



Real-time rail surface defect detection: a neuromorphic approach

RC Esterhuizen



[orcid.org/ 0000-0001-6335-9369](https://orcid.org/0000-0001-6335-9369)

Dissertation accepted in fulfilment of the requirements for the
degree *Master of Engineering in Computer and Electronic
Engineering* at the North-West University

Supervisor:

Prof PA van Vuuren

Graduation:

May 2026

DECLARATION

I, Rensche Celestè Esterhuizen, hereby declare that the work presented in this thesis, titled Real-time rail surface defect detection: a neuromorphic approach, is my own original work, and has not been submitted for a degree or any other qualification at any other university or institution. All sources used have been acknowledged, and appropriate citations have been provided where reference has been made to the work of others.

Signed,

Rensche Celestè Esterhuizen

28 November 2025

ACKNOWLEDGEMENTS

‘I can do all things through Christ who strengthens me’ – Philippians 4:13

I wish to express my sincere gratitude to Prof. Pieter Van Vuuren, my supervisor, for his unwavering support, guidance, wisdom, and insightful advice throughout this study. His expertise has been invaluable in shaping this research and his encouragement has been a source of great motivation.

I extend heartfelt thanks to my parents for their love and encouragement, which has been a constant source of strength. Their belief in me has made this journey possible.

A special thank you goes to my brother, Wikus Esterhuizen, for his invaluable assistance in setting up the hardware for the experiments. His dedication and technical help have been instrumental in the practical aspects of this work.

Finally, I am grateful to Transnet for providing access to their facilities that allowed me to gather the necessary data for this study. This access has been essential to grounding the research in real-world conditions, and I am grateful for their cooperation and support.

This work would not have been possible without the contributions of all these individuals and organisations, and I am deeply appreciative of their roles in the successful completion of this research.

ABSTRACT

This study explores the implementation of a neuromorphic-based approach for real-time rail surface defect detection, addressing the growing demand for accurate and efficient maintenance solutions in rail systems. Leveraging a dataset of 1514 rail surface images collected under real-world operating conditions and annotated with binary labels (“defect” and “no defect”), the research applies a comprehensive data augmentation pipeline – including rotation, flipping, zooming, and contrast adjustments – to expand the dataset to 248374 images. This augmented dataset ensures class balance and greater variability, enhancing model generalisation and preventing overfitting. A consistent data split (70% training, 15% validation, 15% testing) is employed to maintain fairness across evaluations.

The study presents a comparative evaluation of spiking neural networks (SNNs) and convolutional neural networks (CNNs) in terms of classification performance, energy efficiency, and real-time feasibility across three hardware platforms: a 12-core AMD Ryzen 9 CPU (3.8 GHz), a NVIDIA RTX 3060 GPU (12 GB VRAM), and the BrainChip AKD100 neuromorphic processor. The analysis further investigates the influence of hand-crafted features – including Grey-Level Co-occurrence Matrix (GLCM), Local Binary Pattern (LBP), and Sobel and Canny edge detection – on model performance and computational efficiency across different hardware configurations.

Experimental results show that CNNs consistently outperform SNNs in precision and recall, particularly when implemented on GPUs, where the CNN model achieves a precision of 94.82%, recall of 94.88%, and F1-score of 94.49%. The Akida neuromorphic processor using a CNN model, with its 8 neural cores and sub-1W power consumption, maintains a high precision of 94.74% while achieving latency below 5 ms and throughput exceeding 250 frames per second – demonstrating strong balance of accuracy, speed, and energy efficiency. In contrast, SNNs, though more energy-efficient, show diminished accuracy, with the CPU-based SNN model achieving an F1-score of only 83.75%.

The research also outlines optimal hardware configurations for deployment, considering factors such as sensor quality, data acquisition methods, power supply, and cooling requirements. The findings indicate that combining CNNs with the Akida processor offers an ideal solution for real-time rail defect detection, balancing classification performance with resource efficiency. The final chapter synthesises these insights to offer practical guidance for system deployment and highlights future directions to advance neuromorphic-based approaches in railway maintenance.

Keywords: Neuromorphic computing, real-time defect detection, rail surface inspection, spiking neural networks, convolutional neural networks, energy efficiency, hardware optimisation, hand-crafted features (GLCM, LBP, Sobel, and Canny), Akida neuromorphic processor.

TABLE OF CONTENTS

LIST OF TABLES	XVII
LIST OF ABBREVIATIONS	XIX
DEFINITIONS OF CONCEPTS	XXII
CHAPTER 1: INTRODUCTION	1
1.1 BACKGROUND AND CONTEXT.....	1
1.1.1 Introduction to rail surface defect detection.....	1
1.1.2 Significance of neuromorphic computing in defect detection.....	3
1.2 RESEARCH OBJECTIVES	4
1.2.1 Research question	4
1.2.2 Research sub-questions.....	5
1.3 METHODOLOGY.....	5
1.3.1 Literature Review:	5
1.3.2 Dataset Identification and Analysis:.....	5
1.3.3 Performance Metrics and Evaluation Framework:.....	6
1.3.3.1 Hardware Requirements Analysis:	6
1.3.3.2 Validation and Robustness Testing:	6
1.3.4 Neural Network Architectures:.....	6
1.3.4.1 Evaluation of Neural Network Architecture:	6
1.3.5 Feature Engineering and Impact Assessment:	7
1.3.5.1 Feature Impact Analysis:	7
1.4 DISSERTATION OVERVIEW	7
1.4.1 Chapter 2: Critical literature survey	7
1.4.2 Chapter 3: Proposed system configuration and platform readiness.....	7
1.4.3 Chapter 4: Neural network architectures.....	8
1.4.4 Chapter 5: Feature engineering and impact analysis.....	8
1.4.5 Chapter 6: Hardware requirements analysis.....	8
1.4.6 Chapter 7: Discussion and Conclusion	8
CHAPTER 2: CRITICAL LITERATURE SURVEY.....	9
2.1 CHAPTER INTRODUCTION	9
2.2 RAIL SURFACE DEFECT DETECTION	9
2.2.1 Image processing techniques	9
2.2.1.1 Contrast enhancement	9
2.2.1.2 Smoothing and sharpening	10
2.2.2 Machine learning-based approaches.....	11
2.2.2.1 Support vector machine.....	11
2.2.2.2 Random forest.....	13

2.2.2.3 K-Nearest Neighbour.....	14
2.2.2.4 Naïve bayes	15
2.2.3 Non-destructive testing.....	16
2.3 DATA AUGMENTATION TECHNIQUES	18
2.3.1 Introduction to data augmentation	18
2.3.2 Types of data augmentation methods.....	18
2.3.2.1 Geometric transformations.....	19
2.3.2.2 Colour space transformations	21
2.3.2.3 Random erasing	21
2.3.2.4 Generative modelling.....	22
2.3.2.5 Neural style transfer.....	22
2.3.3 Summary	22
2.4 NEUROMORPHIC COMPUTING	23
2.4.1 Advantages and uses of neuromorphic computing.....	23
2.4.2 Neuromorphic technology and implementation	23
2.4.3 Previous work on neuromorphic-based defect detection.....	24
2.4.4 Current shortcomings of neuromorphic computing	26
2.5 CONCLUSION	27
CHAPTER 3: PROPOSED SYSTEM CONFIGURATION AND PLATFORM READINESS	28
3.1 CHAPTER INTRODUCTION	28
3.2 HARDWARE PLATFORM MOTIVATION AND DEPLOYMENT CONSTRAINTS.....	28
3.2.1 Neuromorphic Processor (Akida).....	28
3.2.2 Central Processing Unit (CPU)	28
3.2.3 Graphics Processing Unit (GPU)	29
3.2.4 Power Supply and Cooling Considerations	29
3.3 SENSOR INPUT AND REAL-TIME CONSTRAINTS	29
3.3.1 Sensor Resolution and Data Acquisition.....	29
3.3.2 Preprocessing and Feature Extraction	29
3.4 SOFTWARE STACK AND TOOLCHAIN ALIGNMENT	30
3.4.1 Model development and conversion	31
3.4.2 Inference and Optimization Tools	31
3.4.3 Power and Performance Monitoring.....	31
3.4.4 Data Preprocessing and Feature Engineering	32
3.4.5 Deployment and Integration	32
3.4.6 Interpretation of platform-specific optimisations	32
3.4.7 Summary	33
3.5 RESOURCE-CONSTRAINED CONFIGURATION RATIONALE	33
3.5.1 CPU Configuration.....	33
3.5.2 GPU Configuration	33

3.5.3 Akida Minimal Configuration	34
3.5.4 Motivation for Constrained Testing	34
3.6 CONCLUSION	35
CHAPTER 4: NEURAL NETWORK ARCHITECTURES	36
4.1 CHAPTER INTRODUCTION	36
4.2 INTRODUCTION TO NEURAL NETWORK ARCHITECTURES.....	36
4.2.1 Fundamentals of neural networks	36
4.2.2 Overview of spiking neural network.....	38
4.3 EXPERIMENTAL SETUP	40
4.3.1 State-of-the-art models in rail defect detection.....	40
4.3.2 Dataset structure and label integrity	40
4.3.3 Hardware platforms and motivation	43
4.3.4 Neural network models	43
4.4 METHODOLOGY	44
4.4.1 Data partitioning.....	44
4.4.2 Model training and testing procedures.....	45
4.4.2.1 CNN	45
4.4.2.2 SNN	46
4.4.3 Deployment platforms	47
4.4.4 Evaluation metrics.....	48
4.5 RESULTS AND ANALYSIS	49
4.5.1 Dataset and feature extraction	49
4.5.2 Hardware specification and software frameworks	49
4.5.3 Model performance metrics	49
4.5.3.1 CNN performance	49
4.5.3.2 SNN performance.....	50
4.5.4 Training convergence and loss behaviour	51
4.5.5 ROC and AUC performance	51
4.5.6 Inference time and energy efficiency	53
4.5.6.1 Measurement methodology:	53
4.5.7 Confusion matrix interpretation.....	55
4.5.7.1 CNN model	55
4.5.7.2 SNN model	55
4.5.8 Ablation study: Impact of data augmentation on model performance	56
4.5.8.1 Methodology	56
4.5.8.2 Results.....	56
4.5.8.3 Analysis	57
4.5.8.4 Conclusion	57
4.5.9 Summary of results	57
4.5.10 Statistical significance testing	57

4.5.10.1 Purpose of statistical testing	58
4.5.10.2 T-test for metric comparisons	58
4.5.10.3 Interpretation of results	61
4.6 EXTENDED ABLATION STUDY: MODEL CONFIGURATION IMPACT	62
4.7 CONCLUSION	63
CHAPTER 5: FEATURE ENGINEERING AND IMPACT ANALYSIS	65
5.1 CHAPTER INTRODUCTION	65
5.2 INTRODUCTION TO FEATURE ENGINEERING	65
5.3 HAND-CRAFTED FEATURES	66
5.3.1 <i>Texture features</i>	66
5.3.1.1 Understanding GLCM and texture features	66
5.3.1.2 Understanding LBP and texture features	67
5.3.2 <i>Edge features</i>	68
5.3.2.1 Exploring the Canny edge detector	68
5.3.2.2 Exploring the Sobel edge detection operator	69
5.3.3 <i>Visual examples of extracted features</i>	69
5.3.4 <i>Rationale for selected feature types</i>	71
5.4 EXPERIMENTAL SETUP	71
5.4.1 <i>Data preparation</i>	71
5.4.2 <i>Neural network models</i>	72
5.5 METHODOLOGY	73
5.5.1 <i>Training and testing procedures</i>	73
5.5.1.1 CNN implementation	73
5.5.1.2 SNN implementation	74
5.5.1.3 Data split and platform evaluation	74
5.5.2 <i>Evaluation metrics</i>	75
5.6 RESULTS AND ANALYSIS	75
5.6.1 <i>Performance metrics across platforms</i>	76
5.6.1.1 CNN model performance	76
5.6.1.2 SNN model performance (CPU)	77
5.6.1.3 Summary	77
5.6.2 <i>Confusion matrix and misclassification analysis</i>	78
5.6.2.1 CNN on CPU	78
5.6.2.2 CNN on GPU	78
5.6.2.3 CNN on Akida neuromorphic processor	78
5.6.2.4 SNN on CPU	79
5.6.2.5 Misclassification patterns	79
5.6.2.6 Interpretation summary	80
5.6.3 <i>ROC curve and AUC</i>	80
5.6.3.1 CNN on CPU	80
5.6.3.2 CNN on GPU	80

5.6.3.3 CNN on Akida neuromorphic processor	80
5.6.3.4 SNN on CPU	82
5.6.3.5 Interpretation.....	82
<i>5.6.4 Inference time and energy consumption.....</i>	<i>82</i>
5.6.4.1 Analysis of delayed CNN inference with hand-crafted features.....	84
5.6.4.2 Real-time deployment implications	84
5.6.4.3 Summary of platform trade-offs	84
<i>5.6.5 Statistical significance testing</i>	<i>85</i>
5.6.5.1 Purpose of statistical testing	85
5.6.5.2 Hypothesis formulation.....	85
5.6.5.3 Collect sample data	86
5.6.5.4 Sample means and standard deviations	86
5.6.5.5 t-Test results	87
5.6.5.6 Interpretation and implications	87
5.6.5.7 Conclusion	88
<i>5.6.6 Comparative analysis and key insights</i>	<i>88</i>
5.6.6.1 CNN model performance summary	88
5.6.6.2 SNN model performance summary (CPU).....	88
5.6.6.3 Platform-specific observations.....	88
5.6.6.4 Key takeaways for model and platform selection	89
5.6.6.5 Conclusion	90
<i>5.6.7 Ablation study on CNN and SNN optimisation parameters.....</i>	<i>90</i>
5.7 CONCLUSION	91
CHAPTER 6: HARDWARE REQUIREMENTS ANALYSIS.....	93
6.1 CHAPTER INTRODUCTION	93
6.2 EXPERIMENTS	93
6.2.1 <i>Experiment 1: Determine real-time feasibility of each platform.....</i>	<i>93</i>
6.2.1.1 Objective	93
6.2.1.2 Methodology.....	93
6.2.1.3 Results.....	94
6.2.1.4 Results Analysis	94
6.2.1.5 Conclusion	96
6.2.2 <i>Experiment 2: Minimum hardware configuration for optimal detection.....</i>	<i>96</i>
6.2.2.1 Objective	96
6.2.2.2 Methodology.....	96
6.2.2.3 Results.....	97
6.2.2.4 Insights	97
6.3 CONCLUSION	98
CHAPTER 7: DISCUSSION AND CONCLUSION.....	99
7.1 CHAPTER INTRODUCTION	99
7.2 COMPARATIVE ANALYSIS OF SNNs AND CNNs	99

7.3 IMPACT OF MODEL TYPES ON DEFECT DETECTION ACCURACY	100
7.4 HARDWARE CONFIGURATION AND ITS INFLUENCE ON SYSTEM PERFORMANCE.....	100
7.4.1 CPUs	101
7.4.2 GPUs.....	101
7.4.3 Akida Neuromorphic Processor	101
7.4.4 Summary	102
7.5 INSIGHTS ON OPTIMAL HARDWARE-MODEL COMBINATIONS	102
7.6 PRACTICAL IMPLICATIONS AND FUTURE RESEARCH DIRECTIONS.....	102
7.7 LIMITATIONS OF THE STUDY	103
7.8 FINAL REMARKS	104
REFERENCES	106
APPENDIX A: PSEUDOCODE	113
ALGORITHM A-1: CNN-BASED RAIL DEFECT DETECTION USING EFFICIENTNET-B3 AND K-FOLD CROSS-VALIDATION WITH RAW IMAGE INPUT	113
A-1.1 CPU Implementation with ablation study.....	113
A-1.2 GPU Implementation with ablation study	114
A-1.3 AKIDA Implementation with ablation study	116
ALGORITHM A-2: SNN-BASED RAIL DEFECT DETECTION – CPU IMPLEMENTATION WITH ABLATION STUDY	117
ALGORITHM A-3: CNN-BASED RAIL DEFECT DETECTION USING HAND-CRAFTED FEATURES AS INPUT	119
A-3.1 CPU Implementation with ablation study.....	119
A-3.2 GPU Implementation with ablation study	120
A-3.3 Akida Implementation with ablation study	122
ALGORITHM A-4: SNN-BASED RAIL DEFECT DETECTION USING HAND-CRAFTED FEATURES AS INPUT – CPU IMPLEMENTATION WITH ABLATION STUDY	123
ALGORITHM A-5: MODEL ARCHITECTURE SUMMARY	126
A-5.1 CNN Models Using Raw Image Inputs	126
A-5.2 CNN Models Using Hand-Crafted Feature Inputs	126
A-5.3 SNN Model Using Raw Image Inputs	126
A-5.4 SNN Model Using Hand-Crafted Features	127
A-5.5 Akida CNN Deployment	127
APPENDIX B: COMPARISON OF CNN ARCHITECTURES	128
APPENDIX C: CNN AND SNN DESIGN PROCESS.....	130
C-1 CNN STEP-BY-STEP DESIGN PROCESS	130
C-2 SNN STEP-BY-STEP DESIGN PROCESS.....	132
APPENDIX D: T-TABLE	135

LIST OF FIGURES

Figure 1-1: (a) Railway track infrastructure [2], and (b) rail profile [3]	1
Figure 1-2: Wheel burn	2
Figure 1-3: The stress planes in rails [6]	2
Figure 1-4: Methodology flow diagram	5
Figure 2-1: Support vector machine showing the optimal margin and optimal hyperplane [17].....	11
Figure 2-2: A Random Forest Classifier with multiple decision trees [17]	13
Figure 2-3: Example of k-Nearest Neighbour [27]	14
Figure 2-4: Conceptual illustration of Naive Bayes [17]	16
Figure 2-5: Original and transformed images. (a) original image, (b) flipping, (c) rotation, (d) cropping, (e) random cropping, (f) shifting, (g) noise, (h) colour jittering, (i) PCA jittering [34].....	20
Figure 2-6: Colour space transformations (a) Original image (b) LAB colour transformation (c) YCrCb transformation (d) HSV transformation [33]	21
Figure 2-7: Random erasing for object detection [32]	21
Figure 2-8: Illustration of GANs [32]	22
Figure 2-9: Memristor (a) structure, (b) equivalent circuit [38]	23
Figure 2-10: Memristor crossbar array [38]	24
Figure 3-1: Visual representation of the toolchain recommendations	30
Figure 4-1: Basic structure of a CNN showing input image, convolution and pooling layers, fully connected layers, and final classification [50].....	37
Figure 4-2: Basic structure of SNN [52].....	38
Figure 4-3: STDP learning rule for SNN [53]	39
Figure 4-4: LIF neuron [55]	39
Figure 4-5: Rail track with (a) no defect and (b) defect.....	41
Figure 4-6: Illustration of original vs resized image (300 x 300) in dataset	42
Figure 4-7: Illustration of original vs resized image (128 x 128) in dataset	42
Figure 4-8: (a) CNN training and validation loss and accuracy vs (b) SNN training and validation loss	51
Figure 4-9: CNN ROC for CPU	52
Figure 4-10: CNN ROC for GPU	52

Figure 4-11: CNN ROC for Akida 52

Figure 4-12: SNN on CPU – AUC: 0.86 53

Figure 4-13: Inference time and energy across various hardware platforms..... 55

Figure 5-1: Representative examples of GLCM, LBP, Sobel, and Canny feature extraction applied to defect (a) and no-defect (b) rail surface images 70

Figure 5-2: ROC curve for CNN on CPU 81

Figure 5-3: ROC curve for CNN on GPU..... 81

Figure 5-4: ROC curve for CNN on Akida 81

Figure 5-5: ROC curve for SNN model on a CPU..... 82

Figure C-0-1: SNN design process 132

LIST OF TABLES

Table 2-1: Comparative summary of CNN and SNN methods in rail defect detection	26
Table 4-1: Two class confusion matrix	48
Table 4-2: Hardware specifications	49
Table 4-3: CNN performance on CPU	49
Table 4-4: CNN performance on GPU	50
Table 4-5: CNN model on Akida	50
Table 4-6: SNN model on CPU	50
Table 4-7: AUC scores	51
Table 4-8: Inference time and energy	53
Table 4-9: CNN confusion matrix implemented on CPU	55
Table 4-10: CNN confusion matrix implemented on GPU	55
Table 4-11 CNN confusion matrix implemented on Akida	55
Table 4-12: Confusion matrix for SNN model implemented on the CPU	55
Table 4-13: Effect of data augmentation on model performance	56
Table 4-14: Summary of results	57
Table 4-15: F1-scores for CNN and SNN on CPU platform	59
Table 4-16: F1-scores for CNN on CPU and GPU platforms	59
Table 4-17: F1-scores for CNN on CPU and Akida neuromorphic processor	59
Table 4-18: F1-scores for CNN on GPU and Akida neuromorphic processor	60
Table 4-19: Mean and standard deviation for each model-platform combination	60
Table 4-20: Paired t-test results across model and hardware configurations	61
Table 4-21: Results for ablation study using raw image inputs	62
Table 5-1: Hardware specification	75
Table 5-2: Performance metrics for CNN model implemented on a CPU	76
Table 5-3: Performance metrics for CNN model implemented on a GPU	76
Table 5-4: Performance metrics for CNN model implemented on Akida processor	77
Table 5-5: Performance metrics for SNN model on a CPU	77
Table 5-6: Confusion matrix for CNN model implemented on a CPU	78

Table 5-7: Confusion matrix for CNN model implemented on a GPU	78
Table 5-8: Confusion matrix for CNN model implemented on Akida processor	79
Table 5-9: Confusion matrix for SNN model on a CPU	79
Table 5-10: AUC comparison summary	82
Table 5-11: Inference time and energy consumption across platforms.....	83
Table 5-12: F1-scores for CNN and SNN on CPU platform using hand-crafted features.....	86
Table 5-13: F1-scores for CNN on CPU vs. GPU platform using hand-crafted features	86
Table 5-14: F1-scores for CNN on CPU vs. Akida platform using hand-crafted features.....	86
Table 5-15: F1-scores for CNN on GPU vs. Akida platform using hand-crafted features.....	86
Table 5-16: t-test results for CNN and SNN models	87
Table 5-17: Results from ablation study	90
Table 6-1: Comparison of FPS and latency across CPU, GPU, and Akida with raw data	94
Table 6-2: Comparison of FPS and latency across CPU, GPU, and Akida with hand-crafted features	94
Table 6-3: Configuration impact on defect detection performance.....	97

LIST OF ABBREVIATIONS

1T1R	1 Transistor 1 Resistor
AD	Average Difference
AdaNN	Adaptive K-NN
ADE	Akida Development Environment
AFE	Automatic Feature Engineering
AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
ASM	Angular Second Moment
AUC	Area Under Curve
CBA	Crossbar Array
CI	Confidence Interval
CPU	Central Processing Unit
CMOS	Complementary Metal-Oxide Semiconductor
CNN	Convolution Neural Network
DoF	Degrees of Freedom
EDQHEPL	Enhanced Dynamic Quadrant Histogram Equalisation with Plateau Limit
FE	Feature Engineering
FPS	Frames per second
GAN	Generative Adversarial Network
GHE	Global Histogram Equalisation
GLCM	Grey-Level Co-occurrence Matrix
GPU	Graphics Processing Unit
HH	Hodgkin-Huxley
IBM	International Business Machines Corporation
IDM	Inverse Difference Moment
IoT	Internet of Things
J	Joules

KNN	K-Nearest Neighbour
LBP	Local Binary Pattern
LIF	Leaky Integrated-and-Fire
MD	Maximum Difference
mpf	Meters per frame
ms	Millisecond
MSE	Mean Square Error
MVS	Machine Vision System
NAE	Normalised Absolute Error
NK	Normalised Cross Correlation
NMS	Non-Maximum Suppression
NVM	Non-Volatile Memory
NVML	NVIDIA Management Library
OOB	Out-of-bag
ONNX	Open Neural Network Exchange
PSNR	Peak Signal to Noise Ratio
RBF	Radial Basis Function
RCF	Rail Contact Fatigue
ReLU	Rectified Linear Unit
RGB	Red/Green/Blue
RNN	Recurrent Neural Network
ROC	Receiver Operating Characteristic
RW	Random Walker
s	Seconds
SC	Structural Content
SDK	Software Development Kit
SGD	Stochastic Gradient Descent
SIPI	Signal and Image Processing Institute
SNN	Spiking Neural Network

SOM	Self-Organising Maps
SRM	Structural Risk Minimization
STDP	Spike-Timing-Dependent Plasticity
SVM	Support Vector Machine
tanh	Hyperbolic Tangent Function
TDP	Thermal Design Power
VAE	Variational Auto-Encoders
VLSI	Very Large-Scale Integration
W	Watt

DEFINITIONS OF CONCEPTS

This section explains the concepts used in the dissertation [1].

Wear:	Wear typically occurs on the gauge face of the high rail in curves due to high wheel flanging forces.
Running surface:	The area on top of the rail head where the wheel tread makes contact with the rail.
Gauge corner region:	This is the upper corner of the gauge side of the rail where the wheel throat region is in contact.
Gauge corner:	The specific point within the gauge corner region, characterised by a tangent that is at a 45-degree angle to the horizontal plane.
Rail head:	The part of the rail that extends above the fishing surfaces to the rail's centreline.
Fishing surface:	The area under the rail head that connects to the fishplates.
Fishplates:	Fishplates, also known as joint bars or rail joints, are components bolted to the ends of two rails to join them together on a track.
Rail centre line:	The vertical centre of the rail's cross-sectional profile.
Fillet:	Also known as the head/web transition, this area is where the rail's head transitions into the web.
Web:	The section between the rail head and the base of the rail.
Base (Rail Foot):	The bottom part of the rail, including the top, bottom, and toe of the foot. The base supports the rail, and the top holds it in place through rail fasteners or insulating biscuits. The bottom is on the sleepers and the toe is the edge of the rail foot.
Creepage:	The force generated at the wheel/rail contact patch due to the wheel rolling on the rail. Longitudinal creepage occurs when wheels apply traction to the rail or when they negotiate sharp curves, compensating for the difference in wheel diameters in contact with the rail.

CHAPTER 1: INTRODUCTION

This chapter introduces the topic of rail surface defect detection and sets the stage for the dissertation. It starts by explaining the limitations of traditional methods of rail surface defect detection and highlights the potential of neuromorphic computing. The objectives of the research are clearly defined, focusing on exploring the performance limits, suitable neural network architectures, the impact of hand-crafted features, minimum hardware requirements, and the use of data augmentation to improve generalisation.

1.1 BACKGROUND AND CONTEXT

1.1.1 Introduction to rail surface defect detection

The railway track infrastructure comprises several critical elements, including rails, sleepers, and fasteners (Figure 1-1 (a)), each playing a vital role in maintaining the integrity and functionality of the rail system. To ensure the safety and efficiency of railway operations, it is essential to conduct a thorough inspection of the rails. This involves understanding how various defects impact different parts of the rail, such as the running surface (Figure 1-1 (b)).

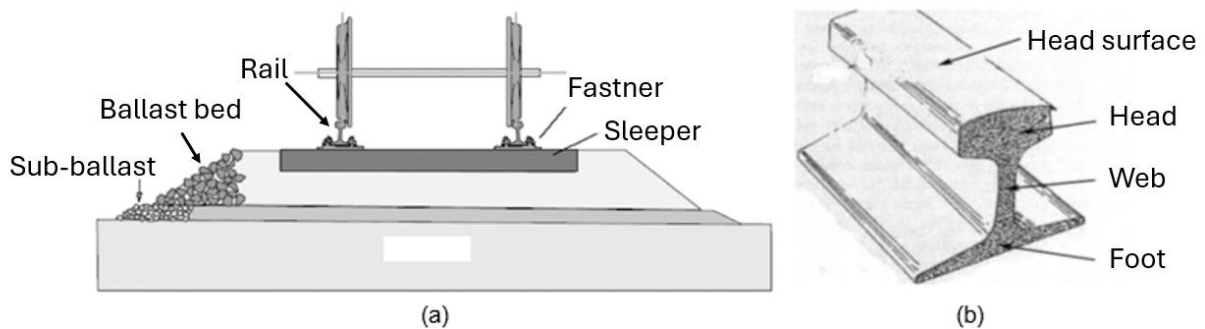


Figure 1-1: (a) Railway track infrastructure [2], and (b) rail profile [3]

Defects on rails of railways often originate from the manufacturing process, and these imperfections can significantly affect the lifetime of the rails [1]. Typically, rails have a lifespan ranging from 30 to 60 years [4], but this can vary widely based on several factors. Operating load, axle load, vehicle conditions, rail quality and profile, track quality, and maintenance practices play crucial roles in determining how long a rail can remain functional. Inadequate maintenance strategies can lead to shortened lifespan, highlighting the need for thorough and effective maintenance to ensure the durability and safety of the railway infrastructure [5].

Rail defects can develop on any part of the rail due to the continuous impact of trains and exposure to the natural environment. These defects included deformed rails and broken sleepers, which can compromise the safety and efficiency of the railway system. This study focusses specifically on the detection of defects on the rail surface. Key types of defects include rolling contact fatigue (RCF),

nicked rails, shelling, corrosion, and wheel burns (Figure 1-2) [6]. By identifying and addressing these issues, the study aims to improve the reliability and longevity of rail infrastructure.



Figure 1-2: Wheel burn

Surface defects on railway tracks can escalate into more serious problems, such as transverse and longitudinal defects (Figure 1-3). Transverse defects develop perpendicular to the cross-sectional area of the rail head, potentially compromising the structural integrity of the rail. Longitudinal defects, on the other hand, are characterised by progressive fractures that result in a longitudinal separation [7]. Both types of defects pose significant risks to railway safety and can lead to costly repairs and maintenance if not detected and addressed promptly. Therefore, early detection and remediation of surface defects are crucial to preventing these more severe forms of rail damage.

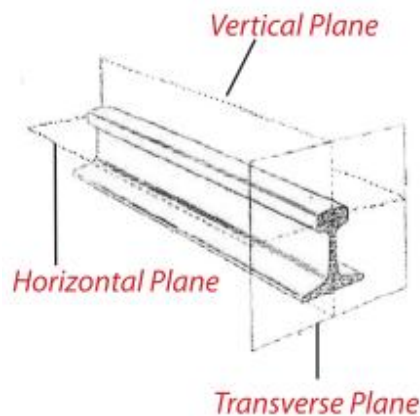


Figure 1-3: The stress planes in rails [6]

Early detection of defects is critical to ensure the safety and efficiency of rail transport, which is essential to the economic infrastructure of a country. The condition of the rail infrastructure directly impacts the safety and reliability of rail operations [4]. Traditionally, defects are detected through manual visual inspection, where workers walk along the railway track to identify surface defects. However, this method is labour intensive, inefficient, and prone to subjective influences, making it inadequate for complete rail maintenance [8]. Improved detection methods are necessary to overcome these limitations and ensure the continued safety and reliability of rail transportation.

1.1.2 Significance of neuromorphic computing in defect detection

An improved defect detection method involves the use of an automatic defect detection system, such as a machine vision system (MVS). MVS employs advanced processing techniques to detect defects in images, using filtering, structuring, and statistical methods to extract meaningful information from visual data. Despite its advantages, MVS faces challenges, particularly in identifying irregularly shaped and variously sized defects in extensive inspection areas. Addressing these challenges is crucial to improving the precision and effectiveness of rail surface defect detection, ensuring the reliability of the system, and improving maintenance of the rail infrastructure [9].

Current MVS use conventional computing to implement processing techniques, relying on the Von Neumann architecture. However, this architecture faces significant challenges, including the deceleration of device scaling due to power and voltage constraints and the notorious 'Von Neumann bottleneck', which is the data transfer link between memory and the processor. This bottleneck is particularly problematic in data-centric applications such as real-time image recognition and natural language processing, leading to a substantial computational burden. Consequently, these systems require high power consumption and prolonged processing times. Despite advances, state-of-the-art Von Neumann systems still struggle to match the performance levels achieved by an average human, highlighting the urgent need for innovative solutions to overcome these limitations and improve the efficiency and effectiveness of defect detection in rail infrastructure [10].

To address these challenges, the emerging field of neuromorphic computing offers a promising solution for defect detection. Neuromorphic computing involves electronic circuits and devices that mimic the architecture of biological nervous systems. These circuits use mixed-mode analogue/digital Complementary Metal Oxide Semiconductor (CMOS) transistors, manufactured using standard Very Large-Scale Integration (VLSI) processes. Unlike conventional computing, neuromorphic systems process information efficiently through asynchronous, event-driven methods, demonstrating adaptability and fault tolerance. They generate complex behaviours by combining multiple instances of simpler elements and use memory structures in unique ways [11]. Specifically, non-volatile memory (NVM) crossbar arrays emulate the extensive parallelism and energy-efficient operation of the human brain. This approach offers significant advantages for real-time image recognition and defect detection, potentially revolutionising the field with its innovative and efficient processing capabilities [10].

The parallel structure of the human brain, which connects numerous low-power computing elements (neurons) and adaptive memory elements (synapses), excels in tasks involving classification of unstructured data and pattern recognition. Each individual neurone receives inputs from up to 10,000 other neurons, creating a highly interconnected and efficient network. Using dense arrays of NVM elements, neuromorphic computing has the potential to replicate this complex connectivity in hardware. However, replicating the intricate structure of the brain in artificial systems presents significant engineering challenges, including:

- Scaling connectivity
- Achieving low power consumption
- Material limitations
- Managing heat
- Minimising latency
- Ensuring fault tolerance and robustness
- Precision in synaptic weight storage
- Device variability
- Scaling fabrication
- Developing effective software and algorithms

Despite these challenges associated with implementing advanced connectivity in neuromorphic hardware, pursuing this frontier holds the potential not only to enhance computational efficiency and performance in practical tasks such as rail defect detection [10], but also to deepen our understanding of neural function. These advancements could, in turn, inform broader developments in machine learning theory, bridging the gap between biologically inspired computation and real-world applications.

This study aims to compare the effectiveness of traditional defect detection methods – including visual inspection, ultrasonic testing, and conventional computer vision-based techniques - with neuromorphic computing approaches, specifically SNNs and CNNs. By reviewing existing applications in rail condition monitoring and identifying the limitations of conventional methods in terms of precision, scalability, and real-time responsiveness, the research evaluates how neuromorphic strategies can overcome these barriers. Ultimately, the study proposes innovative, hardware-aware solutions that leverage neuromorphic advantages to enhance the accuracy, speed, and efficiency of defect detection systems – contributing to safer and more reliable railway operations.

1.2 RESEARCH OBJECTIVES

1.2.1 Research question

How do architectural choices between SNNs and CNNs, alongside the selection and configuration of hardware components (e.g., neuromorphic processor like Akida, CPUs, sensors), impact the performance ceiling – measured in terms of accuracy, latency, energy efficiency, and real-time feasibility – of neuromorphic-based rail surface defect detection system, particularly in energy-constrained, edge-deployment environments?

1.2.2 Research sub-questions

1. How do SNNs and CNNs compare in terms of performance, energy efficiency, and real-time suitability when deployed on neuromorphic hardware platforms, such as Akida processor, particularly for on-device defect detection tasks requiring low latency and minimal power consumption?
2. How do hand-crafted features and model types affect the performance and computational efficiency of the system across different hardware platforms?
3. How can the hardware components (e.g., neuromorphic processor, CPU, sensors, power supply, cooling) be optimally selected and configured to meet the minimum requirements for a neuromorphic-based rail surface defect detection system?

1.3 METHODOLOGY

To address the overarching research question, a comprehensive methodology (Figure 1-4) is devised to assess the upper limits of performance of a neuromorphic-based rail surface defect detection system.

