previous section, leveraging the capabilities of the YOLOv8 architecture. The model is trained on the designated train subset while its performance is periodically validated against the validation set. Throughout the training phase, key metrics such as loss, accuracy, and learning rate are monitored to assess convergence and guide potential adjustments in hyperparameters.

```
model = YOLO("yolov8n-seg.pt")
result = model.train(
    data = os.path.join(root_dir, "00_labeled_dataset", "data.yaml"),
    epochs = 100,
    imgsz = 300,
    batch = 32,
    name = "weakly_supervised_disk",
    device = "0",
)
```

Listing 3.13: Example of training a YOLO segmentation model in Python using a dataset downloaded via the Roboflow API.

3.4.2.6. Model Export and Deployment

Once the model has been successfully trained, it can be exported and deployed for further use. Roboflow provides built-in functionality to host and serve the model directly through its platform, enabling rapid deployment and testing within a production-like environment. Additionally, the trained model can be exported and stored locally in various formats (e.g., ONNX, TorchScript, CoreML), allowing for integration with custom inference pipelines or deployment on edge devices.

3.4.2.7. Annotation Verification and Inference

After training the initial model, the next step involves using it to perform inference and generate pseudo-labels for the second dataset. To facilitate this process, a dedicated Python notebook is prepared, serving as the core environment for executing the inference pipeline. Within this notebook, a custom function is defined to verify whether each file in the dataset has already been manually labelled. If a label is detected, the image is excluded from the inference process to prevent overwriting validated annotations. The resulting combined dataset is thus composed of two distinct subsets: manually labelled images, which serve as high-quality ground truth, and automatically inferred labels, which extend the dataset size and enhance the model's learning capacity for subsequent training cycles.

```
def labeled(file):
  for labeled_root, dirs, labeled_files in os.walk(target_folder):
    for labeled_file in labeled_files:
      if file.replace(".", "_") in labeled_file.replace("cast_ok_cast_ok", "cast_ok").
    replace("cast_def_cast_def", "cast_def"):
        return True
  return False
```

Listing 3.14: Python function to identify whether an image has been manually annotated.

The next step involves using the previously trained model to perform inference on the new, unlabelled data. The output predictions generated by this model serve as the basis for constructing pseudo-labels, which are then integrated into the combined dataset. This process augments the original manually labelled data with additional annotations derived from the model's predictions, enabling the creation of a richer and more diverse training set for subsequent model iterations or fine-tuning.

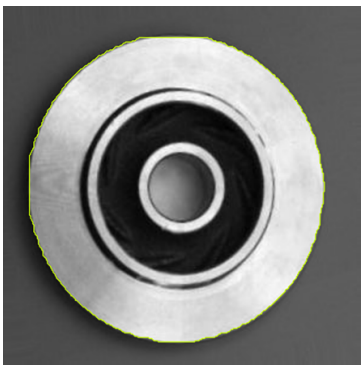
```
dest_images = os.path.join(target_folder, "train", "images")
dest_labels = os.path.join(target_folder, "train", "labels")
n = 0
infered_files = 0
inference_time = 0
for root, dirs, files in os.walk("00_clean_dataset/dataset/casting_data"):
  for file in files:
    n += 1
    if not labeled(file):
      infered_files += 1
      result = clean_model.predict(os.path.join(root, file), conf = 0.9)[0]
      inference_time += result.speed['inference']
      if result.masks is None:
        logger.info("Total Work: {} Low confidence discard. Discarded item: {}.".format
    (n, file) )
        continue
      label = "0"
      for coords in result.masks.xyn[0]:
        label += f" {coords[0]} {coords[1]}"
      with open(os.path.join(dest_labels, file.replace(".jpeg", ".txt")), "w") as f:
        f.write(label)
      shutil.copy(os.path.join(root, file), dest_images)
      logger.info("Total Work: {} | Add pseudo label for {}".format(n, file))

    else:
      logging.info("Item already labeled, discarded item: {}.")
print("Inference time: {} ms.".format(inference_time))
print("Total work: {} | Infered files: {}.".format(n, infered_files))
```

Listing 3.15: Code snippet for merging the manually annotated dataset with the pseudo-labelled data derived from model inference.

3.4.2.8. Manual Filtering

After generating the pseudo-labelled dataset through model inference, a manual review process is carried out to ensure the quality and reliability of the annotations. This step involves visually inspecting the predicted labels and discarding any instances that do not accurately reflect the true characteristics of the images. Such supervision is essential to prevent the propagation of incorrect labels, which could negatively impact the model's performance in future training stages. By retaining only high-confidence and contextually accurate annotations, the integrity of the combined dataset is preserved, thereby reinforcing the robustness of the overall learning pipeline.



(a) Accurate segmentation mask correctly outlining the defect on the piece.

(b) Incorrect segmentation mask failing to capture the actual defect on the piece.

Figure 3.14: Comparison between a correct and an incorrect segmentation result produced by the model.

In general, defects are typically found in components where the circular geometry of the piece is distorted or where the lighting conditions during image acquisition create low-contrast boundaries between the part and the background. These factors introduce visual ambiguities that challenge the segmentation model, increasing the likelihood of inaccurate mask predictions. Such cases underscore the importance of both proper image capture conditions and robust model generalization to handle real-world variability in appearance.

3.4.2.9. Model Training

This final step consists of conducting a training process analogous to that of the initial model. Once again, a supervised learning paradigm is adopted, utilizing the newly constructed and manually verified dataset. The updated dataset—comprising both original manually labelled images and high-confidence pseudo-labels that have passed manual validation—is downloaded and used to retrain the segmentation model. The objective at this stage is to leverage the increased dataset volume and improved label quality to enhance the model's generalization capabilities and segmentation accuracy. All training parameters and evaluation protocols are kept consistent with the initial training phase to ensure comparability of results.

```
rf = Roboflow(api_key=${SECRET})
project = rf.workspace("tfm-ks6ji").project("disc-detection-pseudo-v2")
version = project.version(1)
dataset = version.download("yolov8")
```

Listing 3.16: Roboflow API integration for programmatic dataset download using Python.

At this stage, the model is trained on the expanded dataset that incorporates both manually annotated samples and high-confidence pseudo-labels generated through the inference process. This training phase aims to exploit the additional data volume to improve the model's learning capacity. The inclusion of pseudo-labelled data allows the model to generalize better across a broader range of scenarios, while manual filtering ensures that the quality of the training set remains sufficiently high to avoid degrading performance.

```
model = YOLO("yolov8n-seg.pt")
result = model.train(
    data = os.path.join(root_dir, "00_pseudo_labeled_dataset", "data.yaml"),
    epochs = 100,
    imgsz = 300,
    batch = 32,
```

```
    name = "weakly_supervised_pseudo",
    device = "0",
)
```

Listing 3.17: Training of the segmentation model using the pseudo-labelled dataset.

Finally, the trained model is deployed to the Roboflow platform and also stored locally for future use.

3.4.3. Keras/TensorFlow Classification

The next model to be trained is a classification model implemented using the TensorFlow backend and the integrated Keras API.

The primary objective of this model is to develop a fully self-contained solution, independent of pre-trained architectures such as YOLO, thereby gaining full control over the model design, training process, and deployment strategy. The focus is placed on simplicity and interpretability, while still achieving robust performance in classifying the relevant parts based on defect presence or category.

Exploring this type of custom architecture is particularly valuable in industrial contexts where lightweight and efficient models are required for deployment on embedded systems. TensorFlow, through its TensorFlow Lite format, offers native support for optimized execution on edge devices with constrained computational resources. This capability makes it an ideal choice for real-time inference in manufacturing environments, where latency, reliability, and hardware limitations are critical factors.

The following section provides a detailed explanation of the methodology used to design, train, and evaluate the Keras/TensorFlow classification model.

3.4.3.1. Dataset Loading

The first step in the classification workflow involves loading the preprocessed dataset into memory. This dataset—previously curated and cleaned—is consistently used across all classification experiments to ensure comparability of results. To facilitate this process, a utility function provided by the Keras API is employed. This function enables the efficient loading of image data from a directory structure while automatically partitioning it into training and validation subsets based on a specified split ratio.

This approach not only simplifies dataset handling but also ensures reproducibility, as the splitting mechanism can be controlled using a fixed random seed. Additionally, the function supports automatic labeling based on directory names, resizing of input images to a uniform shape, and shuffling, thereby streamlining the preparation of data for model training.

```
train_dataset, val_dataset = image_dataset_from_directory(
    DATA_DIR,
    validation_split=0.2,
    subset="both",
    seed=42,
    image_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    label_mode='binary'
)
```

Listing 3.18: Loading and partitioning the dataset into training and validation subsets using Keras image_dataset_from_directory utility, with binary labels and predefined split ratio.

3.4.3.2. Data Augmentation

In contrast to the previous approach employed during the training of YOLO models—where augmentation was handled internally by the framework—in this case, we explicitly introduce a data augmentation layer into the model pipeline. This augmentation layer is implemented using Keras' preprocessing utilities and is integrated directly into the training workflow.

The purpose of this step is to artificially increase the diversity of the training data by applying random but

realistic transformations to the input images. These transformations may include operations such as horizontal flipping, rotation, zooming, translation, and brightness adjustments. By exposing the model to a wider range of visual variations, the augmentation layer helps improve its generalization capabilities and robustness to real-world scenarios where lighting, orientation, or positioning of parts may vary.

```
data_augmentation = tf.keras.Sequential([
    layers.RandomFlip("horizontal"),
    layers.RandomRotation(0.1),
    layers.RandomContrast(0.1)
])
```

Listing 3.19: Definition of a data augmentation pipeline using Keras, applying random horizontal flips, slight rotations, and contrast adjustments to enhance model generalization.

3.4.3.3. Normalization

Following the data augmentation stage, input normalization is performed as a critical preprocessing step. Most images used in computer vision tasks are represented in the RGB colour space and consist of three channels—red, green, and blue—each with pixel intensity values ranging from 0 to 255. These values, derived from the standard 8-bit encoding of image data, must be rescaled to a consistent numerical range to facilitate effective training.

To normalize the input data, all pixel values are divided by 255, thus mapping the intensity values to the [0, 1] range. This transformation ensures numerical stability during training, accelerates convergence, and helps avoid issues related to vanishing or exploding gradients. Moreover, it brings the input data into alignment with the assumptions made by many deep learning optimizers and activation functions.

```
preprocess = tf.keras.Sequential([
    layers.Rescaling(1./255)
])
```

Listing 3.20: Definition of a preprocessing layer using Keras to normalize input images by rescaling pixel values to the 0 - 1 range.

3.4.3.4. Model Definition

The next step involves defining the architecture of the classification model using the Keras API, a high-level interface built on top of TensorFlow that simplifies the creation, training, and evaluation of deep learning models. The model is constructed as a sequential stack of layers, enabling a clear and modular design that is easy to interpret and modify.

This definition includes the integration of previously established preprocessing components, such as data augmentation and normalization layers, followed by a series of convolutional, pooling, and fully connected layers designed to extract meaningful features and perform binary classification. The architecture is tailored to strike a balance between computational efficiency and representational capacity, making it suitable for deployment on resource-constrained environments such as embedded systems.

```
model = models.Sequential([
  layers.Input(shape=IMG_SIZE + (3,)),
    data_augmentation,
    preprocess,

    layers.Conv2D(32, 3, activation='relu'),
    layers.MaxPooling2D(),

    layers.Conv2D(64, 3, activation='relu'),
    layers.MaxPooling2D(),

    layers.Conv2D(128, 3, activation='relu'),
    layers.MaxPooling2D(),

    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(1, activation='sigmoid')
```

```
])
```

Listing 3.21: Definition of a sequential Keras classification model incorporating data augmentation, normalization, multiple convolutional and pooling layers, and a fully connected output for binary classification.

The architecture of the convolutional neural network is composed of the following layers:

- **Input Layer:** This layer defines the input shape of the model and must match the format of the incoming image data. In this case, it expects three channels (RGB), even though the original dataset is in grayscale. This decision was made to ensure compatibility with standard image formats and to allow greater flexibility for future applications where RGB images might be used. Adopting a universal input shape enhances the model's reusability and simplifies potential integration with other datasets or systems.

- **Data Augmentation Layer:** This layer applies real-time, randomized transformations to the input images during training, including horizontal flipping, rotation, and contrast adjustment. As previously defined, this technique increases data diversity and helps the model generalize better to unseen examples.

- **Preprocessing Layer:** The normalization layer rescales pixel intensity values from the original [0, 255] range to [0, 1], ensuring numerical stability during training and improving convergence behavior.

- **Convolutional and Pooling Blocks:** These core layers are responsible for hierarchical feature extraction:

  - **Conv2D Layers:** Convolutional layers detect local patterns in the input images, such as edges, textures, and shapes. The number of filters increases progressively ($32 \rightarrow 64 \rightarrow 128$), allowing the model to learn increasingly complex and abstract features at deeper levels of the network. The kernel size is set to 3×3, a commonly used configuration that balances computational cost with the ability to capture spatial patterns effectively. The activation function used is ReLU (Rectified Linear Unit), which introduces non-linearity and enables the model to learn complex decision boundaries.

  - **MaxPooling2D Layers:** Each convolutional block is followed by a pooling layer, which reduces the spatial dimensions of the feature maps. This operation decreases the number of parameters, lowers computational cost, and retains the most salient features, contributing to translational invariance.

- **Flatten Layer:** The flattening operation converts the two-dimensional feature maps into a one-dimensional vector. This transformation is necessary to transition from the convolutional base to the fully connected classification head.

- **Dense Layer:** A fully connected layer with 512 neurons that aggregates and interprets the extracted features. This layer learns global patterns and relationships relevant to the classification task.

- **Dropout Layer:** A dropout rate of 50% is applied during training to randomly deactivate half of the neurons. This technique serves as a form of regularization, preventing overfitting by reducing the model's reliance on specific activations and promoting redundancy in learned representations.

- **Output Layer:** The final dense layer consists of a single neuron with a sigmoid activation function. It outputs a value between 0 and 1, representing the probability that the input image belongs to the positive class. This configuration is well-suited for binary classification tasks, such as identifying whether a part is defective or not.

3.4.3.5. Class Balance

An important consideration—often overlooked during the training of YOLO models due to the nature of object detection frameworks—is the balance between classes in classification tasks. In the context of supervised learning, an imbalanced dataset can significantly bias the model toward the majority class, leading to poor generalization and suboptimal performance, particularly in identifying minority class instances.

For example, if the dataset contains 500 images labelled as “True” (e.g., defective parts) and 1,000 images labelled as “False” (e.g., non-defective parts), a naive model may develop a predisposition to classify new samples as “False.” This imbalance could result in a higher number of false negatives, where actual defects are incorrectly classified as normal, posing serious risks in industrial applications.

To mitigate this issue, it is essential to address class imbalance when training classification models using TensorFlow/Keras. One effective approach involves computing class weights that inversely correlate with the frequency of each class. These weights are then passed to the training function, encouraging the model to treat both classes with equal importance, regardless of their distribution in the dataset.

To compute the class weights, we utilize the `compute_class_weight` utility from the scikit-learn library, as previously discussed. This method ensures that the loss function penalizes errors in the minority class more heavily, thereby guiding the model to learn a more balanced decision boundary.

```
labels = np.concatenate([y for x, y in train_dataset], axis=0)
class_weights = compute_class_weight(
    'balanced',
    classes=np.unique(labels),
    y=labels.flatten()
)
class_weights = {i: w for i, w in enumerate(class_weights)}
```

Listing 3.22: Computation of class weights using `scikit-learn` to address class imbalance in the training dataset, ensuring equal importance is given to both classes during model training.

These weights are subsequently provided to the model during the training phase, where they are integrated into the loss function. This adjustment ensures that misclassifications of the minority class are penalized more heavily, thereby guiding the model to treat both classes with proportional significance. Incorporating class weights is a widely adopted strategy in imbalanced classification scenarios, and it is particularly critical in industrial applications where the cost of false negatives—such as undetected defects—can be substantial.

3.4.3.6. Model Compilation

Once the model architecture is defined, the next step is to compile it by specifying the optimizer, loss function, and evaluation metrics—each of which is selected based on the nature of the task and the characteristics of the dataset.

```
model.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['accuracy']
)
```

Listing 3.23: Compilation of the Keras model using the Adam optimizer, binary crossentropy loss, and accuracy, precision, and recall as evaluation metrics for binary classification.

- **Optimizer – Adaptive Moment Estimation (Adam):** The Adam optimizer is chosen for its efficiency and robustness. It combines the advantages of both AdaGrad and RMSProp by adaptively adjusting the learning rate of each parameter based on estimates of first and second moments of the gradients. This dynamic adjustment accelerates convergence and makes Adam particularly well-suited for training on noisy or sparse datasets. Due to its solid performance across a wide range of deep learning tasks, Adam has become the default choice in many applications, including image classification.
- **Loss Function – Binary Crossentropy:** Since the task at hand is a binary classification problem, the binary crossentropy loss function is employed. It measures the dissimilarity between the predicted probability (ranging from 0 to 1) and the actual binary label. This loss function heavily penalizes predictions that are both incorrect and confident, thus encouraging the model to produce calibrated probabilities that reflect true classification uncertainty.
- **Evaluation Metrics:** During training and validation, accuracy is tracked as the primary evaluation metric. This allows for real-time monitoring of the model’s performance and provides a straightforward indication of how well the model distinguishes between defective and non-defective parts. Additional metrics such as precision, recall, or F1-score may be incorporated in future iterations to provide a more nuanced assessment, particularly in cases of class imbalance.

3.4.3.7. Callbacks

Callbacks in Keras provide a powerful mechanism to execute specific operations at various stages of the training process, particularly at the end of each epoch. They are essential tools for implementing behaviors such as early termination, dynamic learning rate adjustments, model checkpointing, and real-time monitoring. This modularity is one of the key strengths of the Keras API and contributes significantly to the flexibility and control of the training workflow.

For this experiment, a Model Checkpoint callback is configured.

```
callbacks = [
    keras.callbacks.ModelCheckpoint("model.weights.h5",
                                    monitor='val_loss',
                                    save_best_only=True,
                                    save_weights_only=True),
]
```

Listing 3.24: Configuration of Keras callbacks for model checkpointing to save the best-performing model during training.

The ModelCheckpoint callback enables automatic saving of the model during training whenever an improvement is detected. The `save_best_only=True` parameter ensures that only the best-performing model, according to the monitored metric, is stored. In this case, only the model weights are saved, for later on recovery. This feature acts as both a safeguard and a convenience, ensuring that the optimal version of the model is always preserved.

3.4.3.8. Model Training

Once the model architecture, compilation settings, class weights, and callbacks have been properly configured, the training process is initiated. The training is conducted using the prepared training and validation datasets over a defined number of epochs. During this stage, the model is exposed to the input data in batches, adjusting its internal parameters to minimize the binary crossentropy loss function.

The `class_weight` argument ensures that the imbalance between classes is appropriately addressed, allowing the model to give proportionally more attention to the underrepresented class. Additionally, the `callbacks` argument includes the previously defined ModelCheckpoint, which provide dynamic control over the training process and ensure retention of the best-performing model.

```
history = model.fit(
    train_dataset,
    validation_data=val_dataset,
    epochs=EPOCHS,
    class_weight=class_weights,
    callbacks=callbacks
)
```

Listing 3.25: Training of the custom Keras classification model with balanced class weights and integrated callbacks for model checkpointing.

This training strategy results in a lightweight and fully custom classification model, designed from scratch without reliance on pretrained architectures. Such independence enhances interpretability and control, while also making the model more adaptable to deployment-specific constraints. Furthermore, due to its simplicity and reduced number of parameters, the model is well-suited for compression and conversion to formats such as TensorFlow Lite, facilitating deployment on resource-constrained or embedded systems—an essential requirement in many industrial inspection scenarios.

3.4.4. YOLOv8 Segmentation (Crop) + Keras/TensorFlow Classification

An alternative and potentially more effective configuration involves combining segmentation and classification models in a sequential processing pipeline. In this approach, the previously developed YOLOv8 segmentation model is repurposed as a preprocessing tool to refine the input data before classification.

The underlying motivation is to simplify the classification task by isolating and highlighting the region of interest within each image. Specifically, the segmentation model is used to detect and localize the metal disc

in each sample. Once detected, a cropping operation is applied to remove the background and retain only the segmented region. This procedure significantly reduces the visual complexity of the input data, allowing the classification model to focus exclusively on the structural patterns and surface features of the disc itself.

By narrowing the model's attention to the most relevant spatial area, this preprocessing step is expected to enhance the classifier's accuracy and robustness, particularly in scenarios where irrelevant background elements or inconsistent lighting conditions might otherwise introduce noise. Moreover, this method introduces a modular and reusable architecture: the segmentation model performs spatial localization, while the classification model handles the defect prediction task.

This combined strategy not only improves the interpretability and efficiency of the overall pipeline but also aligns with best practices in computer vision, where task decomposition often leads to more reliable and maintainable systems.

(a) Original image with background preserved, containing both the region of interest and irrelevant visual context.

(b) Preprocessed image after segmentation-based cropping, isolating the region of interest.

Figure 3.15: Comparison between the original input and the preprocessed image produced by the YOLOv8 segmentation model. The preprocessing step eliminates background noise, enabling the subsequent classification model to focus solely on the relevant features of the metal disc.

3.4.4.1. Preprocessing

The initial step in this pipeline involves recovering both the cleaned dataset and the trained YOLOv8 segmentation model previously developed using a semi-supervised learning approach. These resources are essential to perform the background removal operation prior to classification.

To begin, the dataset is downloaded and extracted into the local environment using secure file transfer commands. Similarly, the trained segmentation model is retrieved and its weights are loaded using the Ultralytics YOLOv8 interface:

```
!curl -u leprieto:${SECRET} -o 00_dataset.zip "https://cloud.leonprieto.com/remote.php/dav/files/leprieto/University/UEM/TFM/resources/00_clean_filtered_dataset/dataset.zip"
!unzip -q 00_dataset.zip -d 00_dataset
!rm 00_dataset.zip

!curl -u leprieto:${SECRET} -o 01_pseudo_model.zip "https://cloud.leonprieto.com/remote.php/dav/files/leprieto/University/UEM/TFM/resources/jupyter/02_pseudo_model.zip"
!unzip -q 01_pseudo_model.zip -d 01_pseudo_model
!rm 01_pseudo_model.zip
```

Listing 3.26: Download and extraction of the preprocessed dataset and the trained YOLOv8 segmentation model.

Once the resources are available, the segmentation model is instantiated:

```
model = YOLO('01_pseudo_model/weights/best.pt')
```

Listing 3.27: Instantiation of the YOLOv8 segmentation model using the previously trained weights.

With the model loaded, a directory traversal is performed over the dataset. For each image, an inference is executed to obtain the segmentation mask corresponding to the region of interest. This mask is used to isolate the metal disc by removing the background and replacing it with a uniform white canvas. This preprocessed output is then stored in a parallel directory structure, mirroring the original dataset layout:

```
for root, dirs, files in os.walk(root_dataset):
    rel_dir = os.path.relpath(root, root_dataset)
    for file in files:
        image_path = os.path.join(root, file)
        image = cv2.imread(str(image_path))
        if image is None:
            print(f"Could not read {image_path}")
            continue

        results = model.predict(source=image, task='segment')

        if not results or not results[0].masks:
            print(f"No mask found in {image_path}")
            continue

        mask = results[0].masks.data[0].cpu().numpy()
        mask = (mask * 255).astype(np.uint8)

        if mask.shape != image.shape[:2]:
            mask = cv2.resize(mask, (image.shape[1], image.shape[0]))

        mask_3c = cv2.merge([mask, mask, mask])
        white_bg = np.full_like(image, fill_value=255)
        masked_image = np.where(mask_3c == 255, image, white_bg)

        save_path = os.path.join(root_croped_dataset, rel_dir)
        os.makedirs(save_path, exist_ok=True)
        save_path = os.path.join(save_path, file)
        cv2.imwrite(str(save_path), masked_image)
        print(f"Saved masked image to {save_path}")
```

Listing 3.28: Iterative application of the segmentation model to each image for background removal. The region of interest is preserved while the surrounding area is replaced with a white background.

This process generates a new version of the dataset in which all backgrounds have been removed and replaced with a uniform white background, ensuring that the classification model will focus exclusively on the disc's structural features. This preprocessing step enhances both model interpretability and consistency across samples.

3.4.4.2. Manual Postprocessing

Following the automated inference and background removal performed by the YOLOv8 segmentation model, a manual postprocessing step is conducted to validate the quality of the resulting dataset. Although the segmentation model performs reliably in most cases, certain images may exhibit suboptimal results—such as incomplete masks, misaligned boundaries, or missing parts of the object of interest—due to variations in lighting, occlusion, or image artifacts.

This manual inspection ensures consistency with the quality control standards established during the earlier stages of model development, particularly those applied when building the segmentation model for the initial training pipeline. The goal is to preserve only those images in which the region of interest—the metal disc—is clearly and fully represented, while discarding or correcting any samples where the automated cropping has failed.

Figure 3.16: Example of an image with erroneous preprocessing due to inaccurate segmentation.

Once this curated dataset with a white background has been finalized, the classification pipeline is re-executed, following the same procedure as described previously. This includes data loading, augmentation, normalization, model definition, class balancing, compilation, and training. By reusing the previously established architecture and configuration, the experimental conditions remain consistent, allowing for a direct comparison between the baseline model trained on unprocessed images and the one trained on segmentation-refined inputs.

3.4.4.3. Model Training

After completing the manual postprocessing phase and ensuring the quality of the refined dataset, the next step involves training a classification model using TensorFlow and Keras. The architecture and training configuration used in this stage are identical to those previously established in the earlier section. This deliberate reuse of the same model design ensures a controlled experimental setup, allowing for a meaningful comparison between models trained on unprocessed images and those trained on segmentation-refined inputs.

The training dataset now consists exclusively of images where the background has been removed and replaced with a uniform white canvas, as produced by the segmentation model. This focused input allows the classifier to concentrate solely on the structural and surface-level patterns of the metal disc, potentially improving classification accuracy by eliminating irrelevant contextual information.

All aspects of the training pipeline are preserved: data loading via the Keras API, data augmentation to enhance variability, normalization of pixel intensities, use of class weights to address class imbalance, and the integration of callbacks such as EarlyStopping and ModelCheckpoint. By maintaining this consistency, the experiment isolates the effect of the segmentation-based preprocessing on model performance.

This setup is particularly valuable for deployment in industrial environments, where robust defect detection must operate on high-quality, tightly controlled inputs under varying real-world conditions.

3.4.5. YOLOv8 Segmentation (Backbone) + Keras/TensorFlow Classification + Fine-Tuning

An additional strategy for training a classification model involves leveraging transfer learning techniques, which allow for the reuse of features learned by a pre-trained model on a related task. Transfer learning has become a cornerstone of modern deep learning, particularly in scenarios where labelled data is limited or where training from scratch would be computationally expensive and time-consuming.

In this experiment, the approach focuses on extracting the backbone of the previously trained YOLOv8 segmentation model. The term backbone refers to the lower layers of a deep neural network responsible for extracting general-purpose features such as edges, textures, and shapes from input images. These layers typically capture hierarchical and spatially invariant representations that are transferable across various computer vision tasks.

By isolating the backbone of the YOLOv8 segmentation model—already trained to identify regions of interest on metal parts—the goal is to capitalize on its learned representations. A new classification head is then appended to this backbone, enabling the model to perform binary classification of defect presence, while building upon the rich, task-relevant features previously learned during segmentation.

Once the new classification head has been attached, a fine-tuning phase is performed. This involves selectively unfreezing the final layers of the segmentation backbone, allowing them to be retrained alongside the new classification layers. Fine-tuning enables the model to adapt the more abstract and task-specific features to the classification problem at hand, improving performance on subtle, domain-specific variations that may not have been fully captured by the original segmentation training.

This hybrid approach balances the efficiency of transfer learning with the adaptability of end-to-end optimization and is particularly effective in industrial inspection tasks, where models must generalize to nuanced defect patterns under constrained data availability.

3.4.5.1. Backbone Extraction

The first step in implementing transfer learning from the YOLOv8 segmentation model involves extracting its backbone, which serves as the foundational feature extractor within the network. This operation is essential for repurposing the model for classification tasks, as it enables the reuse of low-level and mid-level representations that the model has already learned during its segmentation training.

To carry out this extraction, it is necessary to identify the specific layers of the segmentation model that correspond to the backbone. In the YOLOv8 architecture, the model is organized into three primary components: the backbone, the neck, and the head. The backbone is responsible for hierarchical feature extraction, the neck further refines and fuses multi-scale features, and the head generates the final task-specific outputs (e.g., bounding boxes, masks, or class probabilities).

For YOLOv8 models, the backbone is generally understood to span from the input layer (layer 0) through to the SPPF (Spatial Pyramid Pooling - Fast) layer (layer 9). This segment of the architecture captures multi-scale spatial features and is well-suited for downstream tasks due to its generality and abstraction capabilities.

Once this segment is isolated, the neck and head components—originally tailored for segmentation—can be replaced or modified to support classification. Specifically, the segmentation head is removed and substituted with a new fully connected classification head appropriate for binary decision tasks.

The figure below illustrates the structure of the YOLOv8 architecture, with the backbone section highlighted. This visualization aids in the identification of key layers and facilitates precise architectural modifications for successful transfer learning.

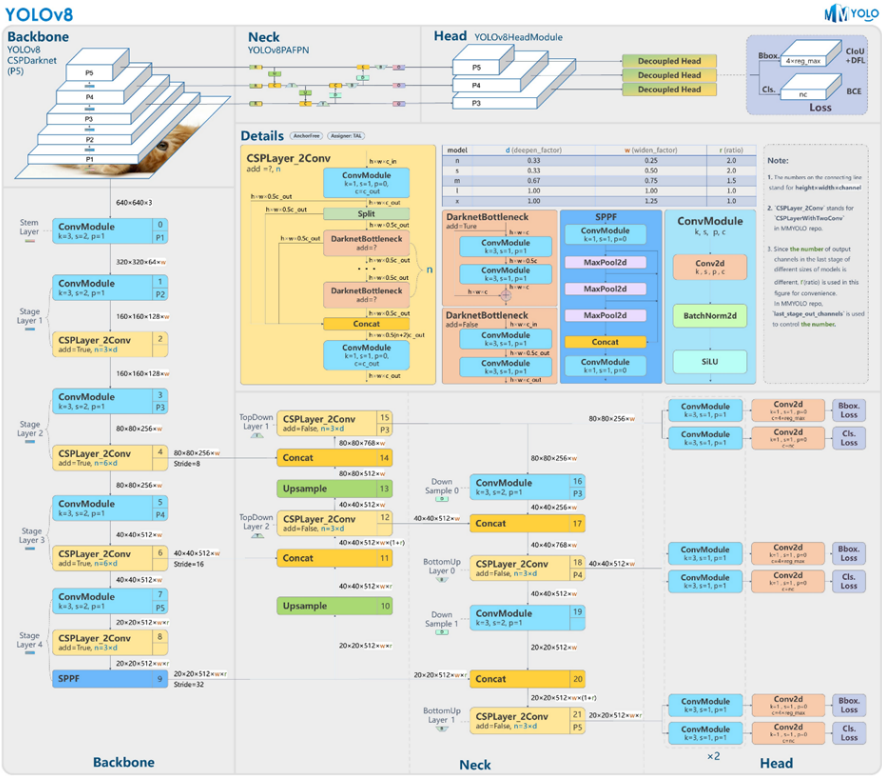


Figure 3.17: Architectural Overview of the YOLOv8 Model, Highlighting the Backbone, Neck, and Head Components. Source: OpenMMLab[54]

Once the range of layers constituting the backbone has been clearly defined, the next step involves attaching a custom classification head to complete the model. This integration enables the transformed model to shift from segmentation to classification, reusing the learned features while adapting to a new objective. To achieve this, several methodological alternatives can be explored, depending on the desired framework and interoperability constraints between PyTorch and TensorFlow/Keras.

The following approaches are considered:

- Exporting to TensorFlow via Conversion Tools: This strategy involves converting the YOLOv8 model, originally implemented in PyTorch, into a format compatible with TensorFlow. Various conversion pipelines are available:
  - Using ONNX as an Intermediary Format: Convert the model from PyTorch to ONNX, and subsequently from ONNX to TensorFlow. ONNX (Open Neural Network Exchange) serves as a widely supported interchange format, facilitating framework-agnostic model translation.
  - Direct Export from YOLO to TensorFlow-Compatible Format: Some implementations of the YOLO framework offer native export capabilities to formats such as `.h5`, which are directly usable in TensorFlow/Keras environments.
  - Third-Party Conversion Libraries: Community-developed tools such as `pytorch2keras` and `nobuco` allow for automated or semi-automated translation of PyTorch models into Keras by replicating architecture and transferring weights.
- Maintaining the Model in PyTorch: An alternative is to forgo the conversion process entirely and retain the model within the PyTorch framework. In this scenario, the classification head is appended directly to the extracted backbone using native PyTorch layers, thus preserving framework consistency and avoiding potential conversion issues.
- Manual Conversion to Keras: A more granular approach consists of iterating over the YOLOv8 architecture and reconstructing its backbone layer-by-layer in Keras. During this process, weights from the

original PyTorch model are explicitly transferred to the equivalent Keras layers. Although this method requires meticulous layer mapping and format handling, it provides full control over the final architecture and ensures compatibility with TensorFlow-based deployment pipelines.

The following section presents an analysis of the aforementioned alternatives, evaluating their practicality, technical complexity, and suitability for the intended classification task and deployment environment.

Methodology	**Pros**	**Cons**
Export to TensorFlow using conversion tools	Enables rapid model conversion using third-party libraries; minimal manual intervention required.	Potential compatibility issues with Keras 3; frequent runtime exceptions due to version mismatches; limited backward compatibility. Some converted models may not be trainable.
Change of framework	Avoids intermediate conversion steps; allows direct modifications in PyTorch while maintaining high accuracy and fidelity.	Requires use of additional backends, increasing system complexity and reducing consistency across the toolchain.
Manual conversion	Ensures precise and transparent conversion without third-party dependencies; provides full control over model architecture and parameters.	Demands deep understanding of model representations across frameworks and the internal mechanics of each backend.

Table 3.1: Comparison of model conversion methodologies

Following the comparative analysis of the available methodologies, the decision was made to proceed with manual model conversion. This approach involves reconstructing the model architecture layer by layer, translating it from one deep learning framework (in this case, PyTorch) to another (TensorFlow/Keras). Each layer must be carefully redefined using the equivalent operations and parameters in the target framework, and the corresponding pre-trained weights must be transferred with precision.

Although this method is arguably the most labor-intensive and technically demanding, it offers maximum control and flexibility over the conversion process. Unlike automated tools, manual conversion ensures full transparency, allowing for architectural modifications, fine-grained optimization, and precise debugging at each stage. This level of control is particularly advantageous when the target deployment environment imposes strict constraints or when compatibility with newer framework versions is required—scenarios where automated tools often fail or introduce silent inconsistencies.

Moreover, manual conversion enhances model interpretability and provides a deeper understanding of the internal operations, which is invaluable for fine-tuning, optimization, and long-term maintenance of the system.

3.4.5.2. Manual Conversion

The manual conversion process involves replicating the architecture and behavior of the YOLOv8 segmentation backbone from its original PyTorch implementation to an equivalent TensorFlow/Keras model. This requires constructing a precise mapping between the functional components of both frameworks, taking into account differences in tensor conventions, layer definitions, initialization methods, and model serialization formats.

To achieve this, a low-level understanding of both PyTorch and TensorFlow is essential.[55][56] The manual process not only involves translating the architecture layer-by-layer but also ensuring that the behavior of each layer—such as activation functions, normalization strategies, and weight initialization—remains consistent across frameworks.

Throughout this section, the full conversion methodology is detailed, including:

- Identification of corresponding layer types across frameworks (e.g., Conv2D in Keras vs. nn.Conv2d in PyTorch),
- Reconstruction of the backbone model architecture in TensorFlow/Keras,
- Consideration of shape conventions (channel-first vs. channel-last),
- Management of static vs. dynamic graph execution models,
- And implementation of any custom layers or operations not natively available in the target framework.

In addition to the architectural conversion, the section also addresses weight transfer, a crucial step to ensure that the pre-learned features from the original YOLOv8 segmentation training are retained. This involves extracting the weights from the PyTorch model and carefully assigning them to the corresponding layers in the Keras model, taking care to match dimensionality and data format.

By the end of the process, a fully functional Keras model is obtained, structurally and parametrically aligned with the original PyTorch backbone. This allows for seamless integration with the new classification head and supports further training or fine-tuning within the TensorFlow ecosystem.

3.4.5.2.1 Weight Transfer

A fundamental step in the manual model conversion process is the correct migration of learned weights from the source framework (PyTorch) to the target framework (TensorFlow/Keras). Preserving these pre-trained weights is essential to retain the representational power and performance achieved during the original training phase, particularly when the backbone has already been optimized through extensive segmentation tasks.

In the specific case of the YOLOv8 architecture, the backbone primarily consists of two core types of layers: convolutional layers (Conv2D) and batch normalization layers (BatchNorm2D). These form the essential building blocks of the model's feature extraction mechanism. Consequently, a successful weight transfer operation must ensure that the parameters of these layers are correctly interpreted and reassigned across frameworks.

The main considerations for this step include:

- Weight format differences: PyTorch uses a `channels-first` format (NCHW), while TensorFlow/Keras defaults to channels-last (NHWC). Therefore, convolutional weights must be transposed accordingly before assignment.
- Parameter mapping: Each `Conv2D` layer in PyTorch contains a weight tensor and optionally a bias vector. Similarly, `BatchNorm2D` layers include parameters such as `gamma` (scale), `beta` (shift), `running_mean`, and `running_var`. These must be carefully extracted from the PyTorch state dictionary and assigned to the corresponding Keras layers with matching semantics and dimensions.
- Ordering and layer alignment: The sequential nature of PyTorch modules may differ from the declarative or graph-based structure in Keras. Ensuring that weights are applied to the correct Keras layers requires a precise mapping of layer names or indices between the source and target models.

The weight transfer process not only enables continuity in the learning process but also reduces the training time required for the classification model, since the backbone already captures domain-specific features. The next section will present a detailed implementation of this weight transfer procedure, including the use of PyTorch's `state_dict()` and Keras' `set_weights()` functionalities.

3.4.5.2.1.1 Conversion of Convolutional (Conv2D) Weights

In PyTorch, the convolutional layer weights are stored using a channels-first format, following the convention:

$$W = \left(out_{ch}, \frac{in_{ch}}{groups}, k_H, k_W \right) \tag{3.1}$$

By contrast, frameworks such as TensorFlow and Keras use a channels-last format for their convolutional kernels, structured as:

$$W = \left(k_H, k_W, \frac{in_{ch}}{groups}, out_{ch}\right) \tag{3.2}$$

To ensure compatibility when transferring weights from PyTorch to TensorFlow/Keras, the weight tensor must be transposed accordingly.

Regarding the bias term, TensorFlow expects the bias to be handled separately from the convolutional weights. Therefore, if a bias is present in the PyTorch layer, it must be extracted and returned as a second element in the list of parameters. These values can then be set using the Keras `set_weights()` method, which accepts a list of arrays corresponding to kernel weights and bias (if present).

The following Python function performs this transformation, taking a PyTorch `Conv2d` layer as input and returning the corresponding TensorFlow-compatible weight tensors:

```python
def torch_conv_weights(conv: torch.nn.Conv2d):
    w = conv.weight.detach().cpu().numpy()
    w = np.transpose(w, (2, 3, 1, 0))  # (H, W, in_c/groups, out_c)
    return [w] + ([conv.bias.detach().cpu().numpy()] if conv.bias is not None else [])
```

Listing 3.29: Python function to convert PyTorch `Conv2d` weights to TensorFlow-compatible format by transposing dimensions and optionally including the bias vector.

3.4.5.2.1.2 Batch Normalization (BatchNorm2D) Weights

For the BatchNorm2D layer, PyTorch stores the parameters in a structure that includes both learnable and running (non-learnable) statistics. The order and meaning of the parameters are as follows:

$$W = \left(weight, bias, running_{mean}, running_{var}\right) \tag{3.3}$$

These correspond to the components of the batch normalization operation:

- *weight*: the scaling factor γ,
- *bias*: the shifting factor β,
- *running_mean*: the estimated mean used during inference,
- *running_var*: the estimated variance used during inference.

In TensorFlow/Keras, the batch normalization layer expects these parameters in the following order:

$$W = \left(\gamma, \beta, moving_{mean}, moving_{var}\right) \tag{3.4}$$

As the mathematical semantics are equivalent between PyTorch and TensorFlow for batch normalization, transferring these weights is a direct process—provided they are extracted and passed in the correct order.

The following Python function extracts the batch normalization weights from a PyTorch `BatchNorm2d` layer and prepares them for assignment to a corresponding TensorFlow/Keras layer:

```python
def torch_bn_weights(bn: torch.nn.BatchNorm2d):
    return [
        bn.weight.detach().cpu().numpy(),          # gamma
        bn.bias.detach().cpu().numpy(),            # beta
        bn.running_mean.detach().cpu().numpy(),    # moving mean
        bn.running_var.detach().cpu().numpy(),     # moving variance
    ]
```

Listing 3.30: Python function for extracting `BatchNorm2d` weights from a PyTorch model and formatting them for direct assignment to a Keras `BatchNormalization` layer.

3.4.5.2.2 Layer Conversion

After successfully transferring the weights from the PyTorch model, the next step involves reconstructing the network architecture by converting each layer block from the YOLOv8 model into its functional equivalent in TensorFlow/Keras.

This layer-by-layer conversion is essential to preserve the model's original behavior and ensure that the TensorFlow-based replica performs identically to its PyTorch counterpart. The process involves replicating both the computational structure and the parameter configuration of each architectural block, as well as assigning the transferred weights in the correct order and format.

In the YOLOv8 architecture, the backbone is composed of several reusable modules that serve as the fundamental building blocks of the network. The four key components requiring conversion are:

- ConvModule: A composite layer that typically includes a `Conv2D` operation followed by `BatchNormalization` and a nonlinear activation function (e.g., SiLU). This forms the most basic unit for feature extraction.
- Bottleneck: A residual-style block consisting of two sequential convolutional layers with an optional skip connection. It is designed to reduce computational cost while preserving representational capacity.
- CSPLayer_2Conv (also known as C2 or C2f): This block combines multiple bottleneck sub-blocks in a cross-stage partial (CSP) configuration, which splits the input into parallel paths, applies transformations, and then concatenates the outputs. It improves gradient flow and enhances feature reuse.
- SPPF (Spatial Pyramid Pooling – Fast): A pooling-based structure that aggregates multi-scale features using a sequence of `MaxPooling` operations with different receptive fields, enabling the network to capture contextual information at various spatial resolutions.

Each of these blocks will be individually reimplemented in Keras, ensuring that their internal operations mirror the logic used in the PyTorch-based YOLOv8 model. The converted layers are then assembled sequentially to reconstruct the backbone architecture, after which the new classification head is appended.

The following sections provide detailed implementations and explanations for each converted module, including architectural diagrams and weight assignment procedures where necessary.

3.4.5.2.2.1 ConvModule

The first architectural block to be converted from PyTorch to Keras is the ConvModule, a fundamental component in the YOLOv8 backbone architecture. This module performs basic feature extraction and is extensively reused across the network.

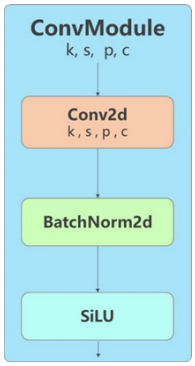


Figure 3.18: Architectural Overview of the ConvModule. Source: OpenMMLab[54]

The ConvModule typically consists of the following three layers:

- Conv2D: Performs spatial convolution over the input image or feature map. This layer is directly available in Keras as `layers.Conv2D`.
- BatchNorm2D: Normalizes the output of the convolution layer to accelerate training and improve stability. In Keras, this is implemented using `layers.BatchNormalization`.
- SiLU Activation: Also known as the Swish activation function, it introduces smooth non-linearity and has been shown to improve performance in many deep architectures. In Keras, this is invoked using `layers.Activation("swish")`.

To accurately replicate the behavior of the ConvModule, a Keras function named `conv_bn_silu` is implemented. This function receives a TensorFlow tensor (`x`) and a pre-trained PyTorch ConvModule block (`tblock`), extracts its structural parameters and weights, and builds the equivalent Keras block.

Padding compatibility between the two frameworks is also considered. PyTorch explicitly defines padding as a tuple, whereas Keras allows specification via either `"same"` or `"valid"`. When explicit padding is present, a `ZeroPadding2D` layer is prepended to align the spatial dimensions.

Weight transfer is handled using helper functions `torch_conv_weights` and `torch_bn_weights`, which convert and load the corresponding parameters into the new Keras layers, as explained in earlier sections.

The complete function is shown below:

```python
def conv_bn_silu(x, tblock, name, *, log=False):
    tconv: torch.nn.Conv2d = tblock.conv
    tbn: torch.nn.BatchNorm2d = tblock.bn

    pad_h, pad_w = tconv.padding
    if pad_h or pad_w:
        x = layers.ZeroPadding2D(padding=(pad_h, pad_w), name=f"{name}_pad")(x)
        keras_padding = "valid"
    else:
        keras_padding = "valid"

    conv = layers.Conv2D(
        filters=tconv.out_channels,
        kernel_size=tconv.kernel_size,
        strides=tconv.stride,
        padding=keras_padding,
        dilation_rate=tconv.dilation,
        groups=tconv.groups,
        use_bias=tconv.bias is not None,
        name=f"{name}_conv",
    )
    x = conv(x)

    bn = layers.BatchNormalization(
        epsilon=tbn.eps,
        momentum=1 - tbn.momentum,
        name=f"{name}_bn",
    )
    x = bn(x)

    x = layers.Activation("swish", name=f"{name}_silu")(x)

    conv.set_weights(torch_conv_weights(tconv))
    bn.set_weights(torch_bn_weights(tbn))

    return x
```

Listing 3.31: Implementation of the ConvModule block in Keras, reproducing PyTorch's convolution, batch normalization, and SiLU activation using compatible layers and transferring the pretrained weights.

3.4.5.2.2.2 Bottleneck

The next architectural unit requiring conversion is the Bottleneck block, a widely used structure in convolutional neural networks for reducing computational complexity while preserving representational capacity. This block is particularly effective in deep architectures, where efficient feature reuse and dimensionality control are essential.

In the context of YOLOv8, the Bottleneck block appears in two primary configurations:

- With a residual connection (bypass): The input is added to the output of the convolutional sequence, forming a residual link that facilitates gradient flow and combats vanishing gradients in deep networks.

- Without a residual connection: The output of the convolutional layers is used directly, without summing with the input.

Structurally, both variants consist of two sequential ConvModules, each implementing a `Conv2D → BatchNorm → SiLU` combination, as described in the previous section.

In the implementation, the Bottleneck block is embedded within higher-level structures—specifically the `C2f` block (also referred to as `CSPLayer_2Conv`). Therefore, the logic for constructing and converting Bottlenecks has been delegated to the C2f implementation, ensuring modular reuse and centralized control over the conversion process.

The boolean attribute `add` within the Bottleneck class determines whether a residual connection is applied. When `add` is `True`, the input tensor is added element-wise to the output of the second convolution. This conditional operation is explicitly reflected in the Keras implementation using layers.Add().

The following code snippet illustrates the conversion logic for a single Bottleneck block:

```
identity = x2
x2 = conv_bn_silu(x2, bottleneck.cv1, f"c2f{idx}_b{j}cv1", log=log)
x2 = conv_bn_silu(x2, bottleneck.cv2, f"c2f{idx}_b{j}cv2", log=log)
if bottleneck.add:
    x2 = layers.Add(name=f"c2f{idx}_b{j}_add")([identity, x2])
parts.append(x2)
```

Listing 3.32: Conversion of a Bottleneck block from PyTorch to Keras, including support for optional residual connections via conditional application of an additive merge.

This implementation ensures that the original functional behavior of the Bottleneck block is preserved, and that both architectural variants are supported within a unified framework.

3.4.5.2.2.3 C2f Block (Cross Stage Partial Layer)

After converting the foundational `ConvModule` and the `Bottleneck` block, the next component to address is the C2f block, also referred to in the YOLOv8 architecture as CSPLayer_2Conv. This block is a critical structural unit designed to improve computational efficiency and gradient flow by splitting the feature map and processing parts of it in parallel, a concept derived from Cross Stage Partial Networks (CSPNet).

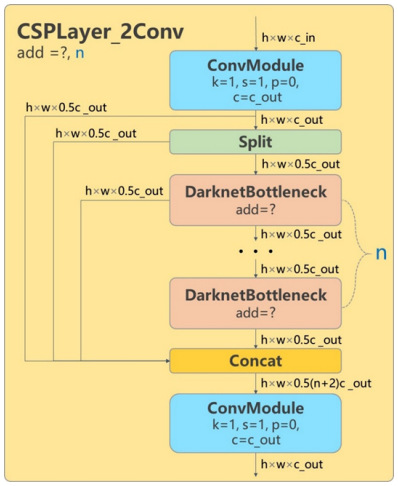


Figure 3.19: Architectural Overview of the CSPLayer_2Conv. Source: OpenMMLab[54]

The C2f block is composed of:

- An initial ConvModule to transform the input,
- A channel-wise split of the resulting feature map,
- A series of Bottleneck blocks applied to one of the feature partitions,
- A concatenation of all paths (including the untouched branch and the outputs of the bottlenecks),
- A final ConvModule to fuse the features,
- An identity `Lambda` layer to name the output explicitly, preserving traceability across layers.

The code below illustrates the full conversion of a C2f block from PyTorch to Keras:

```
def convert_c2f(x, tblock, idx, *, log=False):
    x = conv_bn_silu(x, tblock.cv1, f"c2f{idx}_cv1", log=log)

    half = x.shape[-1] // 2
    x1 = layers.Lambda(lambda z: z[..., :half], name=f"c2f{idx}_split1")(x)
    x2 = layers.Lambda(lambda z: z[..., half:], name=f"c2f{idx}_split2")(x)

    parts = [x1, x2]
    for j, bottleneck in enumerate(tblock.m):
        identity = x2
        x2 = conv_bn_silu(x2, bottleneck.cv1, f"c2f{idx}_b{j}cv1", log=log)
        x2 = conv_bn_silu(x2, bottleneck.cv2, f"c2f{idx}_b{j}cv2", log=log)
        if bottleneck.add:
            x2 = layers.Add(name=f"c2f{idx}_b{j}_add")([identity, x2])
        parts.append(x2)

    x = layers.Concatenate(name=f"c2f{idx}_concat")(parts)
    x = conv_bn_silu(x, tblock.cv2, f"c2f{idx}_cv2", log=log)

    x = layers.Lambda(lambda z: z, name=f"stage{idx}_out")(x)
    return x
```

Listing 3.33: Keras implementation of the C2f (Cross Stage Partial) block, reproducing the split-transform-merge strategy with multiple bottleneck branches and final feature fusion, adapted from the YOLOv8 architecture.

This function ensures that:

- The initial convolution (`cv1`) reduces the input to a manageable number of channels.
- The output is split into two parts (`x1` and `x2`) using `Lambda` layers.
- A loop is used to apply multiple Bottleneck units to one of the branches (`x2`), with optional residual connections as dictated by the `add` attribute.

- All paths are concatenated along the channel axis, preserving feature diversity.
- A final convolution (`cv2`) is applied to merge and normalize the outputs.
- An identity `Lambda` layer labels the final output for structural clarity and debugging purposes.

This implementation ensures architectural fidelity to the original PyTorch version and maintains modularity for weight assignment and integration.

3.4.5.2.2.4 SPPF (Spatial Pyramid Pooling – Fast)

The final component to be converted within the YOLOv8 backbone is the SPPF (Spatial Pyramid Pooling – Fast) block. Positioned at the end of the backbone, this module plays a crucial role in aggregating multi-scale spatial features, enabling the network to capture contextual information with varying receptive fields before passing the representation to the head.

Unlike more complex modules, the SPPF block is relatively simple in structure and implementation. It follows a hierarchical pooling strategy that applies multiple MaxPooling operations with a fixed kernel size and stride, successively stacking pooled feature maps to enrich spatial diversity.

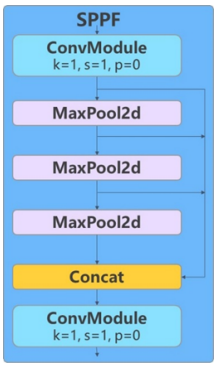


Figure 3.20: Architectural Overview of the SPPF. Source: OpenMMLab[54]

The structure of the SPPF block is as follows:

- A preliminary ConvModule that transforms the input features.
- Three consecutive MaxPooling layers, each applied to the output of the previous one. All use a kernel size of 5 and a stride of 1, maintaining spatial resolution while expanding the receptive field.
- A concatenation operation that merges the outputs of the original input and all three pooled versions along the channel axis.
- A final ConvModule to fuse and normalize the combined features.
- An identity Lambda layer used to label the final output, improving traceability in model inspection.

The Keras-based implementation below replicates this behavior:

```python
def convert_sppf(x, tblock, idx, *, log=False):
    x = conv_bn_silu(x, tblock.cv1, f"sppf{idx}_cv1", log=log)

    p1 = layers.MaxPooling2D(5, 1, "same", name=f"sppf{idx}_p1")(x)
    p2 = layers.MaxPooling2D(5, 1, "same", name=f"sppf{idx}_p2")(p1)
    p3 = layers.MaxPooling2D(5, 1, "same", name=f"sppf{idx}_p3")(p2)
    x = layers.Concatenate(name=f"sppf{idx}_concat")([x, p1, p2, p3])

    x = conv_bn_silu(x, tblock.cv2, f"sppf{idx}_cv2", log=log)
    x = layers.Lambda(lambda z: z, name=f"stage{idx}_out")(x)
    return x
```

Listing 3.34: Keras implementation of the SPPF (Spatial Pyramid Pooling – Fast) block, applying hierarchical max pooling and multi-scale feature fusion, as used in the final stage of the YOLOv8 backbone.

This implementation ensures full architectural fidelity with the original PyTorch version, while leveraging Keras layers for compatibility with the TensorFlow ecosystem. The simplicity and efficiency of this block make it especially well-suited for real-time applications and deployment on resource-constrained hardware.

3.4.5.3. Backbone Assembly

With all individual layer blocks successfully converted and the weight transfer functions defined, the final step in the manual conversion process is to assemble the complete Keras backbone. This stage involves programmatically reconstructing the network by iterating through the PyTorch model's layers and mapping each one to its corresponding Keras implementation.

The assembly process is encapsulated in a function that:

- Accepts a PyTorch backbone model as input.
- Initializes a Keras functional model with the same input shape as the original YOLOv8 backbone (i.e., 320×320 pixels with 3 channels).
- Iterates through each layer of the PyTorch model using `children()` to preserve the sequential structure,
- Dynamically identifies the layer type by name using `layer._get_name()`.
- Dispatches each layer to the appropriate Keras conversion function: `conv_bn_silu`, `convert_c2f`, or `convert_sppf`.
- Appends a `Lambda` layer with a named output to mark stage boundaries and facilitate interpretability,
- Raises an exception if an unsupported or unrecognized layer type is encountered, ensuring model consistency.

The result is a fully reconstructed Keras model functionally equivalent to the YOLOv8 backbone, now compatible with TensorFlow pipelines and ready for integration with a new classification head.

```python
def build_keras_backbone(t_backbone, input_shape=(320, 320, 3), *, verbose=False):
    inputs = layers.Input(shape=input_shape)
    x = inputs

    for idx, layer in enumerate(t_backbone.children()):
        lname = layer._get_name()
        if lname == "Conv":
            x = conv_bn_silu(x, layer, f"conv{idx}", log=verbose)
            x = layers.Lambda(lambda z: z, name=f"stage{idx}_out")(x)
        elif lname == "C2f":
            x = convert_c2f(x, layer, idx, log=verbose)
        elif lname == "SPPF":
            x = convert_sppf(x, layer, idx, log=verbose)
        else:
            raise ValueError(f"Unsupported layer type: {lname}")

    return Model(inputs, x, name="keras_yolov8_backbone")
```

Listing 3.35: Function to assemble the converted YOLOv8 backbone in Keras by iterating through the PyTorch layers and applying the corresponding Keras conversion functions for each block.

3.4.5.4. Validation of the Converted Model

An essential step in the manual model conversion process is the validation of the functional equivalence between the original PyTorch backbone and its Keras counterpart. This ensures that the reconstructed model not only mirrors the architectural structure of the original but also reproduces its behavior when exposed to a wide range of inputs.

To achieve this, a rigorous layer-by-layer comparison is conducted between the PyTorch model and the Keras

implementation. For a given input (typically a synthetic tensor or a real image preprocessed identically in both frameworks), the output of each corresponding stage in the Keras model is compared against the output from the equivalent PyTorch layer.

This validation process serves several purposes:

- Detecting numerical inconsistencies introduced during weight transposition or data format conversion (e.g., channel-first vs. channel-last),
- Ensuring semantic parity in activation functions, padding behavior, and normalization dynamics,
- Verifying successful weight assignment across all layers, especially in cases where no explicit errors are thrown during loading but silent mismatches may occur,
- Identifying potential issues arising from differences in default hyperparameters or floating-point precision between frameworks.

Any discrepancies are analyzed and addressed iteratively, either by adjusting layer configurations, re-transposing weights, or aligning data preprocessing pipelines. Only after the outputs of the two models are aligned to a negligible error margin—typically within the range of floating-point tolerances—can the Keras backbone be deemed a faithful reproduction of the original PyTorch model.

This careful validation provides the confidence needed to proceed with downstream tasks such as classification head integration and fine-tuning, knowing that the core feature extractor behaves identically to the pre-trained YOLOv8 backbone.

3.4.5.5. Integration of the Classification Head

Once the backbone has been successfully converted, validated, and assembled in Keras, the next step is to append a classification head to repurpose the feature extractor for binary classification tasks. The goal is to design a lightweight yet expressive classifier that builds upon the rich feature representations generated by the pre-trained YOLOv8 backbone.

In keeping with the design principles applied in earlier Keras-based classification models, the classification head is constructed as a series of convolutional and dense layers. This modular structure facilitates gradual abstraction and dimensionality reduction, culminating in a binary decision output through a sigmoid activation function.

To connect the head to the backbone, the output from the last backbone stage—specifically the `"stage9_out"` layer corresponding to the SPPF block—is used as the entry point:

```
x = k_backbone.get_layer("stage9_out").output

# Classification head
x = layers.Conv2D(128, kernel_size=3, activation='relu')(x)
x = layers.MaxPooling2D()(x)

x = layers.Flatten()(x)
x = layers.Dense(64, activation='relu')(x)
x = layers.Dropout(0.5)(x)
output = layers.Dense(1, activation='sigmoid')(x)
```

Listing 3.36: Construction of the classification head in Keras, consisting of convolutional, pooling, dense, and dropout layers, appended to the final stage of the converted YOLOv8 backbone.

This architecture introduces two convolutional layers with increasing filter sizes, each followed by max pooling to progressively reduce the spatial dimensions. The resulting feature map is then flattened and passed through a fully connected dense layer with ReLU activation and dropout for regularization. Finally, a single-neuron output layer with a sigmoid activation produces the predicted class probability.

However, before compiling or training this composite model, it is crucial to freeze the backbone weights.

This prevents the newly initialized classification layers from propagating untrained gradients into the pre-trained backbone, which could otherwise overwrite previously learned features and degrade performance.

Freezing is accomplished via:

```
k_backbone.trainable = False
```

Listing 3.37: Freezing the backbone layers to preserve the pre-trained segmentation weights and restrict gradient updates to the classification head only.