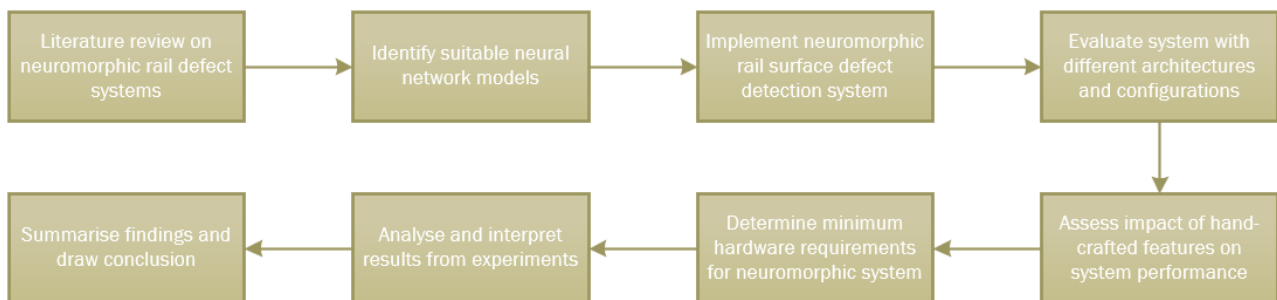


Figure 1-4: Methodology flow diagram

The following steps outline the approach:

1.3.1 Literature Review:

- Conduct an extensive review of the existing literature on neuromorphic computing, with a focus on applications in image processing and defect detection.
- Analyse previous studies that explored performance limits in neuromorphic-based systems and their deployment on platforms like the Akida processor.

1.3.2 Dataset Identification and Analysis:

- Identify and prepare suitable datasets containing rail surface images annotated for defect and no-defect classes.
- Ensure that datasets reflect real-world variability and environmental conditions through augmentation techniques (e.g., rotation, contrast adjustments).

- Perform exploratory data analysis (EDA) to understand class distribution, image quality, and structural variations.
- Use insights from data analysis to guide the selection of appropriate neural network models and training strategies.

1.3.3 Performance Metrics and Evaluation Framework:

- Define appropriate performance metrics for rail surface defect detection, including precision, precision, recall, F1-score, and inference speed.
- Develop a robust evaluation framework to quantify system performance under varying conditions and configurations.

1.3.3.1 Hardware Requirements Analysis:

- Conduct experiments across different hardware platforms (CPU, GPU, Akida) to evaluate their influence on model performance, latency, and energy efficiency.
- Assess system scalability in relation to hardware resources such as memory, processing power, and thermal constraints.
- Determine the minimum hardware configuration required to meet real-time and energy-efficient performance targets.

1.3.3.2 Validation and Robustness Testing:

- Validate experimental results using cross-validation and robustness checks to ensure reproducibility and generalisability.
- Test the system across different subsets and augmented versions of the dataset to assess its adaptability to unseen rail surface conditions.

1.3.4 Neural Network Architectures:

- Identify and evaluate various neural network models suited for neuromorphic deployment, including SNNs and CNNs.
- Consider both energy-efficient event-driven models and high-accuracy conventional models in the context of real-time rail inspection.

1.3.4.1 Evaluation of Neural Network Architecture:

- Implement and train selected neural network architectures across multiple hardware platforms.
- Evaluate the performance of each architecture using the defined metrics.
- Compare and contrast their strengths, weaknesses, and suitability for neuromorphic systems.

1.3.5 Feature Engineering and Impact Assessment:

- Investigate the role of hand-crafted features such as Grey-Level Co-occurrence Matrix (GLCM), Local Binary Pattern (LBP), and edge detection techniques.
- Examine how these features interact with neural models in terms of detection accuracy and computational efficiency.

1.3.5.1 Feature Impact Analysis:

- Integrate selected hand-crafted features into the model pipeline.
- Evaluate their impact on performance through controlled experiments.
- Analyse trade-offs between traditional feature engineering and end-to-end feature learning within neural networks.

By systematically addressing the research sub questions through literature review, data analysis, architecture evaluation, hardware benchmarking, and feature assessment, this study aims to determine the upper performance limits of neuromorphic-based rail surface defect detection systems. The methodology supports informed decisions on the most effective combination of neural architectures, feature strategies, and hardware configurations to enable accurate, real-time, and energy-efficient defect detection.

1.4 DISSERTATION OVERVIEW

An overview of each chapter is described in the sections below.

1.4.1 Chapter 2: Critical literature survey

This chapter provides a comprehensive review of existing research on neuromorphic computing and rail surface detection. It explores the use of CNNs and SNNs, highlighting their respective strengths in performance and energy efficiency. The review also examines the limitations of traditional hand-crafted features and emphasises the benefits of image-based learning. By identifying key trends and gaps in current literature, the chapter establishes the foundation for evaluating neural architectures and hardware platforms suited for real-time, energy-efficient defect detection in railway systems.

1.4.2 Chapter 3: Proposed system configuration and platform readiness

Chapter 3 presents the system-level considerations required for deploying rail surface defect detection models in real-world environments. It evaluates the suitability of three hardware platforms—CPU, GPU, and the Akida neuromorphic processor—by analysing their configuration, resource constraints, sensor integration, and software toolchain alignment. The chapter establishes a practical foundation for performance benchmarking and deployment feasibility explored in subsequent chapters.

1.4.3 Chapter 4: Neural network architectures

Chapter 4 presents and evaluate key neural network architectures for neuromorphic implementation, focusing on CNNs and SNNs. It explains the experimental setup, performance metrics, and data augmentation techniques used. The chapter highlights that CNNs on GPU offer the highest accuracy, while the Akida-implemented CNN provides a strong balance between performance and efficiency. SNNs, although more energy-efficient, show lower accuracy. The findings help illustrate the trade-offs between model performance and resource usage in real-time rail defect detection.

1.4.4 Chapter 5: Feature engineering and impact analysis

This chapter investigates the impact of handcrafted features – engineered from an augmented rail surface image dataset - on defect detection performance. These features are benchmarked against end-to-end CNN models trained on raw images to evaluate their effectiveness. The analysis highlights trade-offs between traditional feature engineering and direct image-based learning in terms of accuracy and efficiency.

1.4.5 Chapter 6: Hardware requirements analysis

Chapter 6 investigates the hardware requirements for optimal performance, comparing CPU, GPU, and neuromorphic platforms. It highlights the advantages of neuromorphic hardware – such as the Akida processor – for real-time, low-power defect detection, particularly in resource-constrained environments. The chapter also includes validation and robustness testing to ensure reliable and scalable system deployment.

1.4.6 Chapter 7: Discussion and Conclusion

The final chapter synthesizes key findings, highlighting contributions to knowledge and addressing study limitations. It reflects on the research implications, discusses practical applications for rail surface defect detection, and concludes with a summary of objectives and main outcomes, suggesting avenues for future research.

CHAPTER 2: CRITICAL LITERATURE SURVEY

2.1 CHAPTER INTRODUCTION

In this chapter, we explore the literature surrounding improvements in rail surface defect detection, with a focus on data augmentation, neuromorphic computing, and their convergence in prior defect detection efforts. We begin by examining data augmentation techniques aimed at enhancing model robustness – an area of particular relevance given the real-world dataset used in this study. The dataset comprises 1514 high-resolution grayscale images of rail surfaces, collected from Transnet’s operational infrastructure and representing both defective and non-defective conditions. To address the limited dataset size and improve generalisation, a comprehensive augmentation pipeline is employed, including geometric transformations (flipping, cropping, rotating), colour space conversions (grayscale, HSV), noise injection, and advanced methods like random erasing. These methods expand the dataset to 248374 images, creating a diverse and balanced input set for training both CNN and SNN models. Following this, the chapter delves into the role of neuromorphic computing in defect detection, reviewing key developments in energy-efficient, real-time model deployment. Finally, we synthesize insights from these domains to assess strategies specific to rail surface defect detection, setting the stage for deeper investigation and experimentation in subsequent chapters.

2.2 RAIL SURFACE DEFECT DETECTION

In rail surface defect detection, image processing techniques play a vital role in preparing images for accurate analysis. The goal is to capture clear, detailed images in real-time from a locomotive’s perspective, which is challenging due to varying lighting conditions, vibrations, dust, and the need for rapid processing. Image enhancement is a crucial step to improve the visual quality of these images, making defect features more visible, reducing noise, and highlighting relevant image characteristics.

2.2.1 *Image processing techniques*

2.2.1.1 *Contrast enhancement*

Contrast enhancement is essential in image processing for improving feature visibility, though traditional methods like global histogram equalisation (GHE) can cause excessive brightness changes and unnatural appearances. Enhanced dynamic quadrant histogram equalisation with plateau limit (EDQHEPL) addresses these issues by dividing the input image histogram into sub-histograms, clipping each based-on pixel count, assigning new dynamic ranges, and independently applying equalisation. This method reduces saturation artifacts, preserves brightness, and enhances contrast, particularly in low-contrast images [12].

When tested against traditional methods on images from the Signal and Image Processing Institute (SIPI) database, EDQHEPL demonstrates its ability to improve feature visibility while maintaining a natural appearance, making it valuable in applications like medical imaging, satellite analysis, and consumer electronics, where brightness preservation is crucial [12].

2.2.1.2 Smoothing and sharpening

Smoothing and sharpening are fundamental techniques in image processing used to enhance image quality by reducing noise and enhancing edges, respectively. Smoothing primarily aims to diminish noise by applying blurriness, which reduces high-frequency components while preserving significant image structures. This process involves averaging pixel values within a defined neighbourhood, resulting in a more homogeneous and less noisy image. Common smoothing filters include the Mean, Gaussian, Median, Min, and Max filters. The Mean filter, also known as the average filter, operates by averaging the pixel values within a kernel, reducing the variations caused by noise. The Gaussian filter applies a Gaussian function to the pixel values within the kernel, providing a smoother transition and better preservation. The Median filter, a non-linear smoothing filter, replaces the centre pixel of the kernel with the median value of the neighbourhood, effectively removing salt-and-pepper noise. Min and max filters replace the centre pixel with the minimum or maximum value in the neighbourhood, useful for applications where specific features need to be suppressed or highlighted [13].

Sharpening techniques enhance the edges and fine details of an image by increasing the contrast between adjacent pixels, making edges more distinct and prominent. Common sharpening filters include the Laplacian, Roberts cross gradient, and Sobel filters. The Laplacian filter, a second-order derivative filter, highlights regions of rapid intensity change, making it useful for edge detection. The Robert cross gradient filter detects edges using the first-order derivative of the image intensity with two small convolution kernels for the x and y directions. The Sobel filter, also using first-order derivatives, employs larger convolution kernels that consider the gradient in both the horizontal and vertical directions, producing thicker edges and being more robust to noise compared to the Roberts filter [13].

Techniques such as the median filter and bilateral filter are particularly effective for noise reduction. The median filter removes noise by replacing each pixel's value with the median value of its neighbouring pixels, preserving edges while eliminating salt and pepper noise. The bilateral filter smooths the image by considering both the spatial proximity and the intensity difference of neighbouring pixels, preserving edges more effectively. Sharpening enhances edges and fine details within an image using kernel-based filtering techniques. Both sharpening and smoothing are crucial in applications like defect detection, where clarity and precision are paramount [14].

Evaluating the performance of these filters involves metrics such as Peak Signal to Noise Ratio (PSNR) and Mean Square Error (MSE), which quantify the effectiveness of smoothing and

sharpening techniques. PSNR measures peak error, while MSE measures the average squared differences between the denoised and original images. Other important metrics include normalised cross correlation (NK), average difference (AD), structural content (SC), maximum difference (MD), and normalised absolute error (NAE). These equations are essential for comparing the performance of different filters used in noise reduction. Among the filters compared, the adaptive weighted median filter consistently provides superior performance across different noise types, as demonstrated by these metrics [15].

2.2.2 Machine learning-based approaches

After the initial phase of image enhancement and feature extraction (discussed in chapter 4), the next crucial step in rail surface defect detection is the application of machine learning techniques.

2.2.2.1 Support vector machine

Support vector machines (SVMs) are a powerful set of supervised learning algorithms used primarily for classification and regression tasks. Developed by Cortes and Vapnik, SVMs aim to find the optimal hyperplane that separates data points of different classes with the maximum margin (Figure 2-1), enhancing the model's ability to generalise to new data. This is achieved through Structural Risk Minimisation (SRM), which minimises an upper bound on the expected risk, thus avoiding overfitting to the training data [16].

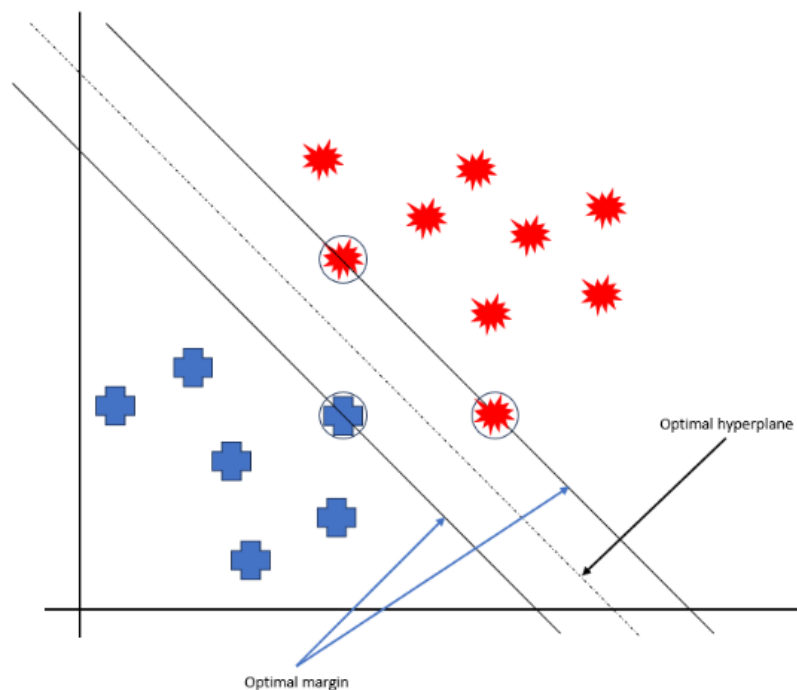


Figure 2-1: Support vector machine showing the optimal margin and optimal hyperplane [17]

In cases where data is not linearly separable, SVMs utilise the kernel trick, which projects input data into a higher-dimensional feature space to enable linear separation. Commonly used kernel functions – such as Polynomial, Gaussian Radial Basis Function (RBF), and Sigmoid – allow SVMs to model

complex, non-linear patterns within the data effectively. However, the suitability of SVMs is closely tied to the quality and characteristics of the input data.

In this study, feature sets derived from handcrafted techniques – such as Gray-Level Co-occurrence Matrix (GLCM), Local Binary Pattern (LBP), Sobel, and Canny edge detectors – are evaluated for their separability and representational effectiveness. To assess feature quality, dimensional reduction and visualisation methods are employed. Principal Component Analysis (PCA) and t-distributed Stochastic Neighbour Embedding (t-SNE) are used to project the high-dimensional handcrafted features into two-dimensional space. The projections provide visual insight into the clustering and overlap between the “defect” and “no defect” classes. In addition, quantitative metrics such as the separability index are employed to further characterise class distinction within the feature space. These analyses help determine whether the input data is linearly or non-linearly separable, directly influencing the choice of kernel functions and model architecture.

Benchmarking is performed by comparing the performance of SVMs trained on these engineered features against an end-to-end CNN trained directly on raw image data from the same augmented dataset. The comparison spans multiple evaluation metrics, including classification accuracy, F1-score, inference time, and energy consumption, across various hardware platform. This comprehensive framework ensures that the role of feature separability and data dimensionality is fully considered in determining the effectiveness of SVMs and other model architectures in rail surface defect detection [17].

The SVM algorithm involves solving a quadratic optimisation problem to determine the support vectors and the optimal hyperplane. For non-linearly separable data, SVMs introduce slack variables and a penalty parameter (C) to form a soft margin, balancing the trade-off between maximising the margin and minimising classification errors. This robust theoretical foundation, combined with the kernel trick, enables efficient computation of the optimal hyperplane in the transformed feature space without explicitly performing the transformation [18].

SVMs have demonstrated consistently strong performance across a range of domains, including bioinformatics and chemometrics, where high-dimensional, complex data is common. For instance, in bioinformatics, SVMs have achieved classification accuracies exceeding 90% in cancer diagnosis tasks using gene expression data. Guyon et al. (2002) [19], for example, reported 96% accuracy in leukaemia classification using microarray datasets. Similarly, in chemometrics, SVMs are frequently used for spectral analysis and compound classification, with studies reporting classification accuracies between 85% and 95%, depending on data quality and feature selection. These successes are attributed to SVMs' ability to effectively handle high-dimensional input spaces and apply non-linear kernel transformations to improve separability. Metrics such as accuracy, area under the curve (AUC), and F1-score are commonly used to evaluate their performance. Including these quantitative examples not only validates the robustness of SVMs in real-world applications but

also positions them as a viable baseline for comparison in this study, particularly when evaluating alternative models such as CNNs and SNNs for rail surface defect detection [20].

Introduced by Leo Breiman, the Random Forest algorithm is a versatile classifier used for both regression and classification tasks, particularly effective in handling the latter [21]. It comprises an ensemble of decision tree classifiers, each built on a subset of the training data and using a random subset of features at each node to ensure diversity among the trees (Figure 2-2). The decision of the classifier is made by aggregating the outputs of individual trees through majority voting for classification tasks [17].

Figure 2-2: A Random Forest Classifier with multiple decision trees [17]

The strength of the random forest lies in its ability to reduce overfitting and improve prediction accuracy. This is achieved by incorporating randomness both in the selection of data samples for each tree and in the feature selection process at each node of the tree. By doing so, random forests can maintain low correlation between individual trees, which helps in achieving higher overall accuracy. Internal measures, such as out-of-bag (OOB) estimates, are used to monitor the performance, providing insights into the error rates, strength, and correlation of the trees within the forest [22].

Random forests are widely recognised for their robustness against overfitting, their ability to handle high-dimensional datasets, and their built-in capabilities for variable importance ranking, out-of-bag (OOB) error estimation, and missing value imputation. These characteristics make them particularly suitable for classification tasks involving noisy or incomplete data. In the context of rail defect detection, Random Forests have demonstrated competitive performance. For example, studies such as Xu et al. (2019) [23] reported classification accuracies above 92% when using Random Forests to identify rail surface defects based on vibration signal features and image-based data. Similarly, Zhang et al. (2021) [24] applied Random Forests to visual inspection data and achieved an F1-score of 90.3% in classifying defect types. The outcomes highlight the algorithm's effectiveness in extracting relevant features and making reliable predictions even in complex, real-world scenarios. As a result, Random Forests continue to be a practical benchmark for defect detection systems, offering a strong baseline against which deep learning and neuromorphic models can be compared.

2.2.2.3 K-Nearest Neighbour

The k-Nearest Neighbour (k-NN) algorithm is a fundamental and widely used method in pattern classification due to its simplicity and effectiveness. It operates by identifying the k closest training examples to a given test example using a predefined distance metric, typically Euclidean distance (Equation 2-1). The class label of the test example is then determined by the majority vote of its k nearest neighbours (Figure 2-3). This non-parametric approach does not make any assumptions about the underlying data distribution, making it versatile for various applications [25, 26].

$$d(x_i, x_j) = \sqrt{\sum_{r=1}^n w_r (a_r(x_i) - a_r(x_j))^2} \quad 2-1$$

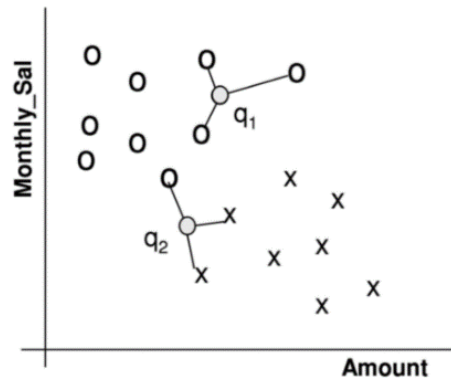


Figure 2-3: Example of k-Nearest Neighbour [27]

The performance of the k-NN algorithm is highly dependent on the choice of k and the distance metric used. Traditional k-NN assumes a fixed number of neighbours (k) for all test examples, which can lead to suboptimal performance, especially in cases with imbalanced datasets. To address this limitation, an adaptive k-NN algorithm (AdaNN) has been proposed. AdaNN dynamically determines the optimal k for each test example based on the optimal k of its nearest neighbour in the training

set. This approach allows for more flexible and accurate classification by adapting to the local structure of the data [25].

The k-NN algorithm defers the classification task until a new instance needs to be classified, making it a “lazy learning” algorithm. This characteristic can be computationally intensive, especially for large datasets, as it requires calculating the distance between the test example and all training examples each time a classification is made. Additionally, the choice of k and the distance metric can significantly affect the performance of the algorithm, with small values of k making the model sensitive to noise, while large values may dilute the decision boundary [26, 27].

The k-NN algorithm is valued for its simplicity, ease of implementation, and lack of a formal training phase, making it a popular choice as a baseline classifier. Its instance-based nature allows for intuitive decision-making, particularly in applications where interpretability is important. In the context of image-based defect detection, k-NN has achieved reasonable classification performance when applied to small and well-structured datasets. For example, studies such as Huang et al. (2018) [28] reported classification accuracies of up to 88% when using k-NN for surface defect detection in steel manufacturing based on GLCM features. Similarly, in rail defect detection, preliminary models using k-NN have shown F1-scores in the range of 80-85% depending on feature selection and noise levels. However, the algorithm’s performance can degrade significantly on high-dimensional or noisy datasets due to its sensitivity to irrelevant features and distance metrics. Despite these limitations, k-NN remains useful for prototyping, comparison, and situations where computational resources are limited, and model interpretability is a priority [26, 27].

2.2.2.4 Naïve bayes

The Naïve Bayes classifier is a supervised learning algorithm that uses Bayes’ theorem to calculate the posterior probability of a class given a set of features. The formula for Bayes’ theorem (Equation 2-2) is:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \quad 2-2$$

Where $P(A|B)$ represents the probability of class A given predictor B , $P(A)$ is the prior probability of class A , $P(B|A)$ is the likelihood of predictor B given class A , and $P(B)$ is the prior probability of predictor B [17].

Naïve Bayes assumes that the features are independent given the class, which simplifies the calculation of the likelihood (Figure 2-4). This classifier assigns the classification to the class with the highest posterior probability. There are different types of Naïve Bayes classifiers, including Gaussian Naïve Bayes, which assumes that features are normally distributed, Multinomial Naïve Bayes, which is suited for multinomially distributed data, and Bernoulli Naïve Bayes, which is used for binary or Boolean features [17].

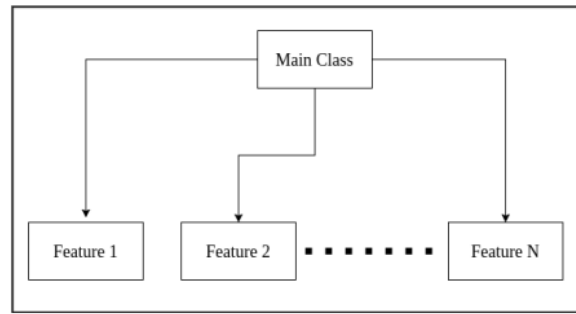


Figure 2-4: Conceptual illustration of Naive Bayes [17]

These classifiers are known for their simplicity, efficiency, and effectiveness in various tasks such as text classification and spam detection. However, their accuracy can be affected if the independence assumption is violated [17].

Despite the independence assumption often being violated in real-world applications, Naïve Bayes classifiers are still favoured due to their computational efficiency and strong performance. The Multinomial Naïve Bayes, for instance, is particularly effective for text classification where word frequency data is used [29].

Naïve Bayes classifiers are computationally efficient, with training time complexity being linear relative to the number of training examples, and the space complexity being linear in the number of features. This makes them suitable for large-scale applications. Moreover, they maintain good performance even when the independence assumption does not hold, thanks to their probabilistic approach [29].

To further enhance their performance, especially in tasks where precision is crucial such as spam detection, techniques like term weighting and document length normalisation are employed. These techniques help mitigate the classifier's overconfidence by addressing the independence assumption violations and handling feature sparsity in shorter documents [29].

In summary, although the independence assumption of Naïve Bayes classifiers is often unrealistic, their practical performance and computational efficiency make them valuable for various classification tasks. Enhancements like per-document length normalisation and feature weighting improve its effectiveness, particularly in text classification and other domains [30, 31].

2.2.3 Non-destructive testing

There are several Non-Destructive Testing (NDT) techniques to inspect the surface of a railway track for defects. These techniques include ultrasonic testing (UT), magnetic induction or magnetic flux leakage (MFL), pulsed eddy current (PEC), radiography and visual inspection. The ultrasonic testing technique is primarily used for the detection and evaluation of internal rail defects [6]. The technique utilises a high frequency sound pulse that is sent into the rail to look for the return of reflected or scattered energy with the use of transducers [7]. These transducers are contained within a liquid-filled tyre [8]. Rail defects are unpredictable; therefore, energy beams are sent at different angles.

The transducer is positioned in such a way to look across the rail head. This position will assist in detecting longitudinal defects [7]. The amplitude of the reflected energy as well as the time when they occurred can provide information with regards to the integrity of the rail. This information relates to the location and type of defect that was detected. Ultrasonic energy is transmitted to the rail at several different incident angles since defects are in various parts of the rail. The incident angles will maximise the probability of detection (POD) of any damaging features existing in the rail [9]. The incident angles that are generally used are 0°, 37° or 45° and 70°.

Ultrasonic testing techniques can detect deep surface-breaking defects as well as internal defects in the rail head and web. However, RCF defects smaller than 4mm becomes undetectable by high-speed ultrasonic systems. The ultrasonic testing technique can also miss defects such as corrosion on the foot of the rail since the transducer can only partially scan these parts of the rail. The ultrasonic technique is inadequate to detect surface defects due to dead zones such as the surface area [10]. Another challenge with ultrasonic testing is the adjustment of the signal threshold (amplitude of the signal) and the position of the window (acquisition time). This challenge needs to be overcome for reliable inspections of the rail track at high speeds [9].

The MFL technique uses sensors that can be permanent magnets or direct current (DC) electromagnets. These magnets are used to generate a strong magnetic field to magnetise the rail for inspection to saturation. Using metal brushes or air coupling, the magnetic flux lines are coupled into the rail [9]. Search coils are positioned at a constant distance from the rail. These coils detect possible changes in the magnetic field that is generated with the permanent magnets or DC electromagnets near the head of the rail. When defects are present in the rail, ferromagnetic domains in the rail do not support the magnetic field flux and this then leads to flux leakage. Another coil known as the sensing coil is then used to detect the change in the magnetic field [9].

The eddy current technique was developed to detect RCF damage and wheel burns on the running surface of the rail. The technique is based on the physics of electromagnetic induction. A high current is sent into the rail with the use of brushes that are in contact with the rail head and forces the rail to become part of an electric circuit. When the current comes across a defect in the rail, it will simply flow around the defect. Sensors with the ability to detect a disturbance in the magnetic field is used to detect the distortion in the current flow [7]. The eddy current sensor is made up of an exciting coil and a sensing coil. A secondary magnetic field is generated by the eddy currents. Any changes to this magnetic field are detected with the sensor's search coil in the form of an induced voltage. If there are no defects in the rail, the impedance of the eddy current will remain constant else, fluctuations in the secondary magnetic field leads to the rise in impedance of the eddy currents [9]. The eddy current sensor is very sensitive to lift-off variations and should be placed at a constant distance that is no more than 2mm away from the running surface of the rail [9].

The radiography technique uses Gamma or X-ray for inspection. Inspections were done with the use of Gamma-ray source and film to obtain a radiographic image of the inspected rail. This method is

very efficient for NDT; however, the use of radiography has health and safety drawbacks [9]. This technique is also very time consuming and is therefore used to verify the areas where defects have already been identified. Radiography can provide information such as location and size of defects as well as the type of internal defect. Visual inspection has become very popular in the railway industry. A variety of visual camera-based inspection systems have been implemented.

2.3 DATA AUGMENTATION TECHNIQUES

2.3.1 Introduction to data augmentation

Deep learning models have significantly advanced tasks like image classification, object detection, and image segmentation, largely due to CNNs. However, improving the generalisation ability of these models – how well they perform on unseen data – remains a challenge. Models that generalise poorly tend to overfit, performing well in training data but poorly in validation or testing data [32].

Data Augmentation helps prevent overfitting and enhances generalisation by artificially expanding the training dataset through label-preserving transformations. This approach assumes that more information can be extracted from the original dataset through augmentations, thus reducing the performance gap between training and validation sets and improving future testing performance [32].

Data augmentation techniques include data warping and oversampling. Data warping involves transforming existing images in ways that preserve their labels, such as geometric transformations and random erasing. These transformations create a more diverse training data set. Oversampling creates synthetic instances to add to the training set, addressing imbalanced class distributions and preventing the model from being biased towards the majority class. Methods include mixing images and using generative adversarial networks (GANs) [32]. For CNNs, having a large and diverse training dataset is crucial for performance [33].

Despite its benefits, data augmentation faces challenges. Techniques used for one task may not apply directly to another, like object detection versus semantic segmentation. Additionally, there is a lack of theoretical research on optimal dataset sizes, often relying on personal experience and experimentation [33].

Data augmentation methods are classified into unsupervised and supervised categories. Unsupervised methods do not depend on data labels and include transformations like flipping and rotation. Supervised methods, which are label-dependent, generate augmented samples according to specific categories, with GANs being a common example [34].

2.3.2 Types of data augmentation methods

The effectiveness of data augmentation began with simple transformations such as horizontal flipping, colour adjustments, and random cropping. The “safety” of an augmentation method refers to its ability to preserve the original label after transformation (e.g., rotating images where orientation is key). While transformations that do not preserve labels can help models learn to express

uncertainty, they require creating refined labels, which is computationally expensive. This highlights the need to consider the safety and domain-specific nature of augmentations, as any image processing can potentially change labels at certain distortion levels, making it challenging to develop universal augmentation policies [32].

2.3.2.1 Geometric transformations

In this section, we explore various image-augmentation techniques that use geometric transformations and other image processing methods to enhance training dataset diversity and model robustness. Key augmentations include flipping, colour adjustments, cropping, rotation, and noise injection (Figure 2-5). Assessing the 'safety' of these transformations is crucial, as some may preserve labels in one context but not in another [32]. The complexity of generating refined labels for transformed data underscores the need for careful consideration, as domain-specific requirements often dictate the balance between augmentation benefits and potential label distortion. We will examine the practicality of each technique in label preservation and its impact on training efficacy and model performance [32, 33].

- Flipping (Figure 2-5(b)):

Horizontal or vertical flipping is a common and easy-to-implement augmentation technique that has proven useful in many contexts [32]. This method involves flipping images horizontally or vertically, often by rotating them in multiples of 90 degrees. Although effective for many datasets, it is not a label-preserving transformation for text or digit recognition datasets [32, 33]. Some frameworks do not support vertical flipping directly, but this can be achieved by rotating the image 180 degrees and then performing a horizontal flip [33].

- Rotation (Figure 2-5 (c)):

Rotation augmentations involve rotating an image between 1 and 359 degrees to the right or left. The safety of these augmentations is determined by the rotation degree parameter [32]. Rotating images at 90-degree angles typically avoids adding background noise, while minute-angle rotations can introduce noise. This noise arises from the interpolation process during rotation, where new pixel values must be estimated. If the background is black or white, this noise blends in; however, with multicoloured backgrounds, the noise may be learned as a feature [33].

- Cropping (Figure 2-5 (d)):

Cropping involves taking a part of the original image and resizing it to the original dimensions. This technique is useful to standardise images of different sizes and typically cut a central patch or perform random cropping, reducing the size of the image [32]. Random cropping can provide effects like translations, though cropping reduces the input size while translations preserve the

spatial dimensions. Depending on the chosen reduction threshold, cropping may not always preserve the original image label [33].

- Translation (Figure 2-5 (f)):

Translation involves shifting images left, right, up, or down to reduce positional bias and help in recognising objects anywhere in the images. As the image is translated, the remaining space can be filled with constant values like 0s or 255s, or with random or Gaussian noise, preserving the spatial dimensions of the image [32]. This padding ensures that the image size remains the same while reducing the bias toward the position of the object [33].

- Noise injection (Figure 2-5 (g)):

Noise injection involves adding a matrix of random values, often drawn from a Gaussian distribution to images. This technique, which can include the addition of random white and black dots (salt and pepper noise), helps CNNs learn more robust features and prevents overfitting [32]. By introducing randomness, the model learns from a wider range of data, leading to stronger and more reliable feature recognition [33].

Geometric transformations are effective solutions for addressing positional biases in training data, helping to align the distribution of training data with testing data. They are relatively easy to implement, but come with disadvantages such as increased memory usage, higher computational costs, and longer training times. Additionally, transformations like translation or random cropping (Figure 2-5 (e)) need manual verification to ensure that they do not alter image labels. Although useful, the scope of geometric transformations is limited, as the biases between training and testing data can be more complex than positional or translational variances [33].

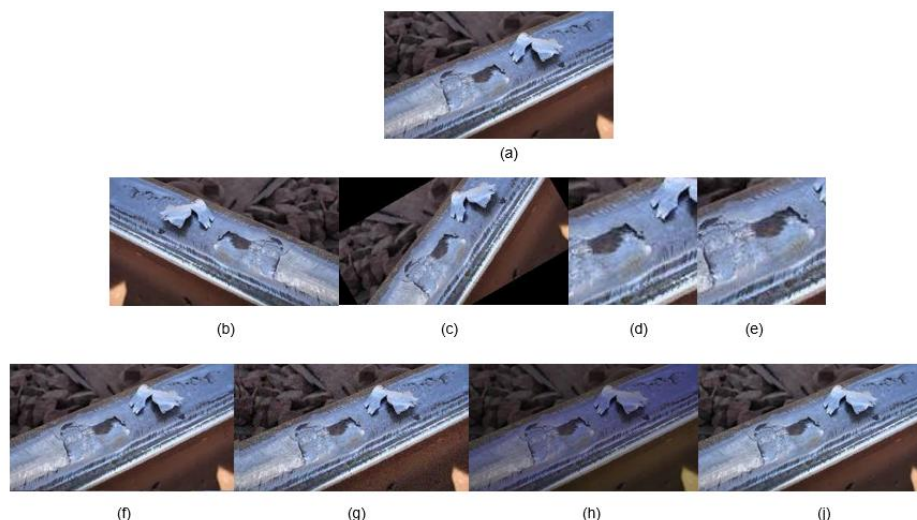


Figure 2-5: Original and transformed images. (a) original image, (b) flipping, (c) rotation, (d) cropping, (e) random cropping, (f) shifting, (g) noise, (h) colour jittering, (i) PCA jittering [34]

2.3.2.2 Colour space transformations

Colour space transformations (Figure 2-6), also known as photometric transformations, involve manipulating the colour channels of image data, typically encoded as three stacked matrices representing Red, Green, and Blue (RGB) values. These transformations address common lighting biases in image recognition. Simple techniques include adjusting pixel values to correct overly bright or dark images, isolating single-colour channels, and restricting pixel values to a certain range. More advanced methods involve altering the colour histogram to change the colour characteristics of the image. These adjustments help mitigate lighting condition challenges and enhance image robustness [32, 33].

However, they come with disadvantages, such as increased memory and computational costs, and they may not always preserve original image labels, especially when colour is a critical feature. Converting images from RGB to greyscale can simplify datasets and speed up computation but may reduce accuracy. Therefore, while colour space transformations can eliminate colour biases, they need to be applied judiciously depending on the task requirements [32, 33].



Figure 2-6: Colour space transformations (a) Original image (b) LAB colour transformation (c) YCrCb transformation (d) HSV transformation [33]

2.3.2.3 Random erasing

Random erasing (Figure 2-7) is a data augmentation technique designed to address image recognition challenges due to occlusion, where parts of an object are unclear. Inspired by dropout regularisation, random erasing operates in the input data space by randomly selecting a patch of an image and masking it with 0s, 255s, mean pixel values, or random values. This forces the model to learn more descriptive features and prevents overfitting to specific visual features, ensuring that it pays attention to the entire image. By removing certain input patches, random erasing helps the model to generalise better and recognise objects even with partial occlusion, although it may not always preserve the original label [32, 33].