By setting the backbone as non-trainable, the model ensures that only the classification head is updated during training, allowing for a fine-grained transfer learning approach that retains the integrity of the original YOLOv8 representation.

3.4.5.6. Manual Dataset Adaptation

Before proceeding with model training, it is necessary to adapt the dataset to meet the input requirements of the converted YOLOv8 backbone. Although the original dataset consists of images with dimensions of 300 × 300 pixels, YOLO models are designed to process input dimensions that are multiples of 32, due to their architectural downsampling structure (based on repeated 2×2 pooling or stride operations). This ensures compatibility with the internal feature map resolutions at each layer.

Consequently, while this dimensional constraint is not strictly required by standard Keras models, it becomes mandatory when reusing a YOLO backbone—since altering its input resolution may result in dimensional mismatches or degraded performance. In practice, YOLO resizes all input images to the nearest upper multiple of 32. For 300 × 300 inputs, the target resolution becomes 320 × 320 pixels.

YOLO applies this resizing by preserving the original image's aspect ratio and then padding the image with a consistent background colour to reach the target size. The default padding colour used in YOLO is:[57]

$$RGB = (114, 114, 114) \tag{3.5}$$

This padding strategy mimics the behavior of OpenCV or similar backends used in real-time object detection pipelines. To replicate this behavior within a TensorFlow data pipeline, the following Python function is implemented:

```python
def resize_and_pad(image, label):
    TARGET_SIZE = 320
    PADDING_COLOR = 114

    image = tf.cast(image, tf.float32)

    shape = tf.shape(image)
    h = shape[0]
    w = shape[1]

    scale = tf.minimum(TARGET_SIZE / tf.cast(h, tf.float32),
                       TARGET_SIZE / tf.cast(w, tf.float32))

    new_h = tf.cast(tf.round(tf.cast(h, tf.float32) * scale), tf.int32)
    new_w = tf.cast(tf.round(tf.cast(w, tf.float32) * scale), tf.int32)

    resized = tf.image.resize(image, [new_h, new_w], method="bilinear")

    pad_h = TARGET_SIZE - new_h
    pad_w = TARGET_SIZE - new_w
    pad_top = pad_h // 2
    pad_bottom = pad_h - pad_top
    pad_left = pad_w // 2
    pad_right = pad_w - pad_left

    padded = tf.pad(resized,
                    paddings=[[pad_top, pad_bottom],
                              [pad_left, pad_right],
                              [0, 0]],
                    constant_values=PADDING_COLOR)
```

```

    padded.set_shape([TARGET_SIZE, TARGET_SIZE, 3])
    padded = padded / 255.0
    return padded, tf.reshape(label, [1])
```

Listing 3.38: TensorFlow function that resizes input images to 320 × 320 pixels while preserving aspect ratio and applying YOLO-style padding using a background colour of RGB(114, 114, 114).

Once this function is defined, it is integrated into the TensorFlow `tf.data` pipeline to transform each image in real time during training and validation:

```
train_dataset = train_dataset.unbatch().map(resize_and_pad, num_parallel_calls=tf.data.
    AUTOTUNE)
val_dataset = val_dataset.unbatch().map(resize_and_pad, num_parallel_calls=tf.data.
    AUTOTUNE)

train_dataset = train_dataset.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)
val_dataset = val_dataset.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)
```

Listing 3.39: Adaptation of the training and validation datasets using the YOLO-compatible resizing function, with batching and prefetching for optimized training throughput.

This preprocessing step guarantees that the dataset is spatially aligned with the YOLOv8 backbone, while maintaining consistency in aspect ratio and ensuring that no artifacts are introduced through naive resizing. Moreover, the use of TensorFlow's `AUTOTUNE` ensures optimal performance through parallel execution and prefetching.

3.4.5.7. Final Configuration and Training

To complete the classification pipeline based on the converted YOLOv8 backbone, the final step consists of defining the training configuration, in alignment with the procedures followed for the previous Keras-based models. This includes setting up class balancing, callback mechanisms, and compiling the model with appropriate optimization parameters.

Class imbalance is addressed by computing class weights, which ensure that minority class samples contribute proportionally more to the loss function during training. Callback functions such as EarlyStopping and ModelCheckpoint can also be included to enhance training stability and preserve the best-performing model configuration.

For the compilation step, the Adam optimizer is selected, which remains one of the most effective and widely adopted optimizers in deep learning due to its adaptive learning rate and robust convergence properties. A learning rate of 1e-3 is chosen, consistent with common practices for training moderately sized convolutional models. The binary crossentropy loss function is used, appropriate for binary classification tasks, and accuracy is tracked as the primary performance metric.

The model is compiled as follows:

```
model.compile(
    optimizer=tf.keras.optimizers.Adam(1e-3),
    loss='binary_crossentropy',
    metrics=['accuracy']
)
```

Listing 3.40: Compilation of the classification model using the Adam optimizer (learning rate 1e-3), binary crossentropy loss, and accuracy as the evaluation metric.

This configuration ensures that the model is ready for training, with parameters tuned to balance learning efficiency and model stability. It also maintains consistency with previous experiments, allowing for direct comparative analysis of performance across different model architectures and preprocessing pipelines.

3.4.5.8. Fine-Tuning

After completing the initial training phase and obtaining a baseline version of the classification model, the next step is to perform fine-tuning—a widely adopted strategy in transfer learning. Fine-tuning allows the model to further adapt its pre-trained feature representations to the specific characteristics of the target

dataset by gradually unfreezing layers in the backbone and continuing training with a reduced learning rate.

The rationale behind fine-tuning is that the lower layers of a deep convolutional network typically capture general-purpose features (e.g., edges, textures), while higher layers encode more task-specific patterns. By unfreezing the final layers of the backbone, the model can refine these high-level representations to better align with the classification task at hand.

The fine-tuning process involves the following steps:

1. Selectively unfreeze layers in the backbone (e.g., the last few convolutional blocks).

2. Set these layers as trainable, allowing their weights to be updated during training.

3. Re-compile the model, using a lower learning rate (e.g., 1×10^{-5}) to avoid disrupting previously learned features.

4. Retrain the model on the same dataset, now with an extended capacity for feature adaptation.

The compilation step for the fine-tuning stage is shown below:

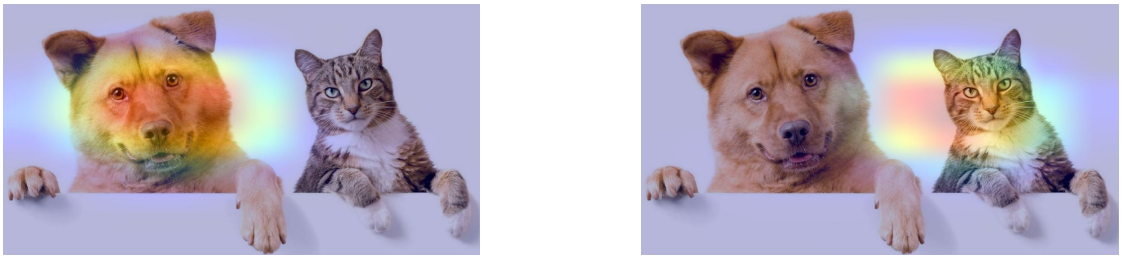
```
model.compile(
    optimizer=tf.keras.optimizers.Adam(1e-5),  # small LR for fine-tuning
    loss='binary_crossentropy',
    metrics=['accuracy']
)
```

Listing 3.41: Recompilation of the model for fine-tuning using a reduced learning rate (1e-5) to safely update the weights of the unfrozen backbone layers.

3.5. Grad-CAM Demonstration

In addition to training the models with the objective of performing inference on the dataset—achieving tasks such as classification or segmentation—an essential aspect of the experimental work involves analysing model interpretability. Specifically, the Grad-CAM (Gradient-weighted Class Activation Mapping) algorithm is employed to gain insights into how the model arrives at its decisions. Grad-CAM is a visualization technique that leverages the gradients of any target concept, flowing into the final convolutional layer of a convolutional neural network, to produce a coarse localization map highlighting the important regions in the input image for predicting the concept.

By applying Grad-CAM, it becomes possible to investigate which areas of an input image most strongly influence the model's predictions. This analysis helps to verify whether the model focuses on relevant features or if it relies on spurious correlations within the data. For instance, consider a convolutional neural network trained for animal detection, capable of distinguishing between different classes such as dogs and cats. When generating a Grad-CAM heatmap for the class "dog," the resulting visualization should ideally highlight the regions of the image containing the dog. Conversely, if a heatmap is generated for the class "cat," the highlighted regions should correspond to the areas occupied by the cat. This targeted visualization enables a qualitative assessment of the model's attention mechanisms, supporting both model validation and debugging.[58]



(a) Grad-CAM heatmap highlighting the region of the image corresponding to the “dog” class.

(b) Grad-CAM heatmap highlighting the region of the image corresponding to the “cat” class.

Figure 3.21: Visualization of Grad-CAM heatmaps for a convolutional neural network trained on animal detection. The highlighted regions indicate which areas of the input image most influenced the model’s predictions for the classes “dog” (left) and “cat” (right). Source: Keras Example.[59]

Furthermore, in the context of defect detection in metallurgical manufacturing, utilizing Grad-CAM assists in determining whether the model attends to actual defect features—such as cracks, surface irregularities, or discolorations—or if it is distracted by irrelevant background elements. Such interpretability tools are critical for ensuring the reliability and trustworthiness of computer vision models deployed in industrial settings, where incorrect or opaque decisions can have significant operational and economic consequences.

3.5.1. Pretrained Model Selection

The initial step in implementing Grad-CAM consists of selecting a suitable pre-trained model. Utilizing pre-trained models is crucial for several reasons. First, these models are typically trained on large-scale datasets, such as ImageNet, and thus possess robust and generalizable feature representations. Their considerable depth and architectural complexity enable them to capture intricate patterns and abstract features from input images. This is particularly important for the effective application of Grad-CAM, which performs optimally when applied to the deeper convolutional layers of the network—layers where high-level semantic features have been learned.

Shallow or lightweight models, while computationally efficient, generally lack the representational capacity required to discern complex patterns in visual data. Their limited depth restricts the ability of convolutional neurons to encode hierarchical or abstract concepts, thereby diminishing the effectiveness of interpretability techniques such as Grad-CAM. For this reason, the use of deeper, high-capacity models is essential when the goal is to visualize and understand the internal decision-making processes of neural networks.

There exists a broad spectrum of pre-trained models, each with distinct architectural features and domain-specific optimizations. Some models may be more suited to certain tasks due to their structural design or training regime. However, a comprehensive benchmarking and comparative analysis of available models for the specific application of defect detection falls outside the scope of this thesis. Instead, the focus here is on leveraging a well-established, high-performing architecture.

In this study, the DenseNet121 model is selected, employing weights pre-trained on the ImageNet dataset. DenseNet121 belongs to the family of Dense Convolutional Networks, which introduce direct connections between any two layers with the same feature-map size. This design allows each layer to receive feature maps from all preceding layers, promoting feature reuse and alleviating the vanishing-gradient problem often encountered in very deep networks. As a result, DenseNet121 is recognized for its efficiency, compactness, and strong representational power, making it an ideal candidate for visualization tasks such as Grad-CAM.[60]

By utilizing DenseNet121, it becomes feasible to exploit the rich, hierarchical features learned during large-scale pre-training and to generate meaningful activation maps that highlight the regions of an image most relevant to the model’s predictions. This approach facilitates a deeper understanding of the decision-making process in convolutional neural networks applied to visual inspection tasks.

3.5.2. DenseNet121 Integration and Processing Pipeline

As described in the preceding section, the DenseNet121 architecture is selected for this stage of the experimental work. Pre-trained models such as DenseNet121 require specific preprocessing steps to be applied to input images to ensure consistency with the data distribution observed during their original training. This preprocessing typically includes resizing images to the required input dimensions, scaling pixel values, and applying normalization parameters that match those used during the ImageNet pre-training phase.

It is essential to standardize the preprocessing pipeline for all phases of the workflow, including training, validation, and inference. Failure to adhere to the recommended preprocessing can result in suboptimal performance or unreliable model outputs, as the model expects inputs with a specific statistical profile.

To streamline this process, the Keras framework provides integrated utilities for both model loading and image preprocessing. Keras includes built-in functions specifically tailored for each pre-trained model, ensuring that images are properly resized, normalized, and formatted before being fed into the network. The typical workflow involves loading the datasets and applying the preprocessing function associated with DenseNet121 as follows:

1. **Dataset Loading:** The dataset is imported into the working environment, typically structured into separate directories for training, validation, and test splits.

2. **Preprocessing:** The images are processed using `the preprocess_input` function from the Keras `applications.densenet` module. This function standardizes the images in a manner consistent with the expectations of the DenseNet121 model.

3. **Model Input:** The preprocessed images are then ready to be passed to the DenseNet121 network for training, evaluation, or inference.

By rigorously following these preprocessing steps, the model is ensured to receive input data in a form that aligns with its training, thereby supporting reliable and reproducible results. In the following code snipet, the practical implementation of these procedures is presented, illustrating how to load the dataset, apply the required preprocessing, and prepare the images for use with DenseNet121 in Keras.

```
image_size = (300, 300)
batch_size = 32
train_ds, val_ds = keras.utils.image_dataset_from_directory(
    "dataset/casting_data/casting_data/train",
    validation_split=0.2,
    subset="both",
    seed=1337,
    image_size=image_size,
    batch_size=batch_size,
    label_mode = 'categorical'
    )

train_ds = train_ds.map(lambda x, y: (keras.applications.densenet.preprocess_input(x),
    y))
val_ds = val_ds.map(lambda x, y: (keras.applications.densenet.preprocess_input(x), y))
```

Listing 3.42: Example of dataset loading and preprocessing for input compatibility with DenseNet121 in Keras.

With the preprocessing steps completed, the next phase involves defining the model architecture. For this purpose, the pre-trained DenseNet121 model is employed as the backbone, serving as a robust feature extractor. To adapt the architecture to the specific classification task, a custom classification head is appended to the network. This head typically consists of one or more dense (fully connected) layers utilizing ReLU activation functions, followed by a final output layer. In image classification tasks, this configuration—combining a deep convolutional base with dense layers culminating in a softmax-activated output—represents a well-established and effective approach.

In contrast to previous models developed for strictly binary classification, the final dense layer in this configuration contains two output neurons. While a single-neuron output with sigmoid activation suffices for binary tasks, preliminary experiments with the Grad-CAM algorithm have shown that a softmax-activated output

layer with multiple neurons yields improved interpretability and more meaningful class-discriminative visualizations, even in the context of binary classification.

Having established these architectural choices, the model is then assembled by integrating the DenseNet121 base with the custom classification head. The following code example illustrates the construction and compilation of this model, demonstrating how to leverage the strengths of both the pre-trained feature extractor and the tailored classification layers for the task at hand.

```
import tensorflow as tf
from keras.applications import DenseNet121

basemodel = DenseNet121(weights = 'imagenet', include_top = False,
                        input_shape = (300, 300, 3), pooling = None)

# for layer in basemodel.layers:
#   layer.trainable = False

x = tf.keras.layers.Flatten()(basemodel.output)
x = tf.keras.layers.Dropout(0.7)(x)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.Dense(16, activation = 'relu')(x)
x = tf.keras.layers.Dropout(0.5)(x)
x = tf.keras.layers.BatchNormalization()(x)
x = tf.keras.layers.Dense(2, activation = 'softmax')(x)
m = tf.keras.models.Model(inputs = basemodel.input, outputs = x)
m.compile(loss = 'binary_crossentropy',
          optimizer = tf.keras.optimizers.Adam(learning_rate = 0.0001),
         metrics = ['accuracy', tf.keras.metrics.Precision(name = 'precision'),
                    tf.keras.metrics.Recall(name = 'recall')])

m.summary()

hist = m.fit(train_ds, epochs = 5, batch_size = 32,
             validation_data = val_ds,
             callbacks = [
                tf.keras.callbacks.EarlyStopping(patience = 10, monitor = 'val_loss',
    mode = 'min', restore_best_weights = True),
                tf.keras.callbacks.ReduceLROnPlateau(patience = 6, monitor = 'val_loss'
    , mode = 'min', factor = 0.1)
            ])
```

Listing 3.43: Construction and compilation of a transfer learning model using DenseNet121 as a fixed feature extractor with a custom classification head in Keras.

3.5.3. Application of the Grad-CAM Algorithm

Once the model has been successfully trained, the next step is to define the methods required for generating heatmap visualizations using the Grad-CAM algorithm. Grad-CAM provides a class-discriminative localization technique, enabling the visualization of the specific regions within an input image that most strongly influence the model's predictions. This not only enhances the interpretability of the model's internal decision-making processes but also assists in validating that the model is focusing on relevant features within the images.

To implement Grad-CAM, a series of utility functions must be defined to extract the necessary gradients and feature maps from the trained model. These functions process the outputs of the final convolutional layers, combine them with the computed gradients, and produce the heatmap overlays. Comprehensive reference implementations and best practices for integrating Grad-CAM with Keras models are provided in the official Keras documentation, ensuring both correctness and compatibility with modern deep learning workflows.

The following section presents a practical example based on these recommended implementations, illustrating how to generate and visualize Grad-CAM heatmaps for a given input image.

```
def get_img_array(img_path, size):
    # `img` is a PIL image of size 299x299
    img = keras.utils.load_img(img_path, target_size=size)
```

```
    # ‘array‘ is a float32 Numpy array of shape (299, 299, 3)
    array = keras.utils.img_to_array(img)
    # We add a dimension to transform our array into a "batch"
    # of size (1, 299, 299, 3)
    array = np.expand_dims(array, axis=0)
    return array

def make_gradcam_heatmap(img_array, model, last_conv_layer_name, pred_index=None):
    # First, we create a model that maps the input image to the activations
    # of the last conv layer as well as the output predictions
    grad_model = keras.models.Model(
        model.inputs, [model.get_layer(last_conv_layer_name).output, model.output]
    )

    # Then, we compute the gradient of the top predicted class for our input image
    # with respect to the activations of the last conv layer
    with tf.GradientTape() as tape:
        last_conv_layer_output, preds = grad_model(img_array)
        if pred_index is None:
            pred_index = tf.argmax(preds[0])
        class_channel = preds[:, pred_index]

    # This is the gradient of the output neuron (top predicted or chosen)
    # with regard to the output feature map of the last conv layer
    grads = tape.gradient(class_channel, last_conv_layer_output)

    # This is a vector where each entry is the mean intensity of the gradient
    # over a specific feature map channel
    pooled_grads = tf.reduce_mean(grads, axis=(0, 1, 2))

    # We multiply each channel in the feature map array
    # by "how important this channel is" with regard to the top predicted class
    # then sum all the channels to obtain the heatmap class activation
    last_conv_layer_output = last_conv_layer_output[0]
    heatmap = last_conv_layer_output @ pooled_grads[..., tf.newaxis]
    heatmap = tf.squeeze(heatmap)

    # For visualization purpose, we will also normalize the heatmap between 0 & 1
    heatmap = tf.maximum(heatmap, 0) / tf.math.reduce_max(heatmap)
    return heatmap.numpy()
```

Listing 3.44: Functions for generating Grad-CAM heatmaps from a given input image, including image preprocessing and class activation map computation for visualization. Source: Keras Example.[59]

With these methods, it becomes possible to generate a NumPy array that represents the structure of the Grad-CAM heatmap corresponding to the input image. However, an additional step is required to superimpose this heatmap onto the original image. This process involves overlaying the Grad-CAM activation map on top of the input image, typically by blending the two using a suitable colormap and adjusting the transparency. The resulting visualization provides an intuitive understanding of which regions in the original image contribute most significantly to the model's decision, facilitating both qualitative analysis and model validation.

The subsequent example demonstrates the procedure for merging the Grad-CAM heatmap with the original image to produce a coherent and interpretable visualization.

```
def save_and_display_gradcam(img_path, heatmap, cam_path="cam.jpg", alpha=0.4):
    # Load the original image
    img = keras.utils.load_img(img_path)
    img = keras.utils.img_to_array(img)

    # Rescale heatmap to a range 0-255
    heatmap = np.uint8(255 * heatmap)

    # Use jet colormap to colorize heatmap
    jet = mpl.colormaps["jet"]

    # Use RGB values of the colormap
    jet_colors = jet(np.arange(256))[:, :3]
    jet_heatmap = jet_colors[heatmap]

    # Create an image with RGB colorized heatmap
```

```
    jet_heatmap = keras.utils.array_to_img(jet_heatmap)
    jet_heatmap = jet_heatmap.resize((img.shape[1], img.shape[0]))
    jet_heatmap = keras.utils.img_to_array(jet_heatmap)

    # Superimpose the heatmap on original image
    superimposed_img = jet_heatmap * alpha + img
    superimposed_img = keras.utils.array_to_img(superimposed_img)

    # Save the superimposed image
    superimposed_img.save(cam_path)

    # Display Grad CAM
    display(Image(cam_path))
```

Listing 3.45: Function for overlaying and visualizing Grad-CAM heatmaps on the original input image, including colorization, superimposition, and saving of the result. Source: Keras Example.[59]

Once the necessary functions for Grad-CAM computation and visualization have been defined, they can be invoked in a straightforward manner to generate and display the heatmaps for new input images. The following example demonstrates the typical usage pattern, illustrating how to pass a trained model and an input image to the implemented functions in order to obtain the corresponding Grad-CAM overlay.

```
!mkdir out_gradcam
m.layers[-1].activation = None
for path, directories, files in os.walk("/content/dataset/casting_data/casting_data/
    test/def_front"):
  for file in files:
    img_path = os.path.join(path, file)
    img_array = keras.applications.densenet.preprocess_input(get_img_array(img_path,
    size=(300,300)))
    heatmap = make_gradcam_heatmap(img_array, m, "relu")
    save_and_display_gradcam(img_path, heatmap, cam_path="/content/out_gradcam/" + file
    )
```

Listing 3.46: Example of batch generation and saving of Grad-CAM visualizations for a set of test images using a trained DenseNet121 model. The code processes each image, computes its Grad-CAM heatmap, and saves the superimposed results.

4

Experimental Results and Analysis

This chapter presents a detailed account of the experimental work undertaken to evaluate the performance of the computer vision models developed for defect detection in metallurgical manufacturing processes. The results reported herein encompass the entire spectrum of the proposed methodology, ranging from the configuration of the computational environment and training protocols to the systematic analysis of model performance using standardized evaluation metrics.

The chapter begins with a thorough description of the experimental setup, including hardware specifications, software environment, and the rationale behind the chosen training protocols and hyperparameter settings. This context ensures the reproducibility of the results and provides a solid foundation for interpreting subsequent findings.

Following this, the core results for each modeling approach are presented. This includes the outcomes of the YOLOv8 classification and segmentation pipelines, as well as the models implemented using Keras and TensorFlow. For each approach, both quantitative metrics—such as accuracy, precision, recall, and F1-score—and qualitative insights are provided to enable a holistic assessment of their strengths and limitations. Special attention is given to hybrid strategies that combine segmentation and classification stages, as well as to the impact of transfer learning and fine-tuning across different stages of training.

A dedicated section is also included to showcase the application of the Grad-CAM algorithm. This segment demonstrates the interpretability of the trained models by visualizing the regions of input images that most strongly influence the model's predictions. Through illustrative examples, the utility of Grad-CAM in understanding, validating, and debugging model behavior is systematically explored.

Overall, this chapter aims to deliver a comprehensive and critical analysis of the experimental results, providing both numerical evidence and visual explanations that collectively validate the effectiveness and reliability of the proposed computer vision solutions in the context of automated defect detection.

4.1. Experimental Setup

Before analyzing the performance of the proposed models, it is essential to establish the experimental conditions under which training, validation, and inference are conducted. The reproducibility and validity of the results depend not only on the models and datasets used but also on the computational environment, training protocols, and evaluation criteria. This section outlines the technical configuration employed throughout the experimental phase, detailing the hardware and software infrastructure, the training strategies applied, and the performance metrics adopted.

The configuration of the computational environment includes the use of cloud-based resources to ensure accessibility and consistency across experiments. Special attention is given to the choice of hardware accelerators, which play a critical role in the efficiency and feasibility of deep learning workflows. Additionally, the

training procedures are defined in terms of hyperparameter selection and image preprocessing standards, enabling fair comparisons across different models and architectural designs.

Finally, the metrics used to evaluate the models are introduced, covering both classification and segmentation tasks, along with computational performance indicators. These metrics serve as the basis for the subsequent analysis, where quantitative and qualitative results are discussed in relation to the configurations presented here.

4.1.1. Computational Environment and Hardware Specifications

The training and evaluation of the proposed models are conducted within the Google Colaboratory (Colab) platform. This cloud-based service, provided by Google, offers a hosted Jupyter notebook environment that supports hardware acceleration and facilitates reproducible experiments without requiring local computational resources. Google Colab proves particularly suitable for academic and research contexts, as it provides free access to computational resources, including CPUs, GPUs, and TPUs, albeit under session time constraints and resource availability conditions.

For the experiments carried out in this work, a GPU-based runtime is selected due to the computational intensity of deep learning training tasks. More specifically, the NVIDIA Tesla T4 graphics processing unit is employed as the primary accelerator. This GPU architecture is part of NVIDIA's Turing generation and is optimized for inference and training tasks involving deep neural networks. The Tesla T4 offers 15 GB of dedicated GDDR6 memory, a critical attribute for handling large batches of high-dimensional image data and enabling complex models to fit in memory without incurring significant swapping overhead.

The environment provided by Google Colab with the T4 accelerator includes the following hardware specifications:

- System RAM: 12.7 GB. This memory supports data pre-processing, batch management, and temporary storage of model activations during training. Sufficient system memory is essential to avoid bottlenecks in data loading and augmentation pipelines, particularly when dealing with large datasets or applying real-time augmentations.
- GPU Memory: 15 GB. High GPU memory is fundamental for training large-scale convolutional based architectures, as it permits the use of larger batch sizes and more complex network configurations without triggering out-of-memory errors.
- Disk Storage: 112.6 GB. The available storage is adequate for hosting all datasets, intermediate files, checkpoints, and model weights. It enables repeated training and testing cycles without requiring frequent data upload operations or external storage dependencies.

This cloud-based setup balances accessibility and performance, allowing for the training and evaluation of modern computer vision models with minimal setup overhead. Although it does not offer the same level of configurability as dedicated on-premise hardware or commercial cloud services (e.g., AWS EC2 or Google Cloud VM instances), it proves sufficient for the scope of the present research, particularly in the prototyping and experimental validation phases.

4.1.2. Training Protocols and Hyperparameter Configuration

To ensure consistency and replicability in the evaluation of all models, a unified training protocol is defined across the different experimental stages. This protocol includes both a general configuration used during the initial training phases and an adapted setup for fine-tuning processes, where pre-trained backbones are further optimized on task-specific data. The hyperparameters selected reflect a balance between computational feasibility and empirical performance, based on established practices in deep learning workflows.

The training configuration is defined as follows:

- Number of Epochs: A total of 100 epochs are used during initial training sessions for all models. This number is chosen to allow sufficient convergence while mitigating overfitting risks, monitored through

validation loss trends. In fine-tuning scenarios, where models are initialized with pre-trained weights, the number of epochs is reduced to 30. This reduction is justified by the typically faster convergence behavior observed during transfer learning.

- Batch Size: A batch size of 32 is selected across all configurations. This value offers a practical compromise between training stability and memory usage, particularly when operating on GPUs with limited VRAM. Moreover, a batch size of 32 maintains statistical efficiency in gradient estimation, which is critical for convergence.
- Image Input Size: During standard training, all images are resized to 300×300 pixels, ensuring compatibility across models and facilitating efficient GPU utilization. An exception is made for the Transfer Learning-based model, which adopts an input resolution of 320×320 pixels, as required by the pre-trained backbone architecture used in its design.
- Learning Rate: The initial learning rate is set to 1×10^{-3} during full model training. This rate enables rapid initial learning without destabilizing the optimization process. In fine-tuning configurations, the learning rate is reduced to 1×10^{-5} to prevent large updates that could disrupt the pre-learned features, ensuring a smooth adaptation to the new data distribution.

This configuration remains consistent across all experiments, ensuring a uniform training procedure that supports meaningful comparison between architectures and variations. The selection of these parameters results from preliminary testing and aligns with widely accepted configurations for convolutional vision models trained in cloud environments.

4.1.3. Evaluation Metrics

The assessment of model performance is conducted using a set of established evaluation metrics that provide insight into different aspects of predictive quality, both at the classification and segmentation levels. These metrics allow for objective comparison between architectures and training strategies, and they serve as the basis for interpreting the effectiveness of the models under various experimental settings.
For classification tasks, the following metrics are computed:

- Accuracy: Measures the proportion of correctly predicted samples over the total number of predictions. It offers a general view of performance but may be misleading in imbalanced datasets.
- Precision: Reflects the proportion of true positive predictions among all positive predictions made by the model. It quantifies the model's ability to avoid false positives.
- Recall: Indicates the proportion of true positive predictions among all actual positive instances in the dataset. It measures the model's capacity to detect relevant cases, minimizing false negatives.
- F1-Score: Represents the harmonic mean of precision and recall. It provides a balanced evaluation when precision and recall are in tension, particularly useful in scenarios where class imbalance is present.
- Confusion Matrix: A tabular representation of prediction outcomes across all classes, offering a detailed breakdown of classification behavior, including misclassifications and class-wise performance.

For semantic segmentation tasks, where pixel-wise classification is required, the following metrics are employed:

- Mean Intersection over Union (mIoU): Calculates the average IoU across all classes. It is obtained by dividing the intersection of predicted and ground truth segments by their union, thus penalizing both false positives and false negatives.
- Dice Coefficient: Also known as the Sørensen–Dice index, this metric emphasizes the overlap between prediction and ground truth. It is especially informative when dealing with class imbalance and small object regions.

In addition to these task-specific indicators, temporal metrics are included to evaluate computational efficiency:

- Inference Time per Image: Measures the average time required by a model to process a single image during inference. This metric is crucial for estimating latency in real-time or near-real-time applications.