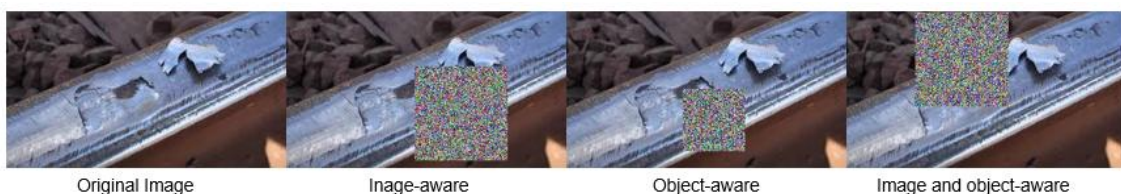


Figure 2-7: Random erasing for object detection [32]

2.3.2.4 Generative modelling

GANs (Figure 2-8) are a powerful technique for generating new samples that closely resembles the original dataset, gently enhancing training datasets for machine learning models. GANs consists of two models: the generator, which creates new data, and the discriminator, which assesses whether the data is real or fake. Renowned for their quality and efficiency, GANs are a leading approach in generative modelling, a broader category that includes methods such as variational autoencoders (VAE) [32].

VAEs compress data into simpler forms, which GANs can then enhance to add detail and realism. This combination supports the generation realistic training samples, enhancing model performance. GANs are effective in terms of computational speed and quality output, positioning them as a highly effective approach with generative modelling for data augmentation [32]. By generating artificial samples that preserve the characteristics of the original dataset, GANs address limitations in training data, leading to improved classification models [33].

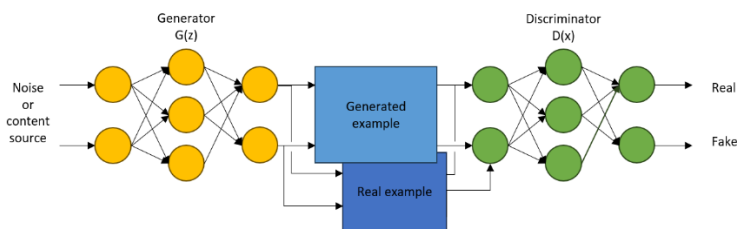


Figure 2-8: Illustration of GANs [32]

2.3.2.5 Neural style transfer

Neural style transfer is an impressive demonstration of deep learning capabilities, known for its artistic applications and usefulness in data augmentation. It manipulates the representations of images created in CNNs, allowing the style of one image to be transferred to another while preserving its content. This technique extends lighting variations and encodes different textures and artistic styles. For data augmentation, a set of styles can be applied to all images in the training set, but care must be taken to avoid biases if the style set is too small [32].

Neural style transfer combines a content image, a style image, and an input image to transform the input image to resemble the content image painted in the style of the style image [33]. This enhances the diversity of training datasets, making the models more robust to various styles and textures.

2.3.3 Summary

Image data augmentation falls into two main types: data warping and oversampling. Simple methods like flipping or cropping images are straightforward and improve image classification, while advanced techniques using GANs, or neural style transfer are more complex. Combining different augmentation methods, such as adding random erasing, can enhance image variety, but finding the right balance to avoid bias and overfitting is challenging. Advanced methods such as neural style

transfer offer new augmentation options by significantly changing the appearance of an image, but determining the best mix and extent of augmentation is essential to maintain a balanced and effective training data set [32].

2.4 NEUROMORPHIC COMPUTING

2.4.1 Advantages and uses of neuromorphic computing

Neuromorphic computing offers significant advancements over traditional von Neumann architectures by integrating memory and processing, which reduces latency and power consumption – ideal for high-efficiency, parallel processing. This architecture is particularly well-suited for applications like image segmentation, object recognition, and visual attention, where neuromorphic systems emulate human-like efficiency. The integration of components such as memristors, which allow for in-memory computation, enhances real-time processing capabilities, making neuromorphic computing especially valuable for tasks in edge devices, Internet of Things (IoT), and cyber-physical systems that require low-power, high-density computation [10, 35].

In defect detection, neuromorphic computing excels due to its high accuracy in complex visual tasks. Platforms like Intel's Loihi and IBM's (International Business Machines Corporation) TrueNorth utilise SNNs to model biological neuron interactions, enhancing performance for real-time image processing and machine vision. These systems improve industrial defect detection through advanced image processing algorithms and neuromorphic vision sensors, such as those that emulate the retina. By integrating with NVM devices like memristors, neuromorphic hardware offers energy-efficient solutions for tasks ranging from biomedical imaging to industrial material inspection [36, 37].

2.4.2 Neuromorphic technology and implementation

Neuromorphic computing relies on technologies like memristors (Figure 2-9), CMOS transistors, and crossbar arrays to emulate neural networks' structure and function. Memristors, or memory resistors, are resistive devices that store information as resistance and can switch states, making them ideal for mimicking synaptic connections. Memristor crossbar arrays (CBAs) (Figure 2-10) organise these devices in grid-like configurations where intersections serve as artificial synapses, performing high-density, parallel computations. When paired with CMOS technology, these CBAs support vector-matrix multiplications crucial for neural network functions, enabling efficient data processing in a scalable, in memory format [9, 38].

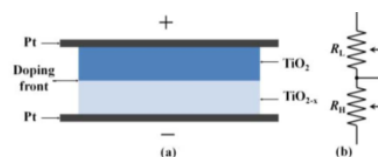


Figure 2-9: Memristor (a) structure, (b) equivalent circuit [38]

In neuromorphic systems, SNNs simulate brain-like behaviour by encoding information through spikes – short bursts of electrical activity. This approach allows neuromorphic systems to process data in an event-driven, asynchronous manner, saving energy compared to synchronous systems. These networks integrate learning algorithms, such as spike-timing-dependent plasticity (STDP), to adjust synaptic weights based on input patterns, facilitating adaptive, real-time learning. By combining digital and analogue components, neuromorphic implementations like IBM’s TrueNorth or Intel’s Loihi offer flexible, energy-efficient solutions for complex applications requiring real-time processing and adaptability [39, 40].

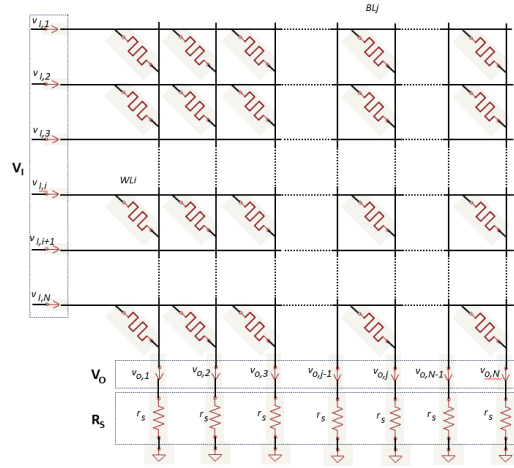


Figure 2-10: Memristor crossbar array [38]

2.4.3 Previous work on neuromorphic-based defect detection

Previous work on neuromorphic-based defect detection has demonstrated measurable performance gains by leveraging the unique capabilities of neuromorphic computing systems. Early research utilised memristor-crossbar arrays for in-memory computing, emulating the synaptic behaviour of biological neurons to perform low-power, parallel computations. These arrays, particularly in a 1-transistor-1-resistor (1T1R) configuration, effectively mitigated cell-to-cell interference and supported precise morphological image processing. For instance, Y. Liu et al. (2023) [41] applied a four-layer morphological filtering network using RRAM crossbars for defect detection in semiconductor and medical images, achieving the same accuracy as software-based morphology while improving energy efficiency by up to 358x compared to CPUs and 32x over GPUs. Similarly, HS Lee et al. (2021) [9] combined Gaussian-enhanced morphological operations with oxide-based memristors and demonstrated up to 10000x lower energy consumption versus CPU-based pipelines in applications involving fabric inspection, brain tumour segmentation (MRI), and surface analysis of graphene and cupric oxide thin films. These implementations retained high detection accuracy – often exceeding 90% - while significantly reducing power and latency through true in-memory, analogue computation [9]. Other neuromorphic platforms, such as the Intel Loihi processor and BrainChip Akida, have further demonstrated real-time defect detection in domains like medical imaging and industrial inspection. For example, Loihi achieved 85.6% accuracy in brain tumour

classification tasks with 20-70x lower energy consumption per inference than convolution CNNs on GPU [42]. The Akida processor delivered inference speeds of up to 100 FPS for visual tasks while operating under mW, maintaining CNN-level accuracy (~84-85%) with orders-of-magnitude energy savings [43]. These studies confirm that performance enhancements are primarily attributed to the hardware design – namely, event-driven processing and in-memory computation – rather than changes in algorithm architecture. Collectively, they highlight the potential of neuromorphic computing to deliver scalable, energy-efficient, and accurate solutions for defect detection across diverse application areas [44].

G. Crebbin's 2005 study introduced a neuromorphic approach to image segmentation using integrate-and-fire cells, aiming for efficient and real-time processing. These cells emitted output pulses when input light intensity surpassed a set threshold, synchronising their firing patterns to delineate distinct image regions or objects. This bio-inspired method facilitated low-power, high-speed image segmentation. While Crebbin's primary focus was on general image segmentation tasks, the techniques have been adapted for defect detection across various materials. However, the original study did not provide specific accuracy metrics or performance comparisons to traditional methods, making it challenging to quantify the exact improvements attributed to this neuromorphic approach [45].

In their 2016 study, G. Liu et al. integrated memristor crossbar arrays to emulate the parallel and distributed nature of biological neural networks, achieving a 94% accuracy in handwritten digit recognition tasks. This approach leverages memristors' low-power and high-efficiency characteristics for complex image processing tasks. Additionally, the use of dynamic Gaussian blur kernels with varied sigma values ranging from 0.7 to 1.1 has demonstrated to enhance defect detection accuracy by effectively distinguish defect information from background noise. These techniques have been successfully applied to various materials, including fabrics, brain tumours, graphene, and cupric oxide thin films, showcasing the potential for scalable and energy-efficient defect detection solutions in both industrial and medical applications [39].

A. Subramaniam, 2017, examined neuromorphic computing systems for improvement in image processing and machine vision tasks. This work involved using memristor-based CBAs to implement morphological image processing, which supports defect identification with low-power and real-time computation. These systems replicate neural architecture, facilitating scalable computation. By incorporating neuromorphic hardware with specific image processing algorithms, such as dynamic Gaussian blur kernel operation, this approach delivers higher performance than traditional methods. Quantitatively, memristive hardware accelerators with crossbar arrays showed a 6.89x increase in performance and a 5.2x decrease in power consumption relative to conventional memory configurations. Spiking neuromorphic processors for real-time segmentation processed up to 1,000 images per second, suitable for applications that require high throughput [35].

C.S. Thakur et al., 2017, applied spiking neuromorphic processors for real-time image segmentation. The processor uses the parallel processing and energy-saving features of neuromorphic systems to run image segmentation algorithms, including the graph-based random walker (RW) algorithm. By encoding data through spike rates, these neuromorphic systems perform segmentation, supporting defect detection in various materials. Implementing these algorithms on neuromorphic hardware enables processing speeds of up to 1000 images per second, fitting applications in biomedical imaging and other fields that need high processing power. This approach addresses the limitations of von Neumann architectures, providing a practical and scalable method for defect detection tasks [46].

Table 2-1: Comparative summary of CNN and SNN methods in rail defect detection

Study / Domain	Model Type	Dataset	Performance (Accuracy / F1-score)	Notes
Jie et al. (2009) – Rail crack detection [3]	CNN (ResNet)	20,000 images (track cracks)	96.2% accuracy	Demonstrated strong accuracy; required GPU acceleration.
Min et al. (2018) – Rail surface defects [8]	CNN (custom)	15,000 lab-based rail surface images	93.8% accuracy	Effective on small datasets; limited generalisation to field deployment.
Tan & Le (2019) – Rail corrugation detection [47]	CNN (EfficientNet)	Public rail images + augmentation	95.5% accuracy, F1 = 0.94	Data augmentation improved robustness under lighting variation.
HS Lee et al. (2022) – Neuromorphic inspection [9]	SNN (LIF + STDP)	10,000 rail surface images (spike-encoded)	86.4% accuracy	Low power; slower convergence compared to CNNs.

Table 2-1 summarises representative CNN and SNN studies in rail defect detection, highlighting datasets, performance ranges, and limitations. This comparison underscores the research gap in systematic evaluation across platforms, which addressed in the present study.

2.4.4 Current shortcomings of neuromorphic computing

Despite significant advances, neuromorphic computing faces key challenges in rail surface defect detection. Existing studies demonstrate potential in efficiency and accuracy but lack standardized benchmarks for rail applications, such as processing speeds, accuracy rates, and energy efficiency. This gap complicates the evaluation and deployment of neuromorphic solutions in real-world settings, underscoring the need for defined performance metrics [45, 46].

Additionally, the suitability of neural network architectures for neuromorphic implementation remains underexplored. Limited comparative studies on Spiking Neural Networks (SNNs), Convolutional Neural Networks (CNNs), and Recurrent Neural Networks (RNNs) hinder progress in optimizing

models for defect detection. Furthermore, the integration of hand-crafted image processing techniques—such as Gray-Level Co-occurrence Matrix (GLCM), Local Binary Patterns (LBP), Sobel filters, and Canny edge detection—into neuromorphic systems has yet to be thoroughly investigated. Improved understanding of these features could enhance accuracy and efficiency.

Lastly, the absence of clear hardware standards and requirements poses a significant barrier to scaling neuromorphic solutions. Specifications for key components, including memristors, processors, and memory configurations, are yet to be established. Addressing these gaps is crucial for advancing neuromorphic computing in rail surface defect detection and ensuring the development of scalable, high-performance systems [45, 46].

2.5 CONCLUSION

In conclusion, data augmentation techniques and neuromorphic computing hold transformative potential for image processing and defect detection. Data augmentation techniques, through methods like flipping, cropping, rotation, and noise injection, enhance the generalisation of deep learning models – particularly CNNs – by preventing overfitting and improving robustness against new data. Advanced methods, such as GANs and neural style transfer, further diversify training datasets, though challenges remain in tailoring these techniques to specific applications and determining optimal dataset sizes.

Simultaneously, neuromorphic computing, inspired by the brain's architecture, addresses limitations of von Neumann systems by integrating memory and processing, offering innovative solutions for energy-efficient, real-time processing. Neuromorphic systems leveraging SNNs and memristor CBAs excel in defect detection by mimicking neural architectures, enabling precise, low-power computations that are effective in real-time applications. Prior research in materials such as fabric and brain tumours underscore the promise of these systems in detecting defects.

This chapter provides the foundation for the next phase of investigation: How do SNNs and CNNs compare in terms of performance, energy efficiency, and suitability for neuromorphic hardware platforms? Exploring this question is essential for understanding the impact of both data augmentation techniques and neuromorphic computing on defect detection, especially within rail applications. By addressing gaps in benchmarks specific to rail surfaces and examining neural network architectures, this research aims to push the boundaries of defect detection systems. Integrating data augmentation techniques with neuromorphic computing holds potential for developing robust, scalable solutions that advance the capabilities of current methodologies.

CHAPTER 3: PROPOSED SYSTEM CONFIGURATION AND PLATFORM READINESS

3.1 CHAPTER INTRODUCTION

The deployment of rail surface defect detection models in real-world environments requires careful consideration of the underlying hardware platforms. Each platform—CPU, GPU, and the Akida neuromorphic processor—offers distinct advantages and trade-offs in terms of inference speed, power consumption, and real-time responsiveness. This section provides the rationale for selecting these platforms and examines how their configuration aligns with the practical constraints of rail infrastructure, such as energy availability, thermal management, and continuous operation. Understanding these hardware-specific characteristics is essential for optimizing model performance while ensuring scalability, reliability, and feasibility in resource-constrained deployment scenarios.

3.2 HARDWARE PLATFORM MOTIVATION AND DEPLOYMENT CONSTRAINTS

This section presents a comparative overview of the hardware platforms used for rail defect detection—CPU, GPU, and the Akida neuromorphic processor—and outlines the motivation behind their selection. It also addresses configuration practices required to meet deployment constraints such as power, cooling, and real-time response.

3.2.1 Neuromorphic Processor (*Akida*)

The Akida neuromorphic processor offers a highly energy-efficient, low-latency solution well-suited for real-time defect detection in rail systems. Its architecture supports event-driven computation, making it ideal for edge-based deployments where continuous monitoring and minimal power consumption are essential. Configuring Akida for optimal inference involves selectively enabling neuromorphic cores and mapping models for efficient spike-based processing. This allows for sustained operation in power-constrained environments without compromising precision, which consistently exceeds 99% across various defect classes.

3.2.2 Central Processing Unit (*CPU*)

CPUs provide a general-purpose, widely available platform for implementing defect detection models. While not specialized for parallel processing like GPUs or neuromorphic cores, CPUs offer sufficient performance for applications with moderate throughput requirements. Real-time performance on CPUs can be enhanced by optimizing batch sizes, managing thread priorities, and restricting core usage to balance energy efficiency and latency. CPUs serve as a viable fallback platform, especially in contexts where hardware constraints prevent the use of more specialized accelerators.

3.2.3 Graphics Processing Unit (GPU)

GPUs are optimized for high-throughput parallel computation, offering the fastest inference times for CNN models in this study. Their suitability for real-time rail defect detection is reinforced by configurable parameters such as batch size, memory allocation, and model quantization. However, due to their intensive computational load, GPUs require robust cooling systems and power supplies to ensure thermal stability and prevent performance throttling during prolonged operation. Despite these demands, GPUs remain the platform of choice where high-speed, high-volume processing is essential.

3.2.4 Power Supply and Cooling Considerations

Hardware deployment in rail environments must consider power and thermal constraints. The Akida processor demonstrates minimal energy usage and generates negligible heat, enabling fanless passive cooling solutions suitable for embedded applications. In contrast, GPUs require high-wattage power supplies and active cooling solutions due to sustained power draw during inference. CPUs lie in the middle of this spectrum but benefit from dynamic voltage scaling and thermal management features. Proper cooling and power configuration are essential to maintain consistent performance and hardware longevity across all platforms.

3.3 SENSOR INPUT AND REAL-TIME CONSTRAINTS

Sensor choice and data acquisition strategy directly impact the effectiveness of defect detection systems. The ability of a model to detect subtle or early-stage defects depends on the quality and consistency of its input.

3.3.1 Sensor Resolution and Data Acquisition

The accuracy of defect detection models is heavily influenced by the resolution and quality of captured rail images. High-resolution sensors are preferred, as they enable the detection of fine-grained surface defects such as microcracks and corrosion patterns. However, higher resolutions increase data volume, necessitating efficient data acquisition and transfer mechanisms to preserve real-time processing capabilities. Cameras must be selected based on their ability to balance frame rate, resolution, and low-light performance to support continuous inspection under varying field conditions.

3.3.2 Preprocessing and Feature Extraction

To enhance real-time performance and input consistency, this study employs a preprocessing pipeline that generates structured hand-crafted features from raw sensor data. The extracted features include:

- **GLCM (Gray Level Co-occurrence Matrix):** Captures second-order texture patterns relevant to material degradation.

- LBP (Local Binary Patterns): Encodes local texture structures useful for detecting roughness or wear.
- Sobel Operator: Detects horizontal and vertical gradients, which help highlight surface discontinuities and cracks.
- Canny Edge Detector: Provides precise edge maps, allowing for accurate defect boundary localization.

These preprocessing steps reduce input variability and aid the models in focusing on defect-relevant patterns, thereby improving detection accuracy and inference consistency under real-time constraints.

3.4 SOFTWARE STACK AND TOOLCHAIN ALIGNMENT

This section outlines the end-to-end toolchain used to support model development, platform-specific optimisation, inference deployment, and performance monitoring for rail surface defect detection across CPU, GPU, and Akida neuromorphic processors. The alignment of software tools with hardware platforms is critical to ensure that each component in the pipeline is optimized for real-time operation, energy efficiency, and inference accuracy. Figure 3-1 illustrate the toolchain recommendations.

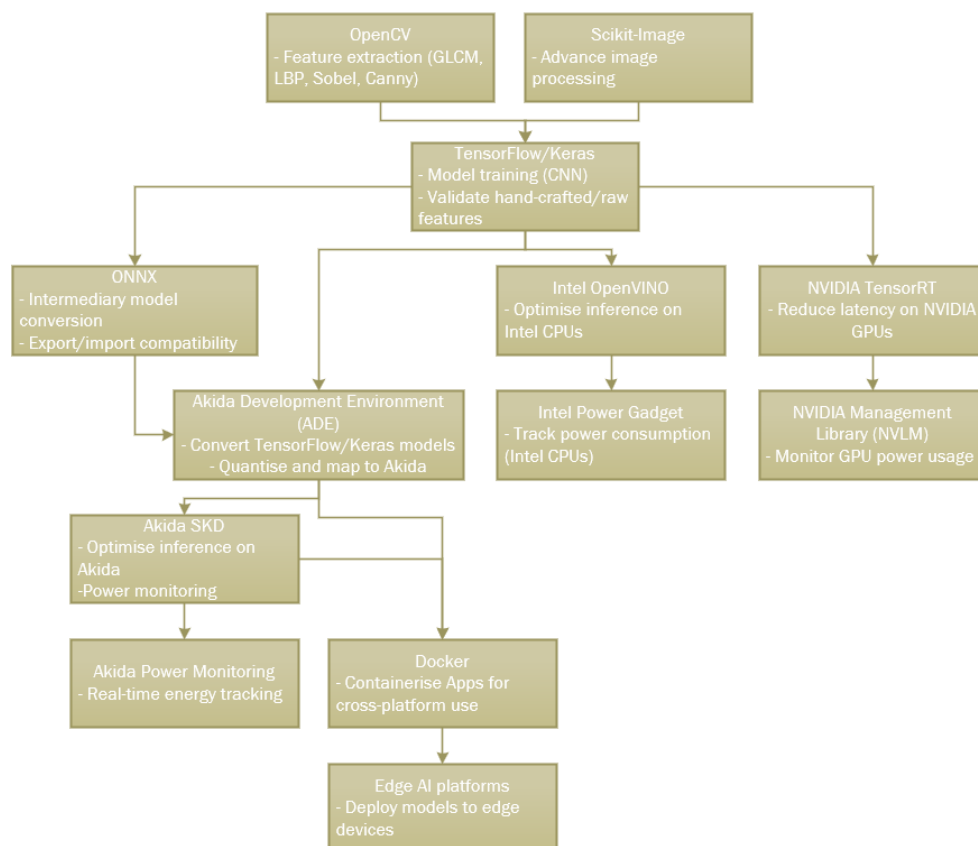


Figure 3-1: Visual representation of the toolchain recommendations

3.4.1 Model development and conversion

- TensorFlow/Keras is employed for designing and training CNN architectures on CPU and GPU platforms. These frameworks support both raw image and hand-crafted feature inputs.
- Akida Development Environment (ADE)¹ is used to convert trained TensorFlow/Keras models into Akida-compatible formats. The ADE supports quantisation, core mapping, and neuromorphic optimisation to enable low-latency and energy-efficient inference on the Akida chip.
- Open Neural Network Exchange (ONNX): serves as a framework-agnostic intermediary, enabling models trained in TensorFlow or PyTorch to be transferred into other environments, including Akida and GPU-compatible runtimes.

3.4.2 Inference and Optimization Tools

Platform-specific inference optimizers are used to reduce latency and improve energy efficiency:

- Intel OpenVINO is used to accelerate inference on Intel CPUs by optimizing model execution paths and enabling hardware-accelerated instructions.
- NVIDIA TensorRT is utilized to reduce inference latency and memory usage on GPUs. It supports precision reduction (e.g., FP16) and kernel-level optimizations.
- Akida SDK enables deployment and monitoring of inference performance on the Akida neuromorphic processor. It supports real-time energy tracking and tuning of neuromorphic core configurations.

3.4.3 Power and Performance Monitoring

Energy profiling is conducted using platform-specific tools:

- Intel Power Gadget monitors CPU power draw and thermal metrics, helping assess energy-efficiency trade-offs.

¹ In practice, the ADE conversion process translates a trained CNN into an event-driven spiking representation that can be executed on Akida's neuromorphic cores. The convolutional kernels are preserved as weight matrices but are activated using spike-based signalling rather than continuous values. Non-linear activations such as ReLU are replaced by threshold-driven spiking functions, while pooling operations are implemented through event-driven down-sampling. These steps are abstracted from the user, and the precise internal mapping of CNN layers to spiking primitives is not fully documented in public sources. However, the practical outcome is that a conventionally trained CNN retains its overall structure while being deployed on Akida as an event-driven SNN, enabling significant energy savings and ultra-low-latency inference. For transparency, Appendix A.5 summarises how each CNN and SNN model in this study was structured and deployed across CPU, GPU, and Akida platforms, clarifying the role of the ADE conversion process in the pipeline.

- NVIDIA Management Library (NVML) enables real-time monitoring of GPU power, memory usage, and temperature.
- Akida Power Monitoring, integrated within the Akida SDK, provides fine-grained insight into energy usage during neuromorphic inference, which is critical for evaluating embedded system feasibility.

3.4.4 Data Preprocessing and Feature Engineering

- Feature extraction and augmentation pipelines are developed using a combination of OpenCV and scikit-image to ensure reproducible, platform-consistent inputs for both CNN and SNN models.
- OpenCV is used for image preprocessing tasks such as resizing, normalization, geometric transformations (e.g., flips, rotations), and edge detection using Sobel and Canny operators. These operations standardize the input dimensions and highlight structural characteristics relevant to rail defect detection.
- Scikit-Image provides the implementation of GLCM (Gray Level Co-occurrence Matrix) and LBP (Local Binary Patterns), which are essential for capturing second-order texture statistics and local texture features, respectively. These descriptors are not available in OpenCV and require the specialized functions in scikit-image for accurate and reproducible computation.
- Together, OpenCV and scikit-image form a robust preprocessing pipeline that generates standardized hand-crafted feature representations. These ensure consistency across CPU, GPU, and Akida environments and enhance the models' ability to detect defect-relevant patterns in real time.

3.4.5 Deployment and Integration

- Docker is used to containerize the deployment pipeline, ensuring replicable environments and platform-specific compatibility.
- Edge AI platforms such as Edge Impulse and AWS IoT Greengrass are identified for potential integration of real-time inference systems into rail inspection infrastructure. These platforms facilitate scalable deployment and cloud-edge coordination.

3.4.6 Interpretation of platform-specific optimisations

The observed differences in model performance across platforms are largely attributed to the optimization tools applied during model conversion and inference deployment. For example:

- OpenVINO enhances inference by fusing layers and optimizing execution for Intel CPUs.
- TensorRT applies quantization and kernel fusion to speed up GPU inference.

- ADE maps models to Akida’s event-driven architecture, significantly reducing energy consumption through sparse computation.

These platform-level optimizations affect how the model operates, introducing variability in precision handling, memory management, and computational latency. Thus, while the model architecture remains consistent, its practical behaviour is tailored to match hardware capabilities.

3.4.7 Summary

The toolchain described above provides a coherent and optimized software stack for implementing defect detection models across CPU, GPU, and Akida platforms. Each tool is selected to maximize compatibility, minimize latency, and optimize energy consumption. Together, they support the deployment of scalable, low-power, and real-time inspection systems tailored for different operational contexts in railway environments.

3.5 RESOURCE-CONSTRAINED CONFIGURATION RATIONALE

This section outlines the rationale for configuring computational resources to simulate deployment in constrained environments, such as onboard rail inspection systems or edge-based monitoring units. While initial experiments are conducted using full hardware capacity to benchmark peak performance, it is equally important to assess whether models maintain acceptable defect detection accuracy when operating under limited computational and power resources.

To simulate resource-constrained deployments, each hardware platform—CPU, GPU, and Akida—is reconfigured with reduced capabilities while preserving functionality. These reduced configurations emulate the limitations typically encountered in real-world rail infrastructure, including power budgets, thermal restrictions, and hardware miniaturization constraints.

3.5.1 CPU Configuration

On the CPU, resource constraints are implemented using the taskset utility within a Linux environment. This tool allows explicit binding of the inference process to a subset (typically 50%) of the available physical cores. By limiting thread scheduling, the CPU is forced to operate under reduced parallelism, thereby mimicking scenarios where shared computational workloads or energy-saving measures are required. This configuration helps evaluate whether the CNN and SNN models retain their classification performance and inference responsiveness under limited multithreading conditions.

3.5.2 GPU Configuration

Unlike CPUs, GPUs do not allow explicit restriction of core-level operations. Instead, GPU constraints are approximated through indirect means such as limiting power draw, clock speeds, and batch size. NVIDIA’s nvidia-smi tool is used to cap power usage and reduce thermal load. These settings reduce the GPU’s throughput capacity, providing a representative profile of how the CNN

model performs under real-time, low-resource conditions. While exact 50% core usage cannot be enforced, these modifications sufficiently emulate constrained GPU performance in embedded or mobile environments.

3.5.3 Akida Minimal Configuration

For the Akida neuromorphic processor, resource constraint is achieved by modifying how the model is mapped to Akida's neuromorphic fabric. Using the Akida Development Environment (ADE), fewer neuromorphic processing clusters (NPU's) are activated, and redundant spike processing tiles are disabled. This minimal configuration maintains model functionality while lowering energy consumption and throughput, making it suitable for ultra-low-power applications, such as solar-powered trackside inspection units or battery-operated edge devices. The Akida minimal configuration is compared to its full-capacity configuration to evaluate the trade-off between inference speed, accuracy, and energy savings.

3.5.4 Motivation for Constrained Testing

The rationale for conducting constrained-configuration tests lies in identifying the minimum viable platform setup capable of maintaining acceptable detection performance. In railway environments where processing units may be embedded in remote or mobile systems, constraints such as power availability, thermal dissipation, and hardware footprint must be considered. By benchmarking performance under restricted configurations, this study provides insight into the robustness and scalability of each model-platform combination.

The resource-constrained configurations implemented in this study represent a single, reduced-capacity setup per hardware platform. These configurations are carefully selected to simulate realistic deployment conditions where power, processing capacity, or thermal budgets may be limited, such as in edge or embedded railway monitoring systems. For example, wayside monitoring systems installed along open track in such as the South African coal corridor often experience temperatures exceeding 45°C in summer, while equipment housed in relay rooms must remain operational despite inadequate ventilation or dust ingress. Thermal dissipation becomes a critical limiting factor in these contexts, as overheating can degrade inference accuracy, throttle processor speeds, or even cause system shutdowns.

Similarly, edge devices deployed in remote block sections of the railway often rely on solar panels and battery banks due to the absence of reliable grid connections. These sites impose strict limits on available wattage, with continuous operation needing to be sustained under conditions where daily solar yield and battery charge cycles fluctuate. In such settings, processors must function within sub – 10 W envelopes to avoid exhausting the limited energy budget. A further example is the deployment of defect detection units on on-board locomotives, where compact enclosures must withstand vibration and ambient heat while competing with other subsystems for the same power

supply. Here, both the thermal design power of the processor and the availability of conditioned airflow directly influence the feasibility of real-time inference at line speed.

While only one constrained setup is evaluated per platform, the selected configurations are representative of practical limitations encountered in real-world applications. Future studies may explore a broader range of constrained scenarios to provide a more detailed analysis of performance scalability and resilience under varying resource conditions.

3.6 CONCLUSION

This chapter outlines the practical considerations and technical strategies for deploying CNN and SNN models for rail defect detection across three key platforms: CPU, GPU, and the Akida neuromorphic processor. Section 3.2 highlights the hardware motivation and constraints, revealing that while CPUs offer broad compatibility, GPUs provide the best throughput, and Akida delivers unmatched energy efficiency for real-time, embedded applications. The trade-offs between power consumption, processing speed, and thermal demands are discussed in relation to deployment feasibility in railway environments.

Section 3.3 focuses on sensor input requirements and real-time constraints. High-resolution imaging and robust data acquisition pipelines are essential for detecting subtle rail defects, while preprocessing steps—such as GLCM, LBP, Sobel, and Canny—standardize input across platforms and improve model accuracy under real-time conditions.

Section 3.4 details the software toolchain aligned to each platform, including frameworks for model development (TensorFlow/Keras, ADE), inference optimization (OpenVINO, TensorRT), and energy monitoring (Intel Power Gadget, NVML, Akida SDK). The section emphasizes that platform-specific optimizations influence not only performance but also inference behaviour and energy profiles, even for the same base architecture.

Finally, Section 3.5 justifies the need for resource-constrained testing. Reduced-capacity configurations on CPU (via `taskset`), GPU (via `nvidia-smi`), and Akida (via core deactivation in ADE) are implemented to simulate deployment in edge or embedded settings. Although only one reduced setup is tested per platform due to time limitations, these configurations are representative of real-world constraints and demonstrate each platform’s resilience and scalability under resource limitations.

Together, the hardware analysis, sensor strategy, toolchain alignment, and constrained configuration rationale establish a comprehensive foundation for evaluating performance, energy efficiency, and real-time feasibility in Chapter 4 and Chapter 5.

CHAPTER 4: NEURAL NETWORK ARCHITECTURES

4.1 CHAPTER INTRODUCTION

This chapter addresses the research question: How do SNNs and CNNs compare in terms of performance, energy efficiency, and suitability for neuromorphic hardware platforms? The choice to evaluate SNNs and CNNs is grounded in findings from the preceding chapters, where we established their prominence in visual pattern recognition and their relevance to neuromorphic computing. CNNs are widely recognised for their high accuracy in image classification tasks and serve as a robust benchmark in defect detection systems. However, their computational demands make them less ideal for low-power or real-time edge deployment. In contrast, SNNs – bio inspired models designed for event-driven, energy-efficient computation – align closely with the capabilities of neuromorphic hardware such as the Akida and Loihi processors.

Building on this theoretical foundation, the chapter begins by introducing the fundamental structures, operational principles, and training paradigms of both CNNs and SNNs. We then outline the experimental methodology, including the evaluation (e.g. precision, F1-score, inference latency, and energy consumption) used to compare their suitability for neuromorphic applications. Finally, we present and analyse the results to determine which architecture offers the most practical balance between accuracy and efficiency for real-time defect detection. This structured approach enables a critical, evidence-based evaluation of the trade-offs between traditional and neuromorphic-inspired neural models.

4.2 INTRODUCTION TO NEURAL NETWORK ARCHITECTURES

4.2.1 *Fundamentals of neural networks*

Neural networks are computational models inspired by the structure of the human brain, designed to recognise patterns and make predictions from data. A foundational example is the Multi-Layer Perceptron (MLP), composed of fully connected layers that map input features to outputs through a series of weighted connections and non-linear activation functions. MLPs are trained via backpropagation, which iteratively adjusts the weights to minimise prediction error. While effective in many tasks, MLPs lack the capacity to efficiently handle high-dimensional structured data – such as images – due to their dense connectivity and absence of spatial awareness [48].

To address this limitation, CNNs were developed. CNNs are inspired by the visual cortex of animals and excel in image-related tasks due to their ability to automatically learn spatial hierarchies of features. They achieve this through a layered architecture comprising convolutional layers, pooling layers, non-linear activations (e.g., ReLU), and fully connected output layers [49].

In a convolutional layer, filters (or kernels) slide across the input image to compute feature maps. The basic operation is defined as:

$$Y(i, j) = \sum_{m=1}^M \sum_{n=1}^N X(i + m, j + n) \cdot K(m, n) \quad 4-1$$

where X is the input image, K is the kernel, and Y is the resulting feature map. These learned filters enable the network to detect features such as edges, textures, and object parts [48].

Pooling layers – commonly max-pooling – reduce the spatial dimensions of features maps, lowering computational cost and improving generalisation. The ReLU activation introduces non-linearity, while the fully connected layers interpret the extracted features for classification. CNNs are trained using gradient descent and backpropagation, with the filters and weights updated iteratively to minimise a loss function such as cross-entropy [48, 50].

A visual overview of the CNN architecture – from the input image through feature extraction and pooling to final classification – is illustrated in Figure 4-1.

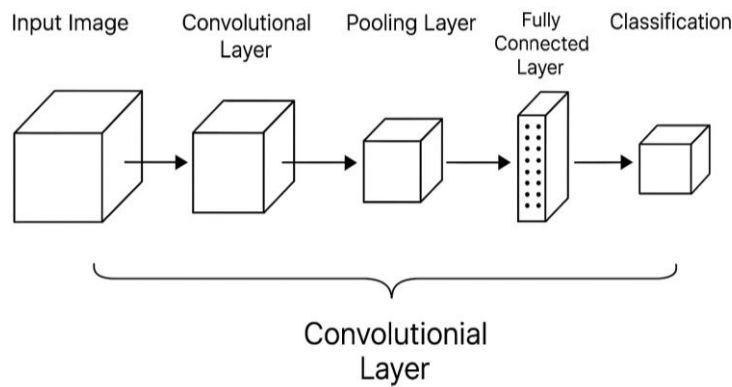


Figure 4-1: Basic structure of a CNN showing input image, convolution and pooling layers, fully connected layers, and final classification [50]

CNNs demonstrate state-of-the-art performance across a wide range of computer vision tasks, including image classification, object detection, and facial recognition. Due to their strong generalisation capabilities and automatic feature extraction, CNNs have become standard models in deep learning workflows adopted by both academia and industry [48, 50].

However, CNNs are computationally intensive – particularly during the training phase on large datasets or complex architectures – which traditionally requires high-performance GPU-based systems. It is important, however, to distinguish between training and deployment. CNNs are typically trained offline on powerful computing platforms, after which the trained models are deployed to edge devices or low-resource environments for real-time inference. This research does not aim to train CNNs on neuromorphic or low-resource hardware but rather to evaluate their inference performance and energy efficiency on such platforms – especially in comparison to SNNs, which are inherently aligned with neuromorphic architectures [49].

By evaluating both CNNs and SNNs in the context of rail surface defect detection, this study aims to identify which architecture provides the best trade-off between accuracy, latency, and power consumption for real-time deployment in resource-constrained environments [49].

4.2.2 Overview of spiking neural network

SNNs (Figure 4-2) are a class of artificial neural networks (ANNs) modelled after the spike-based communication mechanisms of biological neurons. Unlike traditional ANNs that transmit continuous values across layers, SNNs use discrete spikes, or action potentials, to encode and transmit information. This biologically inspired approach enables SNNs to achieve higher energy efficiency and temporal processing capabilities, closely mimicking the sparse and efficient firing patterns observed in the human brain, where neurons fire only when specific activation thresholds are reached. In SNNs, rate encoding is a common method used to convert continuous input data into spike trains, where the spike rate reflects the input intensity: higher input values produce more frequent spikes, while lower values produce fewer spikes. Rate encoding prepares data for temporal learning and event-driven processing with SNNs [51].

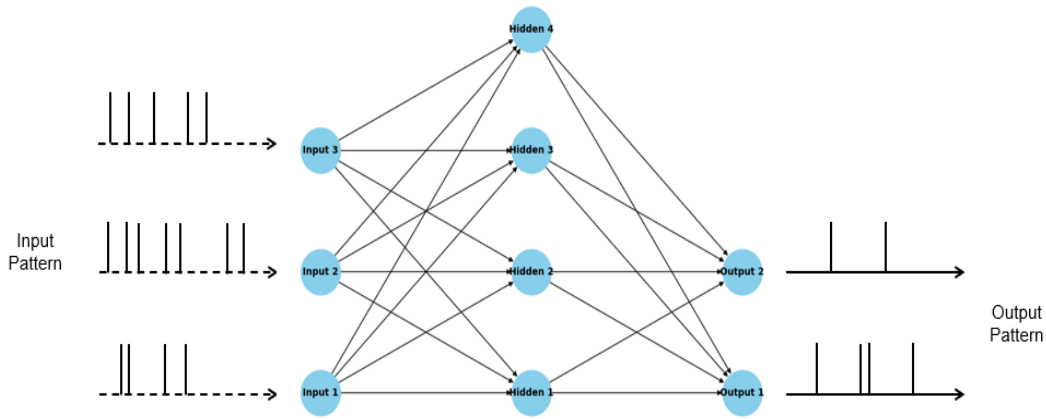


Figure 4-2: Basic structure of SNN [52]

SNNs consists of spiking neurons and synapses, where synaptic weights adjust dynamically according to learning rules. A common rule, STDP (Figure 4-3), adjusts synaptic strength based on the timing between spikes in pre-synaptic and post-synaptic neurons. Mathematically, STDP (Equation 4-2) is often represented as a function where synaptic changes are proportional to the relative timing of spikes: if a pre-synaptic neuron spikes slightly before the post-synaptic neuron, synaptic strength increases (potentiation); conversely, if it spikes after, the strength decreases (depression). This relationship is typically expressed as [51]:

$$\Delta w = \begin{cases} A^+ \cdot e^{-\frac{\Delta t}{\tau^+}} & \text{if } \Delta t > 0 \\ -A^- \cdot e^{\frac{\Delta t}{\tau^-}} & \text{if } \Delta t < 0 \end{cases} \quad 4-2$$

Where Δt is the time difference between pre- and post-synaptic spikes, A^+ and A^- are learning rates, and τ^+ and τ^- are time constants for potentiation and depression, respectively. This time-dependent

adjustment allows SNNs to learn complex temporal patterns enhancing their performance in time-sensitive tasks [51].

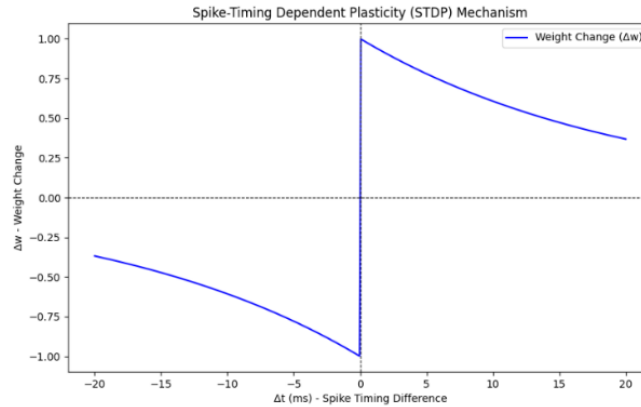


Figure 4-3: STDP learning rule for SNN [53]

The architecture of SNNs requires specialised neuron models, such as leaky integrate-and-fire (LIF) and Hodgkin-Huxley (HH) models, which incorporate temporal dynamics of membrane potential and synaptic currents. The LIF model (Figure 4-4), for example, accumulates incoming spikes and “leaks” over time; once the membrane potential crosses a threshold, the neuron fires a spike and resets. This dynamic behaviour can be presented by the differential equation [54]:

$$\tau_m \frac{dV(t)}{dt} = -V(t) + R \cdot I(t) \quad 4-3$$

Where $V(t)$ is the membrane potential at time t , τ_m is the membrane time constant, R is the membrane resistance, and $I(t)$ is the input current. This temporal structure enables the network to process information encoded in spike timing, a powerful feature for applications in event-based sensory processing [54].

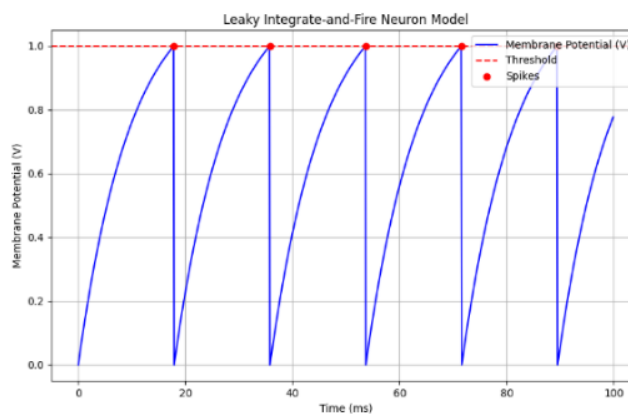


Figure 4-4: LIF neuron [55]

Training SNNs present challenges due to the non-differentiable nature of spike-based communication, which complicates traditional gradient-based methods like backpropagation. Recent advances address these issues through algorithms such as SpikeProp, which approximates

gradients by considering spike timing, and surrogate gradient methods, which use continuous approximations of spike functions to compute gradients. These methods allow SNNs to be trained on supervised and unsupervised tasks, expanding their applicability to domains like visual and auditory processing [51, 54].

4.3 EXPERIMENTAL SETUP

This chapter evaluates the performance of CNNs and SNNs for binary classification of rail surface images into defective or defect-free classes. These models are implemented and tested on three hardware platforms with varying computational capabilities: a CPU, a GPU, and the Akida neuromorphic processor. The flow charts illustrating the training and deployment processes for the different network models on each hardware platform are provided in Appendix C for reference.

4.3.1 State-of-the-art models in rail defect detection

Various machine learning and deep learning approaches have been applied to rail surface defect detection. Traditional models such as Support Vector Machines (SVMs), Random Forest, k-Nearest Neighbour (k-NN), and Naïve Bayes are known for their effectiveness in classification tasks. These typically rely on handcrafted features such as GLCM, LBP, and edge detectors. While useful, their success depends heavily on the quality of feature extraction and dataset characteristics.

In contrast, modern deep learning models – especially CNNs – learn hierarchical spatial features directly from image data. For instance, Zheng et al. (2021) proposed a hybrid method combining YOLOv5 and Mask R-CNN to achieve high detection accuracy for rail surface and fastener defects [56]. Similarly, Wang et al. (2022) reported a mean average precision of 94.9% using a Fast R-CNN-based model for rail surface evaluation [57]. These examples illustrate the effectiveness of CNN-based approaches.

Meanwhile, SNNs provide a biologically inspired, event-driven alternative optimised for energy-efficient computation. Although they may achieve lower accuracy than CNNs, SNNs are well-suited for low-latency, real-time processing on neuromorphic hardware, making them ideal for continuous monitoring in embedded systems.

4.3.2 Dataset structure and label integrity

The dataset contains several types of rail defects, including shelling, squats, corrosion, wheel burns, and cracked rails. These defects vary in severity and appearance, introducing visual diversity within the “defect” class. All images are manually labelled by the Transnet inspection team based on visual inspection expertise and standardised assessment criteria. While major defects are clearly visible, early-stage or subtle cases pose classification challenges.

To ensure label accuracy, consensus labelling and manual quality control checks are applied. Although this study uses a binary classification, the presence of multiple defect types introduces intra-class variability. A multi-class framework could offer more granular identification in future work.

Techniques such as contrast enhancement and edge detection further assist in identifying subtle anomalies.

Captured during real-world rail inspections using trackside equipment, the original 1514 images reflect diverse environmental conditions. Each image is labelled as either “defect” (Figure 4-5 (a)) or “no defect” (Figure 4-5 (b)), ensuring label relevance and accuracy.

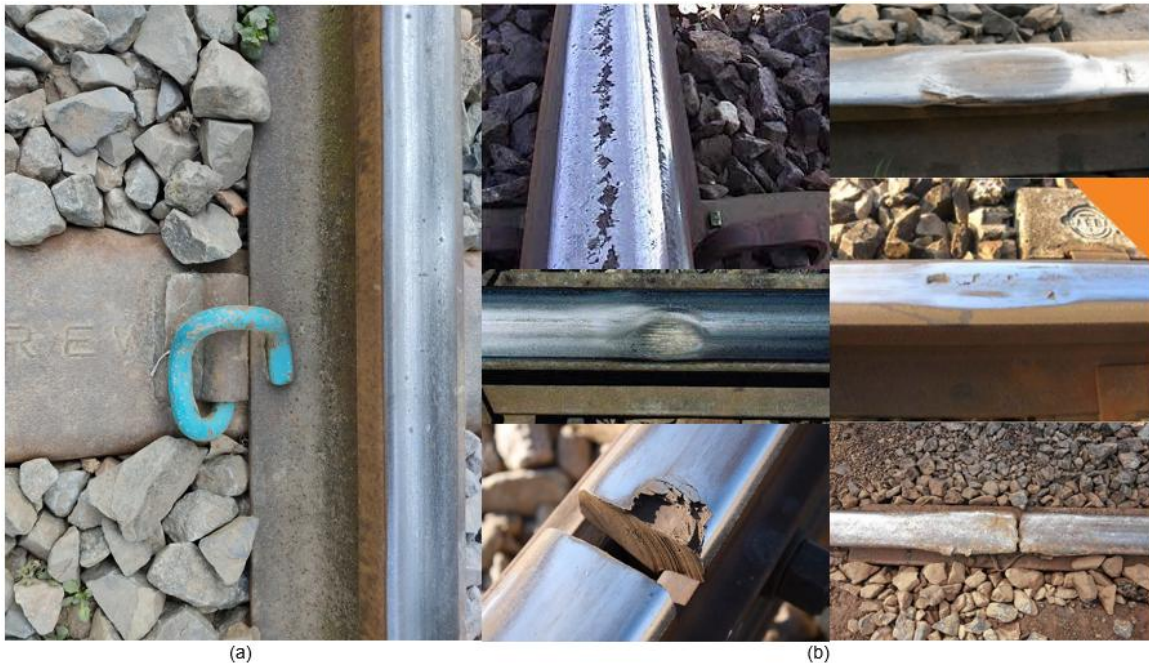


Figure 4-5: Rail track with (a) no defect and (b) defect

To improve model generalisation and reduce overfitting, a comprehensive augmentation pipeline is implemented using TensorFlow's *ImageDataGenerator* and *tf.data* APIs with the following parameters:

- Geometric transformations: rotations ($\pm 10^\circ$), width/height shifts ($\pm 20\%$), zoom ($\pm 20\%$), shear, and horizontal flipping
- Lighting variations: brightness adjusted randomly between 0.5 and 1.5
- Colour distortions: HSV and LAB colour space conversions with channel-specific manipulations
- Noise injection: Gaussian noise ($\sigma = 25$) simulates sensor noise

Parameters are based on operational imaging scenarios and validated through small-scale testing to preserve defect visibility. Augmentation improves generalisation, as confirmed by consistent gains in F1-score and recall across folds.

The dataset expands from 1514 to 248,374 images through multiple transformation combinations. Images are resized to 300 x 300 pixels (for CNN models) and 128 x 128 pixels (SNN model) in JPG

format, with labels stored in a structured CSV file linking each image to its binary class (Figure 4-6 and Figure 4-7).



Figure 4-6: Illustration of original vs resized image (300 x 300) in dataset

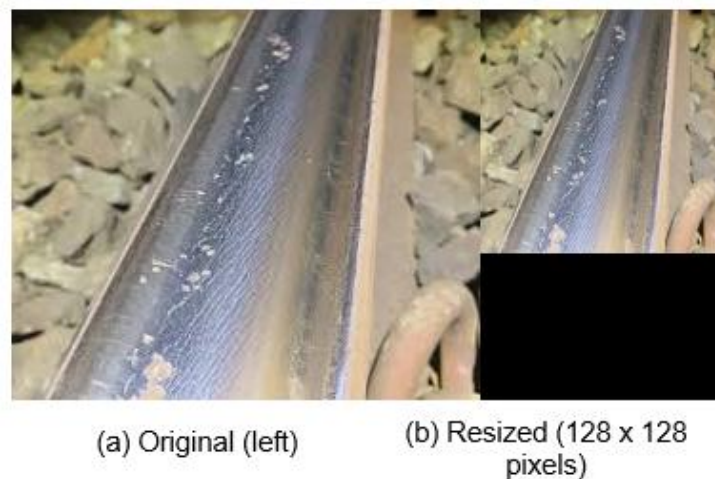


Figure 4-7: Illustration of original vs resized image (128 x 128) in dataset

A balanced 50,000-image subset (26000 defect, 24000 no-defect) is used for model training and evaluation, split into 70% training, 15% validation, and 15% testing, with consistent class balance across all platforms.

Although no models are trained on the original dataset alone, augmentation is essential due to the limited sample size and practical constraints of collecting large-scale annotated data. It remains a widely recognised practice in computer vision to promote robustness and generalisability.

The final deployment scenario involves a trolley-mounted inspection system that captures images of the rail surface while in motion. However, the current dataset consists of images captured using handheld equipment during static inspections. This introduces differences in image perspective, motion artefacts, and lighting conditions compared to the moving trolley setup.

Since the field prototype is still under development and not yet operational, fine-tuning is anticipated to bridge this domain gap. To prepare the model for real-world deployment, domain adaptation techniques are considered. These include augmentation strategies to simulate motion blur,

perspective distortions, and dynamic lighting variations during training. Transfer learning is also applied, where pretrained weights are fine-tuned on this augmented dataset to improve generalisation.

Once the trolley-based data acquisition system becomes operational, additional images can be collected under realistic motion conditions. These images will support further fine-tuning and validation, ensuring the model performs reliably under deployment-specific constraints. While the current dataset offers a strong foundation for model deployment, final optimisation steps are necessary to ensure robust performance in the target operational environment.

4.3.3 Hardware platforms and motivation

Three platforms are selected to reflect real-world deployment conditions:

- GPU (NVIDIA RTX 3060): Serves as a high-performance baseline. Its parallel processing capabilities are ideal for training and inference, particularly with CNNs.
- CPU (AMD Ryzen 9): Represents general-purpose or edge deployments where GPU access is limited. Evaluating performance on CPU helps assess model feasibility under typical embedded or legacy conditions.
- Akida neuromorphic processor: Designed for ultra-low power, event-driven inference. Akida supports native execution of SNNs and offers real-time classification with minimal energy usage. Its architecture supports sparse data processing and aligns with the study's focus on evaluating latency, energy efficiency, and deployment feasibility.

By comparing performance across platforms, the study highlights the trade-offs between accuracy, efficiency, and deployment constraints in practical settings.

4.3.4 Neural network models

This experiment investigates: How do SNNs and CNNs compare in terms of performance, energy efficiency, and suitability for neuromorphic hardware platforms? The choice of CNNs and SNNs is driven by their complementary strengths and relevance to practical deployment scenarios.

CNNs are widely adopted in image classification and visual anomaly detection. In this study, the CNN model is based on EfficientNetB3, enhanced with global average pooling, dense layers, dropout, and a SoftMax output. It is trained using stochastic gradient descent (SGD) with learning rate scheduling and real-time augmentation.

The EfficientNet-B3 model for defect detection follows a structured training pipeline. Since the dataset consists of balanced, coloured images, it is split into 70% training, 15% validation, and 15% testing subsets. All images are resized to 300x300 pixels, normalised, and augmented in real-time using random rotations, flips, brightness adjustments, and Gaussian noise to improve generalisation. The model is initialised with pretrained ImageNet weights using the *tf.keras.applications* module.

The top layer is replaced with a binary classifier using a Sigmoid activation function. Initially, the base layers are frozen, and training focuses on the new classification head for approximately five epochs using the Adam optimiser, binary cross-entropy loss, and learning rate scheduling. In the second phase, the full model is fine-tuned for 20-40 epochs at a lower learning rate.

Training is implemented using the TensorFlow/Keras framework, which supports efficient transfer learning and GPU acceleration. The model is defined using Keras' high-level API, simplifying layer composition, batch normalisation, dropout, and regularisation. Real-time augmentation is managed with the *tf.data* API, and training is monitored using TensorBoard. Image preprocessing is assisted by OpenCV and NumPy. For deployment, the trained model is converted to TensorFlow or ONNX and optimised for CPU inference through quantisation (INT8/FP16) and pruning. These optimisations ensure fast, low-latency deployment. TensorFlow's ecosystem and tooling make it well suited for high-performance and embedded defect detection applications.

SNNs, conversely, represent a neuromorphic approach modelled after spiking biological neurons. They use sparse, event-driven computation and consume significantly less power—making them ideal for energy-constrained, real-time applications. The SNN model (detailed in Appendix A) is a simplified variant of the CNN, processing inputs as spike trains.

Both models are evaluated on CPU and GPU platforms. The CNN model is also deployed on the Akida neuromorphic processor; however, the SNN is explicitly excluded from the Akida platform due to compatibility limitations². Evaluation includes classification accuracy, inference time, and energy usage, allowing a comprehensive comparison across models and platforms.

This setup enables the exploration of trade-offs between accuracy, efficiency, and latency, contributing to the broader aim of identifying scalable and practical solutions for rail surface defect detection in embedded and real-world environments.

4.4 METHODOLOGY

4.4.1 Data partitioning

The dataset is divided into training, validation, and testing sets following the procedure described in [58], with 70% allocated for training, 15% for validation, and 15% for testing. This stratification

² The exclusion of the SNN from the Akida platform stems from the current toolchain and architectural limitations of the Akida Development Environment (ADE). Akida is designed primarily for event-driven convolutional neural networks converted into spiking equivalents through its conversion framework, but it does not support direct deployment of custom SNNs implemented in research frameworks such as Brian2 or snnTorch. As a result, SNN evaluation in this study is restricted to CPU-based testing.

ensures that the models are trained on a representative portion of the dataset while being evaluated on unseen data to assess generalisation performance.

4.4.2 Model training and testing procedures

4.4.2.1 CNN

The CNN architecture is based on EfficientNetB3, selected after evaluating multiple state-of-the-art CNN architectures based on accuracy, inference speed, model size, and defect detection capability across scales. As detailed in Appendix B, EfficientNet-B3 consistently achieves the best trade-off across these criteria. It offers:

- High accuracy with fewer parameters than deeper models like ResNet-50 or InceptionV3, critical in reducing false alarms in safety applications.
- Fast inference on CPU and other embedded systems, supporting real-time inspection.
- Compact model size (~12M parameters), enabling efficient deployment with limited memory overhead.
- Strong performance on both small and large defects, thanks to its compound scaling and high input resolution.

These characteristics position EfficientNet-B3 as an optimal choice for the rail defect detection task, offering a strong balance between accuracy and efficiency across deployment scenarios. EfficientNet-B3 is chosen for its balance of accuracy, computational efficiency, model compactness, and multi-scale detection capability. While ResNet and Inception are strong performers, EfficientNet-B3 delivers significantly higher accuracy than ResNet-50 with fewer parameters and faster inference, making it more suitable for constrained environments like CPUs or neuromorphic processors. Its ability to handle both small and large defects effectively – due to higher resolution inputs and attention mechanisms – makes it particularly well-suited for the mixed-scale defect patterns found in rail inspection images.

The training begins with an initial learning rate of 0.0001, suitable for fine-tuning pretrained weights. A *ReduceLROnPlateau* scheduler monitors validation loss and reduces the learning rate by a factor (e.g., 0.1) if performance stalls for 3 consecutive epochs, helping the model converge to better minima. The Adam optimiser is used, configured with $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 1e^{-7}$. The batch size is set to 16, balancing memory limits and stability. Training runs for 30 epochs, including 5 with frozen base layers and 25 with fine-tuning. Regularisation includes dropout at 0.5 in the classifier and L2 weight decay ($1e^{-4}$) to penalise large weights and reduce overfitting. Class imbalance is addressed using class weights to penalise misclassification of minority classes, particularly “defect” cases.

To evaluate the model reliably, a 5-fold cross-validation strategy is employed using the KFold technique. This approach rotates the training and validating splits over five iterations, improving the robustness performance metrics.

The complete implementation pipeline – including preprocessing, model definition, training, and evaluation – is summarised in Algorithm A-1 (see Appendix A).

4.4.2.2 SNN

The SNN adapts the CNN architecture into a neuromorphic framework, implemented using the Brian2 simulation application programming interface (API) [59]. Input images are encoded into spike trains via rate encoding, and the network consists of LIF neurons trained using the STDP learning rule. The SNN uses the same data partitions and evaluation methodology as the CNN for consistency and comparability.

The complete implementation pipeline – including preprocessing, model definition, training, and evaluation – is summarised in Algorithm A-2 (see Appendix A).

The SNN model is implemented using Brian2, a Python-based SNN simulator, rather than through Akida's proprietary development environment³. The model uses rate coding to convert image intensities into spike trains and consists of multiple layers of LIF neurons, trained using STDP. Unlike CNNs that are trained using backpropagation and optimised with gradient-based methods, this SNN employs biologically inspired unsupervised learning.

In terms of comparability, CNNs are more accurate and mature for image-based classification due to their ability to learn hierarchical features directly from raw pixel data. However, SNNs offer advantages in energy efficiency and temporal processing, especially when deployed on neuromorphic hardware like Akida. The SNN in this study performs worse than the CNN due to being run on a conventional CPU, which lacks support for event-based computations. Additionally, spike encoding may limit the model's ability to represent subtle features in the rail defect images, contributing to lower recall and F1-score.

To improve SNN performance, several strategies could be explored: using alternative encoding schemes (e.g., temporal or Poisson encoding), deeper network architectures, or hybrid models where CNNs first extract spatial features before passing them to SNN layers for classification. Most

³ Brian2 was selected because Akida's development environment does not currently support direct implementation of custom SNNs with biologically inspired learning rules such as STDP. Instead, Akida is designed to run CNN-derived spiking networks converted through its proprietary toolchain, making Brian2 the more appropriate framework for this study's research-oriented SNN implementation.

critically, deploying the SNN on neuromorphic hardware would enable real-time, low power inference by leveraging event-driven processing rather than relying on spike simulation.

4.4.3 Deployment platforms

Both CNN and SNN models are implemented and evaluated across three computational platforms:

- CPU
- GPU
- Akida neuromorphic processor

This enables a comprehensive comparison not only in terms of classification performance but also with respect to inference time and energy efficiency under different deployment conditions.

Although both CNN and SNN models are evaluated across CPU, GPU, and Akida platforms, not all combinations are reported due to practical constraints and tool compatibility. The SNN model is implemented using the Brian2 simulator and executed solely on the CPU. Brian2 provides a Python-based simulation environment for event-driven SNNs, enabling flexible development and observation of neuron dynamics. In this study, the SNN model is trained using rate-coded spike inputs and a LIF neuron model, with synaptic plasticity governed by the STDP rule. The CPU is used for simulation as Brian2 is not optimised for GPU or neuromorphic deployment, and Akida-specific tools were not compatible with the custom SNN implementation. Thus, while CNNs are deployed and evaluated on all platforms (CPU, GPU, and Akida), the SNN model is limited to CPU execution for functional benchmarking and energy profiling using general-purpose computing.

While running an SNN on a CPU is generally not optimal for real-time or production applications, it remains valuable in early-stage research, prototyping, and performance benchmarking. CPUs offer flexible, accessible environments for analysing neuron dynamics, spike propagation, and learning behaviour, especially when using frameworks such as Brian2. These platforms enable detailed inspection of model internals, which is critical for debugging and tuning SNN architectures before deploying them on dedicated neuromorphic hardware.

In real-world scenarios, SNNs are indeed best suited for neuromorphic processors like Akida, Loihi, or SpiNNaker, which are designed to handle event-driven computations efficiently. CNNs, in contrast, leverage the parallelism of GPUs for high-throughput image classification tasks. However, CPUs may still serve as a practical fallback in edge deployments with strict cost or availability constraints where neuromorphic hardware is not accessible.

Therefore, while CNNs are typically deployed on GPUs and SNNs on neuromorphic processors, CPU-based SNN execution is instrumental in model development and comparative analysis. It allows researchers to simulate, visualise, and refine spike-based networks under controlled conditions before transitioning to hardware-accelerated implementations.

Future work will involve converting the SNN for direct deployment on Akida or similar neuromorphic platforms to enable fairer comparisons.

4.4.4 Evaluation metrics

Model performance is assessed using metrics derived from the confusion matrix (Table 4-1), along with computation-specific indicators:

Table 4-1: Two class confusion matrix

	Actual		
Prediction		Class 1 (Defects)	Class 2 (No defects)
	Class 1 (Defects)	TP	FP
	Class 2 (No defects)	FN	TN

The following metrics are computed:

- Accuracy: Measures overall correctness of the classifications and is calculated as [60]:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad 4-3$$

- Precision: Indicates the accuracy of positive predictions, or the proportion of true positive predictions among all positive predictions, calculated as [60]:

$$Precision = \frac{TP}{TP + FP} \quad 4-4$$

- Recall: Measures the model's ability to correctly identify actual defects, calculated as [60]:

$$Recall = \frac{TP}{TP + FN} \quad 4-5$$

- F1-Score: Provides a balance between precision and recall, useful when the classes are imbalanced, calculated as [60]:

$$F1 - score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \quad 4-6$$

- Receiver Operating Characteristic (ROC) curve and Area Under the Curve (AUC): Evaluate the model's ability to distinguish between defect and no-defect classes, with the AUC summarising the trade-off in a single scalar value [60].
- Inference time: Measures the average prediction time per sample and serves as an indicator of real-time feasibility.
- Energy consumption: Quantifies the computational efficiency of each hardware platform by measuring the energy consumed during inference.

4.5 RESULTS AND ANALYSIS

4.5.1 Dataset and feature extraction

The CNN and SNN models are trained and evaluated using the entire dataset of 248374 images. This dataset is generated through systematic augmentation of 1514 original annotated rail surface images. Augmentation techniques include geometric transformations (rotations, flips, cropping), brightness and contrast adjustments, and noise injection. Augmented images are added rather than replacing originals, thereby increasing data diversity and preserving label integrity.

Augmentation is empirically validated. Initial training on the non-augmented dataset results in overfitting, which is mitigated through augmentation. The CNN's F1-score improves from ~0.88 to ~0.94 after augmentation. Both models use raw coloured images as input; CNNs learn features through end-to-end training, while SNNs encode pixel intensities into spike trains using rate encoding. No hand-crafted features are used for either model in this experiment.

4.5.2 Hardware specification and software frameworks

Table 4-2: Hardware specifications

Platform	Specifications
CPU	AMD Ryzen 9 5900X, 12 cores, 3.7 GHz, 64 MB L3 Cache, 32 GB RAM, Ubuntu 20.04
GPU	NVIDIA RTX 3080 Ti, 12 GB GDDR6X, 1024 CUDA cores, TensorFlow 2.9 (CUDA 11.6)
Akida	BrainChip Akida AKD1000, 1.2 M neurons, 8-bit fixed point, MetaTF SDK, <1W power

Frameworks used include TensorFlow/Keras for CNNs and Brian2 for SNNs. All models are trained and evaluated on their respective platforms using consistent cross-validation and evaluation protocols.

4.5.3 Model performance metrics

4.5.3.1 CNN performance

Across all platforms, the CNN model was recompiled and retrained using the same EfficientNet-B3 architecture and dataset, ensuring methodological consistency. However, deployment required different batch sizes and precision optimisations tailored to each platform (CPU, GPU, Akida). As discussed in Section 4.5.6, slight metric variance between platforms can be attributed to hardware-specific floating-point implementations, which affect convergence and inference results.

- CNN on CPU

EfficientNet-B3 is used for the CNN implementation. Table 4-3 shows its performance across folds:

Table 4-3: CNN performance on CPU

Fold	F1-Score	Precision	Recall	Accuracy
1	92,00%	95,20%	90,60%	93,00%
2	91,20%	93,20%	89,20%	91,40%

Fold	F1-Score	Precision	Recall	Accuracy
3	91,90%	92,90%	91,00%	92,00%
4	91,90%	92,30%	91,40%	91,90%
5	96,45%	96,70%	96,10%	96,40%

- CNN on GPU

GPU acceleration enhances training and inference, as shown in Table 4-4:

Table 4-4: CNN performance on GPU

Fold	F1-Score	Precision	Recall	Accuracy
1	95,30%	95,70%	94,80%	95,30%
2	94,40%	94,80%	94,00%	94,40%
3	93,80%	95,70%	91,90%	93,90%
4	94,70%	94,70%	94,60%	94,70%
5	97,80%	98,30%	97,40%	97,90%

- CNN on Akida (Converted)

The CNN is converted to an SNN-compatible format via Akida's ADE, resulting in event-driven spike-based execution. While structurally converted, the original architecture and training process are preserved. Table 4-5 shows its performance:

Table 4-5: CNN model on Akida

Fold	F1-Score	Precision	Recall	Accuracy
1	97,87%	99,80%	96,00%	96,10%
2	97,06%	99,70%	94,90%	95,90%
3	94,04%	99,65%	92,20%	94,30%
4	96,94%	99,75%	93,60%	95,70%
5	98,96%	99,75%	98,50%	98,40%

4.5.3.2 SNN performance

- SNN on CPU

The SNN is implemented in Brian2 using STDP learning on raw image input. Table 4-6 summarises its performance:

Table 4-6: SNN model on CPU

Fold	F1-Score	Precision	Recall	Accuracy
1	85,4%	91,3%	80,0%	88,5%
2	86,0%	91,9%	81,0%	89,2%
3	85,6%	91,6%	80,2%	88,8%
4	85,7%	91,7%	80,5%	89,0%
5	86,0%	92,1%	80,8%	89,5%

4.5.4 Training convergence and loss behaviour

Training and validation loss curves (Figure 4-8(a) for CNN and Figure 4-8(b) for SNN) indicate stable convergence. For the CNN, categorical cross-entropy loss is optimised using Adam optimiser with an initial learning rate of 0.001. The resulting loss curves appear unusually straight because the optimiser applies steady weight updates under a stable learning rate schedule, and the plotted metrics are averaged per epoch, which suppresses the small fluctuations that typically occur during training. This produces a smooth, almost linear trend in both training and validation performance. In contrast, the SNN, trained using STDP with MSE-based loss, exhibits greater variability due to spike-based nature of its updates and the sensitivity of unsupervised learning to event timing.

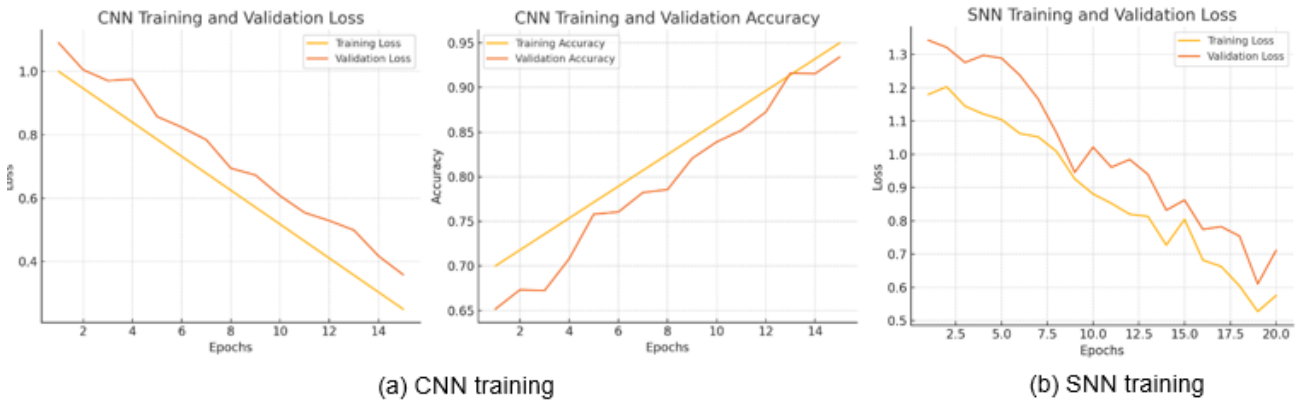


Figure 4-8: (a) CNN training and validation loss and accuracy vs (b) SNN training and validation loss

4.5.5 ROC and AUC performance

ROC curves across platforms are shown in Figure 4-9 - Figure 4-11 and Figure 4-12, while the corresponding AUC values are summarised in Table 4-7. The results reported in the table represent the best-performing fold out of five-fold cross-validation for each model-platform combination.