- Execution Time per Epoch: Captures the duration of a complete pass over the training dataset. It reflects the scalability of the model and its suitability for iterative retraining in production environments.

Together, these metrics provide a comprehensive view of model behavior from multiple perspectives: accuracy, robustness, efficiency, and deployability. The selection is aligned with standard practices in the evaluation of computer vision systems and supports a thorough and comparative analysis of the experimental results presented in subsequent sections.

4.2. Model Results

This section presents the results obtained from the training and evaluation of the various deep learning models developed for the detection of defective components in metallurgical manufacturing processes. The models include both classification and segmentation architectures, as well as hybrid approaches that combine multiple stages of processing. Each model is assessed according to the performance metrics introduced in the previous section, under consistent training protocols and within the same computational environment.

The objective of this section is twofold: first, to quantitatively evaluate the predictive capacity of each model using standard performance indicators; and second, to compare the different architectural choices and training strategies in terms of accuracy, robustness, and computational efficiency. The results are structured to reflect the experimental pipeline, starting from baseline classifiers and progressing toward more advanced, task-specific configurations, including fine-tuned transfer learning models and multi-stage inference systems.

In addition to numerical results, selected visualizations are included to illustrate the qualitative performance of segmentation outputs and classification decisions. These examples provide further insight into the strengths and limitations of each approach, particularly in cases involving ambiguous or borderline samples. The section concludes with a synthesis of the observed patterns and a critical discussion on the implications of the findings for real-world deployment scenarios.

4.2.1. YOLOv8 Classification

The first architecture evaluated is based on the YOLOv8 framework, configured specifically for classification tasks. Unlike its traditional usage for object detection, this version of YOLOv8 leverages its high-performance convolutional backbone to perform binary classification between defective and non-defective components. The model used in this work is initialized with pre-trained weights, allowing it to benefit from a strong prior learned on large-scale image datasets. This transfer learning strategy significantly accelerates convergence and enhances generalization, particularly in scenarios where the domain-specific dataset is relatively limited in size or diversity.

A fundamental aspect of training deep learning models is monitoring their learning dynamics to ensure stable convergence and to prevent overfitting. To this end, particular attention is given to the evolution of the validation loss over the course of training. The validation loss serves as a key indicator of the model's ability to generalize to unseen data. A consistently decreasing validation loss typically signals a healthy learning process, while stagnation or increase may indicate overfitting or suboptimal learning rates.

Throughout training, continuous monitoring of this metric is carried out. The figure below illustrates the progression of the validation loss in relation to the training loss across epochs:

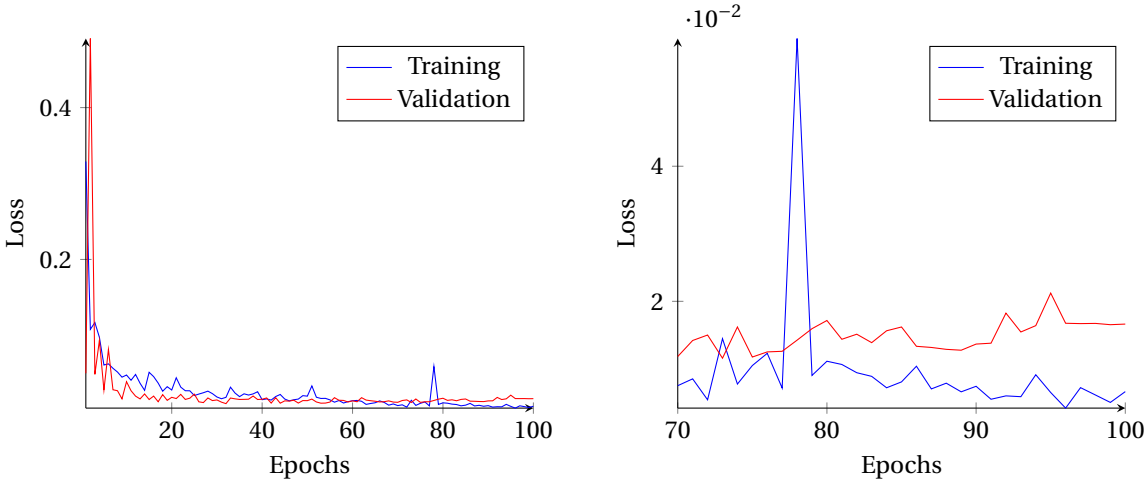


Figure 4.1: Training and validation loss curves over 100 epochs. The left plot shows the entire training process, while the right plot focuses on epochs 70 to 100.

As shown in the figure, the model approaches convergence well before completing the 100 training epochs. However, a slight divergence between the validation loss and the training loss can be observed in the later stages of training. This deviation suggests the initial onset of overfitting, where the model continues to improve on the training data but no longer generalizes as effectively to unseen examples. In practical applications, it becomes relevant to consider the trade-off between training duration and model accuracy. Reducing the number of epochs could mitigate overfitting while significantly decreasing training time—a particularly important consideration in industrial contexts where computational efficiency and rapid iteration are essential.

Another key benchmark during training is the average time per epoch. In this case, the model required 51.96 seconds per epoch, resulting in a total training duration of approximately 1 hour and 26 minutes. This time frame is consistent with the use of a GPU-accelerated environment and reflects the computational demands of the model, dataset size, and augmentation strategies employed during training.

An additional insight provided by the YOLOv8 framework during training is the visualization of representative training batches. These batches allow for a qualitative inspection of the data as it is presented to the model, including the transformations applied through augmentation strategies. As shown in the image below, the framework automatically applies a variety of augmentation techniques directly on the dataset during training.

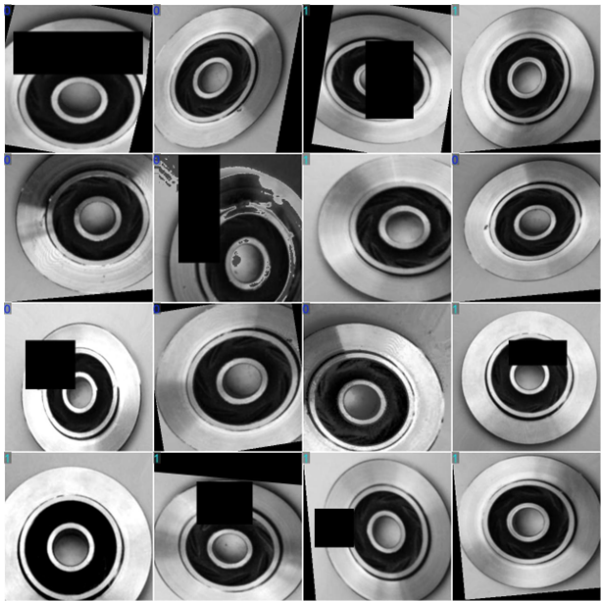


Figure 4.2: Example of a training batch generated by the YOLOv8 framework during model training. The image illustrates the application of online data augmentation techniques—including geometric and photometric transformations—designed to increase dataset variability and improve generalization performance.

These augmentations may include random cropping, horizontal flipping, changes in brightness or contrast, scaling, and other affine transformations. Their purpose is to artificially increase the diversity of the training data, thereby improving the model's ability to generalize and reducing the risk of overfitting. By exposing the model to a broader distribution of input variations, the training process becomes more robust to noise and variability in real-world inspection scenarios.

This augmentation process is performed online—i.e., dynamically at each training step—which avoids the need to store augmented copies and ensures variability across epochs.

To evaluate the generalization performance of the trained YOLOv8 classification model, inference is conducted on the test dataset, which is held out entirely from the training and validation processes. This separation ensures that the reported results reflect the model's true predictive capacity on previously unseen data, free from any influence introduced during optimization.

The inference procedure is executed over the complete test set, and the model achieves an average inference time of 3.98 milliseconds per image, demonstrating strong suitability for real-time or high-throughput industrial inspection systems.

The classification results are summarized in the confusion matrix below, which provides a detailed breakdown of true positives, false positives, false negatives, and true negatives for each class:

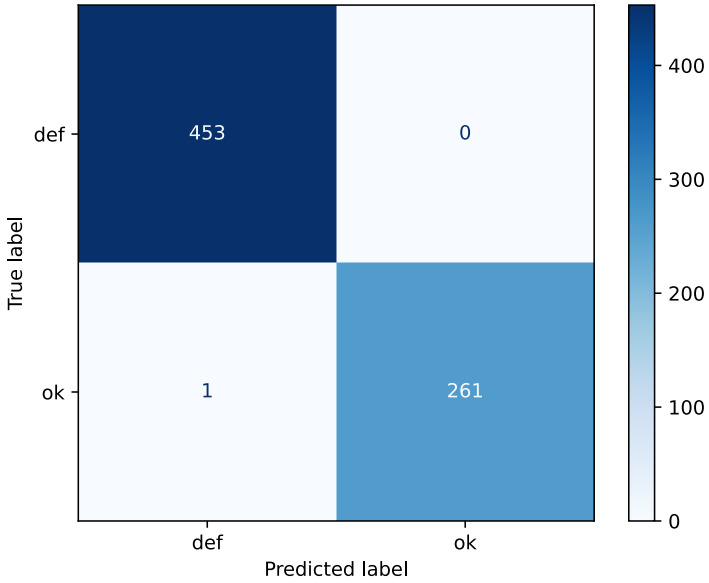


Figure 4.3: Confusion matrix of the YOLOv8 Classification Model on the test dataset.

Based on the classification outcomes, the main performance metrics are computed, including accuracy, precision, recall, and F1-score. These metrics, defined earlier in this section, provide a comprehensive view of the model's predictive capacity across both classes.

Class	**Precision**	**Recall**	**F1-score**	**Support**
Defective	0.9978	1.0000	0.9989	453
Ok	1.0000	0.9962	0.9981	262
Macro Average	0.9989	0.9981	0.9985	—
Weighted Average	0.9986	0.9986	0.9986	—
Accuracy: 99.86%				

Table 4.1: Precision, recall, and F1-score metrics for each class, with overall accuracy reported on the test dataset.

The results clearly demonstrate that the YOLOv8 classification model achieves excellent generalization performance, significantly exceeding the reliability typically associated with manual visual inspection by human operators. Such performance is particularly valuable in industrial contexts, where consistency, speed, and accuracy are critical.

As reflected in the confusion matrix, only one image labelled as "OK" was incorrectly classified as "Defective". This represents the only missclassification within the entire test set. The corresponding sample is shown below:

Figure 4.4: The model's only misclassification among the 715 images in the test dataset, shown with the corresponding predicted probabilities.

4.2.2. YOLOv8 Segmentation

This section presents the results obtained from the two segmentation models trained under a semi-supervised learning regime. Unlike the previous classification-focused experiments, the objective here is not to assign a label to an entire image but to identify and localize specific regions of interest, namely, the target components within complex backgrounds. Segmentation models are particularly relevant in industrial applications where the spatial localization of defects or objects is critical for downstream processing or decision-making.

The task addressed in this experiment focuses on segmenting the component from its background, enabling precise isolation of the object for further inspection or analysis. The segmentation process relies on pixel-level predictions, making the learning task inherently more complex than classification, and demanding careful supervision of training behavior.

As with the classification models, the first step in the evaluation involves analyzing the evolution of training metrics over time to ensure model convergence and detect any signs of overfitting. In segmentation tasks, the loss function is composed of several distinct components, each targeting a specific aspect of the prediction quality. For this experiment, the following losses are monitored:

- Bounding Box Loss: Measures the discrepancy between the predicted and ground truth bounding box coordinates. It evaluates the spatial precision with which the model localizes the object within the image and seeks to minimize positional error.
- Classification Loss: Quantifies the model's ability to correctly classify the content within each predicted bounding box. Although the primary task is segmentation, accurate classification remains important when bounding boxes are used to differentiate multiple object types.
- Segmentation Loss: Assesses the similarity between predicted and actual segmentation masks at the pixel level. This loss component drives the model to generate masks that closely match the true shape and boundaries of the objects.

By continuously monitoring these metrics for both the training and validation datasets, it becomes possible to assess whether the model is learning generalizable patterns or overfitting to the training data. The parallel evolution of these losses across training epochs provides a reliable indication of convergence and learning stability, which is essential in tasks as sensitive as industrial part segmentation.

The following figures present a comparative visualization of the key performance metrics discussed above. These plots facilitate a clearer interpretation of the model's behavior across both classes and provide an intuitive overview of the classification results:

The box loss curve shows a steady and consistent decline in both training and validation phases. Notably, the validation loss remains closely aligned with the training loss, with no significant divergence observed. This behavior indicates that the model maintains good generalization throughout training and is not overfitting to the training set.

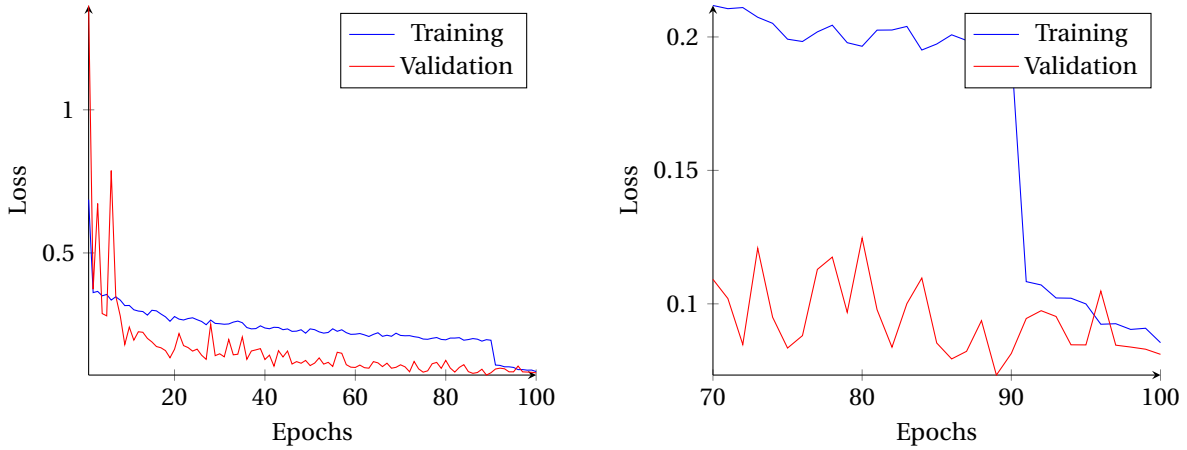


Figure 4.5: Box loss evolution over 100 training epochs. The left plot displays the full training trajectory, while the right plot focuses on epochs 70–100 to highlight the final convergence behavior.

The classification loss exhibits an initial period of instability, particularly in the validation curve, which spikes during early epochs. This is typical in early training stages, especially in transfer learning settings where the classifier head is adapting to new task-specific data. After this phase, both training and validation curves converge progressively. By the final epochs, the validation loss stabilizes at a low value, indicating that the model has successfully learned to differentiate between the classes.

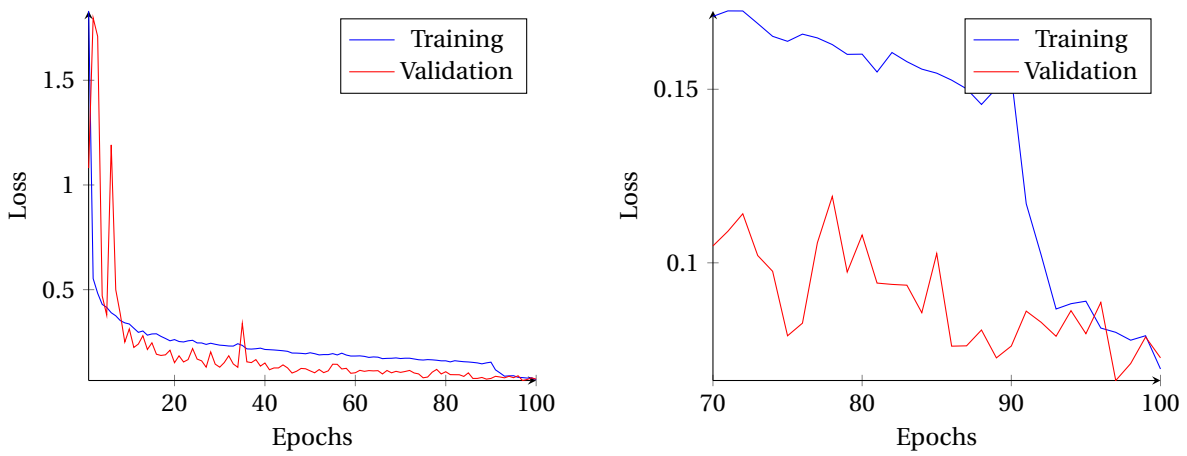


Figure 4.6: Classification loss curves across 100 epochs. The initial fluctuation is followed by smooth convergence, suggesting effective learning of class-level distinctions. The right plot focuses on epochs 70-100 to highlight the final convergence behavior.

The segmentation loss decreases rapidly during the early epochs and continues to decline smoothly across the entire training phase. Both training and validation losses maintain a similar trend, with low variance and no overfitting signs. This consistency demonstrates stable learning and good generalization in the pixel-wise segmentation task.

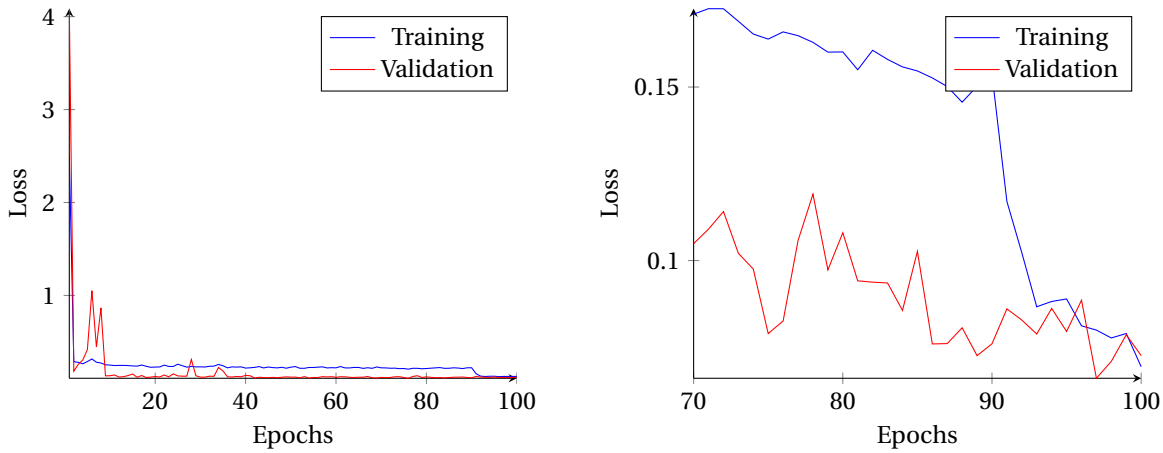


Figure 4.7: Segmentation loss over 100 training epochs. The alignment between training and validation losses indicates strong generalization and model stability. The right plot focuses on epochs 70-100 to highlight the final convergence behavior.

Regarding training efficiency, this model is trained at an average pace of 10.04 seconds per epoch, representing a significant reduction in comparison to the previous classification setup. This improvement is primarily attributed to the smaller size of the dataset used in this particular experiment. Unlike the earlier model, which was trained on the entire training set, this configuration employs only a reduced subset, selected for fine-tuning purposes or for experimentation on segmentation-specific annotations.

As a result, the total training time amounts to only 16 minutes and 44 seconds, making this configuration particularly attractive in scenarios where rapid prototyping or iterative retraining is required.

The images below illustrate representative test samples used during training, showing the applied data augmentation strategies. These transformations increase the dataset's variability, even when working with a reduced sample size, thereby helping the model generalize effectively:

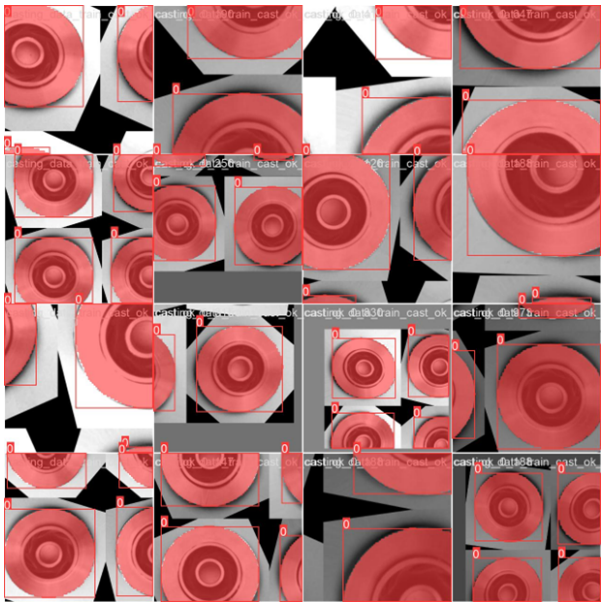
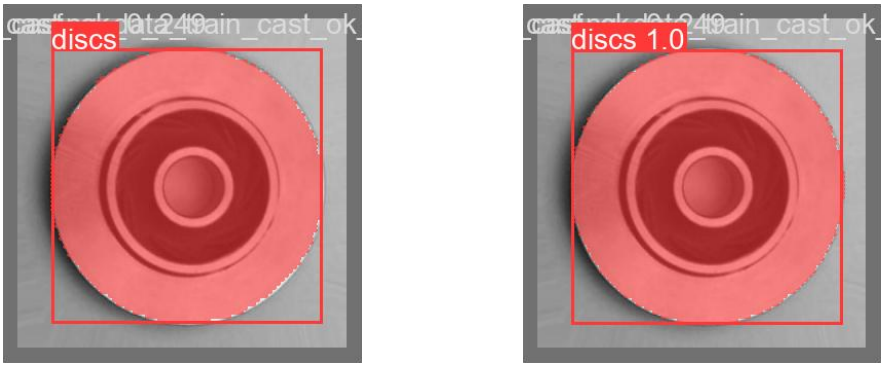


Figure 4.8: Augmented training samples used during segmentation model training.

It is also useful to inspect qualitative examples of the segmentation model's performance on validation data. The image below shows a representative inference result, where the model accurately delineates the object regions:



(a) Ground truth mask from the validation dataset.

(b) Predicted segmentation mask generated by the model.

Figure 4.9: The prediction exhibits a more refined and consistent delineation of the defective region, highlighting the model's capacity to generalize beyond noisy or imperfect annotations.

In this particular example, the predicted segmentation mask demonstrates a level of precision that, in some aspects, surpasses the quality of the original ground truth annotation. This phenomenon highlights the model's strong generalization capability. By learning from patterns across the dataset, the model is able to mitigate inconsistencies or imperfections in the manual labels, effectively compensating for annotation noise and producing more coherent and accurate predictions.

Finally, after completing the training phase, the model is validated on the designated test set. This dataset corresponds to a small, manually labelled subset reserved exclusively for evaluation purposes. Each file in the test set is processed individually, and the corresponding segmentation predictions are generated by the model.

The predicted masks are then compared against the ground truth annotations to compute key performance metrics, including mean Intersection over Union (mIoU) and Dice coefficient, both of which are standard indicators of segmentation quality. In addition, the average inference time per image is recorded to assess the model's efficiency during deployment.

These results provide a quantitative measure of the model's ability to generalize to unseen data and are discussed in the following section.

mIoU	**Dice**	**Inference Time [ms]**
0.9849	0.9924	9.3669

Table 4.2: Segmentation performance metrics obtained on the test dataset. The results reflect the model's accuracy (mIoU and Dice coefficient) and average inference time per image.

mIoU values close to one indicate a high degree of overlap between the predicted masks and the ground truth, reflecting accurate segmentation of the regions of interest. Similarly, high Dice coefficient values signify strong similarity between the masks, confirming the model's spatial consistency and reliability in delineating defective areas.

Subsequently, a second segmentation model is trained using a pseudo-labelled dataset, obtained by running inference with the previously trained model. This approach leverages the generalization ability of the first model to expand the training data beyond the manually annotated samples, allowing for more robust learning on a larger scale.

Unlike the first model, the second segmentation model is trained on a substantially larger dataset, which has a significant impact on training time. The total duration of the training process amounts to 104 minutes and 44 seconds. This increase is expected, given the higher number of samples and the associated computational load during each epoch.

The training dynamics of the second segmentation model can be better understood by analyzing the behavior of the three main loss components: box loss, classification loss, and segmentation loss. Each of these metrics captures a different aspect of the model's learning process, and their joint interpretation provides a holistic view of the convergence process and generalization capabilities.

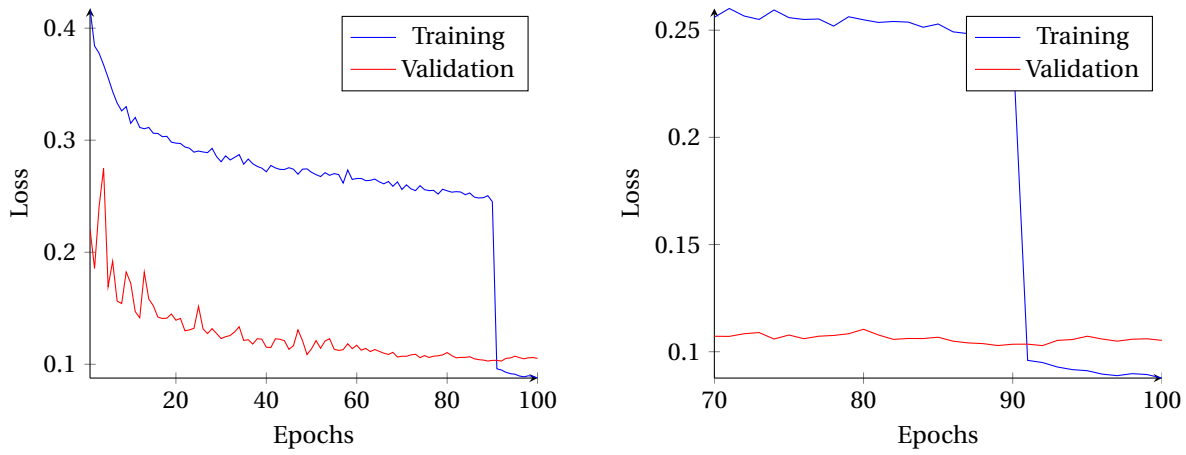


Figure 4.10: Box loss evolution over 100 epochs. The decline in loss demonstrates steady improvements in localization accuracy. The sharp drop after epoch 90 corresponds to the disabling of mosaic augmentation, which improves convergence in the final phase.

Figure 4.10 illustrates the evolution of the box loss across the 100 training epochs. The box loss quantifies the quality of the predicted bounding boxes with respect to the ground truth. Throughout the training process, both the training and validation losses exhibit a smooth and consistent downward trend. The validation curve closely follows the training loss, which suggests that the model is not overfitting and that the learned representations generalize well to unseen data. Around epoch 90, a significant drop in both curves can be observed. This sharp decrease is directly linked to YOLOv8's internal training mechanism, which disables the mosaic augmentation strategy towards the end of training. By switching to unaltered input images, the model receives more stable samples, which facilitates final convergence and allows for rapid refinement in localization accuracy.

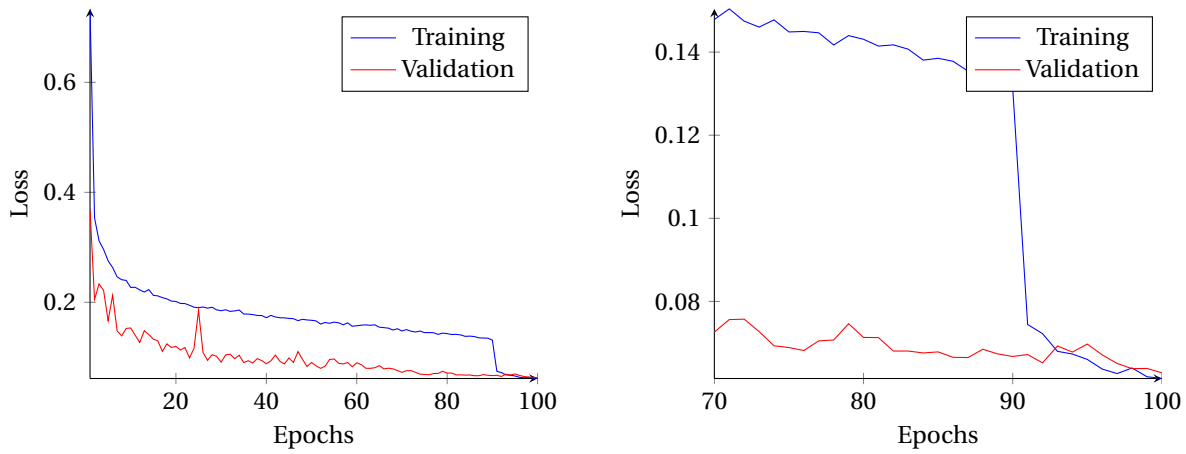


Figure 4.11: Classification loss throughout training. The curve exhibits stable convergence and low variance between training and validation, indicating consistent class prediction performance across samples. A sharp improvement is also observed after mosaic augmentation is turned off.

The classification loss, shown in Figure 4.11, follows a similar trajectory. The initial epochs show rapid improvements in classification performance as the model adapts to the pseudo-labelled dataset. After this early phase, the classification loss gradually continues to decline. Once again, the gap between training and validation losses remains minimal throughout the process, reinforcing the idea that the pseudo-labelled dataset does not introduce harmful bias. Around epoch 90, the loss curves drop more steeply, reflecting the removal

of augmentation noise and the resulting stabilization in class prediction.

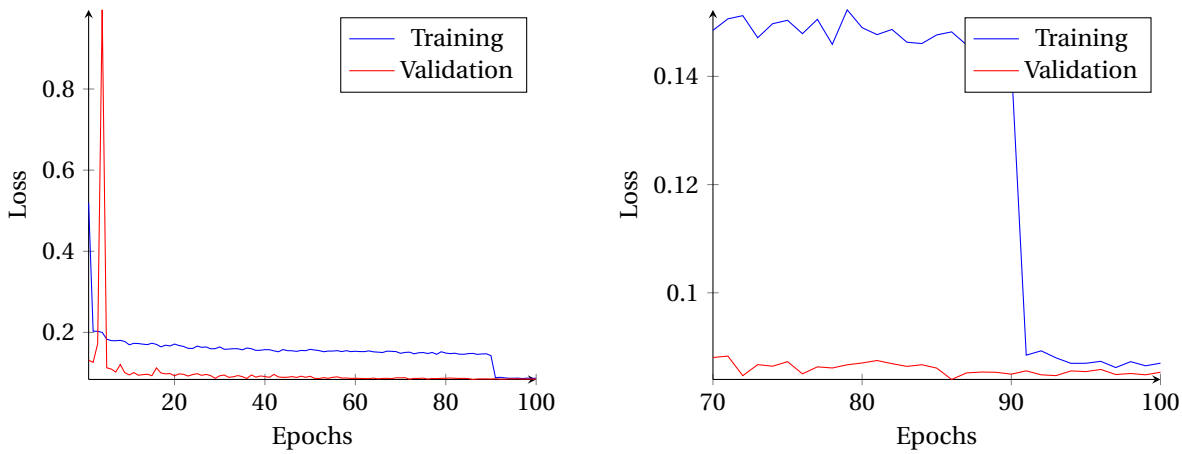


Figure 4.12: Segmentation loss progression over 100 epochs. The model gradually improves its ability to delineate pixel-wise regions, with a marked refinement in loss after epoch 90 as the training transitions away from mosaic-augmented inputs.

Finally, Figure 4.12 depicts the segmentation loss, which measures the pixel-wise accuracy of the predicted masks. This component is particularly critical in the context of semantic segmentation tasks, where spatial precision is essential. As with the previous metrics, the training and validation losses exhibit strong alignment, with consistent improvements across the epochs. The model successfully refines its segmentation capabilities, even in the presence of label noise typical of pseudo-annotations. The pronounced drop near the end of training reaffirms the positive impact of disabling mosaic augmentation: without the distortions introduced by this technique, the model focuses on fine-grained patterns and achieves more stable segmentation outputs.

Evaluating the test dataset enables a quantitative assessment of the model's performance on previously unseen data. By running inference across all test samples and comparing the predicted segmentation masks with the ground truth annotations, it becomes possible to compute the key evaluation metrics. These include the mean Intersection over Union (mIoU), the Dice coefficient, and the average inference time per image.

mIoU	**Dice**	**Inference Time [ms]**
0.9889	0.9944	9.8838

Table 4.3: Segmentation performance metrics obtained on the test dataset. The results reflect the model's accuracy (mIoU and Dice coefficient) and average inference time per image.

As observed in the obtained metrics, the second segmentation model does not exhibit a significant improvement over the first. This outcome suggests that the 100 manually annotated images used in the initial training phase were already sufficient to achieve high performance on the specific segmentation task addressed by this model. Consequently, the semi-supervised training regime—while more extensive in terms of data—has not led to noticeable gains.

Semi-supervised learning typically proves most effective in scenarios where supervised learning alone is insufficient due to limited labelled data. In this case, however, the original labelled dataset already provided enough variability and representative examples, diminishing the added value of the pseudo-labelled samples.

4.2.3. Keras/TensorFlow Classification

This section presents the results obtained by the classification model developed using the TensorFlow framework. The architecture is entirely constructed from scratch, with no reliance on pre-trained weights or transfer learning. Such a design choice aims to evaluate the model's capacity to learn feature representations directly from the metallurgical inspection data, without leveraging prior knowledge from external datasets.

The analysis begins with an examination of the model's behavior throughout the training process, focusing on the evolution of the training and validation losses. These curves provide essential insights into the convergence dynamics, the presence of overfitting, and the overall learning stability. The figure below illustrates the loss trajectories over the training epochs:

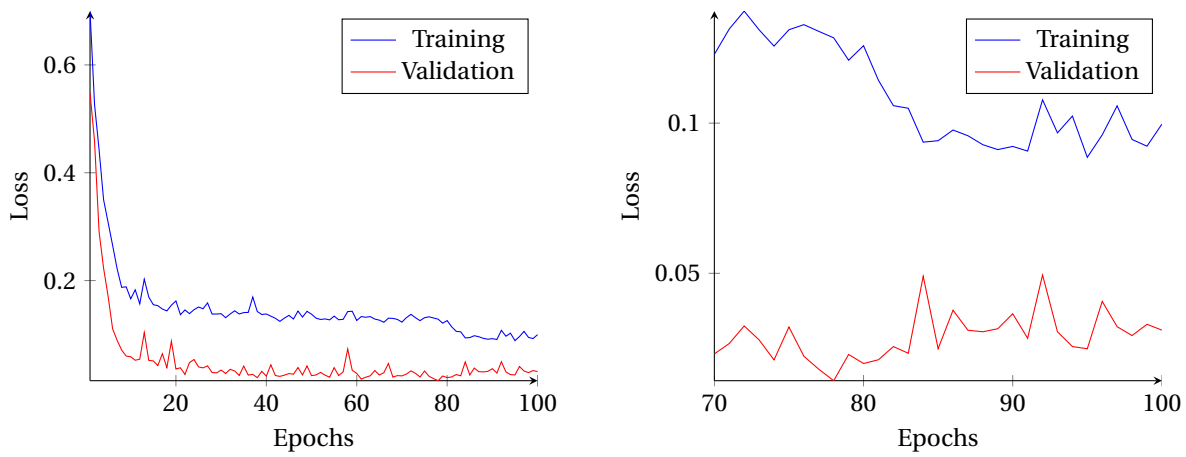


Figure 4.13: Evolution of training and validation losses for the TensorFlow-based classifier. The model demonstrates rapid convergence during the early epochs, achieving low validation loss values. A mild overfitting trend is observed starting around epoch 80, as indicated by the validation losses having an increasing behavior.

As observed with the pre-trained YOLO classification model, the convergence times for the custom TensorFlow model are also relatively short. The loss curves indicate that the model achieves high-quality learning within the first few epochs, quickly stabilizing to produce reliable results.

However, signs of overfitting begin to emerge around epoch 80. From this point onwards, the validation loss exhibits a slight upward trend, diverging from the training loss. This pattern suggests that the model starts to memorize the training data rather than generalize well to unseen examples. Such behavior highlights the importance of implementing early stopping mechanisms or regularization techniques to prevent performance degradation on validation data.

This model, being relatively lightweight due to the limited number of layers and reduced parameterization, trains under a significantly accelerated regime. The overall computational footprint remains low, allowing the model to achieve a remarkable training benchmark compared to the more complex YOLO-based models. Specifically, an average epoch duration of 17.96 seconds is recorded, resulting in a total training time of 29 minutes and 56 seconds.

Once the training phase is complete, the model is evaluated using the test dataset, following the same protocol applied in the evaluation of the YOLO model. However, in contrast to the highly optimized YOLO architecture, the inference speed of this custom TensorFlow model is notably slower. The average inference time per image reaches 35.15 milliseconds, representing an increase of 1124.19% compared to the YOLO classifier.

The classification results for this evaluation are summarized in the confusion matrix below:

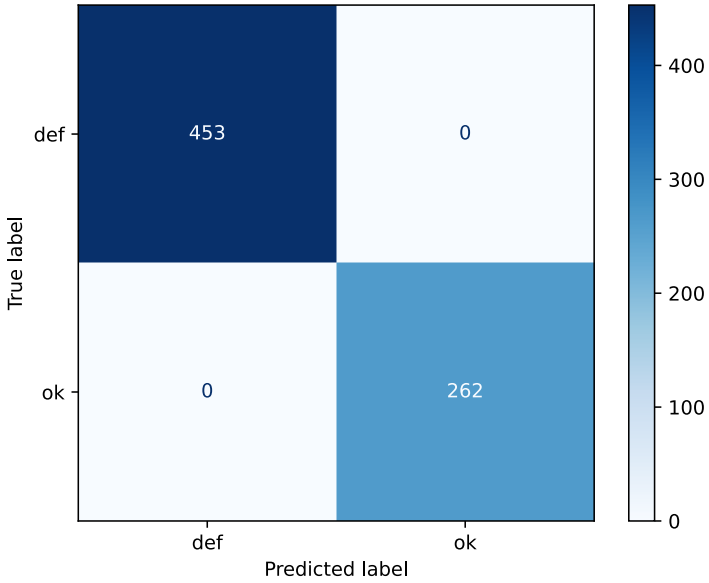


Figure 4.14: Confusion matrix of the Keras/TensorFlow Classification Model on the test dataset.

As observed, the model achieves perfect classification performance on the test dataset, indicating its remarkable capacity to generalize and discriminate between defective and non-defective samples. This result highlights the suitability of the chosen architecture and training configuration for the specific characteristics of the dataset.

The evaluation metrics reflect this exceptional performance, with all key indicators reaching their maximum value:

Class	**Precision**	**Recall**	**F1-score**	**Support**
Defective	1.0000	1.0000	1.0000	453
Ok	1.0000	1.0000	1.0000	262
Macro Average	1.0000	1.0000	1.0000	—
Weighted Average	1.0000	1.0000	1.0000	—
Accuracy: 100%				

Table 4.4: Precision, recall, and F1-score metrics for each class, with overall accuracy reported on the test dataset.

4.2.4. YOLOv8 Segmentation (Crop) + Keras/TensorFlow Classification

The next experiment involves a classification model that incorporates a segmentation-based preprocessing step. Initially, the motivation for this model was to benchmark its performance against the baseline flat classification model. However, given the perfect performance previously achieved on the test dataset, this comparative purpose becomes redundant, as no further improvement can be meaningfully demonstrated.

Nonetheless, the experiment remains relevant as a way to repurpose the prior training scenario and assess how segmentation-based preprocessing impacts model behavior. This approach offers insight into the model's robustness under different data preparation strategies.

The analysis begins with the evolution of the training and validation losses throughout the training process, as shown in the figure below:

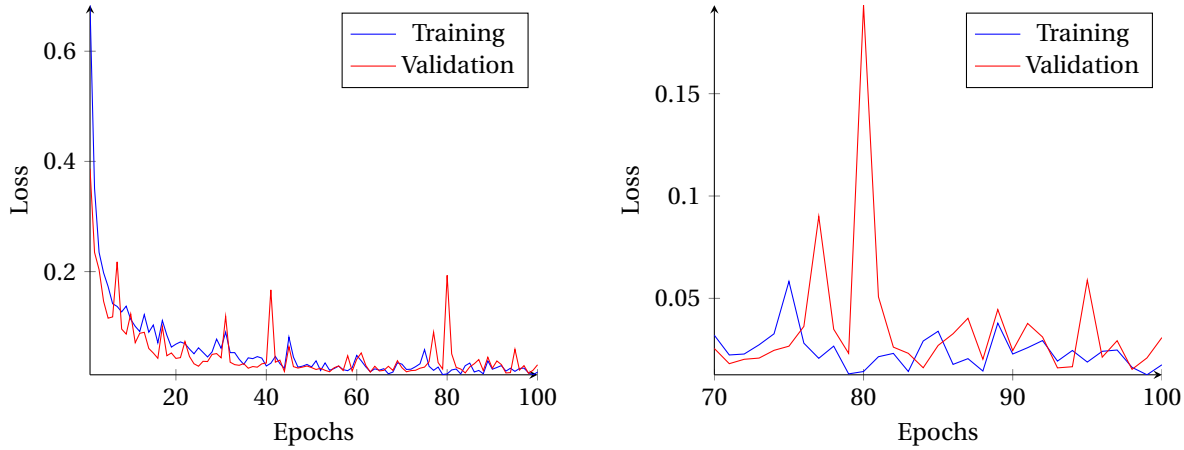


Figure 4.15: Training and validation loss curves for the classification model with segmentation-based preprocessing, showing full training (left) and final epochs (right).

The model demonstrates a relatively fast convergence, although there is still room for improvement, as the best-performing version is reached only at epoch 98.

As in the previous TensorFlow-based scenario, the average training time per epoch remains significantly lower than that observed with the YOLO framework. Specifically, the training completes at an average pace of 18.24 seconds per epoch, resulting in a total training time of 30 minutes and 24 seconds.

Following the training phase, inference is performed on the test dataset to evaluate the model's performance on previously unseen data. The average inference time per image is 35.78 milliseconds, which is consistent with the latency observed in the earlier TensorFlow classification experiment.

The confusion matrix illustrating the classification results is presented below:

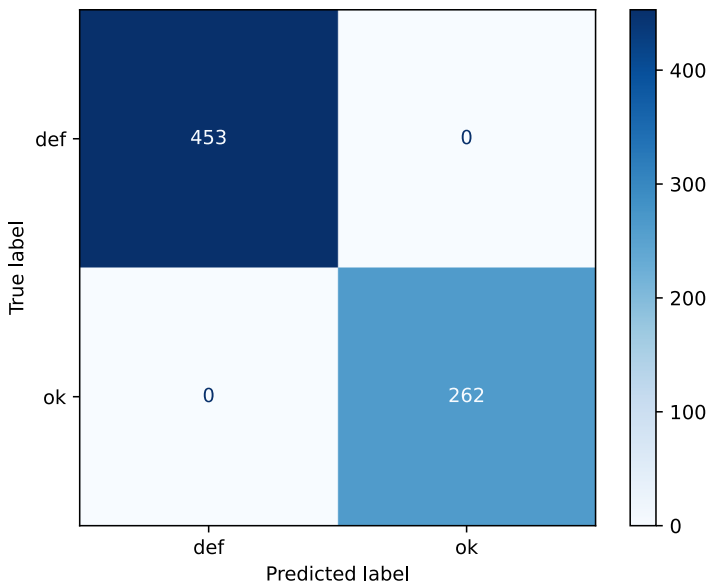


Figure 4.16: Confusion matrix of the YOLOv8 Segmentation (Crop) + Keras/TensorFlow Classification model on the test dataset.

As in the previous experiment, the results indicate perfect classification performance. This confirms that the final model—constructed using the segmentation pipeline trained under a semi-supervised regime—is

equally effective in comparison to the earlier flat classification approach.

The key evaluation metrics obtained from the test dataset are as follows:

Class	Precision	Recall	F1-score	Support
Defective	1.0000	1.0000	1.0000	453
Ok	1.0000	1.0000	1.0000	262
Macro Average	1.0000	1.0000	1.0000	—
Weighted Average	1.0000	1.0000	1.0000	—
Accuracy: 100%				

Table 4.5: Precision, recall, and F1-score metrics for each class, with overall accuracy reported on the test dataset.

4.2.5. YOLOv8 Segmentation (Backbone) + Keras/TensorFlow Classification + Fine-Tuning

The final model in this study is developed using the Keras framework and is based on the backbone extracted from the YOLO segmentation model previously described. This hybrid architecture leverages the feature extraction capabilities of YOLO while integrating a custom classification head in TensorFlow, followed by fine-tuning on the target dataset.

This section presents not only the training results of the Keras-based model but also a detailed validation of the conversion process, comparing the reliability and numerical consistency of the converted model with its original counterpart implemented in PyTorch.

When transferring a model architecture from the PyTorch-based YOLO framework to Keras/TensorFlow, it is essential to validate that the layer-by-layer conversion has been performed correctly. Any discrepancy in layer behavior can lead to significant divergence in the output, ultimately compromising the model's effectiveness.

To ensure correctness, a validation procedure is conducted in which a randomly generated input tensor sampled from a standard normal distribution—is propagated through each layer of both the original PyTorch model and the converted Keras model. For each stage of the forward pass, the outputs are compared using the following metrics:

- `max`$|\Delta|$: The maximum absolute difference between the PyTorch and Keras output tensors at a given layer.
- `mean`$|\Delta|$: The average absolute difference across all elements in the output tensor between both implementations.

The results of this validation are summarized in the table below:

Stage	max$\lvert\Delta\rvert$	mean$\lvert\Delta\rvert$
0	6.104×10^{-5}	1.060×10^{-6}
1	2.136×10^{-4}	5.857×10^{-6}
2	2.337×10^{-5}	5.771×10^{-7}
3	1.907×10^{-5}	4.075×10^{-7}
4	1.526×10^{-5}	2.685×10^{-7}
5	2.670×10^{-5}	4.729×10^{-7}
6	1.812×10^{-5}	3.931×10^{-7}
7	6.676×10^{-5}	7.712×10^{-7}
8	3.052×10^{-5}	6.965×10^{-7}
9	7.153×10^{-6}	2.883×10^{-7}

Table 4.6: Absolute differences between PyTorch and Keras model outputs across each stage during the conversion validation.

The differences observed in this context are, overall, within acceptable bounds. Typical tolerances for intra-framework or cross-framework model conversions generally fall within the range of 10^{-5} to 10^{-3} for most

deep learning tasks, especially when computations are performed using `float32` precision. The reported variations, ranging from 7×10^{-6} to 2×10^{-4}, comply with the stricter limits of this reference interval.

These values indicate that the Keras implementation is numerically very close to the original PyTorch model. In practical terms, such deviations are negligible and have no meaningful impact on model performance or inference behavior. This confirms the reliability of the conversion process and the functional equivalence of the two implementations.

This section presents the performance analysis of the final classification model developed using transfer learning. As previously discussed, the architecture consists of the YOLO segmentation backbone, to which a custom classification head is appended and trained within the Keras framework. The training procedure incorporates fine-tuning, allowing a progressive unfreezing of the YOLO backbone to optimize performance.

To evaluate the effect of fine-tuning on model performance, the training process is divided into the following sequential stages:

1. **Base Training:** The classification head is trained for 100 epochs while keeping the entire YOLO backbone frozen. This prevents early propagation of untrained weights to the feature extractor and allows the classification layers to stabilize.

2. **Unfreezing Stages 8–9:** The final two stages of the backbone are unfrozen, and the model is trained for 30 additional epochs. This enables gradual adaptation of the deeper feature representations.

3. **Unfreezing Stages 6–7:** Two additional stages are unfrozen, and the model is again trained for 30 epochs, allowing further refinement of mid-level features.

4. **Unfreezing Stages 4–5:** The process continues by unfreezing stages 4 and 5, followed by another 30-epoch training phase to incorporate more general feature tuning.

5. **Unfreezing Stages 2–3:** Stages 2 and 3 are unfrozen and the training is extended for 30 more epochs, providing the network with the ability to adjust lower-level features.

6. **Unfreezing Stages 0–1:** Finally, the earliest stages of the YOLO backbone are unfrozen. The model undergoes a last 30-epoch training cycle, enabling full end-to-end fine-tuning of the entire architecture.

This progressive unfreezing strategy allows for a controlled evaluation of how each stage of the backbone contributes to final performance. It also enables a detailed understanding of the impact and efficiency of the transfer learning configuration in this specific metallurgical defect classification task.

4.2.5.1. Base Model

The initial training stage—conducted with the YOLO backbone entirely frozen—achieves an average epoch duration of 20.53 seconds. This training speed is notably efficient, especially considering the architectural complexity and the large number of parameters inherent to the YOLO backbone.

At the end of the 100 training epochs, the model achieves a validation accuracy of 93.07%. This performance is considered solid within the context of defect detection in metallurgical manufacturing, indicating a good initial capacity of the classification head to discriminate between defective and non-defective samples.

However, it is also evident that this baseline configuration does not match the superior accuracy achieved by previous models developed in earlier sections of this work. This suggests that the frozen backbone may not be optimally aligned with the target dataset's characteristics, limiting the model's representational adaptability.

The figure below shows the evolution of the training and validation metrics across epochs during this base training stage:

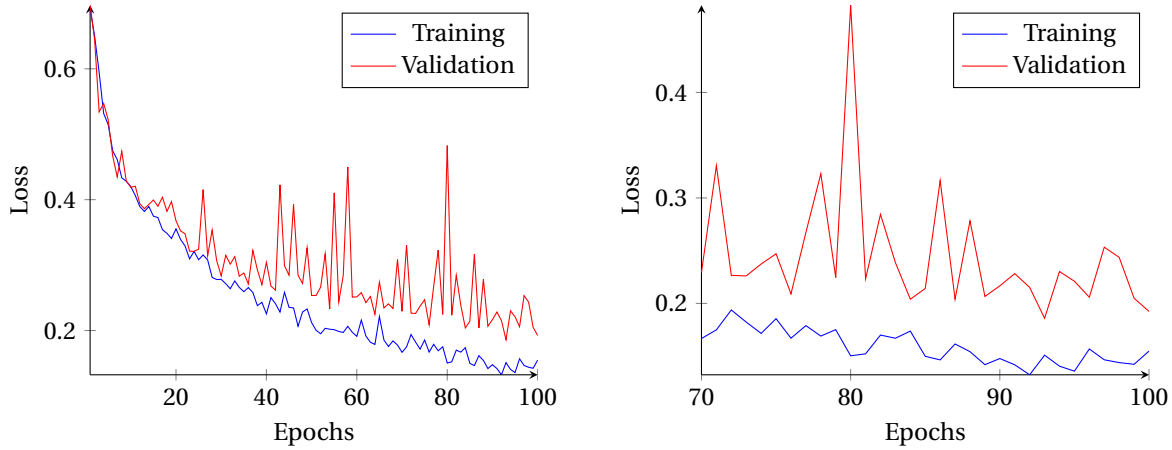


Figure 4.17: Training and validation loss curves for the model with the frozen YOLO backbone, shown across all epochs (left) and the final epochs (right).

As observed, the convergence of this model is noticeably slower compared to previous configurations. This behavior is likely attributed to the larger dimensionality of the model and the difficulty encountered by the classification head in adapting to the pre-trained YOLO backbone under a frozen regime. Moreover, a clear divergence between the training and validation loss is evident, which may indicate that the model is overfitting to the training data. The increased complexity of the model, introduced by the higher number of parameters and feature representations, can contribute to this overfitting, as it allows the model to memorize training examples rather than generalizing effectively to unseen data.

Following training, the model is validated on the test dataset. In this scenario, the inference time is slightly higher relative to the previous solutions. The average inference latency reaches 53.79 milliseconds per image, marking a significant increase compared to earlier approaches and highlighting potential inefficiencies in this configuration.

Based on the inference results, the classification matrix is presented below to offer a visual summary of the model's performance:

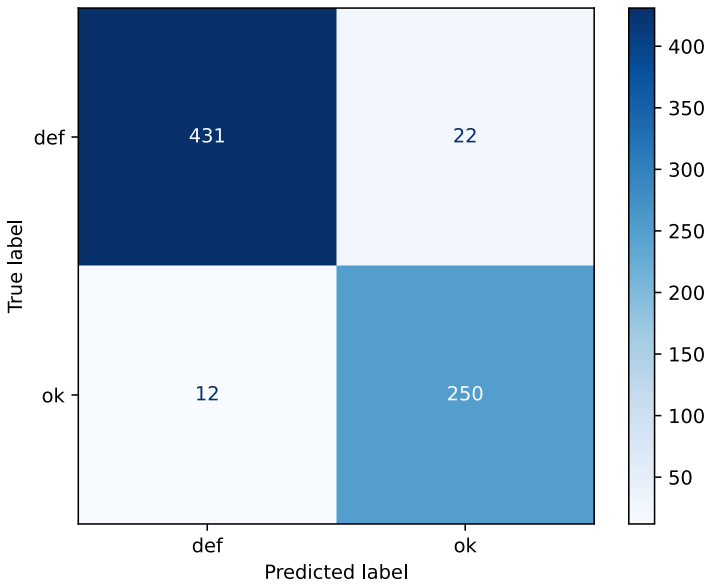


Figure 4.18: Confusion matrix for the model with frozen YOLO backbone, summarizing classification results on the test dataset.

The initial state of the model, with the YOLO backbone entirely frozen, shows a noticeably lower performance compared to the previously explored architectures. Despite this, it still delivers acceptable results within the scope of industrial requirements, maintaining a classification quality that could be considered operationally viable for automated defect inspection.

To illustrate this performance more concretely, the following table presents the detailed classification metrics obtained on the test dataset:

Class	**Precision**	**Recall**	**F1-score**	**Support**
Defective	0.9729	0.9514	0.9621	453
Ok	0.9191	0.9542	0.9363	262
Macro Average	0.9460	0.9528	0.9492	—
Weighted Average	0.9532	0.9524	0.9526	—
Accuracy: 95.24%				

Table 4.7: Precision, recall, and F1-score metrics for each class, with overall accuracy reported on the test dataset.

Compared to the previous models, the performance of this initial configuration falls short, confirming that freezing the entire backbone limits the model's capacity to fully adapt to the target domain. To address this limitation, the next step consists of progressively unfreezing backbone layers to evaluate the impact of fine-tuning on the model's classification performance.

4.2.5.2. Training: Stages 8–9

Following the progressive unfreezing strategy, all layers from the beginning of Stage 8 to the end of Stage 9 are set to trainable for the second phase of training. This enables the uppermost layers of the YOLO backbone to adjust their feature representations to the specific characteristics of the dataset.

The average training time per epoch remains consistent with previous observations, recorded at 20.4 seconds.

After the fine-tuning process, the model reaches a validation accuracy of 94.9%, representing a 1.9-point increase over the initial configuration. This translates to a 26.41% reduction in validation errors, clearly indicating that the additional training and selective unfreezing of the deepest layers enhance the model's capacity to generalize.

The following figure illustrates the evolution of training and validation losses during this stage:

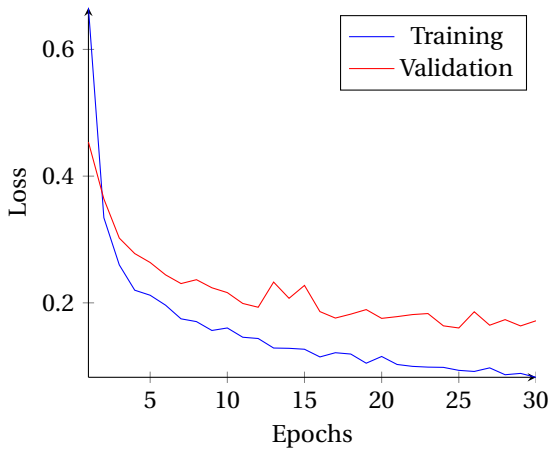


Figure 4.19: Training and validation loss curves for the model during fine-tuning with Stages 8-9 unfrozen.

Test validation is then conducted to assess the model's performance on previously unseen data. The average inference time is measured at 45.65 milliseconds per image, showing an improvement in efficiency compared

to the base configuration.

With the predictions generated, the confusion matrix is presented below to provide a visual summary of the model's classification performance at this stage:

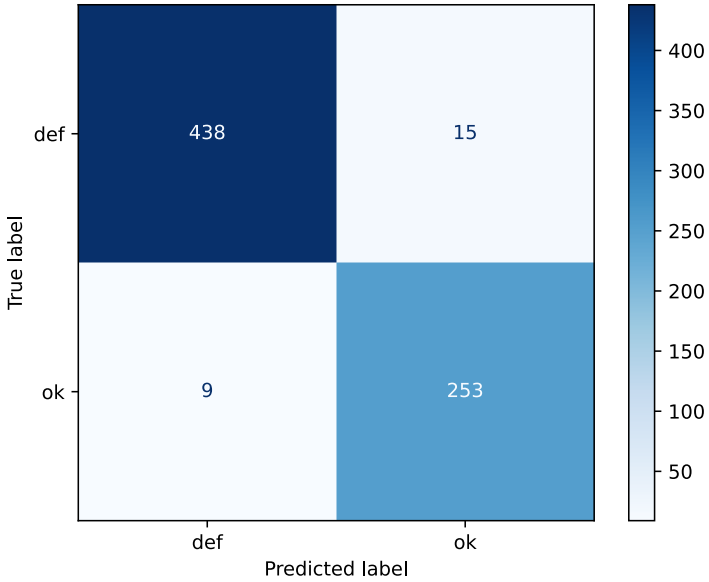


Figure 4.20: Confusion matrix summarizing test classification results after fine-tuning Stages 8-9 of the YOLO backbone.

The following table presents the evaluation metrics computed on the test dataset:

Class	Precision	Recall	F1-score	Support
Defective	0.9799	0.9669	0.9733	453
Ok	0.9440	0.9656	0.9547	262
Macro Average	0.9619	0.9663	0.9640	—
Weighted Average	0.9667	0.9664	0.9665	—
Accuracy: 96.64%				

Table 4.8: Precision, recall, and F1-score metrics for each class, with overall accuracy on the test dataset after fine-tuning Stages 8-9.

These results are consistent with the trends observed during the validation phase. The performance improvements confirm that unfreezing stages 8–9 enhances the model's ability to generalize, resulting in a 29.41% reduction in test classification errors compared to the base model. However, despite this gain, signs of overfitting are present, indicating that the model is specializing too closely to the training data. This suggests a delicate balance between performance improvement and model robustness that must be carefully monitored as additional layers are unfrozen.

4.2.5.3. Training: Stages 6–7

Continuing with the progressive unfreezing strategy, all layers from the beginning of Stage 6 to the end of Stage 7 are set as trainable for the next fine-tuning phase. This adjustment allows a deeper portion of the backbone to adapt to the specific patterns present in the metallurgical inspection dataset.

The average training time per epoch slightly increases to 22.5 seconds, remaining within an efficient range given the added model complexity.

The following figure shows the training and validation loss curves over the epochs for this stage, providing insight into the model's convergence behavior and potential overfitting dynamics:

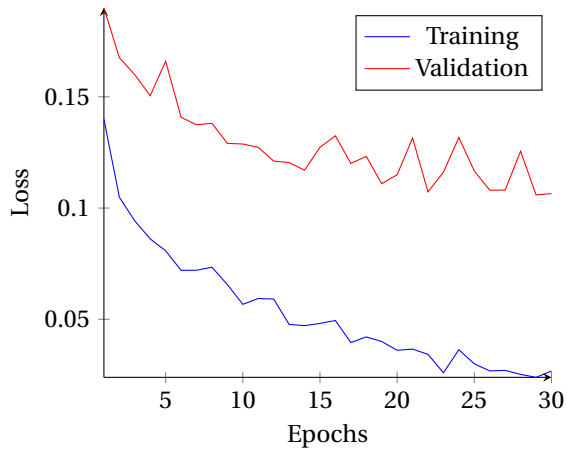


Figure 4.21: Training and validation loss curves for the model during fine-tuning with Stages 6-7 unfrozen.