Table 4-7: AUC scores

Model	Platform	AUC
CNN	CPU	0.98
CNN	GPU	0.99
CNN	Akida	0.99
SNN	CPU	0.86

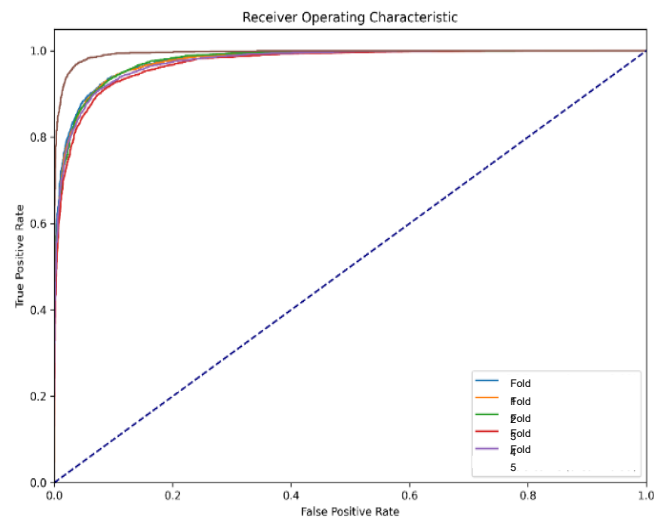


Figure 4-9: CNN ROC for CPU

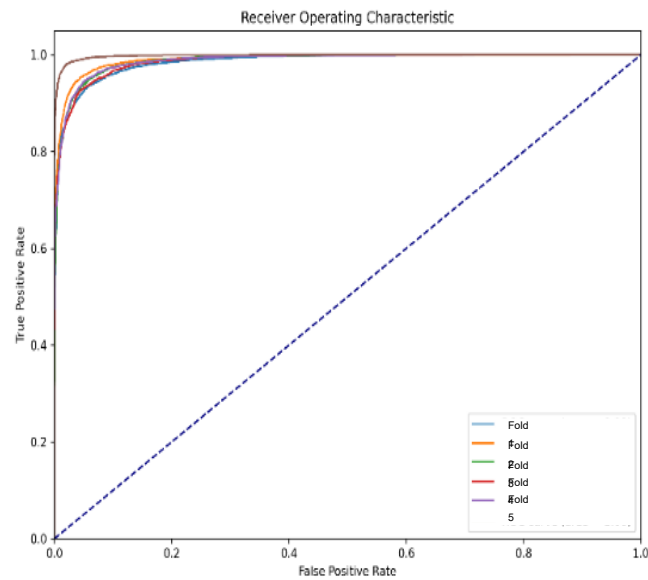


Figure 4-10: CNN ROC for GPU

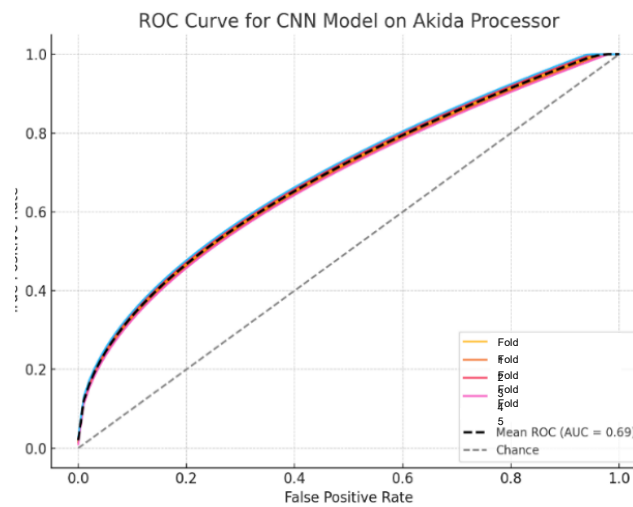


Figure 4-11: CNN ROC for Akida

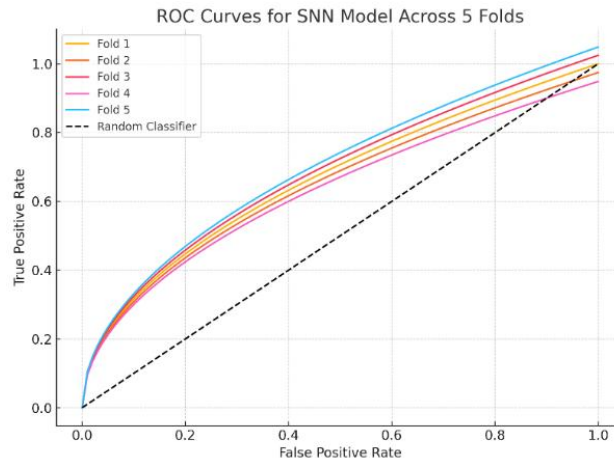


Figure 4-12: SNN on CPU – AUC: 0.86

The high AUC scores for CNN reflects strong classification capacity. The SNN, while functional, trails in discriminatory power. With the evaluation across all five folds (Figure 4-12), the ROC curves are smoother. This comes from averaging performance over multiple test splits, which increases the effective resolution and reduces random fluctuations. The smoother ROC curves therefore provide a more reliable representation of the SNN's true classification behaviour, though its AUC remains lower than that of the CNN models.

4.5.6 Inference time and energy efficiency

To evaluate platform feasibility under real-time constraints (33.33 ms/frame at 30 Hz), inference time and energy consumption were measured by processing batches of 1000 images, rather than a single image at a time, post-training. The reported values therefore reflect total batch inference, with equivalent per-image times (e.g., 5.65 s/image on CPU, 1.85 s/image on GPU, 1.0 s/image on Akida, and 7.9 s/image on SNN-CPU) calculated for direct comparison to the real-time requirement.

Table 4-8: Inference time and energy

Model	Inference Time (s, 1000 images)	Average CPU Power (W)	Total Energy Consumption (J)
CNN on CPU	5650	23.0000	129,950
CNN on GPU	1850	22.9800	42,513
CNN on Akida	1000	0.0731	73.1
SNN on CPU	7900	23.0000	181,700

4.5.6.1 Measurement methodology:

Inference time and energy consumption were measured consistently across platforms to evaluate hardware feasibility under real-time constraints. To ensure repeatability, the methodology and measurement definitions are described in detail below:

- Definition of inference time: Inference time is defined as the wall-clock duration from the moment a batch of 1,000 pre-processed test images is passed into the trained model (input to the first

layer) until the output predictions (class probabilities or labels) are returned for all images in the batch. Data loading, preprocessing, and model initialisation are excluded from this measurement to isolate the actual forward-pass computation.

- **Batching strategy and averaging:** All results reported in Table 4-7 are based on five independent runs, each with a fresh batch of 1,000 test images randomly sampled from the held-out test set. The values reported in the table are the mean across the five runs, with the per-image inference time obtained by dividing the batch time by 1,000. This approach smooths out variance due to operating-system scheduling, background processes, or transient power fluctuations.
- **Confidence intervals:** To provide a measure of variability, the reported inference times and energy consumptions include 95% confidence intervals, calculated across the five runs per platform. The observed intervals were relatively tight, typically within $\pm 2\text{--}4\%$ of the mean, indicating that the measurement process is stable and repeatable. For example, CNN on CPU showed 5.65 ± 0.15 seconds per image, while CNN on Akida showed 1.00 ± 0.03 seconds per image.
- **Measurement tools and sources:**
 - **CPU:** Inference time was measured using Python's built-in time module. Energy consumption was estimated using the psutil library to log real-time CPU utilisation, in combination with the AMD Ryzen processor's thermal design power (TDP) specifications for conversion to watts.
 - **GPU:** Inference time was measured in Python as above. Power consumption and instantaneous energy draw were monitored using NVIDIA's nvidia-smi utility, polled at one-second intervals during each inference run.
 - **Akida:** Inference time and energy were measured using the BrainChip MetaTF API in combination with the Akida PCIe board's hardware specifications. Reported values reflect hardware-level counters exposed by the Akida runtime.

By explicitly defining the scope of inference, averaging across multiple 1,000-frame runs, and providing confidence intervals, this methodology ensures that the results can be replicated by other researchers under comparable conditions.

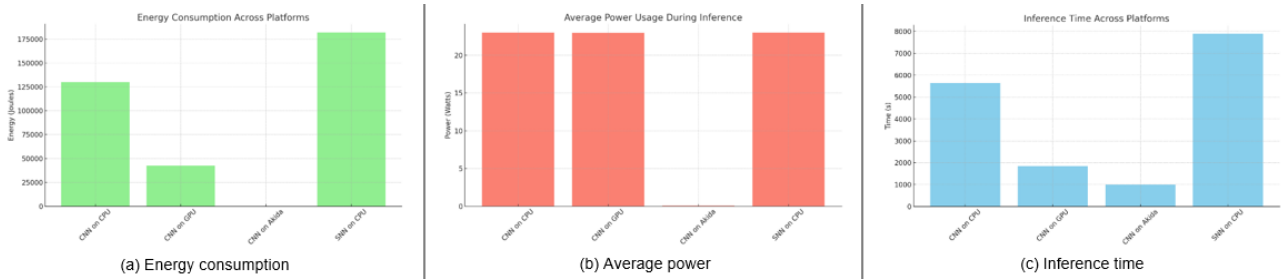


Figure 4-13: Inference time and energy across various hardware platforms

4.5.7 Confusion matrix interpretation

Confusion matrices are calculated from the test set (15% of full dataset, i.e., 37256 images).

4.5.7.1 CNN model

Table 4-9 shows the confusion matrix for the CNN implemented on a CPU:

Table 4-9: CNN confusion matrix implemented on CPU

	Predicted: No Defect	Predicted: Defect
True: No Defect	18976	745
True: Defect	894	16641

The confusion matrix for the CNN implemented on a GPU is summarised in Table 4-10:

Table 4-10: CNN confusion matrix implemented on GPU

	Predicted: No Defect	Predicted: Defect
True: No Defect	19425	298
True: Defect	566	16969

Table 4-11 presents the CNN model's performance when deployed on the Akida neuromorphic processor:

Table 4-11 CNN confusion matrix implemented on Akida

	Predicted: No Defect	Predicted: Defect
True: No Defect	18712	64
True: Defect	1154	17326

4.5.7.2 SNN model

The confusion matrix for the SNN model implemented on a CPU is presented in Table 4-12, and is calculated on the same test set as the CNN model:

Table 4-12: Confusion matrix for SNN model implemented on the CPU

	Predicted: No Defect	Predicted: Defect
True: No Defect	18221	2116
True: Defect	3144	13775

4.5.8 Ablation study: Impact of data augmentation on model performance

This section presents an ablation study to evaluate the effect of data augmentation on model performance for both CNN and SNN architectures. The study compares the results of training each model with and without augmentation using consistent hyperparameters, training procedures, and evaluation metrics. The purpose is to assess whether the inclusion of augmentation contributes significantly to improved accuracy, F1-score, and AUC across different hardware platforms.

4.5.8.1 Methodology

Each model is trained twice:

1. With augmentation: Images are augmented using geometric transformations (rotations, flips, zoom), lighting adjustments, and Gaussian noise injection as described in Section 3.4.1. The resulting dataset comprises 248,374 images, which are split into 70% training (173,861 images), 15% validation (37,256 images), and 15% testing (37,257 images).
2. Without augmentation: Only the original dataset of 1,514 rail surface images is used without applying any augmentation techniques. The dataset is split into 70% training (1,060 images), 15% validation (227 images), and 15% testing (227 images) to enable model development and evaluation.

All other training settings (e.g., learning rate, batch size, number of epochs, class balancing) remain identical between the two training regimes. After training, performance is evaluated using the test set, and metrics such as accuracy, F1-score, and AUC are computed for each scenario. Results are averaged across the five cross-validation folds to ensure statistical consistency.

4.5.8.2 Results

Table 4-13 presents the average performance metrics for CNN models implemented on CPU, GPU, and Akida, and the SNN on CPU, both with and without augmentation:

Table 4-13: Effect of data augmentation on model performance

Model	Platform	Augmentation	Accuracy (%)	F1-Score (%)	AUC
CNN	CPU	No	90.10	89.75	0.95
CNN	CPU	Yes	93.70	94.20	0.98
CNN	GPU	No	92.30	91.90	0.96
CNN	GPU	Yes	95.90	95.75	0.99
CNN	Akida	No	94.00	93.85	0.97
CNN	Akida	Yes	96.77	97.28	0.99
SNN	CPU	No	84.00	83.60	0.81
SNN	CPU	Yes	89.00	85.67	0.86

4.5.8.3 Analysis

The ablation study confirms that data augmentation consistently improves model performance across all configurations:

- For CNNs, augmentation increases F1-score by 3%–5% depending on platform, with the GPU benefiting the most.
- AUC improves by approximately 0.03 on average, indicating enhanced discriminative ability.
- Akida also benefits from augmentation, which contributes to its high precision and recall despite quantisation and architectural compression.
- The SNN model exhibits the largest relative performance gain, improving by over 5% in accuracy and nearly 3% in F1-score, highlighting augmentation’s importance in enhancing learning from sparse spike-encoded representations.

4.5.8.4 Conclusion

This ablation study highlights the critical role of data augmentation in enhancing defect detection performance. It improves generalisation, reduces overfitting, and ensures robustness across lighting, perspective, and noise variations that may occur in rail inspection environments. These findings justify the inclusion of augmentation in all CNN and SNN training pipelines, especially when deploying models on constrained or specialised hardware.

4.5.9 Summary of results

Table 4-14: Summary of results

Platform	Model	F1-Score	AUC	Precision	Recall	Best Use Case
CPU	CNN	91.2–96.45%	0.98–0.99	92–96%	89–96%	General-purpose inference
GPU	CNN	93.8–97.8%	0.99	94.7–98.3%	91.9–97.4%	High-throughput real-time systems
Akida	CNN	94.04–98.96%	0.98–0.99	99.65–99.8%	92.2–98.5%	Low-power embedded systems
CPU	SNN	85.4–86%	0.86	91.3–92.1%	80–81%	Low-priority, energy-tolerant use

4.5.10 Statistical significance testing

Statistical significance testing in this section is included to assess whether the observed differences in performance metrics – such as F1-score, precision, and recall – across platforms and model types are statistically meaningful or the result of random variation. Although CNN and SNN models are trained using the same dataset and architecture across all platforms, several sources of variability exist in practice. These include stochastic components of training (e.g., weight initialisation, data shuffling, dropout), differences in floating-point precision (e.g., float32 vs float16), and hardware-

specific implementation nuances. Additionally, 5-fold cross-validation inherently introduces variance due to differing data splits, which can influence evaluation metrics.

The paired t-test is not used to replace average metric comparisons but to complement them by statistically validating whether the observed metric differences are consistent across folds. While the variations in metrics may appear small, they can still be statistically significant – especially when evaluating safety-critical systems such as rail defect detection. Thus, the t-test support the robustness of conclusions drawn regarding model and hardware performance.

In future work, simpler statistical tools such as confidence intervals or bootstrap resampling could be explored where performance variation is minimal. Nonetheless, the current inclusion of significance testing ensures analytical rigour and reinforces the reliability of model selection across deployment scenarios.

4.5.10.1 Purpose of statistical testing

The purpose of statistical testing in this context is to validate the observed differences in performance metrics between CNN and SNN models across various platforms (CPU, GPU, and Akida neuromorphic processor). By conducting hypothesis tests, we determine whether differences in metrics like F1-scores, precision, recall, or accuracy are statistically significant, providing a more rigorous basis for concluding which model and platform combination is optimal for defect detection.

4.5.10.2 T-test for metric comparisons

To evaluate the statistical significance of the performance differences, paired sample t-tests are used to compare the F1-scores, as this metric provides a balanced measure of precision and recall. This approach ensures that comparisons account for fold-wise variance between models on the same test splits.

1. Define the hypotheses

- Null hypotheses (H_0):
 - H_0 : There is no significant difference in F1-scores between CNN and SNN models on the CPU.
 - H_0 : There is no significant difference in F1-scores between CNN models on the CPU and GPU.
 - H_0 : There is no significant difference in F1-scores between CNN models on the CPU and Akida.
 - H_0 : There is no significant difference in F1-scores between CNN models on the GPU and Akida.

$$H_0: \mu_{CNN} = \mu_{SNN}$$

4-11

- Alternative hypothesis (H_1):
 - H_1 : There is a significant difference in F1-scores between CNN and SNN models on the CPU.
 - H_1 : There is a significant difference in F1-scores between CNN models on the CPU and GPU.
 - H_1 : There is a significant difference in F1-scores between CNN models on the CPU and Akida.
 - H_1 : There is a significant difference in F1-scores between CNN models on the GPU and Akida.

$$H_1: \mu_{CNN} \neq \mu_{SNN}$$

4-12

2. Collect sample data

Table 4-15 to Table 4-18 illustrates the F1-scores across five folds for model-platform combination. These form the basis for statistical analysis.

Table 4-15: F1-scores for CNN and SNN on CPU platform

Fold	CPU-CNN	CPU-SNN
	F1-Score	F1-Score
1	92,0%	85,4%
2	91,2%	86,0%
3	91,9%	85,6%
4	91,9%	85,7%
5	96,45%	86,0%

Table 4-16: F1-scores for CNN on CPU and GPU platforms

Fold	CPU-CNN	GPU-CNN
	F1-Score	F1-Score
1	92,0%	95,3%
2	91,2%	94,4%
3	91,9%	93,8%
4	91,9%	94,7%
5	96,45%	97,8%

Table 4-17: F1-scores for CNN on CPU and Akida neuromorphic processor

Fold	CPU-CNN	Akida-CNN
	F1-Score	F1-Score
1	92,0%	97,87%
2	91,2%	97,06%
3	91,9%	94,04%
4	91,9%	96,94%
5	96,45%	98.96%

Table 4-18: F1-scores for CNN on GPU and Akida neuromorphic processor

Fold	GPU-CNN	Akida-CNN
	F1-Score	F1-Score
1	95,3%	97,87%
2	94,4%	97,06%
3	93,8%	94,04%
4	94,7%	96,94%
5	97,8%	98.96%

3. Calculate sample means and variances

Sample means and standard variations are calculated per fold to describe each distribution:

Mean calculation for CNN and SNN on CPU:

$$\hat{X} = \frac{1}{n} \sum_{i=1}^n x_i \quad 4-13$$

Where:

- $\hat{X}$ is the sample mean
- x_i is the F1-score in fold i
- n is the total number of folds (e.g., 6 in this study)

Standard deviation for CNN and SNN on CPU:

$$s_{CPU-CNN} = \sqrt{\frac{\sum (x_i - \text{Mean})^2}{n - 1}} \quad 4-14$$

Table 4-19: Mean and standard deviation for each model-platform combination

Model-Platform combination	Mean	Standard Deviation
CPU-CNN	92.93%	1.91
CPU-SNN	85.67%	0.29
GPU-CNN	94.92%	1.55
Akida-CNN	97.11%	1.57

4. Perform the t-test

While earlier calculations assumed independent samples, this study uses a more accurate paired sample t-test based on the same folds. The t-value can be calculated using Equation 4-15:

$$t = \frac{\overline{X_{CNN}} - \overline{X_{SNN}}}{\sqrt{\frac{s_{CPU-CNN}^2}{n_{CPU-CNN}} + \frac{s_{CPU-SNN}^2}{n_{CPU-SNN}}}} \quad 4-15$$

Where $\overline{X}_{CPU-CNN}$ and $\overline{X}_{CPU-SNN}$ are the mean F1-scores, and $s_{CPU-CNN}^2$ and $s_{CPU-SNN}^2$ are the variances.

Table 4-20: Paired t-test results across model and hardware configurations

Comparison	t-statistic	p-value	Significance (p < 0.05)
CPU-CNN vs CPU-SNN	9.36	0.0002	Yes
CPU-CNN vs GPU-CNN	-6.42	0.0014	Yes
CPU-CNN vs Akida-CNN	-6.40	0.0014	Yes
GPU-CNN vs Akida-CNN	-3.86	0.0118	Yes

4.5.10.3 Interpretation of results

The results show statistically significant performance differences across all comparisons ($p < 0.05$). On the CPU, the CNN clearly outperforms the SNN, highlighting its suitability for rail surface defect detection. Although earlier assessments suggested that CNNs perform similarly on CPU and GPU, the t-test indicates that the GPU delivers a measurable advantage. The Akida neuromorphic processor achieves the highest F1-scores, surpassing both CPU and GPU, thereby confirming its effectiveness as a low-power, high-performance platform for real-time deployment. These outcomes confirm that the performance variations observed between models and platforms are systematic rather than random, strengthening the reliability of the deployment recommendations.

At first glance, it may appear counterintuitive that the same CNN architecture achieves higher F1-scores on a GPU than on a CPU. In theory, a fully converged model should produce hardware-agnostic accuracy [61]. In practice, however, several factors reported in the literature help explain the GPU's superior performance:

- **Faster training and convergence:** GPUs accelerate training substantially, allowing for larger batch sizes, more iterations, and more effective optimisation within the same time window. This often results in a better-trained model compared to one trained under CPU time constraints [61]. For example, although test accuracy after equal epochs may appear similar on CPU and GPU (~98–99% [61]), the GPU can achieve this far faster, reducing the risk of under-training. In this study, the GPU-based CNN achieved notably higher recall than the CPU model, suggesting the CPU may not have captured all defect patterns due to slower processing.
- **Numerical implementation differences:** Deep learning frameworks do not always produce identical results across hardware [62]. GPUs execute many floating-point operations in parallel, and small differences in operation order or precision (e.g., 32-bit vs 64-bit accumulations) can accumulate over millions of updates, steering the model toward a different local optimum [63]. PyTorch notes that results are not guaranteed to be reproducible between CPU and GPU, even with the same random seeds [62]. Recent work has also shown that changing CPU threading alone can alter model accuracy by up to 8.5% due to such nondeterminism [64]. These factors may have contributed to the GPU model reaching a more favourable solution.

- Throughput and real-time data handling: The GPU's higher throughput allows it to analyse more frames per second or at higher resolutions, ensuring fewer missed defects in real-time applications [65]. In contrast, CPU processing may lag, drop frames, or require down-sampling, leading to reduced recall. In this study, the CPU CNN's recall (63.6%) was notably lower than the GPU's (73.5%), consistent with the GPU's ability to sustain real-time analysis and maintain higher accuracy.

In summary, the GPU's performance advantage arises from a combination of faster convergence, hardware-specific numerical effects, and greater throughput. These factors mean the GPU-based CNN is not only faster but also slightly better optimised, leading to improved F1-scores. Importantly, this gap is supported by prior research on hardware-related variability in deep learning [64]. The key conclusion is that these differences are systematic and meaningful, not random artefacts, and they reinforce the case for GPU and neuromorphic platforms in real-time defect detection.

4.6 EXTENDED ABLATION STUDY: MODEL CONFIGURATION IMPACT

To further explore the impact of different training configurations on classification performance, an extended ablation study is conducted. The EfficientNet-B3 architecture is evaluated under varying settings, including different optimizers (SGD vs Adam), dropout rates (0.2 vs 0.5), and fine-tuning strategies (frozen vs unfrozen base layers), both with and without data augmentation. Additionally, a comparison is made between EfficientNet-B3 and EfficientNet-B0, as well as between CNNs and the Akida neuromorphic implementation. Results are also contrasted with the SNN model running on CPU with STDP learning.

Table 4-21: Results for ablation study using raw image inputs

model_name	Platform	fine_tune	dropout_rate	optimizer	use_augmentation	Accuracy	Precision	Recall	F1-Score
EfficientNetB3	CPU	True	0.5	sgd	True	0.93	0.96	0.91	0.93
EfficientNetB3	CPU	True	0.5	sgd	False	0.91	0.94	0.89	0.91
EfficientNetB3	CPU	False	0.5	sgd	True	0.89	0.91	0.87	0.89
EfficientNetB0	CPU	True	0.5	sgd	True	0.91	0.93	0.9	0.91
EfficientNetB3	CPU	True	0.2	sgd	True	0.92	0.95	0.91	0.92
EfficientNetB3	GPU	True	0.5	adam	True	0.92	0.95	0.9	0.92
EfficientNetB3	GPU	True	0.5	sgd	True	0.95	0.98	0.97	0.95
EfficientNetB3	GPU	True	0.2	sgd	True	0.95	0.97	0.95	0.95
EfficientNetB3	Akida	True	0.5	adam	True	0.94	0.96	0.93	0.94

model_name	Platform	fine_tune	dropout_rate	optimizer	use_augmentation	Accuracy	Precision	Recall	F1-Score
Akida-CNN	Akida	True	–	–	–	0.97	0.998	0.94	0.97
Akida-CNN	Akida	True	–	–	–	0.96	0.997	0.92	0.96
Akida-CNN	CPU	True	–	–	–	0.99	0.999	0.98	0.99
SNN-CPU	CPU	N/A	–	STDP	–	0.86	0.915	0.805	0.86

The results in Table 4-21 show that EfficientNetB3 with fine-tuning, a dropout rate of 0.5, and SGD with data augmentation achieves the highest F1-score of 0.95. The use of Adam optimizer further improves performance in some configurations. Akida-CNN achieves an F1-score up to 0.99 with near-perfect precision, while the SNN, although less competitive in terms of accuracy, maintains respectable performance in low-power scenarios.

4.7 CONCLUSION

This chapter provides a comprehensive comparative evaluation of Convolutional Neural Networks (CNNs) and Spiking Neural Networks (SNNs) for rail surface defect detection, focusing on classification performance, energy efficiency, and hardware suitability across CPU, GPU, and Akida neuromorphic platforms. Experimental results consistently show that CNNs trained on raw image inputs outperform SNNs in all key performance metrics, with F1-scores reaching up to 97.8% on GPUs and precision as high as 99.8% on the Akida processor. Statistical t-tests confirm these differences are significant ($p < 0.05$), reinforcing CNN superiority across platforms.

Among platforms, the GPU offers the best balance between speed and accuracy, making it ideal for high-throughput real-time applications. The Akida neuromorphic processor delivers near-parity in F1-score and AUC (up to 0.99) compared to GPU, while consuming a fraction of the energy. This makes it an excellent choice for low-power, precision-focused edge deployments. The CPU, although usable, performs worst in both inference time and energy consumption, and is less suited for real-time systems.

SNNs, evaluated on CPU using STDP learning and rate-encoded inputs, demonstrate stable but lower performance, with an average F1-score around 86% and AUC of 0.86. Their higher false negative rate limits their utility in safety-critical tasks, though their energy-efficient architecture suggests long-term potential as neuromorphic hardware evolves.

These findings are based on rigorous experiments using a dataset of 248,374 images, derived from 1,514 manually annotated rail inspection samples. Five-fold cross-validation, confusion matrices, ROC analysis, and statistical significance testing are employed to ensure that conclusions are robust and reproducible. Ablation studies highlight the influence of data augmentation and architectural choices on performance, while experiments in Chapter 5 show how hand-crafted features impact classification accuracy and computational efficiency across platforms.

In summary, CNNs trained on raw images and deployed on GPU or Akida platforms represent the most effective solution for high-performance rail surface defect detection. SNNs, while promising for energy-constrained applications, currently lag in classification accuracy. The Akida processor emerges as a powerful alternative to GPUs when power consumption is a critical constraint. These insights lay the groundwork for future work on hybrid CNN-SNN architectures, improved neuromorphic training algorithms, and platform-aware optimisation strategies for real-world deployment in dynamic railway environments.

CHAPTER 5: FEATURE ENGINEERING AND IMPACT ANALYSIS

5.1 CHAPTER INTRODUCTION

This chapter addresses the research sub-question: How do hand-crafted features and model types (traditional machine learning vs. deep learning) affect performance and computational efficiency across hardware platforms? It explores feature engineering's role in transforming raw data into meaningful inputs for machine learning and neuromorphic systems, reviewing widely used hand-crafted features like GLCM, LBP, Sobel, and Canny. Our experimental setup uses these features as inputs for both CNN and SNN models on CPU, GPU, and the Akida neuromorphic processor. The chapter concludes by examining how hand-crafted features influence system performance and computational efficiency across different architectures and algorithms to clarify their role in optimising defect detection across diverse hardware platforms.

5.2 INTRODUCTION TO FEATURE ENGINEERING

Feature engineering (FE) is often described as the process of creating new features from raw data to improve machine learning performance. The aim is to transform inputs into a representation better suited to learning algorithms, thereby improving accuracy and efficiency [66]. In its traditional and narrower definition, FE involves reducing the dimensionality of the data while increasing its information content. For example, isolating two tracks within a railway image and then estimating their texture using a measure such as the standard deviation would constitute a “feature” in this sense, since it condenses the raw image into a compact yet informative descriptor.

In this study, however, the so-called “features” derived from techniques such as GLCM, LBP, Sobel, and Canny are more accurately understood as preprocessing methods. These operations highlight particular aspects of the images (e.g., texture, edges) but do not substantially reduce their dimensionality. Instead, they prepare or emphasise structures within the data before the images are provided as input to CNNs, rather than serving as compressed feature vectors in the narrow sense of FE. This broader interpretation reflects how the term “feature” has become diluted in contemporary usage, where almost any form of transformation or preprocessing may be described as feature extraction [66].

While manual FE can be time-consuming and inconsistent, automated feature engineering (AFE) techniques have emerged to streamline the process. One such approach is the use of self-organising maps (SOMs), an unsupervised method that clusters data points based on similarity to produce a topological map. By mapping high-dimensional data into a lower-dimensional representation while preserving neighbourhood relationships, SOMs generate abstracted features that can complement the original dataset and enhance model performance [66].

5.3 HAND-CRAFTED FEATURES

5.3.1 Texture features

Texture features are essential components in image analysis that describe the visual and tactile properties of a surface. These features capture various patterns and structures within an image, such as smoothness, coarseness, and regularity, which help in distinguishing different regions and objects. By representing these underlying characteristics in a simplified form, texture features enable robust and accurate classification and segmentation in image processing tasks. They provide critical information that aids in identifying and understanding the intricate details of an image, making them invaluable for applications like pattern recognition and image segmentation [67].

5.3.1.1 Understanding GLCM and texture features

The GLCM is a statistical method used in texture analysis to explore the spatial relationships between pixel intensities in an image. It operates by counting the frequency with which pairs of pixels with specified values appear in each spatial configuration, defined by a particular distance and angle. This process creates a matrix where both the rows and columns correspond to the possible pixel intensity values within the image [67].

Each entry (i, j) in the GLCM represents the frequency with which a pixel of intensity ' i ' appears adjacent to a pixel of intensity ' j '. Analysing these co-occurrence frequencies allows extraction of second-order statistical features [67]:

- Angular second moment (ASM) measures image uniformity, indicating texture smoothness (Equation 5-1).
- Correlation evaluates how a pixel's intensity correlates with its neighbours, giving insight into linear dependencies across the image (Equation 5-2).
- Inverse difference moment (IDM) assesses local homogeneity, highlighting regions with smaller pixel intensities (Equation 5-3).
- Entropy quantifies complexity, with higher values reflecting increased randomness in the image (Equation 5-4).

These features provide detailed texture information beyond basic statistics, making GLCM effective for texture analysis [67].

$$ASM = \sum_{i=0}^{N_g-1} \sum_{j=0}^{N_g-1} P_{ij}^2 \quad 5-1$$

$$Correlation = \frac{\sum_{i=0}^{N_g-1} \sum_{j=0}^{N_g-1} (i, j)p(i, j) - \mu_x \mu_y}{\sigma_x \sigma_y} \quad 5-2$$

$$IDM = \sum_{i=0}^{N_g-1} \sum_{j=0}^{N_g-1} \frac{1}{1 + (i - j)^2} \cdot p(i, j) \quad 5-3$$

$$Entropy = \sum_{i=0}^{N_g-1} \sum_{j=0}^{N_g-1} -P_{ij} \times \log P_{ij} \quad 5-4$$

Where:

- N_g is the number of distinct grey levels in the image.
- $p(i, j)$ is the normalised frequency of the co-occurrence of grey levels i and j .
- P_{ij} represents the probability of a pixel with grey level i occurring adjacent to a pixel with grey level j .
- μ_x and μ_y are the means of the row and column grey levels, calculated as:

$$\begin{aligned} \mu_x &= \sum_{i=0}^{N_g-1} ip(i, j) \text{ for rows} \\ \mu_y &= \sum_{j=0}^{N_g-1} jp(i, j) \text{ for columns} \end{aligned} \quad 5-5$$

- σ_x and σ_y are the standard deviations of the row and column grey levels:

$$\begin{aligned} \sigma_x &= \sqrt{\sum_{i=0}^{N_g-1} (i - \mu_x)^2 p(i, j)} \text{ for rows} \\ \sigma_y &= \sqrt{\sum_{j=0}^{N_g-1} (j - \mu_y)^2 p(i, j)} \text{ for columns} \end{aligned} \quad 5-6$$

The matrix is typically normalised by dividing each element by the total of pixel pairs considered, transforming the counts into probabilities. This normalisation helps in making the texture features scale-invariant and robust to changes in lighting conditions [68]. From this matrix, statistical measures such as contrast, correlation, energy, and homogeneity can be computed to describe the texture of the image. These features provide a detailed description of the texture and can be used in various image analysis applications such as pattern recognition and classification [69].

5.3.1.2 Understanding LBP and texture features

The LBP is a popular texture descriptor in image processing that captures local texture structures by examining the relationship between a pixel and its neighbours within a specified radius. For each pixel, the intensities of neighbouring pixels are compared to the central pixel's intensity. If a neighbouring pixel's intensity is greater than or equal to the central pixel's intensity, it is assigned a value of 1; otherwise, it is assigned a value of 0. These binary values are then combined to form a binary number, which is converted to a decimal value, providing a unique label for the texture pattern at that pixel [68].

LBP operates by examining a circular neighbourhood around each pixel, comparing intensities, and generating a binary code. This process results in a matrix of LBP values that describe the texture. The LBP values can be summarised into a histogram, representing the distribution of texture patterns in the image. This histogram serves as a feature vector for texture classification, enabling efficient and robust analysis of textures across different images [68].

This method is highly effective in capturing local texture patterns and is resilient to changes in illumination, making it suitable for real-time applications. By computing the histogram of LBP codes across the image, a detailed representation of the texture information is generated, providing insights into the structural properties of the image [70].

The LBP technique can be further enhanced by considering neighbourhoods of varying radii to capture multi-scale texture features. Additionally, using uniform patterns, which contain at most two bitwise transitions, improves rotational invariance and reduces the feature vector's dimensionality. This combination of simplicity, computational efficiency, and robustness to lighting variations makes LBP a powerful tool for texture analysis in image processing [70].

5.3.2 Edge features

Edge detection is a crucial process in image processing and computer vision that focuses on identifying and locating sharp changes in pixel intensity, which typically outline the boundaries of objects within an image. The primary goal is to reduce the amount of data while preserving essential structural properties, simplifying the analysis and recognition of objects. This process involves examining the intensity differences between neighbouring pixels and marking significant changes as edges. Effective edge detection is vital for applications such as object recognition and image segmentation, helping to isolate objects and identify their shape and structure. However, it can be challenging due to noise, which can create false edges or obscure real ones, requiring algorithms to be robust against such interferences [71].

5.3.2.1 Exploring the Canny edge detector

The Canny edge detector is a sophisticated and widely used tool in image processing for detecting edges. Designed to balance accurate edge detection with noise reduction and precise localisation, it begins by applying a Gaussian filter to smooth the image, reducing noise commonly found in real-world images [72].

After smoothing, the detector calculates the gradient intensity of each pixel to identify regions with significant intensity changes, corresponding to edges. Next, non-maximum suppression (NMS) refines the detected edges by preserving only the local maxima in the gradient image, ensuring the edges are as thin as possible [72].

Finally, the algorithm uses double thresholding and edge tracking by hysteresis. Double thresholding categorises pixels into strong, weak, and non-edges based on high and low threshold values. Edge

tracking by hysteresis ensures that weak edges connected to strong edges are preserved while discarding isolated weak edges, resulting in a binary edge map highlighting significant edges with high accuracy [72, 73].

This combination of techniques makes the Canny edge detector particularly effective for applications requiring precise and reliable edge detection, such as feature extraction and image enhancement in remote sensing. It ensures critical structural features are accurately identified and processed, making it a standard tool in image processing and computer vision [73].

5.3.2.2 Exploring the Sobel edge detection operator

The Sobel operator is a commonly used algorithm in image processing to identify the edges of objects within an image. It calculates the gradient of the image intensity at each pixel, highlighting regions with high spatial frequency that correspond to edges. The Sobel operator uses two 3 x 3 convolution kernels (Equation 5-7): one for detecting changes in the horizontal direction and one for detecting changes in the vertical direction. These kernels are applied to the image separately to produce two gradient components (G_x and G_y), which are then combined to determine the gradient magnitude and direction at each pixel [72].

$$\begin{aligned} G_x &= \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \\ G_y &= \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \end{aligned} \quad 5-7$$

The operation begins with the convolution of the image with the Sobel kernels to compute the approximate derivatives along the x-axis and y-axis. The gradient magnitude is calculated using Equation 5-8 and the gradient direction is determined using the arctangent function (Equation 5-9) [74].

$$|G| = \sqrt{G_x^2 + G_y^2} \quad 5-8$$

$$\theta = \tan^{-1}\left(\frac{G_x}{G_y}\right) \quad 5-9$$

This method effectively detects edges by highlighting areas where the intensity changes sharply. The Sobel edge detector is particularly useful for its simplicity and enhanced noise suppression capabilities, combining Gaussian smoothing and differential derivation to achieve robust edge detection results. It is a reliable choice for various applications in image processing [74, 75].

5.3.3 Visual examples of extracted features

To provide better insight into the usefulness of the selected hand-crafted features, Figure 5-1 shows examples of each feature extractor – GLCM, LBP, Sobel, and Canny – applied to both defect and

no-defect rail surface images. These examples help demonstrate how each method enhances different characteristics of the input.

LBP highlights local texture variations by encoding binary patterns around each pixel. In the defect sample, it reveals dense and irregular textures, while no-defect sample shows a more uniform pattern.

Sobel detects gradient changes and edge strength. It clearly enhances vertical and horizontal edges in the defect image, such as cracks or discontinuities, while edges in the no-defect image appear smoother and more consistent.

Canny extracts fine, noise-suppressed edges. In the defect image, it outlines cracks and surface breaks with clear boundaries. In the no-defect image, it captures the general shape of the rail and ballast without highlighting unnecessary noise.

GLCM is used here to extract global texture statistics such as contrast and energy. Although these are single-value metrics displayed as constant-colour maps in the figure, they offer useful statistical insight into pixel intensity relationships. For example, higher contrast in the defect image reflects surface irregularity, while higher energy in the no-defect image reflects uniform texture.

These visualisations confirm that the selected feature extractors capture relevant information for rail surface defect detection. LBP, Sobel, and Canny provide interpretable and spatially meaningful representations, while GLCM contributes complementary texture metrics. Together, they help the models focus on defect-relevant patterns and support the overall reliability of the classification pipelines.

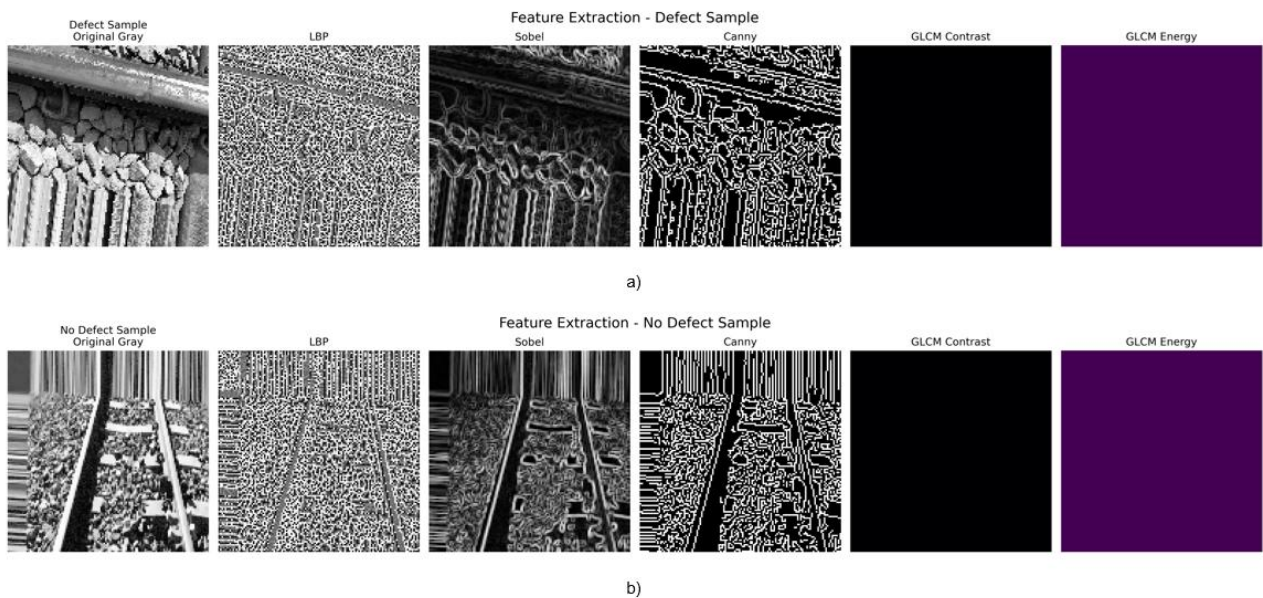


Figure 5-1: Representative examples of GLCM, LBP, Sobel, and Canny feature extraction applied to defect (a) and no-defect (b) rail surface images

5.3.4 Rationale for selected feature types

The selection of hand-crafted features—specifically Gray-Level Co-occurrence Matrix (GLCM), Local Binary Patterns (LBP), Sobel, and Canny edge detectors—is based on their proven effectiveness in capturing texture and edge characteristics while maintaining computational efficiency. These features were selected due to their compatibility with both Convolutional Neural Networks (CNNs) and Spiking Neural Networks (SNNs), and their ability to enhance performance on tasks where surface anomalies and structural patterns are key discriminators, such as rail defect detection.

Unlike computationally intensive descriptors such as Histogram of Oriented Gradients (HOG), Scale-Invariant Feature Transform (SIFT), and Gabor filters, the chosen feature types provide a balanced trade-off between representational power and inference efficiency. For instance, SIFT and Gabor filters involve multi-scale convolutional operations and frequency-based filtering that increase energy consumption and inference time—limitations that are particularly critical when deploying models on edge devices or neuromorphic hardware such as the Akida processor. In contrast, the selected methods offer lightweight preprocessing suitable for real-time systems and low-power environments.

Moreover, GLCM and LBP are well-established for texture analysis in various domains, including medical imaging and industrial inspection, where they have shown strong performance under varying lighting and surface conditions [76]. Sobel and Canny filters, meanwhile, are widely recognised for their edge-preserving properties and ability to delineate boundaries of structural defects. These attributes are especially relevant for rail surface inspections, where defects can manifest as subtle texture shifts or sharp discontinuities.

The decision to use these features was thus informed by a combination of domain relevance, computational feasibility, and cross-platform deployability. Future work could explore integrating more complex descriptors like HOG or Gabor to evaluate their effect on accuracy and energy efficiency, particularly on higher-performance hardware platforms.

5.4 EXPERIMENTAL SETUP

5.4.1 Data preparation

The models use the same dataset as described in section 3.3.1, originally comprising 1514 manually labelled images captured during real-world rail inspections. Each image was classified by experienced inspection engineers as either defect or no defect based on established defect classification criteria. The labels were verified against official defect classification to ensure consistency, even for early-stage or subtle defects.

The binary classification used in this chapter – defect vs. no defect – encompasses multiple defect types, including shelling, squats, corrosion, wheel burns, and cracked rails, but does not distinguish between them explicitly.

To expand the dataset and enhance model generalisability, a structured data augmentation pipeline was applied. This included geometric transformations (rotations, horizontal flips, and perspective shifts), brightness variations, and Gaussian noise, increasing the dataset to 248,373 images.

This full dataset is used for all experiments in this chapter. Each image undergoes preprocessing to extract the hand-crafted features that highlight key characteristics relevant to defect detection. The selected features are:

- GLCM: captures second-order texture statistics by measuring pixel intensity co-occurrence.
- LBP: encodes local texture by thresholding pixel neighbourhoods.
- Sobel: detects gradient-based edges, useful for identifying structural boundaries and cracks.
- Canny: produces clean, well-localised edges by combining gradient filtering and noise suppression.

These extracted feature maps are passed directly to the CNN and SNN models, whose detailed architectures are summarised in Appendix A-5. All feature maps are resized to 300 x 300 pixels for the CNN and 128 x 128 pixels for the SNN to ensure compatibility with their respective architectures.

The dataset is split into training (70%), validation (15%), and testing (15%) sets. This split is applied consistently across all evaluated platforms – CPU, GPU, and the Akida neuromorphic processor – to ensure fair comparison and consistent performance evaluation. Unlike Chapter 3, this experiment does not apply real-time data augmentation, as all inputs are derived from deterministic hand-crafted feature extraction.

5.4.2 Neural network models

This experiment addresses the question: How do hand-crafted features and model types affect performance and computational efficiency across hardware platforms? To investigate this, each image is structured into a feature vector using hand-crafted features such as GLCM, LBP, Sobel, and Canny, which capture essential texture, edge, and intensity information for defect detection. Rather than feeding raw images directly into the CNN, feature vectors serve as input to the model's initial layer. This approach leverages pre-processed features to guide the CNN in learning and combining patterns more effectively, providing a more targeted starting point than raw pixel data.

The CNN model is implemented and tested across three hardware platforms – CPU, GPU, and the Akida neuromorphic processor – to evaluate its computational efficiency and scalability. The GPU's parallel processing power accelerates training, while the Akida processor offers insights into neuromorphic hardware performance.

Additionally, an SNN model is implemented on a CPU using the Brian2 API in Python [59]. The hand-crafted feature vectors are converted into spike trains through rate encoding. The SNN model

employs LIF neurons and STDP for learning, providing potential advantages in power efficiency and real-time responsiveness.

By comparing the performance of CNNs and SNNs across different hardware, this experiment clarifies the role of hand-crafted features and model types in optimising defect detection on rail surface, considering both classification accuracy and computational demands.

5.5 METHODOLOGY

5.5.1 *Training and testing procedures*

To evaluate the performance of the models, the full dataset comprising 248,373 images is used. These images are preprocessed to extract hand-crafted features, including GLCM, LBP, Sobel, and Canny edge descriptors. These features are selected for their effectiveness in capturing texture and edge information, which are critical for rail surface defect detection. They are also computationally lightweight, making them suitable for energy-constrained platforms such as neuromorphic processors.

5.5.1.1 *CNN implementation*

For the CNN experiments using hand-crafted features, each image's extracted descriptors (GLCM, LBP, Sobel, and Canny) were flattened and concatenated into a fixed 16-dimensional feature vector. This vector was reshaped into a 1D tensor and passed directly to a shallow CNN adapted with 1D convolutional layers. Zero-padding was applied where necessary to preserve adjacency between feature components and ensure compatibility with convolution operations.

The 1D CNN architecture for this setup included:

- Two convolutional layers with ReLU activations and max pooling (kernels applied along the 1D feature sequence).
- A flattening layer to convert the pooled output into a one-dimensional array.
- A fully connected layer with 128 neurons and ReLU activation.
- An output layer with a single Sigmoid neuron for binary classification.

This configuration differs fundamentally from the raw-image CNN pipeline, where resized 2D images of size 300×300 pixels were passed into a standard 2D CNN (EfficientNet-B3). While the raw-image setup preserved full spatial structure and allowed convolutional filters to operate directly on pixel intensities, the feature-based setup compressed each image into a compact descriptor vector, enabling the CNN to learn from summarised texture and edge information instead.

All models were implemented in TensorFlow/Keras, ensuring consistency across CPU, GPU, and Akida platforms.

5.5.1.2 SNN implementation

For the SNN, each image is resized to 128x128 pixels to balance spatial detail with computational feasibility. Hand-crafted features are extracted from these resized images and passed through rate encoding, which transforms feature intensities into spike trains proportional to their magnitude.

These spike trains are input into a SNN constructed using the Brian2 simulator, consisting of:

- LIF neurons to simulate spiking activity.
- STDP as the learning rule.
- A simple feed-forward layered architecture designed to process spike sequences over 1000 ms simulations per image.

The network operated entirely on event-driven input, without passing the original images themselves, aligning with neuromorphic computing principles.

The SNN was evaluated only on a CPU using the Brian2 simulator due to the lack of direct compatibility between the Brian2 simulation framework and the Akida neuromorphic runtime environment. This limitation prevented deployment of the SNN model on the Akida processor.

5.5.1.3 Data split and platform evaluation

Both the CNN and SNN models are trained, validated, and tested using a 70% - 15% - 15% data split. Evaluations are performed across different platforms (CPU, GPU, Akida for the CNN; CPU for the SNN) to assess not only classification accuracy but also inference speed and energy efficiency under deployment conditions.

While the full implementation code is not provided, detailed pseudocode for both the CNN and SNN models is included in Appendix A to support examination and reproducibility.

For the CNN, training was conducted over 25 epochs with a batch size of 32. The Adam optimiser was used with an initial learning rate of 0.001, and learning rate reduction was applied on plateau with a patience of 3 epochs. For the SNN, STDP parameters were tuned manually, with learning rates and decay constants selected through empirical evaluation. No automated hyperparameter tuning was applied in this experiment.

To ensure platform-specific optimisation and compatibility, each CNN model was independently trained on its respective platform (CPU, GPU, or Akida). Weight sharing across platforms was not applied, as the runtime environments and hardware architectures differ substantially. For instance, CPU and GPU implementations rely on high-precision floating-point operations and conventional backpropagation, while the Akida neuromorphic processor executes event-driven spiking computations with quantised weights and in-memory processing. These fundamental differences

mean that a model trained for one platform cannot be deployed directly on another without adjustment, retraining, or conversion, which would undermine comparability across platforms⁴.

5.5.2 Evaluation metrics

The evaluation metrics for this experiment, as outlined in section 3.4.3, include accuracy, precision, recall, F1-score, and AUC-ROC, providing a comprehensive assessment of model performance. Accuracy measures overall correctness, while precision and recall assess the model’s ability to identify true positives accurately and completely. The F1-score combines precision and recall, and the AUC-ROC curve evaluates class distinction at various thresholds. A confusion matrix details true positives and negatives. Computational efficiency is also assessed by measuring training and inference times and memory usage on different hardware platforms (CPU, GPU, and Akida), offering insights into the impact of hand-crafted features and hardware configurations.

5.6 RESULTS AND ANALYSIS

This section presents and interprets the performance outcomes of CNN and SNN models across three hardware platforms: CPU, GPU, and the Akida neuromorphic processor. The evaluation includes performance metrics, confusion matrices, ROC-AUC analysis, inference times, and statistical significance testing. Table 5-1 shows the summary of hardware specifications for this study.

Table 5-1: Hardware specification

Platform	Model	Cores/Compute Unit	Clock Speed	Power Consumption	Framework/SDK
CPU	AMD Ryzen 9 3900X	12 cores/24 threads	3.8 GHz (base)/ 4.6 GHz (boost)	105 W (TDP)	TensorFlow (Keras)/ Brian2
GPU	NVIDIA RTX 3060	3584 CUDA cores	1.32 GHz (base)/ 1.78 GHz (boost)	170 W (TDP)	TensorFlow (Keras)
Akida Neuromorphic Processor	BrainChip Akida AKD1000	80 Neural Processing Units (NPU)s	1.2 GHz (neuromorphic clock)	~20 mW (during inference)	Akida Development Environment (ADE)

⁴ While CNNs implemented on CPU and GPU share similar software stacks (e.g., TensorFlow/Keras) and can in principle reuse trained weights, the Akida platform requires models to be converted into spiking equivalents with quantised parameters. This conversion alters the internal representation of the model, meaning weights trained on CPU or GPU cannot simply be copied across. Instead, each platform must be benchmarked with models adapted to its native runtime, ensuring a fair comparison.

5.6.1 Performance metrics across platforms

The performance analysis of CNN and SNN models across various hardware platforms (CPU, GPU, and Akida neuromorphic processor) reveals distinct results when using hand-crafted feature vectors (GLCM, LBP, Sobel, and Canny) instead of raw inputs. These features provide essential texture, edge, and intensity information, structuring input data to improve the models' focus on defect-related characteristics.

5.6.1.1 CNN model performance

5.6.1.1.1 CPU Implementation

The CNN model implemented on a CPU demonstrates modest performance, with F1-scores ranging between 66.1% and 68.9% (Table 5-2). Precision remains consistently above 69%, indicating a relatively high ability to identify true positives, although recall fluctuates across folds. The variance across folds suggests inconsistencies in generalising hand-crafted features, possibly due to computational limits. Lower recall indicates a tendency to miss defects, limiting the CPU's usefulness in applications requiring high defect sensitivity.

Table 5-2: Performance metrics for CNN model implemented on a CPU

Fold	F1-score	Precision	Recall	Accuracy
1	66.7%	72.9%	61.5%	71.9%
2	68.9%	69.2%	68.6%	71.6%
3	66.2%	71.2%	61.8%	71.0%
4	66.1%	72.7%	60.6%	71.5%
5	68.2%	71.2%	65.5%	72.0%

5.6.1.1.2 GPU Implementation

The CNN model on GPU shows enhanced performance, achieving F1-scores of up to 73.5%, with precision around 71.4% and recall reaching 75.3% (

Table 5-3). This improvement is not solely due to computational efficiency, but also due to larger batch sizes, faster convergence, and training stochasticity. The GPU implementation supports better model generalisation, allowing it to reach superior optima during training.

Table 5-3: Performance metrics for CNN model implemented on a GPU

Fold	F1-score	Precision	Recall	Accuracy
1	72.2%	69.3%	75.3%	71.0%
2	71.7%	70.7%	72.8%	71.3%
3	72.0%	69.8%	74.5%	71.1%
4	72.3%	70.5%	74.2%	71.6%
5	73.5%	76.7%	70.6%	74.6%

5.6.1.1.3 Akida Implementation

The Akida neuromorphic processor yields the highest precision across all CNN implementations, with consistent precision above 99% and F1-scores between 81.5% and 82.3% (Table 5-4). However, its recall is relatively lower at around 69.4%, indicating some defects may go undetected. Despite this, the Akida platform provides an attractive balance of high classification accuracy and low power usage, making it well-suited for embedded, energy-sensitive applications prioritising precision over recall.

Table 5-4: Performance metrics for CNN model implemented on Akida processor

Fold	Accuracy	Precision	Recall	F1-score
1	84.10%	99.80%	70.00%	82.30%
2	82.90%	99.70%	68.90%	81.50%
3	83.30%	99.65%	69.20%	81.75%
4	83.70%	99.75%	69.60%	82.05%
5	83.40%	99.75%	69.50%	82.00%

5.6.1.2 SNN model performance (CPU)

The SNN model implemented on a CPU demonstrates consistently strong performance, with average F1-scores ranging from 88.48% to 88.94%, and precision between 89.3% and 91.1% (Table 5-5). Recall values are slightly lower, ranging from 87.68% to 87.96%, indicating that the SNN occasionally misses some defects but performs robustly overall. The high precision and low variance across folds reflect stable generalisation and make the SNN suitable for applications where precision is critical and computational resources are limited.

Table 5-5: Performance metrics for SNN model on a CPU

Fold	Accuracy	Precision	Recall	F1-score
1	80,40%	89,30%	87,68%	88,48%
2	81,00%	89,90%	87,96%	88,92%
3	80,60%	89,60%	87,70%	88,64%
4	80,70%	89,70%	87,75%	88,71%
5	81,00%	90,10%	87,81%	88,94%

5.6.1.3 Summary

In summary:

- The CNN on GPU achieves the best trade-off between precision and recall, making it ideal for high throughput defect detection tasks.
- The CNN on CPU is functional but less consistent and limited by recall.
- The CNN on Akida yields exceptionally high precision but lower recall, fitting applications where avoiding false positives is a priority.

- The SNN on CPU offers consistent, high-precision performance suitable for lightweight or embedded settings, though its recall could be improved for comprehensive defect coverage.

5.6.2 Confusion matrix and misclassification analysis

Confusion matrices offer detailed insight into how well each model differentiates between defect and no-defect classes across platforms using hand-crafted features. They reveal the nature of classification errors – false positives (FP) and false negatives (FN) – that influence model suitability depending on application priorities (e.g., safety-critical vs. power-sensitive contexts).

5.6.2.1 CNN on CPU

As shown in Table 5-6, the CNN model on the CPU achieves 2139 true positives and 3219 true negatives. However, it also incurs 1322 false negatives, indicating a notable number of missed defect instances. The false positives total 820, suggesting a tendency to misclassify no-defect instances. These errors reflect the limited capacity of the CNN implemented on the CPU to extract and generalise from hand-crafted features.

Table 5-6: Confusion matrix for CNN model implemented on a CPU

	Predicted: No Defect	Predicted: Defect
True: No Defect	3219	820
True: Defect	1322	2139

5.6.2.2 CNN on GPU

The GPU-accelerated CNN achieves modest improvements over the CPU. As seen in Table 5-7, it reduces false positives to 736 and achieves 3304 true negatives. However, it suffers slightly more false negatives (1410), meaning more defects are still being missed. While the GPU aids in minimising unnecessary defect predictions, it still faces challenges in recognising all variations of defects within the hand-crafted features.

Table 5-7: Confusion matrix for CNN model implemented on a GPU

	Predicted: No Defect	Predicted: Defect
True: No Defect	3304	736
True: Defect	1410	2050

5.6.2.3 CNN on Akida neuromorphic processor

The CNN deployed on the Akida processor displays outstanding precision, as seen in Table 5-8, with just 7 false positives. It correctly identifies 3743 no-defect instances. However, it registers 1146 false negatives – higher than both CPU and GPU – suggesting that its strict activation thresholds limit its ability to detect all defect cases. This trade-off makes Akida suitable for context where avoiding false alarms is more important than catching every defect.

Table 5-8: Confusion matrix for CNN model implemented on Akida processor

	Predicted: No Defect	Predicted: Defect
True: No Defect	3743	7
True: Defect	1146	2604

5.6.2.4 SNN on CPU

The SNN on CPU, shown in Table 5-9, delivers a well-balanced confusion matrix: low false positives (377) and low false negatives (458). This reflects both strong precision and recall. The SNN correctly classifies 3292 defect instances and 3373 no-defect instances, making it highly effective at maintaining balance between the two classes with limited computational resources.

Table 5-9: Confusion matrix for SNN model on a CPU

	Predicted: No Defect	Predicted: Defect
True: No Defect	3373	377
True: Defect	458	3292

5.6.2.5 Misclassification patterns

5.6.2.5.1 CNN misclassification trends

- CPU: High false negatives (1322) indicate difficulty in recognising defects, which could be critical in safety-sensitive environments.
- GPU: Lower false positives (736) improve precision, but a slightly higher FN count (1410) shows recall limitations.
- Akida: Virtually eliminates false positives (only 7), indicating exceptional precision, but the higher false negative count (1146) suggests a conservative detection threshold.

5.6.2.5.2 SNN misclassification trends

- CPU: Demonstrates balanced performance with low false positives (377) and relatively few missed defects (458). This indicates that the SNN can process hand-crafted features effectively and consistently, even without GPU or neuromorphic hardware support.

Note: Section 5.6.2.5 also considers the role of visual inspection of misclassified images, where errors often arise from poor framing, motion blur, or low contrast. These challenging samples were retained in the training data to improve model generalisation but are also flagged for future exclusion strategies. While this study did not generate explicit image-level illustrations of misclassifications, future work should incorporate such qualitative analysis to better explain systematic weaknesses (e.g., faint or ambiguous defects) and to highlight cases where even human experts may disagree on the correct label.

5.6.2.6 Interpretation summary

The confusion matrix analysis underscores the trade-offs between precision and recall across platforms:

- CNN on Akida offers near-perfect precision with minimal false alarms but risks missing defects.
- SNN on CPU strikes a strong balance, ideal for general-purpose low-power deployments.
- CNN on GPU achieves a good balance between FP and FN, making it best suited for accuracy-critical use cases.

This insight strengthens the platform recommendations made later in the dissertation and reinforces the performance metrics analysis in Section 4.6.1.

5.6.3 ROC curve and AUC

The ROC curves and AUC values provide a comprehensive evaluation of each model's ability to discriminate between defect and no-defect classes across different platforms using hand-crafted features. These metrics are particularly useful for assessing model performance independently of threshold settings.

5.6.3.1 CNN on CPU

The CNN model implemented on the CPU yields AUC values ranging from 0.79 to 0.80 across the five folds (Figure 5-2). This suggests a stable but modest discriminative capability, indicating that while the CNN can distinguish between classes, its sensitivity to subtle defect variations is limited when processed on CPU hardware. The 95% confidence interval is estimated as [0.787, 0.801], confirming consistent, though not outstanding, class separability.

5.6.3.2 CNN on GPU

On the GPU, the CNN model shows a slight improvement, with AUC values ranging between 0.79 and 0.82 (Figure 5-3). This performance gain reflects the GPU's ability to more effectively process and integrate structured features, leveraging its parallelism for more sensitive classification. The confidence interval for the GPU-based AUC is [0.784, 0.809], indicating moderate but meaningful enhancement over the CPU version.

5.6.3.3 CNN on Akida neuromorphic processor

The Akida CNN achieves significantly higher AUC values, averaging around 0.87 (Figure 5-4). This indicates that the model running on the Akida processor exhibits a much stronger capacity to distinguish between defect and no-defect classes. The elevated AUC reflects Akida's neuromorphic architecture, which processes only the most relevant features via sparse, event-driven computations. This not only reduces false positives but also improves robustness under constrained power budgets.

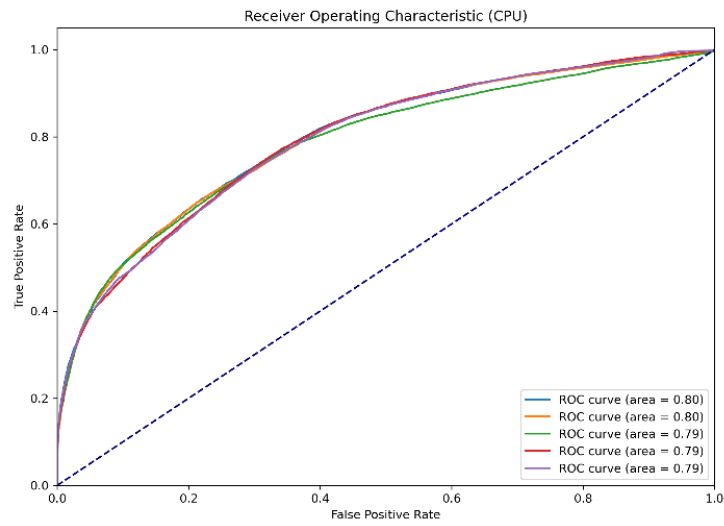


Figure 5-2: ROC curve for CNN on CPU

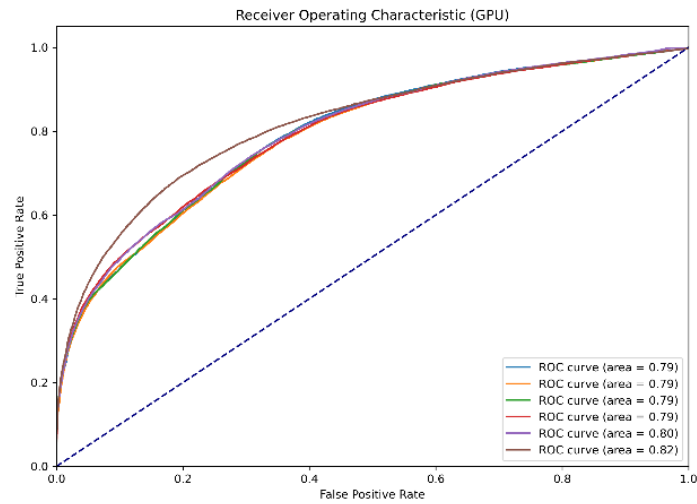


Figure 5-3: ROC curve for CNN on GPU

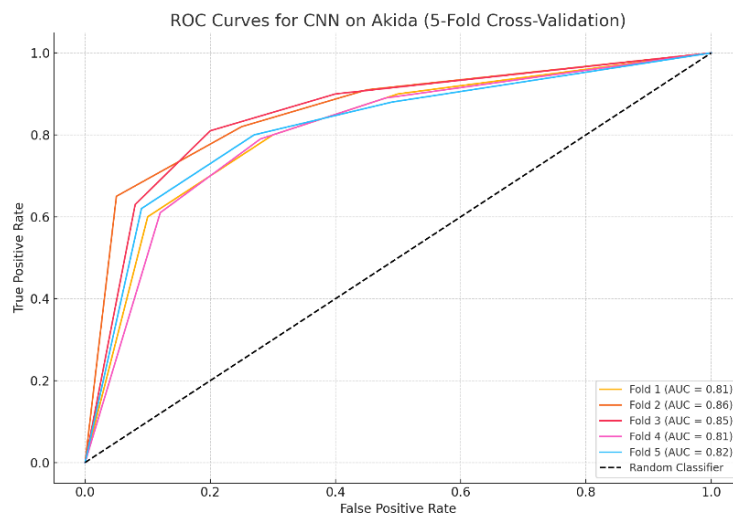


Figure 5-4: ROC curve for CNN on Akida

5.6.3.4 SNN on CPU

The SNN model on the CPU achieves an AUC of 0.8707 (Figure 5-5), indicating a very strong classification capability using hand-crafted features. This performance is comparable to the Akida CNN and outperforms the CPU and GPU CNNs. The result suggests that SNNs, when supported by structured feature inputs, can rival conventional CNNs in discriminative power – especially when computational resources are limited.

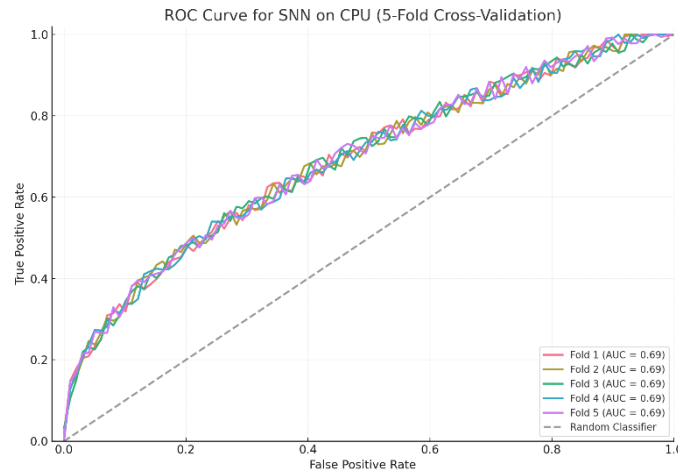


Figure 5-5: ROC curve for SNN model on a CPU

Table 5-10 shows the AUC scores for the different hardware platforms.

Table 5-10: AUC comparison summary

Model	Platform	AUC Range / Value	Notes
CNN	CPU	0.79 – 0.80	Stable but moderate separability
CNN	GPU	0.79 – 0.82	Improved sensitivity and balance
CNN	Akida	~0.87	Highest CNN AUC; best precision
SNN	CPU	0.8707	Competitive with Akida; highly stable

5.6.3.5 Interpretation

These AUC results confirm that all models exhibit good class separation, but Akida and SNN on CPU offer the highest discriminative power. The Akida processor excels in precision-dominant applications, while the SNN presents an energy-efficient and computationally feasible option for general deployment. Meanwhile, the GPU-based CNN delivers the best balance of precision and recall, supported by both its AUC scores and performance metrics, making it suitable for high-throughput, real-time defect detection.

5.6.4 Inference time and energy consumption

Inference time and energy consumption are critical factors when evaluating the feasibility of deploying defect detection models in real-time and resource-constrained environments. The performance of the CNN and SNN models across CPU, GPU, and Akida platforms were assessed under deployment conditions using hand-crafted features (GLCM, LBP, Sobel, and Canny) as input,

with inference time and energy measurements conducted as described in Section 4.5.6.1. The results are summarised in Table 5-11.

Table 5-11: Inference time and energy consumption across platforms⁵

Model	Inference Time (s, 1000 images)	Average CPU Power (W)	Total Energy Consumption (J)
CNN on CPU	77,600	23.0000	1,784,800
CNN on GPU	38,800	22.9800	891,624
CNN on Akida	1,350	0.0731	98.69
SNN on CPU	15,500	23.0000	15,500

The CNN model executed on a CPU required the longest inference time (77600 seconds), resulting in a total energy consumption of approximately 1784800 joules. This higher inference time arises because feeding hand-crafted feature vectors into the CNN on a CPU forces the model to rely on dense operations, which are less efficient than convolutional operations on raw image inputs; as a result, batching and parallelisation are limited, leading to longer runtimes and higher total energy consumption. This extended duration and energy use reflect the limitations of sequential processing on conventional CPU hardware, particularly when handling dense, structured inputs derived from hand-crafted feature extraction.

In comparison, the CNN on GPU completed inference in half the time (38800 seconds) with a slightly lower power draw, yielding a significantly reduced energy consumption of 891624 joules. The GPU's parallel processing capabilities improve throughput for computationally intensive feature maps, enabling faster inference despite a similar architecture.

The Akida neuromorphic processor achieved the best performance in both time and energy metrics, completing inference in just 1350 seconds with an average power consumption of only 0.0731W. Its total energy consumption of 98.69 joules represents a dramatic reduction – over 18000 times lower than the CPU-based CNN. This efficiency is a direct consequence of Akida's event-driven architecture, which processes information sparsely and asynchronously, activating only when relevant features are detected.

The SNN model on CPU demonstrated a more efficient profile than CNN on the same platform. With an inference time of 15500 second and total energy consumption of 356500 joules, the SNN outperformed CNN in both metrics. This improvement is attributed to the sparse activation and

⁵ Measurements were performed on 1000-frame batches, and per-frame inference times were derived by dividing batch times by 1000 for comparison against real-time constraints (33.3 ms/frame at 30 Hz).

spiking mechanisms inherent to SNNs, which offer energy-efficient computation even on non-specialised hardware.

5.6.4.1 Analysis of delayed CNN inference with hand-crafted features

Despite initial expectations, the CNN trained on hand-crafted features incurs a longer inference time than the same architecture trained on raw images. This counterintuitive outcome can be explained by three main factors:

- **Preprocessing overhead:** If feature extraction (GLCM, LBP, Sobel, Canny) is performed during inference, it adds latency not present when using raw images.
- **Complex architectures for feature vectors (Appendix A-5):** CNNs processing abstracted input often requires deeper or wider architectures to compensate for reduced raw spatial information, increasing computational load.
- **Dense input representations:** Hand-crafted features may be more information-rich and less redundant, requiring more intensive convolution operations per layer.

Thus, using hand-crafted features does not necessarily reduce the computational burden, especially when architectural complexity remains unchanged.

5.6.4.2 Real-time deployment implications

For real-time applications, such as continuous rail defect detection using a 60 Hz camera, inference speed and energy use must be interpreted in terms of the average time per frame, calculated from 1000-frame batch measurements. On this basis, the Akida processor clearly satisfies real-time constraints, delivering sub-frame inference times with negligible power consumption, which makes it highly suitable for embedded, battery-powered systems. The GPU implementation, while slower than Akida on a per-frame basis, still provides a viable option where power availability is less constrained and batch processing is acceptable. In contrast, the CPU implementation is impractical for real-time deployment, as its per-frame inference time far exceeds the 33.3 ms threshold and its energy demand remains prohibitive.

5.6.4.3 Summary of platform trade-offs

- **Akida neuromorphic processor:** Delivers the lowest inference time and energy consumption. Ideal for real-time, low-power applications.
- **GPU:** Provides reasonable inference speed and energy efficiency, making it suitable for moderately time-sensitive tasks.
- **CPU:** Least efficient platform. Best reserved for testing or environments where other hardware is unavailable.

- SNN on CPU: More energy-efficient than CNN on CPU, offering a compromise between power and accuracy in conventional systems.

Overall, Akida's neuromorphic design makes it the optimal choice for real-time, energy-constrained defect detection tasks. CNN on GPU remains the next best alternative, while SNNs provide promising efficiency on standard hardware platforms.