As expected, continued fine-tuning of deeper layers leads to increased signs of overfitting, evidenced by a growing absolute difference between the training and validation losses. Nevertheless, the model continues to show performance improvements, suggesting that the additional trainable capacity is still beneficial in refining its decision boundaries.

The model is then validated on the test dataset, where the average inference time is measured at 46.13 milliseconds per image, consistent with prior configurations.

The following confusion matrix summarizes the model's classification results on the test set:

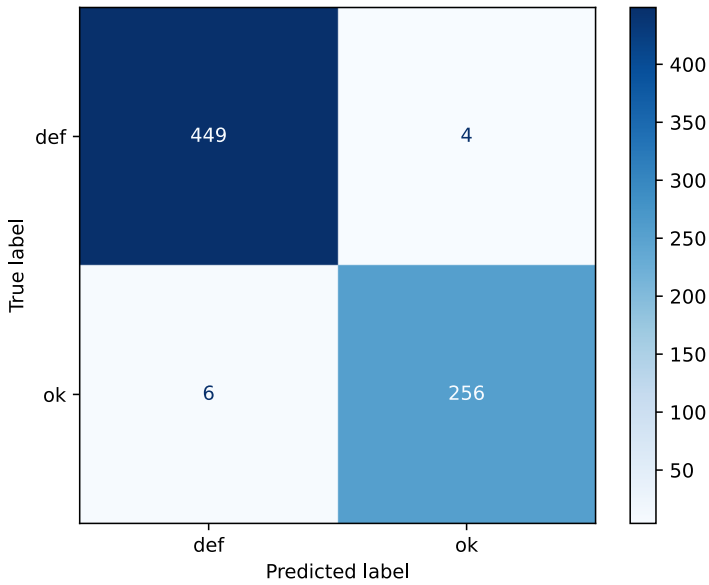


Figure 4.22: Confusion matrix summarizing test classification results after fine-tuning Stages 6-7 of the YOLO backbone.

The following table reports the evaluation metrics obtained from the test dataset:

Class	Precision	Recall	F1-score	Support
Defective	0.9868	0.9912	0.9890	453
Ok	0.9846	0.9771	0.9888	262
Macro Average	0.9857	0.9841	0.9849	—
Weighted Average	0.9860	0.9860	0.9860	—
Accuracy: 98.60%				

Table 4.9: Precision, recall, and F1-score metrics for each class, with overall accuracy on the test dataset after fine-tuning Stages 6-7.

The results demonstrate a notable increase in accuracy, accompanied by a 58.33% reduction in classification errors compared to the previous fine-tuning stage. This confirms that enabling deeper layers—specifically stages 6–7—to adapt contributes positively to the model's ability to capture relevant features. However, the growing divergence between training and validation performance continues to signal the onset of overfitting, which must be carefully balanced against accuracy gains.

4.2.5.4. Training: Stages 4–5

Continuing with the fine-tuning strategy, all layers from the beginning of Stage 4 to the end of Stage 5 are set as trainable in this phase of training. By doing so, a larger portion of the YOLO backbone becomes involved in adapting to the specific characteristics of the classification task.

The average training time per epoch slightly increases again, reaching 22.8 seconds, consistent with the gradual rise in computational cost as deeper layers are unlocked.

The following plot displays the training and validation loss curves, allowing assessment of convergence stability and potential overfitting:

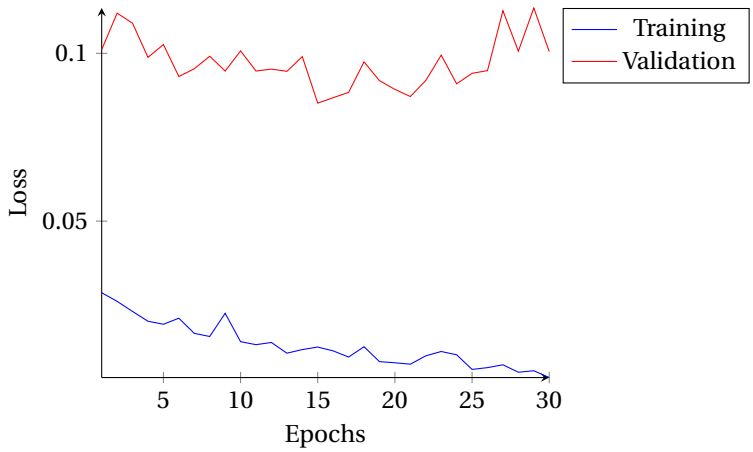


Figure 4.23: Precision, recall, and F1-score metrics for each class, with overall accuracy on the test dataset after fine-tuning Stages 4-5.

As additional layers are unfrozen and the model is trained further, overfitting becomes more pronounced. The gap between training and validation loss widens, and in the final epochs, the validation loss exhibits an upward trend, indicating that the model starts to lose generalization capability as it increasingly fits the training data.

The model is then evaluated on the test dataset, yielding an average inference time of 46.27 milliseconds per image, which remains consistent with previous fine-tuning stages.

The corresponding confusion matrix is shown below to visualize the model's classification performance on unseen samples:

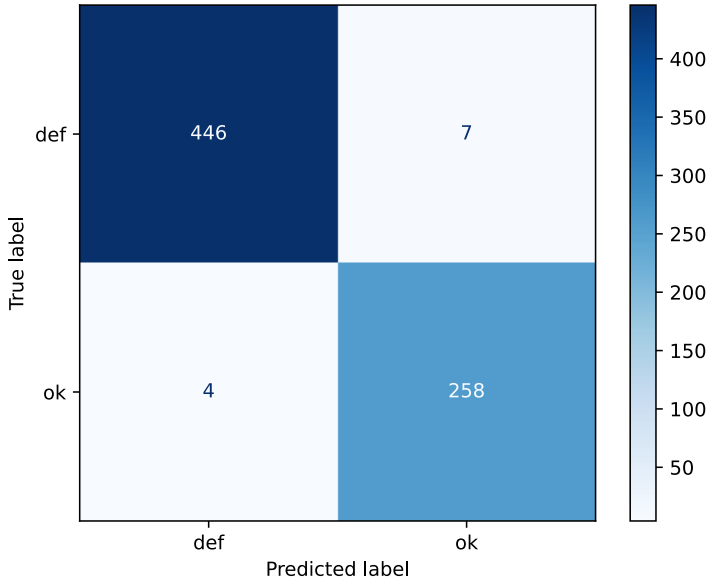


Figure 4.24: Confusion matrix summarizing test classification results after fine-tuning Stages 4-5 of the YOLO backbone.

The following table presents the evaluation metrics derived from the test dataset:

Class	Precision	Recall	F1-score	Support
Defective	0.9911	0.9845	0.9878	453
Ok	0.9736	0.9847	0.9791	262
Macro Average	0.9823	0.9846	0.9835	—
Weighted Average	0.9847	0.9846	0.9846	—
Accuracy: 98.46%				

Table 4.10: Precision, recall, and F1-score metrics for each class, with overall accuracy on the test dataset after fine-tuning Stages 4-5.

In this case, however, the results show a 10% decrease in performance compared to the previous fine-tuning stage. Given the small size of the test set, such variations are plausible, and individual samples can significantly influence the metrics. While the model appeared to generalize better during validation, its behavior on unseen test data reveals that unfreezing stages 4-5 did not result in a meaningful improvement. This outcome highlights the need for cautious layer-wise tuning, especially in low-data regimes where additional trainable parameters may increase the risk of overfitting without consistent performance gains.

4.2.5.5. Training: Stages 2-3

The fine-tuning process continues by unfreezing all layers from the beginning of Stage 2 to the end of Stage 3, thereby deepening the level of adaptation within the YOLO-derived backbone.

The average epoch duration remains aligned with previous stages, measured at 21.83 seconds per epoch. This stability in execution time indicates that the model's complexity increase is moderate, even with additional layers being made trainable.

The training and validation loss curves for this stage are displayed below, offering insight into the training dynamics:

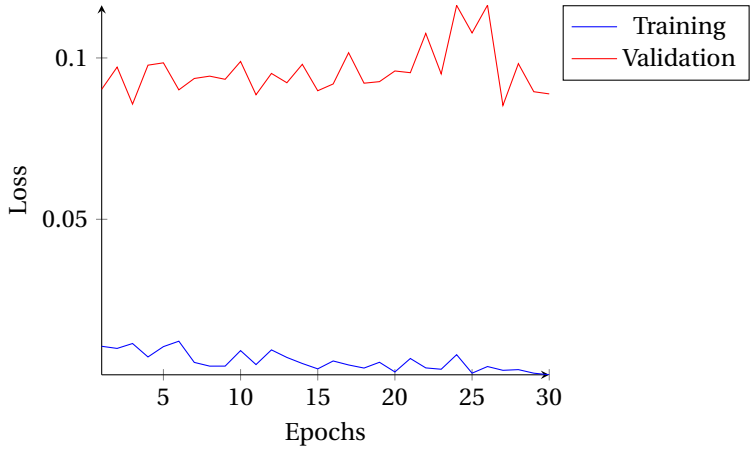


Figure 4.25: Training and validation loss curves for the model during fine-tuning with Stages 2-3 unfrozen.

As observed from the loss curves, no significant improvement is achieved in terms of validation metrics. The model exhibits a slight refinement in the later epochs, where the validation loss reaches marginally lower values than the plateau seen during the main training phases.

The average inference time recorded during test evaluation is 49.17 milliseconds per image, indicating a modest increase in computational cost.

The corresponding confusion matrix is displayed below to visualize classification outcomes for this fine-tuning stage:

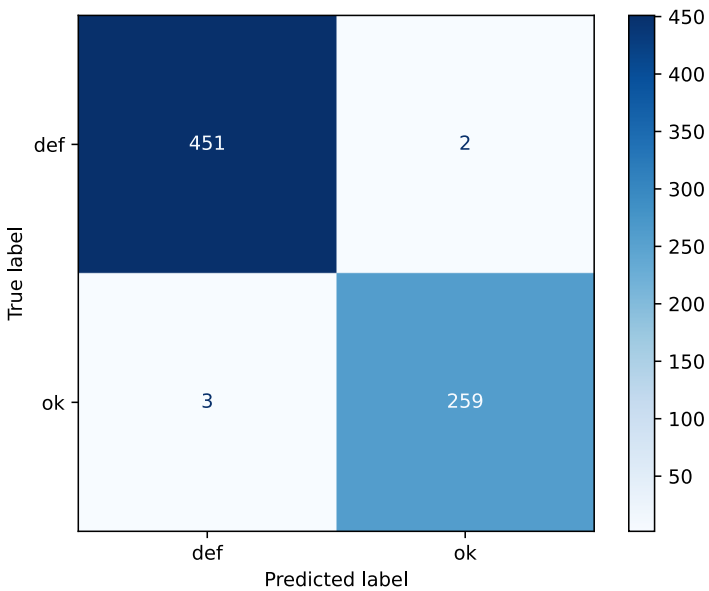


Figure 4.26: Confusion matrix summarizing test classification results after fine-tuning Stages 2-3 of the YOLO backbone.

The following evaluation metrics are computed for this training stage:

Class	Precision	Recall	F1-score	Support
Defective	0.9934	0.9956	0.9945	453
Ok	0.9923	0.9885	0.9904	262
Macro Average	0.9929	0.9921	0.9925	—
Weighted Average	0.9930	0.9930	0.9930	—
Accuracy: 99.30%				

Table 4.11: Precision, recall, and F1-score metrics for each class, with overall accuracy on the test dataset after fine-tuning Stages 2-3.

When compared with the best-performing configuration so far (unfreezing stages 6–7), this setup yields a 50% reduction in classification errors, which is surprisingly significant given the marginal improvements observed in training and validation losses. This outcome highlights a recurring theme: due to the relatively small number of misclassified samples, test metrics can be strongly influenced by just a few critical cases, rather than by overall training behavior.

Nevertheless, the results confirm that further unfreezing of YOLO backbone layers continues to enhance the model's performance, validating the effectiveness of progressive fine-tuning even when apparent training metrics offer only slight gains.

4.2.5.6. Training: Stages 0-1

The final configuration involves completely unfreezing the YOLO backbone, allowing all layers—both shallow and deep—to be fine-tuned. It is important to note that this strategy diverges from the classical definition of transfer learning, which typically aims to retain the learned representations from a pre-trained model while only adjusting the final layers to a new task. Despite this, full unfreezing has been explored here to assess whether further performance gains can be achieved by allowing the entire network to adapt to the specific defect classification task.

The average training time per epoch in this setting remains consistent with previous stages, registering at 23.7 seconds per epoch, indicating a manageable computational cost despite the added complexity.

The following figure shows the evolution of the training and validation losses throughout the full fine-tuning process:

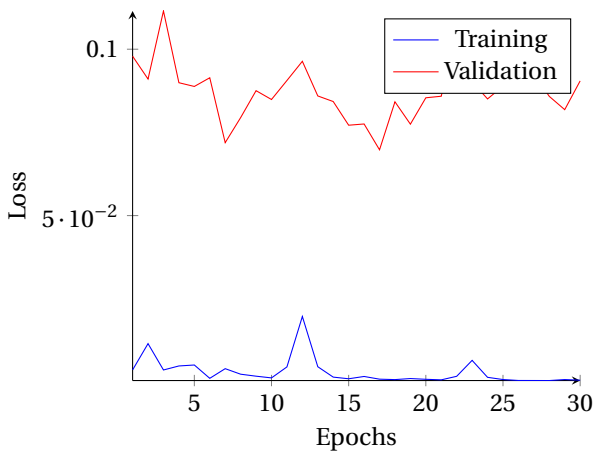


Figure 4.27: Training and validation loss curves for the model during fine-tuning with Stages 0-1 unfrozen.

The training dynamics follow a similar pattern observed in the previous stage. The validation loss exhibits a horizontal oscillation without notable downward trends, while the training loss continues to decrease steadily. This behavior is a clear indicator of overfitting, suggesting that the model is fitting increasingly well to the training data at the expense of generalization.

The average inference time recorded during test evaluation is 46.02 milliseconds per image, consistent with earlier stages and indicating no major degradation in runtime performance despite the full unfreezing.

The resulting confusion matrix is displayed below, offering a visual summary of the model's classification outcomes on the test dataset:

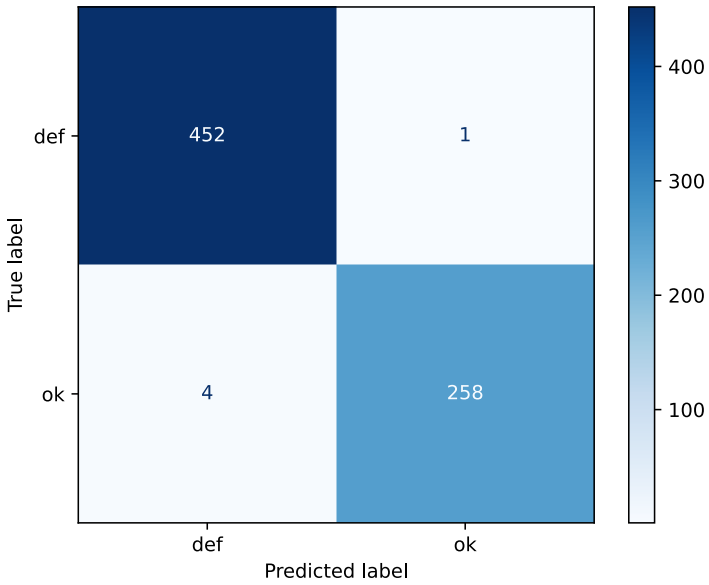


Figure 4.28: Confusion matrix summarizing test classification results after fine-tuning Stages 0-1 of the YOLO backbone.

The evaluation metrics for this final configuration are summarized below:

Class	Precision	Recall	F1-score	Support
Defective	0.9912	0.9978	0.9945	452
Ok	0.9962	0.9848	0.9904	263
Macro Average	0.9937	0.9913	0.9925	—
Weighted Average	0.9930	0.9930	0.9930	—
Accuracy: 99.30%				

Table 4.12: Precision, recall, and F1-score metrics for each class, with overall accuracy on the test dataset after fine-tuning Stages 0-1.

As shown, the number of misclassified images remains unchanged compared to the previous fine-tuning stage, indicating that no further improvement is achieved by fully unfreezing the entire backbone. This plateau in performance reinforces the idea that increasing model capacity beyond a certain point no longer yields meaningful gains, especially when working with a relatively limited dataset.

This concludes the training and fine-tuning phase of the model composed of a YOLO backbone with a custom Keras classification head. While the model demonstrates strong performance overall, it consistently exhibits signs of overfitting throughout training. This is largely attributed to the high capacity of the backbone, which allows it to extract complex features very rapidly. As a result, the model begins to over-specialize on the training data even in the earliest epochs, highlighting the challenges of employing very deep architectures on moderate-sized datasets in industrial scenarios.

4.3. Grad-CAM Demonstration Results

The following section presents the results obtained from applying the Grad-CAM algorithm using the model based on the pre-trained DenseNet121 architecture. Although the primary objective of this example is not to achieve perfect classification accuracy, it remains essential to monitor key performance metrics throughout the training process. Ensuring that the model demonstrates adequate performance is crucial, as the reliability and interpretability of the Grad-CAM heatmaps directly depend on the model's ability to make meaningful predictions.

Consequently, in line with the methodology established in previous experiments, the relevant metrics—such as accuracy, loss, precision, and recall—are systematically tracked and analyzed during the training phase. This careful evaluation serves a dual purpose: it validates the functional integrity of the model and guarantees that the generated activation maps faithfully reflect the true discriminative features learned by the network.

The results reported below therefore provide not only the visual outputs of the Grad-CAM algorithm, but also an assessment of the model's quantitative performance, establishing a comprehensive framework for interpreting both the qualitative and quantitative aspects of the experiment.

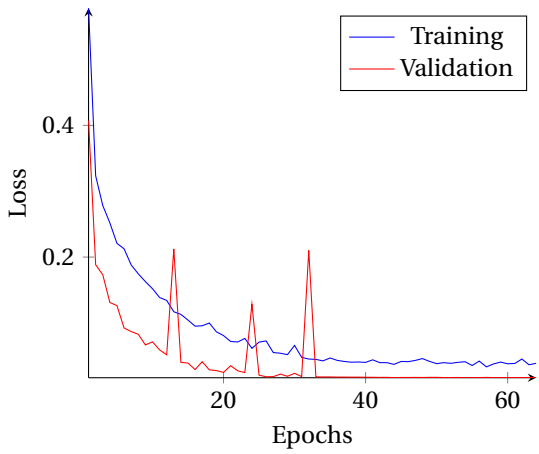


Figure 4.29: Training and validation loss curves for the DenseNet121 model used in the Grad-CAM demonstration. The model exhibits stable convergence, with early stopping applied to prevent overfitting and ensure generalisation.

As observed from the training curves, the model exhibits relatively rapid convergence. In this experiment, early stopping mechanisms have been employed to prevent overfitting and to maintain continuous oversight of the model's performance, as the primary focus is on evaluating the effectiveness of this specific architecture rather than conducting direct comparisons with alternative models. The use of early stopping allows the training process to terminate automatically once the validation performance ceases to improve, thereby ensuring efficient utilization of computational resources and preserving the generalizability of the model.

With respect to the average training time per epoch, there is a noticeable increase compared to previous experiments. This substantial rise in computational cost is attributable to the significantly higher number of trainable parameters present in the DenseNet121 architecture. The increased model complexity, while beneficial for learning intricate data representations, inherently results in longer training times. In this case, the mean epoch duration is recorded at 117.41 seconds, reflecting the computational demands associated with deep, high-capacity neural networks.

Having established the training behavior and computational considerations, the next step involves evaluating the model's performance on the test dataset. This analysis will provide further insights into the model's generalization capability and the validity of the activation maps generated via the Grad-CAM algorithm.

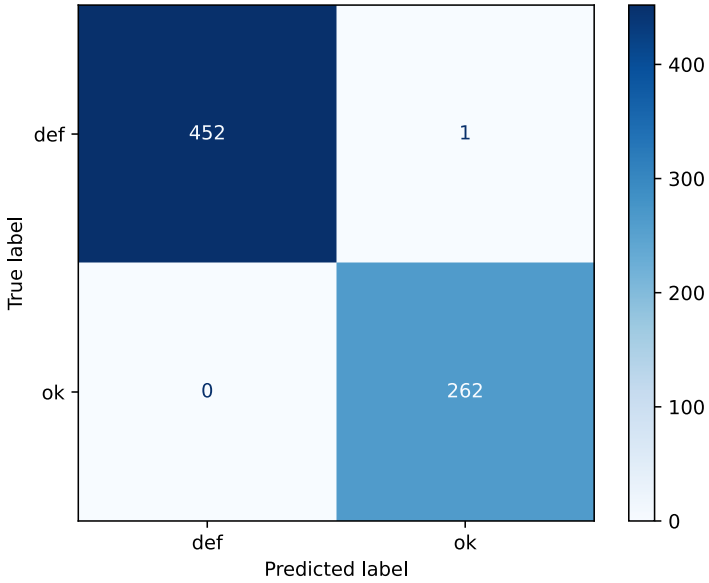


Figure 4.30: Confusion matrix and classification report for the DenseNet121 model on the test dataset. The model achieves high precision, recall, and F1-scores across both classes, indicating strong generalisation and reliable performance for Grad-CAM interpretability.

Finally, the results are presented using the standard evaluation metrics established in previous experiments. By verifying values such as accuracy, precision, recall, and F1-score, a clear picture of the model's strengths and potential limitations is obtained, thereby supporting the interpretability and reliability of the subsequent Grad-CAM visualizations.

Class	**Precision**	**Recall**	**F1-score**	**Support**
Defective	1.0000	0.9978	0.9989	453
Ok	0.9962	1.0000	0.9981	262
Macro Average	0.9981	0.9989	0.9985	—
Weighted Average	0.9986	0.9986	0.9986	—
Accuracy: 99.86%				

Table 4.13: Classification performance of the DenseNet121 model on the test dataset. The reported precision, recall, and F1-score values demonstrate excellent predictive capability across both classes, supporting the reliability of subsequent Grad-CAM visualisations.

Subsequently, an analysis of the heatmaps generated by the Grad-CAM algorithm is conducted. For this purpose, the same test dataset previously used to evaluate the model's quantitative performance is employed. The Grad-CAM technique is systematically applied to each individual image within the test set, thereby enabling a comprehensive visual assessment of the model's interpretability and its ability to localize relevant features associated with defects. Selected examples of these Grad-CAM outputs are presented below to illustrate the range and effectiveness of the method.

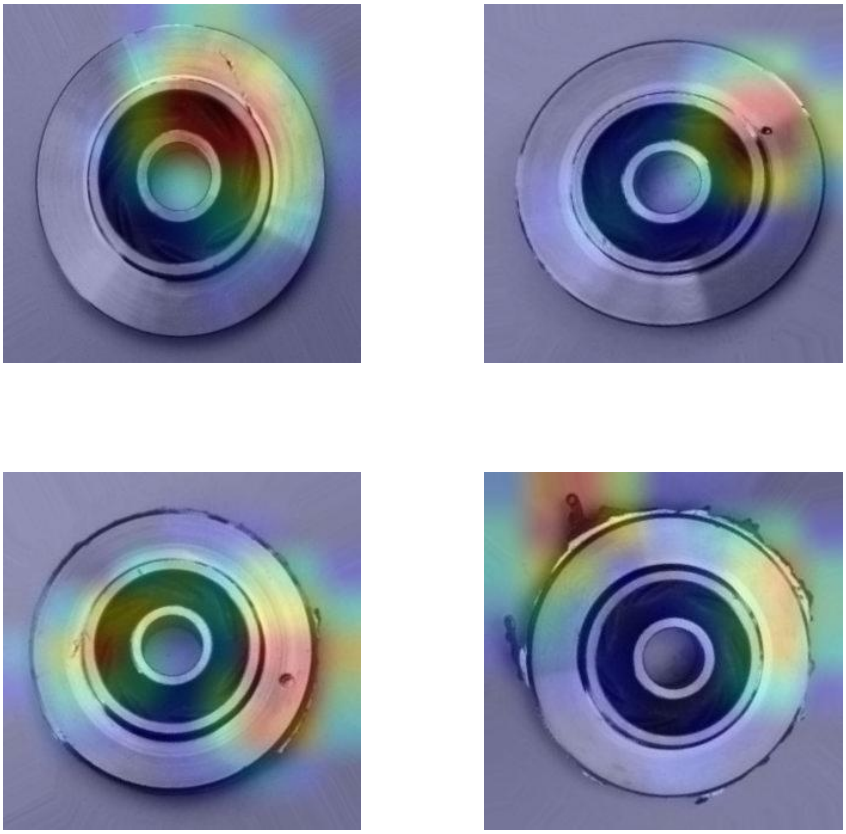


Figure 4.31: Grad-CAM heatmaps for selected test images exhibiting evident surface defects. The model consistently highlights the most relevant defective regions, closely aligning with typical human inspection criteria.

In the initial set of results, the generated heatmaps reveal a striking similarity to human inspection behavior. The Grad-CAM algorithm consistently highlights regions of the image that correspond to the most obvious defects—areas that a human inspector would naturally focus on when assessing the presence of faults. This alignment between the model's attention and intuitive human reasoning strongly supports the credibility and practical value of the Grad-CAM visualizations. The interpretability afforded by these activation maps not only enhances trust in the automated system, but also provides a transparent means of validating the model's decision-making process in the context of defect detection.

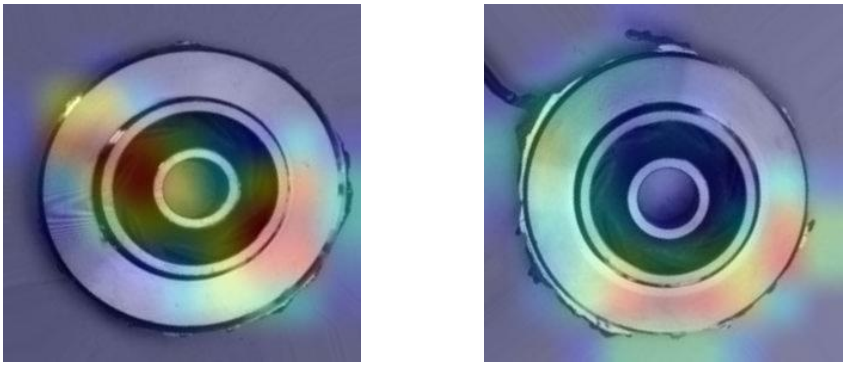


Figure 4.32: Grad-CAM heatmaps for samples with subtle or distributed defects. The model displays a broader distribution of attention, indicating sensitivity to multiple minor anomalies rather than a single dominant feature.

In cases where the defects are less apparent or are distributed throughout the image, the Grad-CAM heatmaps exhibit a different pattern. The resulting visualizations often lack a single, clearly defined region of high ac-

tivation. Instead, the attention of the model appears more diffuse, with the heatmap highlighting multiple areas of the image where potential imperfections are present. This phenomenon is particularly common in instances where several minor defects coexist, rather than a single prominent fault.

The model's indecision, as reflected in the Grad-CAM output, suggests that its classification is influenced by the cumulative effect of several subtle anomalies rather than by a dominant, localized feature. As a result, the heatmaps do not converge on a specific area, but instead indicate a broader distribution of attention. This behavior aligns with the inherent complexity of such cases, where the presence of multiple, scattered defects complicates both automated and manual inspection. It also underscores the utility of Grad-CAM in providing insight into the internal reasoning of the model, especially in challenging scenarios where the classification outcome is less straightforward.

It is important to acknowledge that, as noted by the creator of the dataset, the process of identifying defects in these specific types of components often requires a degree of specialized experience. A human inspector without prior expertise in detecting such faults may encounter difficulties in accurately recognizing or classifying the defects based solely on the visual information contained in the images. This limitation highlights the inherent subjectivity present in manual inspection tasks and emphasizes the potential value of automated approaches.

In conclusion, while not all heatmaps appear entirely self-explanatory, nor do they always encompass every region that might be considered defective from a human perspective, the overall results obtained from the Grad-CAM experiment are notably representative. The visualizations produced demonstrate a strong alignment between the areas highlighted by the computer vision model and the regions that would intuitively draw the attention of a human inspector. This outcome not only underscores the effectiveness of the model in mimicking human diagnostic processes but also establishes the viability of Grad-CAM as a tool for interpreting the behavior of the final convolutional layers within deep neural networks. Furthermore, these results illustrate the algorithm's potential for facilitating the debugging and refinement of model decision-making, thereby enhancing the transparency and trustworthiness of automated defect detection systems in industrial applications.

5

Conclusions

This work demonstrates the transformative potential of deep learning in advancing visual inspection processes within the metallurgical industry. By integrating convolutional neural networks (CNNs) and state-of-the-art object detection frameworks such as YOLOv8, the proposed methodology surpasses the capabilities of traditional rule-based and manually supervised inspection methods. The transition to AI-driven systems not only enhances detection accuracy and operational efficiency but also aligns with the foundational pillars of Industry 4.0. The incorporation of intelligent vision systems into manufacturing workflows supports the paradigm of smart factories, enabling real-time monitoring, data-driven decision-making, and adaptive quality control. This technological shift reflects a broader industrial movement towards automation, connectivity, and digitalisation, positioning the proposed solution at the forefront of modern manufacturing strategies.

The methodological framework developed throughout this thesis reflects a deliberate effort to explore the breadth and depth of modern computer vision strategies in the context of industrial defect detection. Multiple modelling paradigms are examined, including pure classification, semantic segmentation, and hybrid approaches that sequentially integrate both. The experimentation begins with YOLOv8-based classification and segmentation models, which serve as a robust and efficient baseline due to their unified architecture, real-time performance, and support for transfer learning. In parallel, TensorFlow and Keras are employed to build custom classification pipelines, allowing for greater architectural control and interpretability, particularly when tailored to the specific nature of metallurgical defect types.