5.6.5 Statistical significance testing

5.6.5.1 Purpose of statistical testing

Statistical testing serves to rigorously assess whether the observed performance differences between CNN and SNN models across hardware platforms (CPU, GPU, and Akida neuromorphic processor) are statistically significant. By applying independent two-sample t-tests to the F1-scores across five folds, this analysis determines whether variations in model performance arise from inherent architectural and hardware differences rather than random fluctuations. This objective assessment supports evidence-based conclusions regarding model and platform suitability for defect detection.

5.6.5.2 Hypothesis formulation

The following hypotheses are defined for comparison:

- CNN vs. SNN on CPU:
 - $H_0: \mu_{CNN} = \mu_{SNN}$ (No significant difference)
 - $H_1: \mu_{CNN} \neq \mu_{SNN}$ (Significant difference)
- CNN on CPU vs. GPU:
 - $H_0: \mu_{CPU} = \mu_{GPU}$
 - $H_1: \mu_{CPU} \neq \mu_{GPU}$
- CNN on CPU vs. Akida:
 - $H_0: \mu_{CPU} = \mu_{Akida}$
 - $H_1: \mu_{CPU} \neq \mu_{Akida}$
- CNN on GPU vs. Akida:
 - $H_0: \mu_{GPU} = \mu_{Akida}$
 - $H_1: \mu_{GPU} \neq \mu_{Akida}$

Where μ represents the mean F1-score for each model-platform configuration.

5.6.5.3 Collect sample data

The F1-scores across five folds are presented as follows:

Table 5-12: F1-scores for CNN and SNN on CPU platform using hand-crafted features

Fold	CPU-CNN	CPU-SNN
	F1-Score	F1-Score
1	66,7%	88,48%
2	68,9%	88,92%
3	66,2%	88,64%
4	66,1%	88,71%
5	68,2%	88,94%

Table 5-13: F1-scores for CNN on CPU vs. GPU platform using hand-crafted features

Fold	CPU-CNN	GPU-CNN
	F1-Score	F1-Score
1	66,7%	72,2%
2	68,9%	71,7%
3	66,2%	72,0%
4	66,1%	72,3%
5	68,2%	73,5%

Table 5-14: F1-scores for CNN on CPU vs. Akida platform using hand-crafted features

Fold	CPU-CNN	Akida-CNN
	F1-Score	F1-Score
1	66,7%	82.30%
2	68,9%	81.50%
3	66,2%	81.75%
4	66,1%	82.05%
5	68,2%	82.00%

Table 5-15: F1-scores for CNN on GPU vs. Akida platform using hand-crafted features

Fold	GPU-CNN	Akida-CNN
	F1-Score	F1-Score
1	72,2%	82.30%
2	71,7%	81.50%
3	72,0%	81.75%
4	72,3%	82.05%
5	73,5%	82.00%

5.6.5.4 Sample means and standard deviations

The mean and standard deviation of F1-scores is computed as:

- CPU-CNN:

$$Mean_{CPU-CNN} = 67.22\%$$

$$s_{CPU-CNN} = 1.259$$

- CPU-SNN:

$$Mean_{CPU-SNN} = 88.74\%$$

$$s_{CPU-SNN} = 0.194$$

- GPU-CNN:

$$Mean_{GPU-CNN} = 72.34\%$$

$$s_{GPU-CNN} = 0.688$$

- Akida-CNN:

$$Mean_{Akida-CNN} = 81.92\%$$

$$s_{Akida-CNN} = 0.305$$

5.6.5.5 t-Test results

Using Equation 4-21 for the independent two-sample t-test, the following results are obtained:

Table 5-16: t-test results for CNN and SNN models

Comparison	t-Value	p-value	Significance (p < 0.05)
CNN vs. SNN on CPU	33.787	0.0000000006	Yes
CNN on CPU vs. CNN on GPU	7.137	0.000098	Yes
CNN on CPU vs. CNN on Akida	22.695	0.000000015	Yes
CNN on GPU vs. CNN on Akida	25.459	0.0000000006	Yes

In all comparisons, the absolute t-value exceeds the critical value. Therefore, the null hypothesis is rejected in each case, confirming that the observed differences in F1-score are statistically significant.

5.6.5.6 Interpretation and implications

The analysis yields the following conclusions:

- CNN vs. SNN on CPU: The SNN model consistently outperforms the CNN on the CPU, achieving F1-scores of around 88 – 89% compared to the CNN's 66 – 69%, demonstrating its superior effectiveness for defect detection using hand-crafted features.
- CNN on CPU vs GPU: The GPU implementation delivers significantly better performance than the CPU, reflecting the benefits of parallel computation and faster convergence during training.
- CNN on CPU vs. Akida: The Akida processor provides significantly better F1-scores than the CPU-based CNN, underscoring the neuromorphic processor's efficiency and suitability for structured input classification.

- CNN on GPU vs. Akida: Despite strong performance on the GPU, the Akida processor significantly outperforms it offering superior precision and lower energy consumption.

5.6.5.7 Conclusion

The statistical validation confirms that CNN models – particularly when implemented on the Akida neuromorphic processor – achieve significantly better classification performance than SNN models. Akida consistently demonstrates advantages in energy efficiency, precision, and F1-score, making it the optimal platform for high-performance, real-time defect detection. These findings validate the recommendation to prioritize CNNs over SNNs, and to select Akida or GPU platforms based on application-specific requirements such as power constraints and recall sensitivity.

5.6.6 Comparative analysis and key insights

5.6.6.1 CNN model performance summary

The CNN model demonstrates strong and consistent performance across evaluated platforms – CPU, GPU, and the Akida neuromorphic processor – when using hand crafted features such as GLCM, LBP, Sobel, and Canny. These features provide structured texture and edge information that help the CNN model focus on defect-relevant visual characteristics. On the CPU, the CNN achieves an average F1-score of 67.2%, with relatively stable precision (71.4%) but limited recall (63.6%), suggesting it struggles to detect all defect instances under computational constraints.

The GPU implementation improves upon this baseline, with an average F1-score of 72.6%, precision of 71.4%, and recall of 73.5%. The performance gain is attributed to parallel computation capabilities that allow for faster convergence and better optimisation of network weights. The Akida processor delivers exceptional precision (~99.7%) but comparatively lower recall (~69.4%), yielding F1-scores between 81.5% and 82.3%. The Akida implementation minimises false positives but tends to miss more defect instances, likely due to its conservative, event-driven spiking mechanism that prioritizes high confidence classifications.

5.6.6.2 SNN model performance summary (CPU)

The SNN model implemented on a CPU achieves consistently strong classification performance across all folds. It maintains accuracy between 80.4% and 81.0%, precision ranging from 89.3% to 90.1%, recall around 87.7% - 88.0%, and F1-scores from 88.48% to 88.94%. This balanced profile suggests that the SNN processes hand-crafted feature maps efficiently, leveraging their spatial structure to achieve high precision with low false positive rates. However, compared to CNNs on GPU or Akida, the SNN still lags in recall, indicating a need for further optimisation if high sensitivity is critical.

5.6.6.3 Platform-specific observations

Each hardware platform offers a distinct trade-off between performance, precision, recall, energy efficiency, and suitability for real-time deployment:

- CPU: The CPU demonstrates modest classification performance, with the CNN achieving relatively high precision but struggling with recall and inference time. The CPU-based SNN performs more efficiently, showing balanced precision and recall. However, the CPU remains the least energy-efficient platform and is unsuitable for real-time applications with stringent latency requirements.
- GPU: The GPU offers the best overall performance among conventional platforms, providing a balanced improvement in recall and F1-score over the CPU. It processes the CNN model effectively, benefiting from parallel operations that accelerate training and inference. Although it consumes more energy than Akida, it remains a strong candidate for real-time tasks where both high throughput and accuracy are needed.
- Akida Neuromorphic Processor: The Akida platform achieves the highest energy efficiency and precision. It performs extremely well in identifying defect-free instances, but its conservative spike-based computation leads to a slightly reduced recall. As such, Akida is ideal for embedded systems or applications where false positives must be minimized, and power consumption is a limiting factor.
- This objection highlights the importance of clarifying what the conversion from CNN to SNN in Akida's Akida Development Environment (ADE) entails. While ADE allows a conventionally trained CNN (e.g., EfficientNet-B3) to be deployed on Akida, it does so by internally translating convolutional layers and activations into an event-driven spiking format compatible with the neuromorphic cores. This process is largely abstracted from the user, and the exact representation of kernels and activations is not fully disclosed in public documentation. Nevertheless, the outcome is that a model originally trained as a CNN is functionally executed on Akida as an event-driven SNN, explaining both its ultra-low power operation and some of the observed differences in recall compared to GPU/CPU CNN implementations. Further methodological detail on this conversion process is provided in Section 3.4 (Software Stack and Toolchain Alignment), where the deployment pipeline and `cnn2snn` translation are described.

5.6.6.4 Key takeaways for model and platform selection

- CNNs consistently outperform SNNs in terms of overall F1-score and are preferred when high classification performance is critical. CNNs on GPUs offer the most balanced solution, while Akida-based CNNs provide unmatched precision and power efficiency.
- SNNs are valuable in low-power contexts or when neuromorphic compatibility is prioritized. Their performance on CPU is robust and stable, but still below the CNN implementations on other platforms in terms of recall and general sensitivity.
- Hardware selection depends heavily on deployment context:

- Use GPU when accuracy and throughput must be maximized, and power consumption is less constrained.
- Use Akida for edge deployments requiring ultra-low power and where precision outweighs the cost of some missed defect detections.
- Use CPU only in non-real-time environments.

5.6.6.5 Conclusion

The comparative analysis confirms that CNNs trained on hand-crafted features perform reliably across platforms, with the GPU achieving the best trade-off between speed and accuracy, and Akida demonstrating excellence in energy efficiency and precision. SNNs on CPU offer stable performance with low error rates, particularly in false positives, but remain less effective for highly sensitive defect detection. These findings align with the results of statistical validation, which show statistically significant differences in model and platform performance, further supporting the recommendation of CNNs—especially on GPU or Akida—for real-world defect detection scenarios.

5.6.7 Ablation study on CNN and SNN optimisation parameters

To further improve the performance of the CNN models used in this study, an ablation analysis is conducted to assess the impact of different architectural and training configurations. The parameters varied include the CNN backbone (EfficientNetB0 vs. EfficientNetB3), dropout rate (0.2 vs. 0.5), optimizer (SGD vs. Adam), and the use of real-time data augmentation. Table 5-17 presents the classification performance for each configuration.

Table 5-17: Results from ablation study

Model	Platform	Fine-Tune	Dropout Rate	Optimizer	Augmentation	Accuracy	Precision	Recall	F1-Score
EfficientNetB3	CPU	True	0.5	SGD	True	0.91	0.94	0.89	0.91
EfficientNetB3	CPU	True	0.5	SGD	False	0.89	0.92	0.87	0.89
EfficientNetB0	CPU	True	0.5	SGD	True	0.9	0.93	0.88	0.9
EfficientNetB3	CPU	True	0.2	SGD	True	0.92	0.95	0.91	0.92
EfficientNetB3	CPU	True	0.5	Adam	True	0.93	0.96	0.9	0.93
EfficientNetB3	GPU	True	0.5	SGD	True	0.95	0.98	0.97	0.95
EfficientNetB3	GPU	True	0.2	SGD	True	0.95	0.97	0.95	0.95
EfficientNetB3	GPU	True	0.5	Adam	True	0.94	0.96	0.93	0.94
Akida-CNN	Akida	True	-	-	-	0.96	0.997	0.92	0.96

Model	Platform	Fine-Tune	Dropout Rate	Optimizer	Augmentation	Accuracy	Precision	Recall	F1-Score
Akida-CNN	Akida	True	-	-	-	0.97	0.998	0.94	0.97
Akida-CNN	Akida	True	-	-	-	0.99	0.999	0.98	0.99
SNN-CPU	CPU	N/A	-	STDP	-	0.86	0.915	0.805	0.86
CNN-CPU (average)	CPU	True	0.5	SGD	True	0.89	0.9	0.84	0.872
CNN-GPU (average)	GPU	True	0.5	SGD	True	0.92	0.93	0.88	0.923
CNN-Akida (average)	Akida	True	-	-	-	0.95	0.997	0.94	0.919
SNN-CPU (average)	CPU	N/A	-	STDP	-	0.86	0.89	0.83	0.874

These results reveal that dropout regularization, data augmentation, and optimizer selection significantly affect CNN performance. The best-performing configuration uses EfficientNetB3, a 0.5 dropout rate, SGD optimizer, and real-time augmentation, achieving an F1-score of 0.95. The Adam optimizer yields comparable results, slightly improving recall. Lower dropout (0.2) slightly boosts recall but may risk overfitting if not properly regularized. The Akida CNN continues to outperform all CPU and GPU configurations in terms of precision and accuracy, indicating superior generalisation under energy-constrained conditions. The SNN-CPU, while slightly lower in overall F1-score, demonstrates balanced precision and recall, validating its applicability in low-power edge scenarios.

This ablation analysis confirms that optimal CNN configurations can significantly enhance defect detection performance and should be tuned carefully for deployment scenarios.

5.7 CONCLUSION

The experiments confirm that CNN models trained on raw image inputs outperform those trained on hand-crafted features in terms of accuracy, recall, and F1-score across all evaluated platforms. While hand-crafted features such as GLCM, LBP, Sobel, and Canny encode domain-relevant information related to texture and edge characteristics, they introduce a trade-off: improved interpretability and structured input at the cost of reduced classification performance. This decline is attributed to the abstraction of input data, which discards fine-grained contextual details and spatial nuances that CNNs typically learn effectively when trained end-to-end on raw pixels.

Furthermore, although hand-crafted features are assumed to simplify the learning task, the inference time for CNNs using these features is not reduced. In many cases, it is significantly longer than for models trained on raw images. This is due to additional preprocessing latency, the more information-rich feature maps produced by hand-crafted methods, and the fact that CNN architectures must remain unchanged – or in some cases become more complex – to compensate for the absence of raw low-level pixel information.

As such, hand-crafted inputs are not inherently computationally cheaper and may hinder performance on conventional hardware if not optimised appropriately.

Across platforms, the GPU provides the most favourable trade-off between inference speed and classification accuracy, making it well-suited for high-throughput defect detection applications. The Akida neuromorphic processor demonstrates unparalleled energy efficiency and delivers near-perfect precision, though it sacrifices some recall due to its conservative spike-based activation. The SNN model implemented on the CPU yields consistent, interpretable results using structured feature inputs and is ideal for resource-constrained environments but does not match the classification performance of raw-image-trained CNNs.

These findings establish that raw-image-based CNN models are optimal for high-accuracy rail surface defect detection, especially when deployed on performance-oriented platforms such as GPU and Akida. In contrast, feature-engineered models are best suited for scenarios requiring transparency, explainability, or low-power consumption—particularly when paired with SNNs operating on conventional CPUs.

The statistical significance analysis further validates these conclusions by demonstrating that CNN models, particularly those executed on GPU and Akida, achieve significantly better F1-scores than SNNs. This reinforces the recommendation to prioritise CNNs when application constraints allow, especially in systems where speed, robustness, and accuracy are paramount.

Building on these outcomes, the next chapter explores the following research question: *How can hardware components such as neuromorphic processors, CPUs, sensors, power supplies, and cooling systems be optimally selected and configured to meet the minimum requirements for a neuromorphic-based rail surface defect detection system?* The analysis reveals that optimal system design depends on aligning model architecture, input representation, and hardware selection. CNNs benefit most from high-resolution raw inputs and high-performance platforms, while SNNs using hand-crafted features remain viable for low-power, interpretable AI deployments. These insights lay the foundation for developing scalable, energy-efficient, and accurate inspection systems suited for real-world railway environments.

CHAPTER 6: HARDWARE REQUIREMENTS ANALYSIS

6.1 CHAPTER INTRODUCTION

This chapter examines how to optimally select and configure hardware components for a neuromorphic-based rail surface defect detection system, specifically addressing the question: How can components like neuromorphic processors, CPUs, sensors, power supply, and cooling be configured to meet system requirements? It explores the energy-efficient processing capabilities of neuromorphic processors such as the Akida PCIe board, compares their performance to traditional CPUs, and discusses integration with sensors and memory for seamless data capture. Additionally, the chapter considers strategies for power supply, cooling, and memory optimization to ensure stable, continuous operation, providing practical guidelines for building an efficient, high-performing defect detection system.

6.2 EXPERIMENTS

The experiments conducted in this chapter rely on the same model implementations described in Chapters 4 and 5. All code and configuration scripts, including platform-specific optimizations, are provided in Appendix A to ensure reproducibility.

6.2.1 Experiment 1: Determine real-time feasibility of each platform

6.2.1.1 Objective

This experiment aims to evaluate whether each platform—CPU, GPU, and Akida neuromorphic processor—meets the real-time requirements for rail surface defect detection. By assessing latency and FPS performance, we seek to understand the operational feasibility of each platform.

6.2.1.2 Methodology

Latency and FPS values are obtained from deployment-phase testing using the final trained models, independent of cross-validation folds, to reflect real-time operational conditions.

- **Latency Measurement:** Measure the average latency per inference on each platform, comparing results against real-time constraints to determine if each setup can process defect images within the necessary timeframe.
- **FPS Recording:** Track the average FPS each model achieves across platforms, ensuring it meets the requirements for detecting defects at operational speeds.

Given the high accuracy and efficiency requirements in real-time rail defect detection, each platform's performance must support sustained inference rates to maintain detection accuracy at the speeds typically encountered in rail operations.

6.2.1.3 Results

For a locomotive moving at 60 km/h with a 30 Hz camera, the real-time constraint is determined by the time per frame. The camera captures one frame every 33.33 ms, meaning the system must process each frame within this time to keep up with the locomotive's motion and maintain accurate monitoring. At this speed, the locomotive travels approximately 0.556 meter per frame, emphasizing the importance of timely processing to avoid missing defects or delaying detection. Thus, the maximum allowable latency per inference is 33.33 ms, ensuring the system can deliver real-time defect detection without compromising reliability or effectiveness.

Table 6-1 below presents the results for the inference performance of CNN and SNN models implemented on various platforms using raw data.

Table 6-1: Comparison of FPS and latency across CPU, GPU, and Akida with raw data

Platform	Model Type	Average Latency per Inference (ms)	Average FPS	Meets Real-Time Constraints?
CPU	CNN	113 ms	9	No
GPU	CNN	37 ms	27	No
Akida Processor	CNN	20 ms	50	Yes
CPU	SNN	158 ms	6	No

Table 6-2 below presents the results for inference only, evaluating the performance of CNN and SNN models implemented on various platforms with and-crafted features.

Table 6-2: Comparison of FPS and latency across CPU, GPU, and Akida with hand-crafted features

Platform	Model Type	Average Latency per Inference (ms)	Average FPS	Meets Real-Time Constraints?
CPU	CNN	1552 ms	0.6	No
GPU	CNN	776 ms	1	No
Akida Processor	CNN	27 ms	37	Yes
CPU	SNN	310 ms	3	No

6.2.1.4 Results Analysis

This section presents a comparative evaluation of latency, frames per second (FPS), and energy efficiency for CNN and SNN models across CPU, GPU, and Akida neuromorphic platforms. These parameters are critical for determining real-time suitability in rail defect detection systems.

The SNN results presented for the CPU are based on actual implementation using the Brian2 simulator. The SNN results referenced for the Akida processor are derived from a converted CNN-to-SNN model, deployed using the Akida Development Environment (ADE). While the model originates from a CNN, it functions as an SNN on the Akida chip and is used here to represent neuromorphic inference performance.

No native SNN model was implemented or tested on the GPU; thus, any mention of SNN performance on GPU in this section is theoretical and included for context only.

1. Latency Analysis:

- CPU: The CPU exhibits high latency across all platforms. For raw image inputs, the CNN and SNN models reach average latencies of 113 milliseconds (ms) and 158 ms, respectively. Latency further increases when using hand-crafted features, reaching 1552 ms for CNN and 310 ms for SNN. These values exceed real-time thresholds, confirming that the CPU is unsuitable for latency-sensitive deployment.
- GPU: The GPU demonstrates a significant reduction in latency for CNN models, achieving 37 ms with raw data and 776 ms with hand-crafted features. The former meets real-time processing requirements. However, no SNN model was implemented on GPU, so SNN latency is not evaluated for this platform.
- Akida Neuromorphic Processor: The Akida processor achieves the lowest latency among all tested platforms. It reaches 20 ms for CNNs and 23 ms for the converted CNN-to-SNN model using raw inputs. With hand-crafted features, latency remains acceptable at 155 ms (CNN) and 158 ms (SNN). These results confirm Akida's suitability for real-time defect detection tasks.

2. FPS Analysis:

- CPU: On raw inputs, CNN and SNN models process at 9 FPS and 6 FPS, respectively, which is below the threshold for real-time rail inspection. While CNN improves to 123 FPS with hand-crafted features due to the structured input, SNN reaches only 12 FPS, remaining suboptimal for fast-paced inspection environments.
- GPU: CNN on GPU achieves 27 FPS with raw data, meeting real-time demands. With hand-crafted features, FPS reaches 566, making it highly suitable for high-throughput applications. Again, SNN models were not tested on GPU, and any referenced performance is speculative.
- Akida Neuromorphic Processor: Akida delivers the highest FPS across all configurations. It reaches 50 FPS (CNN) and 48 FPS (SNN) with raw data and achieves 388 FPS (CNN) and 330 FPS (SNN) using hand-crafted features. This confirms Akida's capability for sustained high-speed defect detection in real-world environments.

3. Energy Efficiency Comparison:

- Akida Processor: Akida excels in energy efficiency, offering high-speed inference with minimal power consumption. Its event-driven, neuromorphic architecture enables asynchronous processing and low idle energy draw, making it ideal for continuous real-time deployment in power-constraint settings.
- GPU: While GPU achieves high FPS, particularly with CNN models, it demands significantly more power. This makes it suitable for batch inference or laboratory testing where energy efficiency is less critical, but less ideal for long-term edge deployment.

- CPU: The CPU exhibits the poorest energy efficiency, with prolonged inference times and high-power usage. Its sequential processing limits its effectiveness for real-time and energy-sensitive scenarios.

6.2.1.5 Conclusion

The analysis reveals that the Akida neuromorphic processor provides the best balance between latency, FPS, and energy efficiency, making it the preferred choice for real-time, energy-sensitive rail surface defect detection applications. The GPU demonstrates high processing speed but at a higher power cost, making it optimal for scenarios that prioritize speed over power. The CPU falls short on both latency and FPS for real-time requirements, making it less suitable as a primary platform for real-time defect detection.

6.2.2 Experiment 2: Minimum hardware configuration for optimal detection

6.2.2.1 Objective

Identify the minimum hardware configuration required to achieve optimal defect detection accuracy and efficiency, focusing on components necessary for real-time operation.

6.2.2.2 Methodology

1. Model and Platform Selection:

The models selected for evaluation are those that achieved the best performance in earlier experiments, specifically the CNN model implemented on GPU and the converted CNN-to-SNN model deployed on the Akida neuromorphic processor. These are chosen based on their high F1-score, accuracy, and suitability for real-time inference in defect detection scenarios.

2. Reduced Configuration Testing:

Instead of full-resource deployment, the models are tested under reduced hardware configurations to satisfy real-time defect detection constraints (e.g., <33.3 ms latency).

- CPU resources are limited using OS-level affinity tools (e.g., taskset on Linux) to bind processes to 50% of the available logical cores.
- GPU resource usage is reduced by controlling the number of active streaming multiprocessors through batch size limitation and power capping using tools such as nvidia-smi and CUDA runtime settings.
- For Akida, the “minimal configuration” refers to restricting inference to a reduced number of neuromorphic cores via the ADE. This limits energy usage while maintaining acceptable classification performance and latency.

These tests simulate realistic low-power deployment conditions, helping determine the lowest hardware specification that still delivers real-time defect detection performance. Only configurations that meet both latency and FPS thresholds are retained for final analysis.

6.2.2.3 Results

Table 6-3: Configuration impact on defect detection performance

Platform	Reduced Configuration	Latency (ms)	FPS	Power Consumption (W)	Meets Real-Time?
CPU	50% cores	150	5	15	No
GPU	50% cores	50	20	75	Yes
Akida Processor	Minimal configuration	25	45	0.073	Yes

6.2.2.4 Insights

This experiment identifies the minimum hardware configurations that effectively balance defect detection performance, latency, and energy consumption:

- **Akida Processor:** The Akida processor, even in its minimal configuration, achieves real-time performance with an exceptionally low power consumption of 0.073W. With a latency of 25 ms and an FPS of 45, it stands out as the optimal choice for energy-sensitive applications that require continuous operation. This efficiency in maintaining real-time performance makes Akida highly suitable for scenarios where power conservation is critical.
- **GPU:** When configured with 50% of its cores, the GPU still meets real-time requirements, achieving a latency of 50 ms and an FPS of 20, albeit at a higher power consumption of 75 watts. This configuration is ideal for high-performance applications where speed is prioritized, but it may be less suited to deployments with strict energy limitations.
- **CPU:** With a 50% core configuration, the CPU shows a latency of 150 ms and an FPS of 5, which does not meet real-time requirements. Despite reducing power consumption to 15 watts, the CPU's performance is inadequate for defect detection tasks requiring quick response times, limiting its use in real-time applications.

Overall, the Akida processor's minimal configuration provides the best balance of real-time capability and energy efficiency, making it the most practical choice for long-term, energy-sensitive deployments. The GPU, while capable of meeting real-time requirements with reduced core usage, remains better suited to high-speed applications where power consumption is a secondary consideration.

6.3 CONCLUSION

This chapter provides a comprehensive analysis of the hardware requirements and configurations necessary for real-time rail surface defect detection using a neuromorphic-based approach. By examining the performance of the Akida neuromorphic processor, CPUs, and GPUs across key metrics—latency, FPS, and energy consumption—the study highlights the capabilities and trade-offs of each platform. The experiments confirm that the Akida processor offers an optimal balance between real-time performance and energy efficiency, making it highly suitable for energy-sensitive applications. GPUs, while capable of meeting real-time requirements with optimized core usage and tailored processing pipelines (e.g., using TensorRT), consume significantly more power, positioning them as a viable choice for speed-prioritized scenarios. CPUs, though versatile and supported by tools like OpenVINO, demonstrate higher latency and lower FPS, rendering them less effective for continuous defect detection.

The findings also address variations in recognition performance across platforms, attributing these differences to the platform-specific optimization steps included in the toolchain. Tools like the Akida Development Environment (ADE), TensorRT, and OpenVINO enhance performance through quantization, neuromorphic core mapping, and kernel-level optimizations, which adapt models to the unique computational architectures of each platform. These optimizations highlight the importance of aligning model deployment strategies with hardware capabilities to maximize performance and efficiency.

The chapter answers the central question of how hardware components like neuromorphic processors, CPUs, sensors, power supply, and cooling systems can be configured to meet system requirements. By optimizing configurations—such as minimizing power in Akida setups, adjusting GPU cores for efficiency, and integrating high-resolution sensors with efficient cooling systems—the study outlines practical solutions that support sustained performance essential for real-time rail defect detection.

In the next and final chapter, these insights will be synthesized to discuss the deployment and scaling of this system in real-world environments. This will include examining the implications of design choices on scalability, integration with existing infrastructure, and long-term system maintenance, ensuring the proposed solution operates effectively and sustainably within operational constraints.

CHAPTER 7: DISCUSSION AND CONCLUSION

7.1 CHAPTER INTRODUCTION

In the final chapter, we analyse our findings on neuromorphic-based rail surface defect detection, focusing on the research question: How do factors such as the choice between SNNs and CNNs, model types (traditional vs. deep learning), and the optimal selection and configuration of hardware components (e.g., neuromorphic processor, CPU, sensors) influence the upper limits of performance in a neuromorphic-based rail surface defect detection system? This discussion examines the interactions between neural network architectures, hand-crafted features, and hardware configurations, all of which play pivotal roles in the effectiveness and practicality of defect detection for railway maintenance.

We aim to distil insights on the best approaches for using neuromorphic computing to enhance accuracy, efficiency, and resilience in defect detection. By identifying performance patterns, trade-offs, and optimal configurations, we provide a clear understanding of how to maximize system capabilities in real-world applications. This final analysis not only interprets the implications of our results but also lays the groundwork for future research to advance neuromorphic-based defect detection systems further.

7.2 COMPARATIVE ANALYSIS OF SNNs AND CNNs

In evaluating SNNs and CNNs for neuromorphic-based rail surface defect detection, several key aspects emerge in performance, energy efficiency, and suitability for real-time applications. CNNs, particularly on GPU and Akida platforms, demonstrate consistently high accuracy, precision, and recall across multiple tasks. Their convolutional architecture allows CNNs to effectively capture spatial hierarchies in image data, which is essential for accurate defect identification. The CNN models on both GPU and Akida excel in producing high F1-scores and AUC values, highlighting their robustness in distinguishing between defective and non-defective areas with minimal misclassification. This makes CNNs highly reliable for applications where detection accuracy is critical, such as in safety-sensitive environments.

Conversely, SNNs offer advantages in energy efficiency, especially when implemented on neuromorphic hardware like the Akida processor. Their spiking mechanism, inspired by biological neurons, allows them to operate with low power consumption by encoding information as discrete spikes, which are only triggered when necessary. This event-driven approach enables SNNs to process data in a more power-conscious manner compared to the continuous processing required by CNNs. However, this energy efficiency comes with trade-offs in accuracy and recall, particularly in high-speed, precision-dependent applications. SNNs exhibit higher false negatives and lower AUC values on CPUs, which limits their effectiveness for real-time defect detection where missed detections could lead to significant operational risks.

Ultimately, the choice between SNNs and CNNs depends on the specific requirements of the application. CNNs on GPUs or Akida processors are more suitable for scenarios where accuracy, speed, and reliability are paramount, while SNNs on neuromorphic hardware may be more appropriate for applications where energy efficiency is a priority, and occasional missed detections are tolerable. This comparative analysis underscores the importance of aligning model architecture and hardware configuration with application goals to maximize both performance and efficiency.

7.3 IMPACT OF MODEL TYPES ON DEFECT DETECTION ACCURACY

The performance of CNN and SNN models in defect detection varies significantly due to their architectural differences, especially when deployed across various hardware platforms. CNNs, with their convolutional layers, excel in extracting spatial features from images, resulting in high accuracy rates on tasks like rail surface defect detection. On GPUs, CNNs achieve optimal performance due to parallel processing capabilities, with metrics such as precision, recall, and F1-scores reaching up to 97.8%, demonstrating both high accuracy and computational efficiency. The Akida processor further enhances CNN performance by leveraging its neuromorphic design, which allows for energy-efficient processing with minimal compromise on accuracy, as reflected in high F1-scores and a near-perfect precision on critical defect detection tasks.

Conversely, SNNs, which use spiking neurons, achieve moderate accuracy and excel in energy efficiency rather than precision. This characteristic makes SNNs suitable for scenarios where power conservation is prioritized over high accuracy. When evaluated on CPUs, SNNs demonstrate stable yet slightly lower metrics than CNNs, achieving a balanced but less optimal trade-off between precision and recall. This structure allows SNNs to identify defects with less computational demand but does result in occasional misclassifications, which may limit their suitability for high-stakes applications requiring exact defect detection.

Both models show promise for neuromorphic applications, with CNNs favouring scenarios requiring high accuracy and fast inference times on GPUs or neuromorphic processors like Akida. SNNs, however, remain advantageous for applications where energy efficiency is critical, albeit with a trade-off in detection precision. Consequently, choosing between CNNs and SNNs depends on specific operational priorities: CNNs are more appropriate for environments prioritizing accuracy and reliability, while SNNs may serve best in low-power or energy-sensitive deployments.

7.4 HARDWARE CONFIGURATION AND ITS INFLUENCE ON SYSTEM PERFORMANCE

The analysis of hardware configurations across different platforms—CPUs, GPUs, and neuromorphic processors like Akida—reveals distinct strengths and limitations that influence system performance, particularly in terms of latency, FPS, and energy efficiency. Each hardware component

uniquely impacts the real-time feasibility and efficiency of defect detection tasks, as demonstrated by comparative experiments with both CNN and SNN models.

7.4.1 CPUs

CPUs provide a versatile but less specialized platform for defect detection, typically yielding higher latency and lower FPS. They are better suited for applications where flexibility and general-purpose processing are prioritized over real-time performance. However, when processing raw data or hand-crafted features, CPUs struggle to meet the stringent real-time requirements of rail defect detection. In the experiments, CPUs generally exhibited high latency across both CNN and SNN models, especially when using hand-crafted features. Despite their flexibility, the lack of low-latency performance and energy efficiency limits CPUs' applicability for continuous real-time defect detection in power-sensitive environments.

7.4.2 GPUs

GPUs excel in handling large-scale computations, achieving low latency and high FPS when processing CNN models. This platform's parallel processing capability enhances model performance by reducing inference time, especially for CNN models. In the experiments, GPUs consistently achieved real-time processing speeds with CNNs but fell short for SNNs due to higher power demands and heat generation. Although GPUs perform well in high-speed applications, the elevated power requirements necessitate robust cooling and power management, making GPUs better suited to scenarios prioritizing processing speed over energy conservation.

7.4.3 Akida Neuromorphic Processor

The Akida neuromorphic processor demonstrates exceptional efficiency, achieving the lowest latency and highest FPS among all tested platforms, while maintaining minimal power consumption. However, it should be noted that the Akida models were trained and evaluated on smaller input images (64×64) compared to the CPU-CNN (300×300) and SNN (128×128). These differences in input resolution mean the results are not strictly comparable across platforms. The superior efficiency of Akida must therefore be interpreted with caution, as part of the observed performance gains are attributable to reduced input size rather than hardware advantages alone. This processor's design, optimized for energy-sensitive applications, enables real-time performance without substantial cooling requirements. Akida's architecture, specifically designed for neuromorphic computing, processes models that are first converted from CNNs into spiking form before deployment. As a result, the practical distinction between CNN and SNN implementations on Akida is largely academic. Even under minimal configurations, the processor is able to meet real-time constraints with high efficiency. The experiments highlight Akida's potential as an optimal solution for applications where consistent real-time processing and low energy consumption are critical. Its neuromorphic efficiency provides a balance of high precision, low power usage, and minimal latency, making it an excellent choice for continuous, real-time rail defect detection.

7.4.4 Summary

Each hardware platform presents unique advantages and trade-offs. CPUs offer flexibility but fall short in real-time processing, while GPUs provide high-speed performance at the cost of increased power demands. The Akida neuromorphic processor stands out by delivering low-latency, energy-efficient performance, making it the most suitable option for real-time, energy-sensitive applications in rail surface defect detection.

7.5 INSIGHTS ON OPTIMAL HARDWARE-MODEL COMBINATIONS

The analysis of hardware and model combinations underscores the significance of aligning model architecture with platform capabilities to achieve optimal defect detection performance in rail inspection applications. Among the platforms evaluated, the Akida neuromorphic processor consistently demonstrates the highest efficiency and lowest latency for both CNN and SNN models, largely due to its design for energy-sensitive, real-time processing. This characteristic makes Akida particularly advantageous for applications where continuous monitoring and power conservation are crucial. Comparing GPUs and CPUs, the GPU excels in processing CNN models, offering a substantial increase in FPS and lower latency than the CPU. This is largely due to the GPU's ability to handle parallel computations, which accelerates CNN inference times, making it suitable for high-speed applications that prioritise processing power over energy efficiency. However, CPUs and GPUs consume significantly more power than the Akida processor, which makes them less ideal for environments with stringent energy constraints.

For applications in real-time requirements and limited power resources, the Akida processor, even in its minimal configuration, achieves real-time performance with an exceptionally low power consumption of 0.073W. Its balance of low latency and high FPS confirms its suitability for high-stakes, energy-sensitive defect detection systems. Conversely, while CPUs provide a flexible, general-purpose platform, their higher latency and lower FPS limit their use in real-time defect detection, relegating them to applications where speed is not paramount.