This methodological diversity not only strengthens the empirical foundation of the work but also highlights the capacity for modular integration between frameworks. The process of combining YOLOv8 for initial defect localisation with Keras-based classifiers exemplifies a hybrid design that benefits from spatial precision and semantic discrimination. Training strategies are adapted accordingly to maximise generalisation and mitigate overfitting. These include data augmentation techniques, such as geometric and photometric transformations, as well as advanced practices like transfer learning, fine-tuning, and manual backbone extraction. In particular, the reuse of YOLOv8 backbones within Keras models introduces a complex but effective mechanism for knowledge transfer, requiring manual conversion of convolutional, normalization, and composite layers to ensure architectural compatibility.

The integration of multiple frameworks, training protocols, and data preprocessing strategies illustrates a rigorous and comprehensive approach to model development. This methodological robustness provides a strong basis for comparative analysis across different architectures and facilitates reproducibility, which is essential for real-world deployment in industrial contexts.

The experimental design of this thesis extends beyond conventional training routines by incorporating advanced techniques that prioritise both performance and architectural innovation. A particularly novel contribution involves the manual integration of YOLOv8 backbones into Keras-based classification pipelines. This process entails a meticulous conversion of learned weights and architectural components, such as convolutional blocks, batch normalization layers, and composite modules including bottlenecks, cross-stage partial layers (C2f), and spatial pyramid pooling units (SPPF). The success of this manual transfer demonstrates a

deep understanding of the internal mechanisms of both frameworks and reflects a deliberate effort to optimise model reusability across ecosystems.

Additionally, the application of Gradient-weighted Class Activation Mapping (Grad-CAM) provides a means of interpreting the learned representations within the classification models. By visualising activation regions that influence class predictions, Grad-CAM enhances the transparency of the decision-making process and supports the identification of possible sources of bias or overfitting. This commitment to model explainability reinforces the practical value of the proposed solutions, particularly in industrial settings where trust and interpretability are critical for adoption.

Overall, the experimentation not only validates the functional performance of the models but also reveals an iterative design philosophy that prioritises adaptability, precision, and transparency. The combination of architectural modification, interpretability tools, and hybrid design principles results in a technically sophisticated and practically relevant contribution to the field of computer vision in manufacturing.

Beyond model architecture and training strategies, this thesis addresses practical aspects essential for the real-world deployment of computer vision systems in industrial environments. The workflow integrates tools and platforms that are accessible and scalable, including Roboflow for dataset management and annotation, and Google Colab for model training and experimentation. This configuration allows for the effective handling of large image datasets and facilitates rapid prototyping without the need for dedicated hardware, making the approach suitable for small- and medium-scale industrial settings.

Although the discussion of deployment remains at a high level, critical factors such as model quantisation and inference on edge devices are considered. These considerations acknowledge the necessity of efficient and lightweight models that can operate in real-time within resource-constrained environments, such as on production lines or embedded systems integrated with programmable logic controllers (PLCs). The exploration of these topics, even in a preliminary form, demonstrates awareness of the limitations often encountered in transitioning from laboratory-scale models to production-ready systems.

By incorporating these deployment-aware practices, the thesis ensures that the proposed solutions are not only technically sound but also pragmatically aligned with the operational constraints and performance demands of modern manufacturing environments.

The performance evaluation confirms the effectiveness of the proposed models, with the highest accuracy obtained from the YOLOv8 classification model, the baseline Keras classifier, and the Keras classifier trained on cropped and preprocessed images. These architectures outperform the more complex hybrid design that integrates YOLOv8 backbones into Keras classifiers, both in terms of simplicity and predictive performance. Evaluation metrics such as accuracy, precision, recall, Intersection over Union (IoU), and F1-score provide a comprehensive assessment of each model's behaviour, confirming the robustness of the selected approaches.

The experimental results underscore the importance of a rigorous validation framework. Careful dataset partitioning and consistent use of relevant metrics ensure that the models generalise well across various defect types. Data augmentation further enhances the diversity of training samples, while techniques such as transfer learning and proper class weighting—automatically handled through the training framework—ensure effective learning even when certain defect types are less frequent. These findings demonstrate that the proposed models achieve high predictive accuracy while maintaining efficiency and ease of deployment.

These findings collectively demonstrate that the proposed pipeline not only achieves high predictive accuracy but also maintains strong generalisation across various defect categories. This capacity to perform reliably under realistic conditions reinforces the applicability of the models in dynamic industrial environments where defect variability is common and model robustness is essential.

The work presented in this thesis aligns closely with broader societal and industrial objectives, particularly those related to safety, sustainability, and quality assurance in manufacturing. By automating the detection of surface defects through advanced computer vision techniques, the proposed system contributes to reducing human involvement in hazardous inspection tasks. This shift not only improves occupational safety but

also minimises the risk of human error, thereby enhancing overall product reliability.

From an environmental perspective, early and accurate defect detection enables more efficient use of raw materials by reducing the production of defective components, which would otherwise be scrapped or reprocessed. This contributes to the sustainability of manufacturing processes by lowering material waste and energy consumption. In doing so, the approach supports key tenets of sustainable development and responsible industrial innovation.

Moreover, the integration of intelligent inspection systems aligns with the strategic goals of smart manufacturing and Industry 4.0, promoting the adoption of autonomous and interconnected production systems. The ability to ensure high product quality through scalable and interpretable AI models strengthens competitiveness and compliance with increasingly stringent regulatory standards. Thus, the proposed framework not only addresses technical challenges but also contributes to a safer, more sustainable, and economically viable manufacturing landscape.

Despite the promising results and methodological innovations, this thesis acknowledges several limitations that constrain the current implementation and suggest directions for future research. A notable technical challenge involves the manual integration of YOLOv8 backbones into Keras pipelines. This process requires detailed weight conversion and architectural mapping, increasing complexity and limiting scalability. Although this approach is explored as a proof of concept, the additional implementation burden, combined with the fact that it does not outperform simpler architectures, suggests limited practical value in its current form.

The pre-processing pipeline also includes manual steps, such as annotation validation and crop selection. While these interventions improve dataset quality and enhance model focus, they reduce the overall level of automation and may hinder scalability in industrial-scale applications. Moreover, while deployment considerations such as quantisation and edge inference are acknowledged, their implementation remains conceptual, leaving room for future work in hardware-specific optimisation and real-time system integration.

These limitations point to several promising research directions, including the development of automated backbone conversion utilities, streamlined annotation workflows, and more comprehensive deployment pipelines tailored for industrial settings.

Future Work

While the current work establishes a solid foundation for automated visual inspection in metallurgical manufacturing, several avenues remain open for future development. The models and methodologies explored in this thesis demonstrate both technical feasibility and industrial relevance; however, further enhancements could improve scalability, adaptability, and integration into real-world production environments.

Many of the techniques employed—such as manual backbone conversion, separate model components, and offline training strategies—serve as effective prototypes but also reveal opportunities for refinement and automation. As the complexity and variability of industrial processes grow, so too does the need for more flexible, generalisable, and deployment-ready vision systems.

The following subsections outline key directions for future research and development, focusing on architectural automation, system integration, deployment optimisation, and long-term adaptability. These proposals aim to expand the scope and impact of the current work, ensuring that computer vision-based defect detection continues to evolve in alignment with industrial needs and technological advancements.

Automation of Backbone Conversion Pipelines

The manual transfer of YOLOv8 backbones into Keras-based classification models represents one of the most intricate and labor-intensive components of this work. This task involves reconstructing layer-by-layer equivalents of convolutional, batch normalization, and composite modules—such as bottlenecks, C2f blocks, and SPPF structures—while preserving the integrity of pretrained weights. Although this process enables the

reuse of learned representations in a different training framework, it introduces significant complexity and is inherently prone to human error.

Automating this conversion would substantially improve the scalability and reproducibility of similar research pipelines. By developing tools capable of parsing, mapping, and exporting model architectures and parameters across deep learning libraries (e.g., from PyTorch to TensorFlow/Keras), future work could reduce the barrier to integrating pretrained components into customised architectures. Such tools would not only accelerate experimentation but also enable more systematic comparisons between model variants, ultimately fostering a more modular and interoperable ecosystem for industrial computer vision development.

Fully Integrated End-to-End Pipeline

The current approach separates segmentation and classification into distinct stages, often requiring intermediate data handling, manual preprocessing, and sequential model invocation. While effective for prototyping and comparative analysis, this design introduces latency and limits the responsiveness of the system in real-time industrial environments. Future work should explore the development of a fully integrated, end-to-end pipeline that seamlessly combines defect localisation, classification, and decision-making within a single architecture.

Such a pipeline could be realised through the use of modular network designs or multi-task learning frameworks, which enable a shared feature extractor to support multiple downstream tasks. This architectural consolidation would not only streamline inference but also reduce redundancy in model parameters, potentially improving generalisation and computational efficiency. Furthermore, integrating post-processing logic—such as thresholding, defect severity assessment, or decision rules—within the same model graph could facilitate real-time deployment and simplify the system's integration into production workflows. Advancing toward this level of architectural unification represents a critical step in translating deep learning models from research prototypes into industrial-grade solutions.

Deployment on Industrial Hardware

Although the models developed in this thesis have been trained and evaluated using cloud-based platforms such as Google Colab, practical deployment in industrial environments demands execution on embedded or resource-constrained hardware. Target platforms may include NVIDIA Jetson modules, industrial PCs, or FPGA-based systems, all of which offer varying degrees of computational capacity, power efficiency, and integration capabilities for edge-level processing.

To meet the strict latency and reliability requirements of real-time production lines, model optimisation techniques must be applied. These include quantisation to reduce numerical precision and memory footprint, pruning to eliminate redundant parameters, and conversion to hardware-specific runtimes such as TensorRT for accelerated inference on NVIDIA devices. Such optimisations enable the deployment of deep learning models without sacrificing predictive performance, while also ensuring compatibility with embedded controllers and industrial communication protocols.

Pursuing this line of development would bring the system closer to production readiness, enabling continuous defect inspection directly on the shop floor, independent of cloud infrastructure. This transition is essential for achieving low-latency decision-making, high availability, and seamless integration into existing industrial automation architectures.

Active Learning and Online Adaptation

The static training paradigm adopted in this thesis relies on a fixed dataset and offline learning cycles, which limits the model's ability to adapt to new defect types or evolving production conditions. In dynamic industrial environments, where defect characteristics may shift over time due to changes in materials, machinery, or processing parameters, maintaining high model performance requires continuous learning capabilities.

Incorporating active learning frameworks presents a promising solution. By enabling the model to identify

and query instances of high uncertainty, such systems can solicit human annotations only when necessary, thereby reducing the annotation burden while focusing effort where it is most impactful. This strategy facilitates a more efficient use of expert time and accelerates the refinement of the dataset.

Complementing active learning with online adaptation techniques would further enhance robustness. Online learning allows the model to incrementally update its parameters as new labelled data becomes available, without requiring full retraining. Together, these mechanisms would support a self-improving inspection system capable of maintaining accuracy and relevance in the face of changing operational conditions, making the solution more resilient and cost-effective in long-term deployments.

Synthetic Data Generation and Domain Randomisation

A key limitation in industrial defect detection is the scarcity of annotated data for rare or emerging defect types. Collecting and labelling such samples is often time-consuming, costly, and sometimes infeasible due to the unpredictability of certain failure modes. To address this, future work could explore the use of synthetic data generation techniques to augment the training dataset and improve model robustness.

Synthetic imagery can be produced using a variety of approaches, including physics-based simulation, 3D rendering, and generative adversarial networks (GANs). These techniques offer the potential to create controlled, high-fidelity examples of specific defect types under diverse lighting, texture, and background conditions. When combined with domain randomisation—where non-essential features such as colour, orientation, or texture are deliberately varied—models can be trained to focus on relevant structural features and generalise more effectively to real-world data.

By expanding the variability and volume of training data in this way, synthetic augmentation has the potential to mitigate overfitting, enhance performance on edge cases, and increase resilience to distributional shifts. This direction holds particular promise for improving the detection of rare defects, where real examples may be limited or entirely unavailable during training.

Integration with Industrial Control Systems

For computer vision-based defect detection systems to achieve practical utility in production environments, seamless integration with existing industrial control infrastructures is essential. Future work should focus on interfacing the proposed models with systems such as programmable logic controllers (PLCs), manufacturing execution systems (MES), and supervisory control and data acquisition (SCADA) platforms. These systems govern real-time decision-making, production scheduling, and quality control, and thus represent critical points of interaction for any automated inspection pipeline.

Achieving this level of integration would require the development of lightweight, real-time APIs capable of transmitting prediction outputs, triggering actions (e.g., rejecting defective parts), and logging results into production databases. Moreover, the system must comply with industrial communication protocols (e.g., OPC UA, Modbus, or MQTT) to ensure compatibility and reliability.

Embedding AI-driven inspection into these control layers would enable immediate feedback and response within the production cycle, significantly enhancing process efficiency and traceability. It would also support closed-loop manufacturing systems, where decisions are continuously informed by live data, contributing to the broader goals of smart manufacturing and operational excellence.

Multimodal Learning

While this thesis focuses exclusively on visual inspection through RGB imagery, many industrial scenarios present defect signatures that are either subtle in the visible spectrum or obscured by surface conditions. To address these limitations, future research could investigate the incorporation of multimodal learning strategies, where visual data is combined with complementary sensor modalities such as acoustic signals, thermal imaging, X-rays, or hyperspectral data.

By integrating diverse information sources, multimodal models can capture a richer representation of the inspected objects, enhancing the system's ability to detect complex or hidden anomalies. For example, thermal patterns might reveal internal stress fractures, while acoustic emission data could indicate structural defects not visible on the surface. Fusing such data streams requires advanced architectures capable of learning cross-modal correlations and handling heterogeneous input formats.

This line of work promises to increase model robustness, particularly in cases where visual features alone are insufficient or ambiguous. Furthermore, multimodal approaches align with the objectives of predictive maintenance and comprehensive quality assurance, enabling deeper insights into material integrity and process stability across the manufacturing lifecycle.

Bibliography

[1] A. Younesi, M. Ansari, M. Fazli, A. Ejlali, M. Shafique, and J. Henkel, "A comprehensive survey of convolutions in deep learning: Applications, challenges, and future trends," 2024.

[2] C. J. Costa, M. Aparicio, S. Aparicio, and J. T. Aparicio, "The democratization of artificial intelligence: Theoretical framework," *Applied Sciences*, vol. 14, no. 18, 2024.

[3] X. Li, W. Wang, L. Sun, B. Hu, L. Zhu, and J. Zhang, "Deep learning-based defects detection of certain aero-engine blades and vanes with ddsc-yolov5s," *Scientific Reports*, vol. 12, 07 2022.

[4] 3HLE, "Image analysis solution for a.i. quality inspection in factories." `https://3hle.ch/project/retina-deep-learning-image-analysis-software/`, 2020. Accessed: 2025-06-06.

[5] A. Saberironaghi, J. Ren, and M. El-Gindy, "Defect detection methods for industrial products using deep learning techniques: A review," *Algorithms*, vol. 16, no. 2, 2023.

[6] J. Walker, "Impact of artificial intelligence adoption on manufacturing efficiency in the united states," *European Journal of Technology*, vol. 8, no. 1, pp. 38–48, 2024.

[7] International Institute of Non Destructive Testing, "Inspection services." `https://iindttraining.com/inspection/`. Accessed: 2025-06-06.

[8] C. Smith and K. Adendorff, "Advantages and limitations of an automated visual inspection system," *The South African Journal of Industrial Engineering*, vol. 5, 01 2012.

[9] J. Bai, D. Wu, T. Shelley, P. Schubel, D. Twine, J. Russell, X. Zeng, and J. Zhang, "A comprehensive survey on machine learning driven material defect detection," *ACM Computing Surveys*, Apr. 2025.

[10] D. K G and G. Sunitha, *Computer Vision and Its Intelligence in Industry*, pp. 119–142. 01 2024.

[11] KFactory, "The industrial revolution: A short history of manufacturing," 2022. Accessed: 2025-06-02.

[12] S.-H. Park, K.-H. Lee, J.-S. Park, and Y.-S. Shin, "Deep learning-based defect detection for sustainable smart manufacturing," *Sustainability*, vol. 14, no. 5, 2022.

[13] J. Yang, S. Li, Z. Wang, H. Dong, J. Wang, and S. Tang, "Using deep learning to detect defects in manufacturing: A comprehensive survey and current challenges," *Materials*, vol. 13, no. 24, 2020.

[14] Wikipedia contributors, "Canny edge detector — wikipedia," 2024. Accessed: 2025-06-02.

[15] Saturn Cloud, "A comprehensive guide to convolutional neural networks — the eli5 way," 2023. Accessed: 2025-06-02.

[16] Z. Ren, F. Fang, N. Yan, *et al.*, "State of the art in defect detection based on machine vision," *International Journal of Precision Engineering and Manufacturing-Green Technology*, vol. 9, pp. 661–691, 2022.

[17] R. Ameri, C.-C. Hsu, and S. S. Band, "A systematic review of deep learning approaches for surface defect detection in industrial applications," *Engineering Applications of Artificial Intelligence*, vol. 130, p. 107717, 2024.

[18] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Learning deep features for discriminative localization," 2015.

[19] Y. Zou and Y. Fan, "An infrared image defect detection method for steel based on regularized yolo," *Sensors*, vol. 24, no. 5, 2024.

[20] L. Alzubaidi, J. Bai, A. Al-Sabaawi, *et al.*, "A survey on deep learning tools dealing with data scarcity: Definitions, challenges, solutions, tips, and applications," *Journal of Big Data*, vol. 10, p. 46, 2023.

[21] Z. Yang, Y. Yu, C. You, J. Steinhardt, and Y. Ma, "Rethinking bias-variance trade-off for generalization of neural networks," 2020.

[22] O. Rainio, J. Teuho, and R. Klén, "Evaluation metrics and statistical tests for machine learning," *Scientific Reports*, vol. 14, no. 1, p. 6086, 2024.

[23] S. Sah, "Machine learning: A review of learning types," 2020. Preprint.

[24] IBM, "Overfitting vs. underfitting," 2020. Accessed: 2025-06-03.

[25] X. Ying, "An overview of overfitting and its solutions," *Journal of Physics: Conference Series*, vol. 1168, p. 022022, 02 2019.

[26] M. Belkin, D. Hsu, S. Ma, and S. Mandal, "Reconciling modern machine-learning practice and the classical bias–variance trade-off," *Proceedings of the National Academy of Sciences*, vol. 116, no. 32, pp. 15849–15854, 2019.

[27] J. T. Barron and J. Malik, "Shape, illumination, and reflectance from shading," 2020.

[28] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1, pp. 886–893 vol. 1, 2005.

[29] scikit-image developers, "Histogram of oriented gradients (hog)," 2024. Accessed: 2025-06-03.

[30] A. Alfalou, M. Jridi, Y. Kortli, and M. Atri, *A comparative study of CFs, LBP, HOG, SIFT, SURF, and BRIEF for security and face recognition.* 04 2018.

[31] Ultralytics, "Ultralytics: Ai vision for the future," 2025. Accessed: 2025-06-01.

[32] F. Chollet, *Deep Learning with Python.* Shelter Island, NY: Manning Publications, 2 ed., 2021.

[33] T. Zhang, X. Zhang, J. Shi, and S. Wei, "Depthwise separable convolution neural network for high-speed sar ship detection," *Remote Sensing*, vol. 11, no. 21, 2019.

[34] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 6, pp. 1137–1149, 2017.

[35] 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 (CVPR)*, June 2016.

[36] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015* (N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, eds.), (Cham), pp. 234–241, Springer International Publishing, 2015.

[37] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," 2021.

[38] M. Jeong, M. Yang, and J. Jeong, "Hybrid-dc: A hybrid framework using resnet-50 and vision transformer for steel surface defect classification in the rolling process," *Electronics*, vol. 13, no. 22, 2024.

[39] J. Terven, D.-M. Córdova-Esparza, and J.-A. Romero-González, "A comprehensive review of yolo architectures in computer vision: From yolov1 to yolov8 and yolo-nas," *Machine Learning and Knowledge Extraction*, vol. 5, no. 4, pp. 1680–1716, 2023.

[40] L. Yao, H. Xu, W. Zhang, X. Liang, and Z. Li, "Sm-nas: Structural-to-modular neural architecture search for object detection," 2019.

[41] IBM, "Data augmentation," 2021. Accessed: 2025-06-03.

[42] A. Boikov, V. Payor, R. Savelev, and A. Kolesnikov, "Synthetic data generation for steel defect detection and classification using deep learning," *Symmetry*, vol. 13, no. 7, 2021.

[43] A. Semitela, M. Pereira, A. Completo, N. Lau, and J. P. Santos, "Improving industrial quality control: A transfer learning approach to surface defect detection," *Sensors*, vol. 25, no. 2, 2025.

[44] G. Onorato, "Bayesian optimization for hyperparameters tuning in neural networks," 2024.

[45] M. Chaabi and M. Hamlich, "A sight on defect detection methods for imbalanced industrial data," *ITM Web of Conferences*, vol. 43, p. 01012, 01 2022.

[46] P. V. Dantas, W. Sabino da Silva, L. C. Cordeiro, *et al.*, "A comprehensive review of model compression techniques in machine learning," *Applied Intelligence*, vol. 54, pp. 11804–11844, 2024.

[47] O. Vermesan, M. Coppola, R. Bahr, R. O. Bellmann, J. E. Martinsen, A. Kristoffersen, T. Hjertaker, J. Breiland, K. Andersen, H. E. Sand, and D. Lindberg, "An intelligent real-time edge processing maintenance system for industrial manufacturing, control, and diagnostic," *Frontiers in Chemical Engineering*, vol. Volume 4 - 2022, 2022.

[48] NVIDIA Corporation, "Jetson orin: Embedded systems for autonomous machines," 2024. Accessed: 2025-06-03.

[49] S. Rustamova and F. Rustamov, "Problems of integrating artificial intelligence with scada systems," *ScienceRise*, pp. 106–117, Dec. 2024.

[50] G. Van Rossum and F. L. Drake, "Python programming language," 2009. Version 3. Available at https://www.python.org/.

[51] R. Inc., "Roboflow: Computer vision dataset management and preprocessing," 2023. Available at https://roboflow.com/.

[52] G. Research, "Google colaboratory," 2023. Available at https://colab.research.google.com/.

[53] Ravirajsinh45, "Real-life industrial dataset of casting product." `https://www.kaggle.com/datasets/ravirajsinh45/real-life-industrial-dataset-of-casting-product`, 2020. Accessed: 2025-06-01.

[54] M. Contributors, "Yolov8 configuration documentation." `https://github.com/open-mmlab/mmyolo/blob/dev/configs/yolov8/README.md`, 2025. Accessed: 2025-06-01.

[55] PyTorch Team, "PyTorch Documentation." `https://docs.pytorch.org/docs/stable/index.html`, 2025. Accessed: 2025-06-01.

[56] TensorFlow Developers, "TensorFlow API Documentation." `https://www.tensorflow.org/api_docs`, 2025. Accessed: 2025-06-01.

[57] U. Contributors, "Configuring padding pixel color in yolo for compatibility with nvidia deepstream." `https://github.com/ultralytics/ultralytics/issues/19209`, 2025. Accessed: 2025-06-01.

[58] 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," *International Journal of Computer Vision*, vol. 128, p. 336–359, Oct. 2019.

[59] Keras Team, "Grad-CAM Class Activation Visualization." `https://keras.io/examples/vision/grad_cam/`, 2020. Accessed: 2025-06-01.

[60] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," 2018.

A

Source Code for Model Training

A.1. YOLOv8 Classification

```
!pip install ultralytics
!pip install scikit-learn

import os
import shutil
from sklearn.model_selection import train_test_split
from ultralytics import YOLO
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay,
    classification_report

def split_train_val(base_dir, val_size=0.2):
    classes = [folder for folder in os.listdir(base_dir) if os.path.isdir(os.path.join(
    base_dir, folder))]

    train_images = []
    val_images = []

    for class_name in classes:
        class_folder = os.path.join(base_dir, class_name)
        images = [os.path.join(class_folder, f) for f in os.listdir(class_folder) if f.
    endswith('.jpeg')]

        if len(images) == 0:
            continue

        train, val = train_test_split(images, test_size=val_size, random_state=42)
        train_images.extend([(img, class_name) for img in train])
        val_images.extend([(img, class_name) for img in val])

    if len(train_images) == 0:
        raise ValueError("No images were selected for the training set.")
    if len(val_images) == 0:
        raise ValueError("No images were selected for the validation set.")

    train_dir = "/content/split/train"
    val_dir = "/content/split/val"

    for class_name in classes:
        os.makedirs(os.path.join(train_dir, class_name), exist_ok=True)
        os.makedirs(os.path.join(val_dir, class_name), exist_ok=True)
    for img, class_name in train_images:
        shutil.move(img, os.path.join(train_dir, class_name, os.path.basename(img)))
    for img, class_name in val_images:
        shutil.move(img, os.path.join(val_dir, class_name, os.path.basename(img)))

```

```
    return train_dir, val_dir


def train_yolo(epochs=100, batch_size=32):
    model = YOLO("yolov8n-cls.pt")
    model.train(data="/content/split", epochs=epochs, batch=batch_size, imgsz = 300, device="0")
    return model

def test_and_generate_matrix(model, test_dir):
    results = model.predict(os.path.join(test_dir, "ok_front")) + model.predict(os.path.join(test_dir, "def_front"))

    y_true = []
    y_pred = []
    inference_time = 0
    i = 0
    for result in results:

      true_label = result.path.split('/')[-2]
      y_true.append(true_label)

      predicted_class = result.probs.top1
      predicted_label = model.names[predicted_class]
      y_pred.append(predicted_label)
      if true_label != predicted_label:
        result.show()
        result.save(f"missclassification_{i}.png")

      inference_time += result.speed['inference']
      i += 1

    inference_time = np.mean(inference_time)
    print(f"Inference time: {inference_time:.2f} ms")

    cm = confusion_matrix(y_true, y_pred, labels=['def_front', 'ok_front'])
    cm_display = ConfusionMatrixDisplay(cm, display_labels=model.names.values())

    cm_display.plot(cmap=plt.cm.Blues)
    plt.show()

    report = classification_report(y_true, y_pred, target_names=model.names.values())
    print(report)

!rm -rf *
!curl -u leprieto:${SECRET} -o dataset.zip "https://cloud.leonprieto.com/remote.php/dav/files/leprieto/University/UEM/TFM/resources/00_clean_filtered_dataset/dataset.zip"
!unzip -q dataset.zip
!rm dataset.zip

base_dir = '/content/dataset/casting_data/casting_data'

train_data_dir, val_data_dir = split_train_val(os.path.join(base_dir, 'train'))

model = train_yolo()

test_dir = '/content/dataset/casting_data/casting_data/test'
test_and_generate_matrix(model, test_dir)
```

A.2. YOLOv8 Segmentation (Pure)

```
    root_dir = "/content"
runtime_dir = !pwd
if runtime_dir[0] != root_dir:
    raise Exception("Runtime folder does not match root folder.")
!rm -rf *
import locale
locale.getpreferredencoding = lambda: "UTF-8"
```

```
!pip install roboflow
!pip install ultralytics==8.0.196

import torch
import logging

orig_torch_load = torch.load

def torch_wrapper(*args, **kwargs):
    kwargs['weights_only'] = False

    return orig_torch_load(*args, **kwargs)

torch.load = torch_wrapper

NODE_CLASS_MAPPINGS = {}
__all__ = ['NODE_CLASS_MAPPINGS']

from roboflow import Roboflow
from ultralytics import YOLO
import numpy as np
import os
import cv2

from ultralytics import settings
settings.update({"wandb": False})

rf = Roboflow(api_key="{SECRET}")
project = rf.workspace("tfm-ks6ji").project("disc-detection-v2-zqrln")
version = project.version(1)
dataset = version.download("yolov8")

!mv Disc-detection-v2-$version.version "00_labeled_dataset"

model = YOLO("yolov8n-seg.pt")
result = model.train(
    data = os.path.join(root_dir, "00_labeled_dataset", "data.yaml"),
    epochs = 100,
    imgsz = 300,
    batch = 32,
    name = "weakly_supervised_disk",
    device = "0",
)

def compute_iou(pred_mask, true_mask):
    intersection = (pred_mask & true_mask).float().sum((1, 2))
    union = (pred_mask | true_mask).float().sum((1, 2))
    iou = (intersection + 1e-6) / (union + 1e-6)
    return iou

def compute_dice(pred_mask, true_mask):
    intersection = (pred_mask & true_mask).float().sum((1, 2))
    dice = (2. * intersection + 1e-6) / (pred_mask.float().sum((1, 2)) + true_mask.
    float().sum((1, 2)) + 1e-6)
    return dice

def polygon_to_mask(coords, height, width):
    coords = np.array(coords, dtype=np.float32)
    coords = coords.reshape((-1, 1, 2)) * [width, height]
    coords = coords.astype(np.int32)

    mask = np.zeros((height, width), dtype=np.uint8)
    cv2.fillPoly(mask, [coords], 1)
    return torch.tensor(mask, dtype=torch.bool)

n = 0
inference_time = 0
ious = []
dices = []
for root, dirs, files in os.walk("/content/00_labeled_dataset/test/images"):
  for file in files:
```

```
    path = os.path.join(root, file)
    result = model.predict(path)[0]
    inference_time += result.speed['inference']
    with open(os.path.join(root, file).replace("images", "labels").replace(".jpg", ".
    txt")) as f:
      true_label = f.read()
      true_label = true_label.split(" ")

    label = []
    for coords in result.masks.xyn[0]:
      label.append(coords[0])
      label.append(coords[1])
    del true_label[0]

    iou = compute_iou(polygon_to_mask(label, 320, 320).unsqueeze(0), polygon_to_mask(
    true_label,320,320).unsqueeze(0))
    dice = compute_dice(polygon_to_mask(label, 320, 320).unsqueeze(0), polygon_to_mask(
    true_label,320,320).unsqueeze(0))

    ious.append(iou.item())
    dices.append(dice.item())

print("IOU: {} | DICE: {}.".format(np.mean(ious), np.mean(dices)))
print("Inference time: {} ms.".format(inference_time/len(ious)))

project.version(dataset.version).deploy(model_type="yolov8", model_path=result.save_dir
    )
```

A.3. YOLOv8 Segmentation (Pseudo)

```
root_dir = "/content"
runtime_dir = !pwd
if runtime_dir[0] != root_dir:
    raise Exception("Runtime folder does not match root folder.")
!rm -rf *
import locale
locale.getpreferredencoding = lambda: "UTF-8"
!pip install roboflow
!pip install ultralytics==8.0.196

import torch
import logging

orig_torch_load = torch.load

def torch_wrapper(*args, **kwargs):
    kwargs['weights_only'] = False

    return orig_torch_load(*args, **kwargs)

torch.load = torch_wrapper

NODE_CLASS_MAPPINGS = {}
__all__ = ['NODE_CLASS_MAPPINGS']

from roboflow import Roboflow
from ultralytics import YOLO
import numpy as np
import os
import cv2

from ultralytics import settings
settings.update({"wandb": False})

rf = Roboflow(api_key="{SECRET}")
project = rf.workspace("tfm-ks6ji").project("disc-detection-v2-zqrln")
version = project.version(1)
dataset = version.download("yolov8")
```

```python

!mv Disc-detection-v2-$version.version "00_labeled_dataset"

model = YOLO("yolov8n-seg.pt")
result = model.train(
    data = os.path.join(root_dir, "00_labeled_dataset", "data.yaml"),
    epochs = 100,
    imgsz = 300,
    batch = 32,
    name = "weakly_supervised_disk",
    device = "0",
)

def compute_iou(pred_mask, true_mask):
    intersection = (pred_mask & true_mask).float().sum((1, 2))
    union = (pred_mask | true_mask).float().sum((1, 2))
    iou = (intersection + 1e-6) / (union + 1e-6)
    return iou

def compute_dice(pred_mask, true_mask):
    intersection = (pred_mask & true_mask).float().sum((1, 2))
    dice = (2. * intersection + 1e-6) / (pred_mask.float().sum((1, 2)) + true_mask.float().sum((1, 2)) + 1e-6)
    return dice

def polygon_to_mask(coords, height, width):
    coords = np.array(coords, dtype=np.float32)
    coords = coords.reshape((-1, 1, 2)) * [width, height]
    coords = coords.astype(np.int32)

    mask = np.zeros((height, width), dtype=np.uint8)
    cv2.fillPoly(mask, [coords], 1)
    return torch.tensor(mask, dtype=torch.bool)

n = 0
inference_time = 0
ious = []
dices = []
for root, dirs, files in os.walk("/content/00_labeled_dataset/test/images"):
  for file in files:
    path = os.path.join(root, file)
    result = model.predict(path)[0]
    inference_time += result.speed['inference']
    with open(os.path.join(root, file).replace("images", "labels").replace(".jpg", ".txt")) as f:
      true_label = f.read()
      true_label = true_label.split(" ")

    label = []
    for coords in result.masks.xyn[0]:
      label.append(coords[0])
      label.append(coords[1])
    del true_label[0]

    iou = compute_iou(polygon_to_mask(label, 320, 320).unsqueeze(0), polygon_to_mask(true_label,320,320).unsqueeze(0))
    dice = compute_dice(polygon_to_mask(label, 320, 320).unsqueeze(0), polygon_to_mask(true_label,320,320).unsqueeze(0))

    ious.append(iou.item())
    dices.append(dice.item())

print("IOU: {} | DICE: {}.".format(np.mean(ious), np.mean(dices)))
print("Inference time: {} ms.".format(inference_time/len(ious)))

project.version(dataset.version).deploy(model_type="yolov8", model_path=result.save_dir)
```

A.4. Keras/TensorFlow Classification

```
!pip install tensorflow
!pip install keras
!pip install scikit-learn

root_dir = "/content"
runtime_dir = !pwd
if runtime_dir[0] != root_dir:
    raise Exception("Runtime folder does not match root folder.")
!rm -rf *

import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
import matplotlib.pyplot as plt
from tqdm import tqdm
from keras.layers import BatchNormalization
import os
import tensorflow as tf
from keras.callbacks import EarlyStopping
import numpy as np
from sklearn.utils.class_weight import compute_class_weight
from tensorflow.keras import layers, models
from tensorflow.keras.preprocessing import image_dataset_from_directory
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix, classification_report,
    ConfusionMatrixDisplay

!curl -u leprieto:${SECRET} -o 00_dataset.zip "https://cloud.leonprieto.com/remote.php/
    dav/files/leprieto/University/UEM/TFM/resources/00_clean_filtered_dataset/dataset.
    zip"
!unzip -q 00_dataset.zip -d 00_dataset
!rm 00_dataset.zip

IMG_SIZE = (300, 300)
BATCH_SIZE = 32
EPOCHS = 100
DATA_DIR = '00_dataset/dataset/casting_data/casting_data/train/'

train_dataset, val_dataset = image_dataset_from_directory(
    DATA_DIR,
    validation_split=0.2,
    subset="both",
    seed=42,
    image_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    label_mode='binary'
)

class_names = train_dataset.class_names
print(f"Classes: {class_names}")
print(f"Training samples: {train_dataset.cardinality().numpy() * BATCH_SIZE}")
print(f"Validation samples: {val_dataset.cardinality().numpy() * BATCH_SIZE}")

data_augmentation = tf.keras.Sequential([
    layers.RandomFlip("horizontal"),
    layers.RandomRotation(0.1),
    layers.RandomContrast(0.1)
])

preprocess = tf.keras.Sequential([
    layers.Rescaling(1./255)
])

model = models.Sequential([
```

```
  layers.Input(shape=IMG_SIZE + (3,)),
    data_augmentation,
    preprocess,

    layers.Conv2D(32, 3, activation='relu'),
    layers.MaxPooling2D(),

    layers.Conv2D(64, 3, activation='relu'),
    layers.MaxPooling2D(),

    layers.Conv2D(128, 3, activation='relu'),
    layers.MaxPooling2D(),

    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(1, activation='sigmoid')
])

labels = np.concatenate([y for x, y in train_dataset], axis=0)
class_weights = compute_class_weight(
    'balanced',
    classes=np.unique(labels),
    y=labels.flatten()
)
class_weights = {i: w for i, w in enumerate(class_weights)}

model.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['accuracy']
)

callbacks = [
    keras.callbacks.ModelCheckpoint("model.weights.h5",
                                    monitor='val_loss',
                                    save_best_only=True,
                                    save_weights_only=True),
]

history = model.fit(
    train_dataset,
    validation_data=val_dataset,
    epochs=EPOCHS,
    class_weight=class_weights,
    callbacks=callbacks
)

model.load_weights("model.weights.h5")
loss, accuracy= model.evaluate(val_dataset)
print(f"\nValidation Accuracy: {accuracy*100:.2f}%")

train_loss = history.history['loss']
val_loss = history.history['val_loss']
print(f"Loss {train_loss}")
print(f"Validation Loss: {val_loss}")

test_dataset = image_dataset_from_directory(
    '00_dataset/dataset/casting_data/casting_data/test/',
    seed=42,
    image_size=IMG_SIZE,
    batch_size=1,
    label_mode='binary'
)

true_labels = []
predictions = []
for images, labels in test_dataset:
    true_labels.extend(labels.numpy())
    batch_pred = model.predict(images)
    predictions.extend((batch_pred > 0.5).astype(int).flatten())
```

```

true_labels = np.array(true_labels)
predictions = np.array(predictions)

cm = confusion_matrix(true_labels, predictions, labels=[0, 1])
cm_display = ConfusionMatrixDisplay(cm, display_labels=['def', 'ok'])
report = classification_report(true_labels, predictions, target_names=['def', 'ok'])
cm_display.plot(cmap=plt.cm.Blues)
plt.savefig("confusion_matrix.svg")
plt.show()
print(report)
```

A.5. YOLOv8 Segmentation (Crop) + Keras/TensorFlow Classification

```
!pip install tensorflow
!pip install keras
!pip install scikit-learn

root_dir = "/content"
runtime_dir = !pwd
if runtime_dir[0] != root_dir:
    raise Exception("Runtime folder does not match root folder.")
!rm -rf *

import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
import matplotlib.pyplot as plt
from tqdm import tqdm
from keras.layers import BatchNormalization
import os
import tensorflow as tf
from keras.callbacks import EarlyStopping
import numpy as np
from sklearn.utils.class_weight import compute_class_weight
from tensorflow.keras import layers, models
from tensorflow.keras.preprocessing import image_dataset_from_directory
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix, classification_report,
    ConfusionMatrixDisplay

!curl -u leprieto:fnnXN-KRkr9-dWJqy-x2GEE-E3P5N -o 00_dataset.zip "https://cloud.
    leonprieto.com/remote.php/dav/files/leprieto/University/UEM/TFM/resources/00
    _clean_filtered_dataset/dataset.zip"
!unzip -q 00_dataset.zip -d 00_dataset
!rm 00_dataset.zip

IMG_SIZE = (300, 300)
BATCH_SIZE = 32
EPOCHS = 100
DATA_DIR = '00_dataset/dataset/casting_data/casting_data/train/'

train_dataset, val_dataset = image_dataset_from_directory(
    DATA_DIR,
    validation_split=0.2,
    subset="both",
    seed=42,
    image_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    label_mode='binary'
)

class_names = train_dataset.class_names
print(f"Classes: {class_names}")
print(f"Training samples: {train_dataset.cardinality().numpy() * BATCH_SIZE}")
```

```python
print(f"Validation samples: {val_dataset.cardinality().numpy() * BATCH_SIZE}")

data_augmentation = tf.keras.Sequential([
    layers.RandomFlip("horizontal"),
    layers.RandomRotation(0.1),
    layers.RandomContrast(0.1)
])

preprocess = tf.keras.Sequential([
    layers.Rescaling(1./255)
])

model = models.Sequential([
  layers.Input(shape=IMG_SIZE + (3,)),
    data_augmentation,
    preprocess,

    layers.Conv2D(32, 3, activation='relu'),
    layers.MaxPooling2D(),

    layers.Conv2D(64, 3, activation='relu'),
    layers.MaxPooling2D(),

    layers.Conv2D(128, 3, activation='relu'),
    layers.MaxPooling2D(),

    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(1, activation='sigmoid')
])

labels = np.concatenate([y for x, y in train_dataset], axis=0)
class_weights = compute_class_weight(
    'balanced',
    classes=np.unique(labels),
    y=labels.flatten()
)
class_weights = {i: w for i, w in enumerate(class_weights)}

model.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['accuracy']
)

callbacks = [
    keras.callbacks.ModelCheckpoint("model.weights.h5",
                                    monitor='val_loss',
                                    save_best_only=True,
                                    save_weights_only=True),
]

history = model.fit(
    train_dataset,
    validation_data=val_dataset,
    epochs=EPOCHS,
    class_weight=class_weights,
    callbacks=callbacks
)

model.load_weights("model.weights.h5")
loss, accuracy= model.evaluate(val_dataset)
print(f"\nValidation Accuracy: {accuracy*100:.2f}%")

train_loss = history.history['loss']
val_loss = history.history['val_loss']
print(f"Loss {train_loss}")
print(f"Validation Loss: {val_loss}")

test_dataset = image_dataset_from_directory(
```

```
    '00_dataset/dataset/casting_data/casting_data/test/',
    seed=42,
    image_size=IMG_SIZE,
    batch_size=1,
    label_mode='binary'
)

true_labels = []
predictions = []
for images, labels in test_dataset:
    true_labels.extend(labels.numpy())
    batch_pred = model.predict(images)
    predictions.extend((batch_pred > 0.5).astype(int).flatten())

true_labels = np.array(true_labels)
predictions = np.array(predictions)

cm = confusion_matrix(true_labels, predictions, labels=[0, 1])
cm_display = ConfusionMatrixDisplay(cm, display_labels=['def', 'ok'])
report = classification_report(true_labels, predictions, target_names=['def', 'ok'])
cm_display.plot(cmap=plt.cm.Blues)
plt.savefig("confusion_matrix.svg")
plt.show()
print(report)
```

A.6. YOLOv8 Segmentation (Backbone) + Keras/TensorFlow Classification + Fine-Tuning

```
!pip install ultralytics --no-deps
!pip install --upgrade nvidia-cuda-nvcc-cu12==12.5.40

root_dir = "/content"
runtime_dir = !pwd
if runtime_dir[0] != root_dir:
    raise Exception("Runtime folder does not match root folder.")
!rm -rf *
import locale
locale.getpreferredencoding = lambda: "UTF-8"

import torch
import logging

orig_torch_load = torch.load

def torch_wrapper(*args, **kwargs):
    kwargs['weights_only'] = False
    return orig_torch_load(*args, **kwargs)

torch.load = torch_wrapper

NODE_CLASS_MAPPINGS = {}
__all__ = ['NODE_CLASS_MAPPINGS']

import os
import tensorflow as tf
from tensorflow import keras
from ultralytics import YOLO
from keras import layers, models, optimizers
from keras.preprocessing import image_dataset_from_directory
from ultralytics import settings
from sklearn.utils.class_weight import compute_class_weight
from sklearn.metrics import (
    confusion_matrix,
    ConfusionMatrixDisplay,
    classification_report
)
import matplotlib.pyplot as plt
```

```python

settings.update({"wandb": False})

!curl -u leprieto:{SECRET} -o 00_pseudo_model.zip "https://cloud.leonprieto.com/remote.
    php/dav/files/leprieto/University/UEM/TFM/resources/jupyter/02_pseudo_model.zip"
!unzip -q 00_pseudo_model.zip -d 00_pseudo_model
!rm 00_pseudo_model.zip

!curl -u leprieto:{SECRET} -o 01_dataset.zip "https://cloud.leonprieto.com/remote.php/
    dav/files/leprieto/University/UEM/TFM/resources/00_clean_filtered_dataset/dataset.
    zip"
!unzip -q 01_dataset.zip -d 01_dataset
!rm 01_dataset.zip

import numpy as np
import tensorflow as tf
import torch
from tensorflow.keras import Model, layers
from ultralytics import YOLO

def _torch_conv_weights(conv: torch.nn.Conv2d):
    w = conv.weight.detach().cpu().numpy()
    w = np.transpose(w, (2, 3, 1, 0))  # (H, W, in_c/groups, out_c)
    return [w] + ([conv.bias.detach().cpu().numpy()] if conv.bias is not None else [])

def _torch_bn_weights(bn: torch.nn.BatchNorm2d):
    return [
        bn.weight.detach().cpu().numpy(),
        bn.bias.detach().cpu().numpy(),
        bn.running_mean.detach().cpu().numpy(),
        bn.running_var.detach().cpu().numpy(),
    ]

def conv_bn_silu(x, tblock, name, *, log=False):
    tconv: torch.nn.Conv2d = tblock.conv
    tbn: torch.nn.BatchNorm2d = tblock.bn

    pad_h, pad_w = tconv.padding
    if pad_h or pad_w:
        x = layers.ZeroPadding2D(padding=(pad_h, pad_w), name=f"{name}_pad")(x)
        keras_padding = "valid"
    else:
        keras_padding = "valid"

    conv = layers.Conv2D(
        filters=tconv.out_channels,
        kernel_size=tconv.kernel_size,
        strides=tconv.stride,
        padding=keras_padding,
        dilation_rate=tconv.dilation,
        groups=tconv.groups,
        use_bias=tconv.bias is not None,
        name=f"{name}_conv",
    )
    x = conv(x)

    bn = layers.BatchNormalization(
        epsilon=tbn.eps,
        momentum=1 - tbn.momentum,
        name=f"{name}_bn",
    )
    x = bn(x)

    x = layers.Activation("swish", name=f"{name}_silu")(x)

    conv.set_weights(_torch_conv_weights(tconv))
    bn.set_weights(_torch_bn_weights(tbn))

    if log:
        k = conv.kernel.shape
        print(f"{name:20s} | g={tconv.groups:<2d} bias={tconv.bias is not None} |
```

```
    stride={tconv.stride}  pad=({pad_h})  W->{k}")

    return x

def convert_c2f(x, tblock, idx, *, log=False):
    x = conv_bn_silu(x, tblock.cv1, f"c2f{idx}_cv1", log=log)

    half = x.shape[-1] // 2
    x1 = layers.Lambda(lambda z: z[..., :half], name=f"c2f{idx}_split1")(x)
    x2 = layers.Lambda(lambda z: z[..., half:], name=f"c2f{idx}_split2")(x)

    parts = [x1, x2]
    for j, bottleneck in enumerate(tblock.m):
        identity = x2
        x2 = conv_bn_silu(x2, bottleneck.cv1, f"c2f{idx}_b{j}cv1", log=log)
        x2 = conv_bn_silu(x2, bottleneck.cv2, f"c2f{idx}_b{j}cv2", log=log)
        if bottleneck.add:
            x2 = layers.Add(name=f"c2f{idx}_b{j}_add")([identity, x2])
        parts.append(x2)

    x = layers.Concatenate(name=f"c2f{idx}_concat")(parts)
    x = conv_bn_silu(x, tblock.cv2, f"c2f{idx}_cv2", log=log)

    x = layers.Lambda(lambda z: z, name=f"stage{idx}_out")(x)
    return x

def convert_sppf(x, tblock, idx, *, log=False):
    x = conv_bn_silu(x, tblock.cv1, f"sppf{idx}_cv1", log=log)

    p1 = layers.MaxPooling2D(5, 1, "same", name=f"sppf{idx}_p1")(x)
    p2 = layers.MaxPooling2D(5, 1, "same", name=f"sppf{idx}_p2")(p1)
    p3 = layers.MaxPooling2D(5, 1, "same", name=f"sppf{idx}_p3")(p2)
    x = layers.Concatenate(name=f"sppf{idx}_concat")([x, p1, p2, p3])

    x = conv_bn_silu(x, tblock.cv2, f"sppf{idx}_cv2", log=log)
    x = layers.Lambda(lambda z: z, name=f"stage{idx}_out")(x)
    return x

def build_keras_backbone(t_backbone, input_shape=(320, 320, 3), *, verbose=False):
    inputs = layers.Input(shape=input_shape)
    x = inputs

    for idx, layer in enumerate(t_backbone.children()):
        lname = layer._get_name()
        if lname == "Conv":
            x = conv_bn_silu(x, layer, f"conv{idx}", log=verbose)
            x = layers.Lambda(lambda z: z, name=f"stage{idx}_out")(x)
        elif lname == "C2f":
            x = convert_c2f(x, layer, idx, log=verbose)
        elif lname == "SPPF":
            x = convert_sppf(x, layer, idx, log=verbose)
        else:
            raise ValueError(f"Unsupported layer type: {lname}")

    return Model(inputs, x, name="keras_yolov8_backbone")

def _make_k_debug(model, n_stages):
    outs = [model.get_layer(f"stage{i}_out").output for i in range(n_stages)]
    return Model(model.input, outs)

# ---------------------------------------------------------------------------
# -- Checkers -------------------------------------------------------------
# ---------------------------------------------------------------------------

def compare_backbones(t_backbone, k_backbone, shape=(320, 320, 3), n=5, seed=0, *,
    verbose=False):
    if verbose:
        return compare_backbones_layerwise(t_backbone, k_backbone, shape, n, seed)
    mx_abs = mx_rel = 0.0
    rng = np.random.default_rng(seed)
    torch.manual_seed(seed)
```

```python
    for _ in range(n):
        x_np = rng.standard_normal((1, *shape), dtype=np.float32)
        with torch.no_grad():
            y_t = t_backbone(torch.from_numpy(x_np).permute(0, 3, 1, 2)).cpu().numpy()
            y_t = np.transpose(y_t, (0, 2, 3, 1))
        y_k = k_backbone(x_np, training=False).numpy()
        d = np.abs(y_k - y_t)
        r = d / (np.abs(y_t) + 1e-8)
        mx_abs = max(mx_abs, d.max())
        mx_rel = max(mx_rel, r.max())
    print(f"overall max |d| = {mx_abs:.3e}\tmax reld = {mx_rel:.3e}")
    return mx_abs, mx_rel


def compare_backbones_layerwise(t_backbone, k_backbone, shape=(320, 320, 3), n=1, seed=0):
    k_debug = _make_k_debug(k_backbone, len(list(t_backbone.children())))
    rng = np.random.default_rng(seed)
    torch.manual_seed(seed)
    x_np = rng.standard_normal((1, *shape), dtype=np.float32)
    with torch.no_grad():
        x_t = torch.from_numpy(x_np).permute(0, 3, 1, 2)
        t_outs = []
        for m in t_backbone.children():
            x_t = m(x_t)
            t_outs.append(x_t.cpu().numpy().transpose(0, 2, 3, 1))

    k_outs = k_debug(x_np, training=False)

    for idx, (t_o, k_o) in enumerate(zip(t_outs, k_outs)):
        d = np.abs(k_o - t_o)
        print(f"stage {idx:<2d} | max |d| {d.max():.3e}  mean |d| {d.mean():.3e}")

    return None

yolo = YOLO("/content/00_pseudo_model/weights/best.pt")
t_backbone = torch.nn.Sequential(*list(yolo.model.model.children())[:10]).eval()

k_backbone = build_keras_backbone(t_backbone, verbose=True)
k_backbone.save("yolov8_backbone.keras", overwrite=True)

compare_backbones(t_backbone, k_backbone, n=1, verbose=True)

k_backbone.summary()

def resize_and_pad(image, label):
    TARGET_SIZE = 320
    PADDING_COLOR = 114

    image = tf.cast(image, tf.float32)

    shape = tf.shape(image)
    h = shape[0]
    w = shape[1]

    scale = tf.minimum(TARGET_SIZE / tf.cast(h, tf.float32),
                       TARGET_SIZE / tf.cast(w, tf.float32))

    new_h = tf.cast(tf.round(tf.cast(h, tf.float32) * scale), tf.int32)
    new_w = tf.cast(tf.round(tf.cast(w, tf.float32) * scale), tf.int32)

    resized = tf.image.resize(image, [new_h, new_w], method="bilinear")

    pad_h = TARGET_SIZE - new_h
    pad_w = TARGET_SIZE - new_w
    pad_top = pad_h // 2
    pad_bottom = pad_h - pad_top
    pad_left = pad_w // 2
    pad_right = pad_w - pad_left

    padded = tf.pad(resized,
```

```
                    paddings=[[pad_top, pad_bottom],
                              [pad_left, pad_right],
                              [0, 0]],
                    constant_values=PADDING_COLOR)

    padded.set_shape([TARGET_SIZE, TARGET_SIZE, 3])
    padded = padded / 255.0
    return padded, tf.reshape(label, [1])

def set_trainable_layers(model, start: int, end: int):
    """Enable training for specified layer range in the model."""
    for layer in model.layers[start:end]:
        layer.trainable = True

def compile_model(model, learning_rate: float = 1e-5):
    """Compile the model with Adam optimizer and binary crossentropy loss."""
    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate),
        loss='binary_crossentropy',
        metrics=['accuracy']
    )
    model.summary()

def get_callbacks(filepath: str):
    """Create ModelCheckpoint callback to save best model weights."""
    return [
        tf.keras.callbacks.ModelCheckpoint(
            filepath,
            monitor='val_loss',
            save_best_only=True,
            save_weights_only=True
        )
    ]

def train_model(model, train_dataset, val_dataset, class_weights, epochs, callbacks):
    """Train the model and return the training history."""
    history = model.fit(
        train_dataset,
        validation_data=val_dataset,
        epochs=epochs,
        class_weight=class_weights,
        callbacks=callbacks
    )
    return history

def evaluate_model(model, val_dataset, weight_file):
    """Load best weights and evaluate the model."""
    model.load_weights(weight_file)
    loss, accuracy = model.evaluate(val_dataset)
    print(f"\nValidation Accuracy: {accuracy*100:.2f}%")
    return loss, accuracy

def print_loss_curves(history):
    """Print training and validation loss curves."""
    train_loss = history.history['loss']
    val_loss = history.history['val_loss']
    print(f"Loss: {train_loss}")
    print(f"Validation Loss: {val_loss}")

def evaluate_performance(model, test_dataset, weight_file):
  true_labels = []
  predictions = []
  for images, labels in test_dataset:
      true_labels.extend(labels.numpy())
      batch_pred = model.predict(images)
      predictions.extend((batch_pred > 0.5).astype(int).flatten())

  true_labels = np.array(true_labels)
  predictions = np.array(predictions)

  cm = confusion_matrix(true_labels, predictions, labels=[0, 1])
```

```
  cm_display = ConfusionMatrixDisplay(cm, display_labels=['def', 'ok'])
  report = classification_report(true_labels, predictions, target_names=['def', 'ok'])
  cm_display.plot(cmap=plt.cm.Blues)
  plt.savefig(f"confusion_matrix_{weight_file}.svg")
  plt.show()
  print(report)

IMG_SIZE = (300, 300)
BATCH_SIZE = 32
EPOCHS = 100
DATA_DIR = '01_dataset/dataset/casting_data/casting_data/train/'

train_dataset, val_dataset = image_dataset_from_directory(
    DATA_DIR,
    validation_split=0.2,
    subset="both",
    seed=42,
    image_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    label_mode='binary'
)

test_dataset = image_dataset_from_directory(
    '01_dataset/dataset/casting_data/casting_data/test/',
    seed=42,
    image_size=IMG_SIZE,
    batch_size=1,
    label_mode='binary'
)

train_dataset = train_dataset.unbatch().map(resize_and_pad, num_parallel_calls=tf.data.
    AUTOTUNE)
val_dataset = val_dataset.unbatch().map(resize_and_pad, num_parallel_calls=tf.data.
    AUTOTUNE)
test_dataset = test_dataset.unbatch().map(resize_and_pad, num_parallel_calls=tf.data.
    AUTOTUNE)
train_dataset = train_dataset.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)
val_dataset = val_dataset.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)
test_dataset = test_dataset.batch(1).prefetch(tf.data.AUTOTUNE)

k_backbone.trainable = False  # Freeze all layers

x = k_backbone.get_layer("stage9_out").output

# Classification head
x = layers.Conv2D(64, kernel_size=3, activation='relu')(x)
x = layers.MaxPooling2D()(x)

x = layers.Conv2D(128, kernel_size=3, activation='relu')(x)
x = layers.MaxPooling2D()(x)

x = layers.Flatten()(x)
x = layers.Dense(64, activation='relu')(x)
x = layers.Dropout(0.5)(x)
output = layers.Dense(1, activation='sigmoid')(x)

model = Model(inputs=k_backbone.input, outputs=output)
compile_model(model, 1e-3)

labels = np.concatenate([y for x, y in train_dataset], axis=0)
class_weights = compute_class_weight(
    'balanced',
    classes=np.unique(labels),
    y=labels.flatten()
)
class_weights = {i: w for i, w in enumerate(class_weights)}
weight_file = "model.weights.h5"
callbacks = get_callbacks(weight_file)
history = train_model(model, train_dataset, val_dataset, class_weights, EPOCHS,
    callbacks)
evaluate_model(model, val_dataset, weight_file)
```

```
print_loss_curves(history)
evaluate_performance(model, test_dataset, weight_file)

# Stages composing the backbone.
i = 0
for layer in model.layers:
  if "_out" in layer.name:
    print(f"Layer: {layer.name} Index: {i}")
  i += 1

# Fine Tuning: Stage 8-9.
weight_file = "model89.weights.h5"
set_trainable_layers(model, 101, 130)
compile_model(model)
callbacks = get_callbacks(weight_file)
history = train_model(model, train_dataset, val_dataset, class_weights, 30, callbacks)
evaluate_model(model, val_dataset, weight_file)
print_loss_curves(history)
evaluate_performance(model, test_dataset, weight_file)

# Fine Tuning: Stage 6-7.
weight_file = "model67.weights.h5"
set_trainable_layers(model, 68, 100)
compile_model(model)
callbacks = get_callbacks(weight_file)
history = train_model(model, train_dataset, val_dataset, class_weights, 30, callbacks)
evaluate_model(model, val_dataset, weight_file)
print_loss_curves(history)
evaluate_performance(model, test_dataset, weight_file)

# Fine Tuning: Stage 4-5.
weight_file = "model45.weights.h5"
set_trainable_layers(model, 35, 67)
compile_model(model)
callbacks = get_callbacks(weight_file)
history = train_model(model, train_dataset, val_dataset, class_weights, 30, callbacks)
evaluate_model(model, val_dataset, weight_file)
print_loss_curves(history)
evaluate_performance(model, test_dataset, weight_file)

# Fine Tuning: Stage 2-3.
weight_file = "model23.weights.h5"
set_trainable_layers(model, 11, 34)
compile_model(model)
callbacks = get_callbacks(weight_file)
history = train_model(model, train_dataset, val_dataset, class_weights, 30, callbacks)
evaluate_model(model, val_dataset, weight_file)
print_loss_curves(history)
evaluate_performance(model, test_dataset, weight_file)

# Fine Tuning: Stage 0-1.
weight_file = "model01.weights.h5"
set_trainable_layers(model, 0, 10)
compile_model(model)
callbacks = get_callbacks(weight_file)
history = train_model(model, train_dataset, val_dataset, class_weights, 30, callbacks)
evaluate_model(model, val_dataset, weight_file)
print_loss_curves(history)
evaluate_performance(model, test_dataset, weight_file)
```

B

Source Code and Asset Repository

The complete source code and related resources for this thesis are available in the following GitHub repository:

`https://github.com/leonprietobailo/msc-cv-thesis.git`

The repository includes:

- Preprocessing and utility scripts.
- Jupyter notebooks defining the model architecture, and performing training, validation, and testing, along with their corresponding outputs.
- Datasets used for model training and evaluation.
- Grad-CAM heatmap visualizations for model inference analysis.