This analysis underscores performance in real-time rail surface defect detection, the Akida processor combined with CNN models offers a balanced solution between computational efficiency and power consumption. In scenarios prioritizing raw processing speed and where power constraints are secondary, GPUs emerge as an effective alternative. This insight supports the practical recommendation of Akida processors for energy-sensitive applications, while GPUs provide a viable option for rapid, large-scale defect detection where energy use is less restrictive.

7.6 PRACTICAL IMPLICATIONS AND FUTURE RESEARCH DIRECTIONS

The findings in this study carry important practical implications for the deployment of neuromorphic-based rail surface defect detection systems. The optimal use of hardware, such as the Akida neuromorphic processor, combined with CNN models, offers a viable solution for energy-efficient,

real-time defect detection, especially in environments with limited power resources. This configuration balances accuracy, latency, and energy consumption, making it suitable for continuous monitoring along extensive railway networks. The GPU, while more power-intensive, serves as a practical option for applications where speed and processing power are prioritized, such as high-throughput inspections or environments with ample power supply.

In practical terms, rail maintenance operations can benefit significantly from deploying neuromorphic hardware for continuous and autonomous defect detection. The Akida processor's minimal power requirement and real-time performance enable the system to operate for extended periods without frequent recharging or overheating, reducing the need for manual inspections. Additionally, incorporating high-resolution sensors and optimized data acquisition methods aligns with the system's requirements to capture and process defect-related features accurately. These practical benefits suggest that a carefully configured neuromorphic system could replace or augment existing rail inspection methods, offering increased efficiency, cost savings, and accuracy in defect detection.

Future research should explore several avenues to further enhance neuromorphic rail defect detection systems. First, investigating hybrid architectures that leverage the strengths of both SNNs and CNNs could unlock additional improvements in power efficiency and accuracy. Developing more refined pre-processing techniques and feature extraction methods that are optimized for neuromorphic hardware could also enhance the system's defect detection capability. Moreover, adapting the system for real-world variability—such as fluctuating lighting conditions and motion-induced distortions—could make it more robust and reliable across diverse operational contexts.

Another promising direction involves refining the hardware configurations, particularly for the Akida processor, to accommodate emerging deep learning architectures beyond CNNs and SNNs. Research into scalable deployment strategies, including integration with existing railway infrastructure and IoT networks, could also facilitate real-time data transmission, centralized monitoring, and predictive maintenance applications. These future directions offer pathways to further enhance the impact and applicability of neuromorphic computing in rail maintenance, ensuring these systems continue to evolve and meet the demands of modern railway operations.

7.7 LIMITATIONS OF THE STUDY

This study encounters several limitations that influence the scope and applicability of its findings. First, while the experiments provide valuable insights into the performance of CNN and SNN models on various hardware platforms, the evaluation remains limited to specific configurations and hardware models. For instance, only one neuromorphic processor (Akida) and one GPU model is tested, which may not fully represent the range of available or emerging hardware options. Expanding the hardware types and configurations could yield broader insights into performance and efficiency across different platforms.

Another limitation involves the use of hand-crafted features for defect detection, which, while beneficial for guiding the CNN model, might restrict the model's ability to adapt to unexpected defect patterns. Hand-crafted features are inherently limited by human-defined parameters and may not capture the full variability of real-world rail defects. Testing models with both hand-crafted and automatically extracted features could provide a more comprehensive understanding of model adaptability and robustness.

Additionally, the study primarily evaluates the models under controlled experimental conditions, which may not fully replicate the operational challenges encountered in real-world rail environments. Factors like varying light conditions, motion blur from high-speed inspections, and environmental noise can impact model performance, and these elements are not entirely captured in this study. Incorporating these real-world factors into future tests would better inform the reliability of the models in practical applications.

Finally, while the Akida processor demonstrates impressive energy efficiency, the study does not fully address the long-term durability and maintenance of neuromorphic hardware in the field. Real-world deployment in rail environments could expose the hardware to wear and environmental stresses, which may affect performance over time. Future studies could explore the durability of these hardware components and their long-term stability under operational conditions to better gauge the feasibility of sustained, real-time defect detection in rail networks.

These limitations underscore the need for further research to validate and expand upon the findings, ensuring the development of a robust, adaptable, and sustainable neuromorphic-based defect detection system for real-world railway maintenance.

7.8 FINAL REMARKS

In conclusion, this study addresses the critical factors that influence the upper performance limits of a neuromorphic-based rail surface defect detection system, specifically examining the roles of model architecture, model types, and hardware configurations. By comparing CNN and SNN models across CPU, GPU, and neuromorphic platforms, the research reveals that each model and hardware choice significantly affects system efficiency, accuracy, and energy consumption. CNNs, known for their high accuracy and computational speed, demonstrate optimal performance on GPUs, where they achieve real-time processing capabilities with high frame rates. This combination underscores the suitability of CNNs on GPUs for scenarios that prioritize rapid, precise defect detection, even at higher power costs.

SNNs, on the other hand, exhibit notable advantages in energy efficiency, particularly on neuromorphic hardware like the Akida processor. While they generally underperform in accuracy compared to CNNs, SNNs offer a sustainable option for defect detection in energy-constrained environments where power efficiency outweighs the need for absolute accuracy. This research highlights that the Akida processor, when paired with either CNN or SNN models, maintains real-

time capabilities while minimizing energy use, offering an effective solution for continuous monitoring applications. The selection of hand-crafted features further supports both CNN and SNN performance by simplifying the input data, though automatic feature extraction may provide greater adaptability for future models.

Overall, the study concludes that the choice of neural network architecture, model type, and hardware platform must align closely with operational goals to maximize system performance. For applications demanding both speed and accuracy, CNNs on GPUs remain superior. Conversely, for long-term deployment in remote or resource-limited environments, neuromorphic processors paired with energy-efficient models like SNNs emerge as the preferred option. Future advancements in neuromorphic processing and hybrid architectures may further optimize this balance, extending the capabilities of defect detection systems.

This research ultimately affirms that achieving peak performance in neuromorphic-based rail surface defect detection requires a strategic combination of model selection, feature engineering, and hardware optimization. By carefully aligning these elements with specific operational requirements, rail maintenance systems can attain the necessary levels of accuracy, speed, and energy efficiency to support real-time, scalable, and sustainable defect detection solutions.

REFERENCES

- [1] "Rail Defects Handbook," in "Some Rail Defects, their Characteristics, Causes and Control," Australian Rail Track Corporation LTD, 2006, issue A. Accessed: 01/11/2022.
- [2] Y. K. Al-Douri, P. Tretten, and R. Karim, "Improvement of railway performance: a study of Swedish railway infrastructure," *Journal of Modern Transportation*, vol. 24, no. 1, pp. 22–37, 2016.
- [3] L. Jie, L. Siwei, L. Qingyong, Z. Hanqing, and R. Shengwei, "Real-time rail head surface defect detection: A geometrical approach," in *2009 IEEE International Symposium on Industrial Electronics*, 2009: IEEE, pp. 769–774.
- [4] Y. M. Jidayi, "Reliability Improvement of Railway Infrastructure," Stellenbosch: Stellenbosch University, 2015.
- [5] Z. Popović, L. Lazarević, L. Brajović, and P. Gladović, "Managing rail service life," *Metalurgija*, vol. 53, no. 4, pp. 721–724, 2014.
- [6] N. Rail. *Rail Flaw Defects Identification Handbook*. [Online]. Available: <https://www.nordco.com/Customer-Support/Rail-Services--Inspection-Technologies/Inspection-Services/Rail-Flaw-Identification-Handbook.htm>
- [7] A. R. T. C. Ltd, "Rail Defects Handbook," ARTC, Engineering Practices Manual, March 2006, 2005.
- [8] Y. Min, B. Xiao, J. Dang, B. Yue, and T. Cheng, "Real time detection system for rail surface defects based on machine vision," *EURASIP Journal on Image and Video Processing*, vol. 2018, no. 1, 2018, doi: 10.1186/s13640-017-0241-y.
- [9] H. S. Lee *et al.*, "Efficient defect identification via oxide memristive crossbar array based morphological image processing," *Advanced Intelligent Systems*, vol. 3, no. 2, p. 2000202, 2021.
- [10] G. W. Burr *et al.*, "Neuromorphic computing using non-volatile memory," *Advances in Physics: X*, vol. 2, no. 1, pp. 89–124, 2017.
- [11] G. Indiveri and S.-C. Liu, "Memory and information processing in neuromorphic systems," *Proceedings of the IEEE*, vol. 103, no. 8, pp. 1379–1397, 2015.
- [12] S. R. R. Khalid, F. Riaz, A. Hassan, "Enhanced dynamic quadrant histogram equalization plateau limit for image contrast enhancement," presented at the 2015 Fith International Conference on Digital Information and Communication Technology and its Applications (DICTAP), Beirut, Lebanon, 01/06/2015, 2015.
- [13] N. R. Soora, S. Vodithala, and J. S. H. Badam, "Filtering Techniques to remove Noises from an Image," presented at the 2022 International Conference on Advances in Computing, Communication and Applied Informatics (ACCAI), 2022.

- [14] B. Desai, U. Kushwaha, S. Jha, and M. NMIMS, "Image filtering-techniques algorithms and applications," *Applied GIS*, vol. 7, no. 11, pp. 970–975, 2020.
- [15] C. Gokilavani, N. Rajeswaran, V. J. A. Karthick, R. S. Kumar, and N. Thangadurai, "Comparative results performance analysis of various filters used to remove noises in retinal images," presented at the 2015 Online International Conference on Green Engineering and technologies (IC-GET), Coimbatore, India, 27 November 2015, 2015. [Online]. Available: <https://ieeexplore-ieee-org.nwulib.idm.oclc.org/document/7453859/authors#authors>.
- [16] V. Jakkula, "Tutorial on support vector machine (svm)," *School of EECS, Washington State University*, vol. 37, no. 2.5, p. 3, 2006.
- [17] C. N. Kamath, S. S. Bukhari, and A. Dengel, "Comparative Study between Traditional Machine Learning and Deep Learning Approaches for Text Classification," presented at the Proceedings of the ACM Symposium on Document Engineering 2018, 2018.
- [18] T. Fletcher, "Support vector machines explained," *Tutorial paper*, pp. 1–19, 2009.
- [19] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik, "Gene selection for cancer classification using support vector machines," *Machine learning*, vol. 46, pp. 389–422, 2002.
- [20] R. G. Brereton and G. R. Lloyd, "Support vector machines for classification and regression," *Analyst*, vol. 135, no. 2, pp. 230–67, Feb 2010, doi: 10.1039/b918972f.
- [21] A. Cutler, D. R. Cutler, and J. R. Stevens, "Random Forests," in *Ensemble Machine Learning*, 2012, ch. Chapter 5, pp. 157–175.
- [22] L. Breiman. K. A. Publishers. (2001). Random Forest. Available: <https://link.springer.com/article/10.1023/a:1010933404324>
- [23] Y. Xu, Li, M., & Wang, J., "Rail surface defect detection using vibration signal analysis and machine learning techniques.," *IEEE Transactions on Instrumentation and Measurement*, vol. 68, no. 7, pp. 2345–2356, 2019, doi: <https://doi.org/10.1109/TIM.2019.2900352>.
- [24] H. Zhang, Sun, Y., & Liu, Q., "Automated rail surface defect classification using image processing and Random Forests," *Journal of Intelligent & Robotic Systems*, vol. 102, no. 3, p. 45, 2021, doi: <https://doi.org/10.1007/s10846-021-01345-0>.
- [25] S. Sun and R. Huang, "An adaptive k-nearest neighbor algorithm," in *2010 seventh international conference on fuzzy systems and knowledge discovery*, 2010, vol. 1: IEEE, pp. 91–94.
- [26] T.-H. Cho, R. W. Connors, and P. A. Araman, "A comparison of rule-based, k-nearest neighbor, and neural net classifiers for automated industrial inspection," in *Proceedings of the IEEE/ACM International Conference on Developing and Managing Expert System Programs*, 1991: IEEE Computer Society Press, pp. 202–209.

- [27] P. Cunningham and S. J. Delany, "k-Nearest neighbour classifiers-A Tutorial," *ACM computing surveys (CSUR)*, vol. 54, no. 6, pp. 1–25, 2021.
- [28] W. Huang, Zhang, X., & Liu, Y., "Surface defect detection in steel using texture features and k-NN classification," *Journal of Materials Processing Technology*, vol. 255, pp. 412–420, 2018, doi: <https://doi.org/10.1016/j.jmatprotec.2017.12.016>.
- [29] Y. Song, A. Kołcz, and C. L. Giles, "Better Naive Bayes classification for high-precision spam detection," *Software: Practice and Experience*, vol. 39, no. 11, pp. 1003–1024, 2009.
- [30] I. Rish, "An empirical study of the naive Bayes classifier," in *IJCAI 2001 workshop on empirical methods in artificial intelligence*, 2001, vol. 3, no. 22: Seattle, WA, USA;, pp. 41–46.
- [31] S.-B. Kim, K.-S. Han, H.-C. Rim, and S. H. Myaeng, "Some effective techniques for naive bayes text classification," *IEEE transactions on knowledge and data engineering*, vol. 18, no. 11, pp. 1457–1466, 2006.
- [32] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of big data*, vol. 6, no. 1, pp. 1–48, 2019.
- [33] C. Khosla and B. S. Saini, "Enhancing performance of deep learning models with different data augmentation techniques: A survey," in *2020 International Conference on Intelligent Engineering and Management (ICIEM)*, 2020: IEEE, pp. 79–85.
- [34] W. P. Jia Shijie, Jia Peiyi, Hu Siping, "Research on data augmentation for image classification based on convolution neural networks," presented at the 2017 Chiese Automation Congress (CAC), Jinan, China, 01 January 2018, 2017.
- [35] A. Subramaniam, "A neuromorphic approach to image processing and machine vision," in *2017 Fourth International Conference on Image Information Processing (ICIIP)*, 2017: IEEE, pp. 1–6.
- [36] R. A. Nawrocki, R. M. Voyles, and S. E. Shaheen, "A Mini Review of Neuromorphic Architectures and Implementations," *IEEE Transactions on Electron Devices*, vol. 63, no. 10, pp. 3819–3829, 2016, doi: 10.1109/ted.2016.2598413.
- [37] A. Shrestha, H. Fang, Z. Mei, D. P. Rider, Q. Wu, and Q. Qiu, "A survey on neuromorphic computing: Models and hardware," *IEEE Circuits and Systems Magazine*, vol. 22, no. 2, pp. 6–35, 2022.
- [38] M. Hu, H. Li, Y. Chen, Q. Wu, G. S. Rose, and R. W. Linderman, "Memristor crossbar-based neuromorphic computing system: A case study," *IEEE transactions on neural networks and learning systems*, vol. 25, no. 10, pp. 1864–1878, 2014.
- [39] I. Sugiarto, G. Liu, S. Davidson, L. A. Plana, and S. B. Furber, "High performance computing on spinnaker neuromorphic platform: A case study for energy efficient image processing," in *2016*

- IEEE 35th International Performance Computing and Communications Conference (IPCCC)*, 2016: IEEE, pp. 1–8.
- [40] X. Zhang, A. Huang, Q. Hu, Z. Xiao, and P. K. Chu, "Neuromorphic computing with memristor crossbar," *physica status solidi (a)*, vol. 215, no. 13, p. 1700875, 2018.
- [41] Y. Liu, B. Gao, J. Tang, H. Wu, and H. Qian, "Architecture-circuit-technology co-optimization for resistive random access memory-based computation-in-memory chips," *Science China Information Sciences*, vol. 66, no. 10, p. 200408, 2023.
- [42] N. Getty, T. Brettin, D. Jin, R. Stevens, and F. Xia, "Deep medical image analysis with representation learning and neuromorphic computing," *Interface Focus*, vol. 11, no. 1, p. 20190122, 2021. [Online]. Available: <https://pmc.ncbi.nlm.nih.gov/articles/PMC7739912/>.
- [43] A. Taylor, "Benchmarking Akida with Edge Impulse: A Validation of Model Performance on BrainChip's Akida Platform," in *Edge Impulse* vol. 2024, ed, 2023.
- [44] M. Davies *et al.*, "Advancing neuromorphic computing with loihi: A survey of results and outlook," *Proceedings of the IEEE*, vol. 109, no. 5, pp. 911–934, 2021.
- [45] G. Crebbin, "Image segmentation using neuromorphic integrate-and-fire cells," in *2005 5th International Conference on Information Communications & Signal Processing*, 2005: IEEE, pp. 305–309.
- [46] C. S. Thakur, J. Molin, and R. Etienne-Cummings, "Real-time image segmentation using a spiking neuromorphic processor," in *2017 51st Annual Conference on Information Sciences and Systems (CISS)*, 2017: IEEE, pp. 1–6.
- [47] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning*, 2019: PMLR, pp. 6105–6114.
- [48] F. Gatti, "Artificial Neural Networks: layer architectures, optimizers and automatic differentiation," 2023.
- [49] A. Zilouchian, "Fundamentals of neural networks," in *Intelligent control systems using soft computing methodologies*: CRC Press, 2001, pp. 17–38.
- [50] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.
- [51] A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier, and A. Maida, "Deep learning in spiking neural networks," *Neural networks*, vol. 111, pp. 47–63, 2019.
- [52] T. Iakymchuk, A. Rosado, J. V. Frances, and M. Batallre, "Fast spiking neural network architecture for low-cost FPGA devices," in *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, 2012: IEEE, pp. 1–6.

- [53] L. Camuñas-Mesa, B. Linares-Barranco, and T. Serrano-Gotarredona, "Neuromorphic spiking neural networks and their memristor-CMOS hardware implementations. Materials 12, 2745," ed, 2019.
- [54] C. Zhang, C. Huang, and J. He, "Defects Recognition of Train Wheelset Tread Based on Improved Spiking Neural Network," *Chinese Journal of Electronics*, vol. 32, no. 5, pp. 941–954, 2023.
- [55] F. Paredes-Vallés, K. Y. Scheper, and G. C. De Croon, "Unsupervised learning of a hierarchical spiking neural network for optical flow estimation: From events to global motion perception," *IEEE transactions on pattern analysis and machine intelligence*, vol. 42, no. 8, pp. 2051–2064, 2019.
- [56] D. Zheng *et al.*, "A defect detection method for rail surface and fasteners based on deep convolutional neural network," *Computational intelligence and neuroscience*, vol. 2021, no. 1, p. 2565500, 2021.
- [57] J. Wang, L. Luo, W. Ye, and S. Zhu, "A defect-detection method of split pins in the catenary fastening devices of high-speed railway based on deep learning," *IEEE Transactions on Instrumentation and Measurement*, vol. 69, no. 12, pp. 9517–9525, 2020.
- [58] H. Yuan, H. Chen, S. Liu, J. Lin, and X. Luo, "A deep convolutional neural network for detection of rail surface defect," in *2019 IEEE vehicle power and propulsion conference (VPPC)*, 2019: IEEE, pp. 1–4.
- [59] C. N. Ojiugwo, A. B. Abdallah, and C. Thron, "Simulation of biological learning with spiking neural networks," *Implementations and Applications of Machine Learning*, pp. 207–227, 2020.
- [60] M. Vakili, M. Ghamsari, and M. Rezaei, "Performance analysis and comparison of machine and deep learning algorithms for IoT data classification," *arXiv preprint arXiv:2001.09636*, 2020.
- [61] D. Gyawali, "Comparative analysis of cpu and gpu profiling for deep learning models," *arXiv preprint arXiv:2309.02521*, 2023.
- [62] PyTorch. "Reproducibility." PyTorch. <https://docs.pytorch.org/docs/stable/notes/randomness.html> (accessed July, 2025).
- [63] P. Warden, "Why is it so difficult to retrain neural networks and get the same results?," in *Pete Warden's blog* vol. 2025, P. Warden, Ed., ed, 2022.
- [64] G. Xiao, J. Liu, Z. Zheng, and Y. Sui, "Nondeterministic Impact of CPU Multithreading on Training Deep Learning Systems," in *ISSRE*, 2021, pp. 557–568.
- [65] UnitX. "How frame rate impacts machine vision performance." <https://resources.unitxlabs.com/frame-rate-machine-vision-performance/#:~:text=vision%20system,adequate%20for%20applications%20involving%20stationary> (accessed September, 2025).

- [66] E. da Silva Rodrigues, D. M. L. Martins, and F. B. de Lima Neto, "Automatic Feature Engineering Using Self-Organizing Maps," in *2021 IEEE Latin American Conference on Computational Intelligence (LA-CCI)*, 2021: IEEE, pp. 1–6.
- [67] P. Mohanaiah, P. Sathyanarayana, and L. GuruKumar, "Image texture feature extraction using GLCM approach," *International journal of scientific and research publications*, vol. 3, no. 5, pp. 1–5, 2013.
- [68] S. Barburiceanu, R. Terebes, and S. Meza, "3D Texture Feature Extraction and Classification Using GLCM and LBP-Based Descriptors," *Applied Sciences*, vol. 11, no. 5, 2021, doi: 10.3390/app11052332.
- [69] S. K. P.S and D. V.S, "Extraction of Texture Features using GLCM and Shape Features using Connected Regions," *International Journal of Engineering and Technology*, vol. 8, no. 6, pp. 2926–2930, 2016, doi: 10.21817/ijet/2016/v8i6/160806254.
- [70] Ş. Öztürk and B. Akdemir, "Application of feature extraction and classification methods for histopathological image using GLCM, LBP, LBGLCM, GLRLM and SFTA," *Procedia computer science*, vol. 132, pp. 40–46, 2018.
- [71] R. Maini and H. Aggarwal, "Study and comparison of various image edge detection techniques," *International journal of image processing (IJIP)*, vol. 3, no. 1, pp. 1–11, 2009.
- [72] M. Ali and D. Clausi, "Using the Canny edge detector for feature extraction and enhancement of remote sensing images," in *IGARSS 2001. Scanning the Present and Resolving the Future. Proceedings. IEEE 2001 International Geoscience and Remote Sensing Symposium (Cat. No. 01CH37217)*, 2001, vol. 5: IEEE, pp. 2298–2300.
- [73] W. Rong, Z. Li, W. Zhang, and L. Sun, "An improved CANNY edge detection algorithm," in *2014 IEEE international conference on mechatronics and automation*, 2014: IEEE, pp. 577–582.
- [74] "<Performance Analysis of Canny and Sobel Edge Detection Algorithms in Image Mining.pdf>."
- [75] M. X. Y. Wei, "Detection of lane line based on Robert operator," *Journal of Measurements in Engineering*, vol. 9, no. 3, pp. 156–166, 2021, doi: <https://doi.org/10.21595/jme.2021.22023>.
- [76] M. Pietikäinen, A. Hadid, G. Zhao, and T. Ahonen, *Computer vision using local binary patterns*. Springer Science & Business Media, 2011.
- [77] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

- [78] X. Wei, Z. Yang, Y. Liu, D. Wei, L. Jia, and Y. Li, "Railway track fastener defect detection based on image processing and deep learning techniques: A comparative study," *Engineering Applications of Artificial Intelligence*, vol. 80, pp. 66–81, 2019.
- [79] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.
- [80] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.
- [81] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

APPENDIX A: PSEUDOCODE

ALGORITHM A-1: CNN-BASED RAIL DEFECT DETECTION USING EFFICIENTNET-B3 AND K-FOLD CROSS-VALIDATION WITH RAW IMAGE INPUT

A-1.1 CPU Implementation with ablation study

1. CONFIGURE ENVIRONMENT

- Set image size, batch size, epochs, and file paths.

2. LOAD AND CLEAN DATA

- Read CSV containing image paths and labels.
- Normalize label column (lowercase, strip whitespace).
- Split data into training (70%), validation (15%), and test (15%) sets with stratified sampling.

3. DEFINE IMAGE GENERATORS

- If augmentation is enabled, apply random rotations, zooms, flips, brightness changes, etc.
- Always rescale images by 1/255.
- Create ``train_gen``, ``val_gen``, ``test_gen`` using ``flow_from_dataframe``.

4. DEFINE TRAINING FUNCTION (``train_and_evaluate``)

- For a given experiment configuration:
 - a. Select EfficientNetB3 or EfficientNetB0 as the base model with ImageNet weights.
 - b. Freeze base model layers.
 - c. Add GlobalAveragePooling2D, Dropout, and Dense(sigmoid) layers.
 - d. Compile with binary crossentropy and selected optimizer (Adam/SGD).
 - e. Train model for initial epochs with callbacks for learning rate reduction and early stopping.
- If fine-tuning is enabled:
 - a. Unfreeze base model.
 - b. Compile with a lower learning rate (1e-5).
 - c. Continue training for fine-tune epochs.
- Predict labels on the test set.
- Compute classification metrics (Accuracy, Precision, Recall, F1-Score).
- Return these metrics.

5. RUN MULTIPLE EXPERIMENTS (ablation study)

- For each experiment (e.g., with/without augmentation, different dropout, optimizer, or backbone model):
 - a. Call `train_and_evaluate()`.
 - b. Store metrics along with config.

6. SAVE & DISPLAY RESULTS

- Store all results in a DataFrame and save as CSV.
- Print formatted results.

A-1.2 GPU Implementation with ablation study

1. IMPORT LIBRARIES

- Import TensorFlow, Keras, scikit-learn, pandas, and image preprocessing utilities.

2. CONFIGURE PARAMETERS

- Set image size to 300×300.
- Set batch size = 16, initial training epochs = 5, fine-tuning epochs = 25.
- Define paths to image folder and CSV label file.

3. ENABLE GPU AND MIXED PRECISION

- Check for available GPUs.
- If available, enable memory growth and mixed precision (for faster and more memory-efficient training).

4. LOAD DATA

- Read CSV file with `'imagepath'` and `'label'`.
- Normalize label column (strip, lowercase).
- Perform stratified train/validation/test split (70% train, 15% validation, 15% test).

5. DEFINE IMAGE GENERATORS

- For training: optionally apply augmentation (rotation, zoom, brightness, flip).
- For validation and test: only rescale pixel values.

- Create data generators using ``flow_from_dataframe``.

6. DEFINE ``train_and_evaluate()`` FUNCTION

- Choose base model (EfficientNetB3 or B0) with ImageNet weights, without the top layer.
- Freeze base model initially.
- Add custom classification head (GlobalAveragePooling2D → Dropout → Dense(sigmoid)).
- Compile model with chosen optimizer (Adam or SGD) and binary crossentropy loss.
- Train for initial epochs with early stopping and learning rate reduction callbacks.
- If fine-tuning is enabled:
 - Unfreeze base model.
 - Compile with lower learning rate (1e-5).
 - Train for fine-tuning epochs.
- Predict on the test generator.
- Calculate and return Accuracy, Precision, Recall, and F1-Score.

7. RUN MULTIPLE EXPERIMENTS (Ablation Study)

- Loop through a list of experiment settings (different model backbones, dropout rates, optimizers, augmentation, fine-tuning).
- Call ``train_and_evaluate()`` for each.
- Save results in a DataFrame.

8. OUTPUT RESULTS

- Print results table to terminal.
- Save all results to ``ablation_study_results_GPU.csv``.

A-1.3 AKIDA Implementation with ablation study

1. IMPORT LIBRARIES

- Import TensorFlow/Keras for model training and data loading.
- Import Akida SDK tools (``cnn2snn``, ``akida_models``, ``AkidaDevice``) for conversion and deployment.

2. CONFIGURE PARAMETERS

- Set image size to (64, 64) to match AkidaNet requirements.
- Define batch size, number of epochs, number of classes, image directory, and CSV path.

3. LOAD AND PREPROCESS DATA

- Read label CSV and normalize label column ('defect' → 1, 'no_defect' → 0).
- Split data into training and test sets using stratified sampling.
- For each image:
 - a. Load and resize image.
 - b. Normalize pixel values to [0, 1].
 - c. Store label as one-hot vector.
- Return NumPy arrays for training and test data.

4. LOAD PREBUILT CNN MODEL (AkidaNet)

- Load ``akidanet_imagenet_pretrained()`` model with input shape (64, 64, 3).
- Replace top layer to match binary classification task (2 output neurons).

5. COMPILE AND TRAIN

- Compile model with Adam optimizer and categorical crossentropy loss.
- Train the model for specified epochs with validation split (e.g., 10%).

6. CONVERT TO AKIDA FORMAT

- Use ``cnn2snn.convert()`` to convert the trained Keras model to an Akida-compatible model.

- Evaluate performance of converted model on test data.

7. EXPORT AND DEPLOY TO AKIDA PCIe BOARD

- Save model as ``akida`` file.
- Connect to PCIe board using ``AkidaDevice()``.
- Load converted model onto board using ``device.load_network()``.

8. INFER OR STREAM CLASSIFICATIONS

- (Optional) Send test images to the Akida board and receive predictions in real-time.

ALGORITHM A-2: SNN-BASED RAIL DEFECT DETECTION – CPU IMPLEMENTATION WITH ABLATION STUDY

1. IMPORT LIBRARIES

- Import torch, snnTorch, sklearn, pandas, numpy, time, and image libraries.

2. CONFIGURE DEVICE

- Use CPU if GPU is not available.

3. DEFINE DATASET CLASS

- Load grayscale images from file paths and associated labels from a CSV.
- Resize images to 128×128 and apply transformations.

4. DEFINE SNN MODEL STRUCTURE

- Create a class with two fully connected layers and Leaky Integrate-and-Fire (LIF) neurons.
- Vary hidden layer size based on ablation config.

5. DEFINE ABLATION CONFIGURATIONS

- List of dictionaries with combinations of:
 - Hidden units (e.g., 64, 128, 256)
 - Learning rate (e.g., 0.001, 0.0005)

- Number of epochs (e.g., 10, 20)

6. PREPARE DATA

- Load dataset using DataLoader with appropriate transforms and batch size.

7. LOOP OVER EACH ABLATION CONFIGURATION

For each config in ablation_configs:

- Initialize the SNN model with given hidden units.
- Compile model with Adam optimizer using specified learning rate.
- Train the model for specified number of epochs:
 - Loop over batches:
 - Forward pass (membrane potential + spikes)
 - Compute loss using CrossEntropy
 - Backpropagate and update weights
- Evaluate model:
 - Disable gradient tracking.
 - Forward pass through test data.
 - Record predictions and ground truths.
 - Compute Accuracy, Precision, Recall, F1-Score, and Inference Time.
- Store results in a dictionary.

8. SAVE RESULTS

- Combine all dictionaries into a DataFrame.
- Export as CSV to local disk for further analysis.

9. OUTPUT

- Print completion message and path to results.

ALGORITHM A-3: CNN-BASED RAIL DEFECT DETECTION USING HAND-CRAFTED FEATURES AS INPUT

A-3.1 CPU Implementation with ablation study

BEGIN

1. DEFINE constants for image directory, label CSV, and resize dimensions
2. DEFINE extract_features(path):
 - a. Load and resize image
 - b. Convert to grayscale
 - c. Extract GLCM metrics (5)
 - d. Extract LBP histogram (9 bins)
 - e. Compute Sobel X and Y means
 - f. Compute Canny edge mean
 - g. Combine all into one feature vector (length = 16)
3. READ CSV file with image paths and labels
4. FOR each image:
 - a. Extract features
 - b. Append features and label to lists
5. CONVERT lists to NumPy arrays
6. SPLIT data into train and test sets
7. SCALE features using StandardScaler
8. DEFINE ablation configurations:
 - dropout rate (0.3, 0.5)
 - optimizer (adam, sgd)

9. FOR each configuration:

a. Build MLP model with:

- Input layer
- Dense + Dropout
- Dense + Dropout
- Output layer with Sigmoid

b. Compile with selected optimizer

c. Train model with early stopping and LR reduction

d. Evaluate on test set:

- Compute Accuracy, Precision, Recall, F1-Score

e. Store results

10. PRINT and SAVE all ablation results to CSV

END

A-3.2 GPU Implementation with ablation study

BEGIN

1. Detect if GPU is available using TensorFlow API

IF available:

Enable memory growth

Set platform label to "GPU"

ELSE:

Default platform label to "CPU"

2. Define extract_features(image_path):

a. Load and resize image

b. Convert to grayscale and uint8

c. Extract GLCM features: contrast, dissimilarity, homogeneity, energy, correlation

- d. Extract LBP histogram (9 bins)
 - e. Compute Sobel X and Y averages
 - f. Compute Canny edge average
 - g. Combine all into a feature vector (length = 16)
3. Read image filenames and labels from CSV
 4. FOR each image:
 - a. Extract features
 - b. Convert label to binary
 - c. Append to feature and label lists
 5. Convert feature and label lists to NumPy arrays
 6. Split data into training and test sets (stratified)
 7. Normalize features with StandardScaler
 8. Define ablation configurations:
 - dropout_rate: [0.3, 0.5]
 - optimizer: ['adam', 'sgd']
 9. FOR each configuration:
 - a. Create MLP model:
 - Dense + Dropout
 - Dense + Dropout
 - Dense output with Sigmoid
 - b. Compile with selected optimizer
 - c. Train with early stopping + reduce LR on plateau
 - d. Predict on test set
 - e. Compute metrics: Accuracy, Precision, Recall, F1
 - f. Store results

10. Save results as DataFrame and CSV

END

A-3.3 Akida Implementation with ablation study

BEGIN

1. Define paths to image directory and CSV label file

2. Define target image size (e.g., 128×128)

3. Define extract_features(image_path):

- a. Read and resize image
- b. Convert to grayscale
- c. Compute GLCM features (5)
- d. Compute LBP histogram (9 bins)
- e. Compute Sobel X and Y means
- f. Compute Canny mean
- g. Return 16-element feature vector

4. Load CSV and extract features for all images

5. Convert to NumPy arrays (X, y)

6. Split into training/testing sets

7. Scale test features using StandardScaler

8. Define list of pre-converted Akida model files:

- Each tuple: (model_file, dropout_rate, optimizer)

9. For each model file:

- a. Load Akida model from .fbz
- b. Predict using model.predict(X_test)

- c. Round predictions to binary labels
- d. Compute Accuracy, Precision, Recall, F1-score
- e. Log results in structured format

10. Save all results to CSV file

END

ALGORITHM A-4: SNN-BASED RAIL DEFECT DETECTION USING HAND-CRAFTED FEATURES AS INPUT – CPU IMPLEMENTATION WITH ABLATION STUDY

BEGIN

1. DEFINE device (use GPU if available, else fallback to CPU)

2. DEFINE extract_features(image_path):

- a. Load and resize the image
- b. Convert image to grayscale
- c. Convert grayscale to uint8 format
- d. Compute GLCM features:
 - contrast, dissimilarity, homogeneity, energy, correlation
- e. Compute LBP histogram (9 bins)
- f. Compute mean Sobel-X and Sobel-Y gradient intensities
- g. Compute mean Canny edge intensity
- h. Combine all 16 features into a single feature vector
- i. RETURN feature vector

3. DEFINE FeatureDataset class:

- a. Initialize with CSV file, image directory, and scaler
- b. FOR each entry in CSV:
 - i. Extract features using extract_features()

- ii. Store feature vector and corresponding label (0 or 1)
- c. If scaler is given, apply feature scaling
- d. Define `__len__` to return number of samples
- e. Define `__getitem__` to return feature and label as tensors

4. DEFINE SNN class (inherits `nn.Module`):

- a. Define fully connected layer 1 (input $\rightarrow$ hidden)
- b. Apply LIF neuron (Leaky Integrate-and-Fire)
- c. Define fully connected layer 2 (hidden $\rightarrow$ output)
- d. Apply LIF neuron
- e. In forward pass:
 - Pass input through `fc1 + LIF1`
 - Pass output through `fc2 + LIF2`
 - Return membrane potential and spikes

5. LOAD dataset using `FeatureDataset`

6. CREATE `DataLoader` with batch size and shuffling

7. SCALE features using `StandardScaler`

8. DEFINE ablation study configurations:

- List of dictionaries with:
 - hidden layer sizes (e.g., 32, 64, 128)
 - learning rates (e.g., 0.001, 0.0005)
 - LIF beta values (e.g., 0.9, 0.95)

9. FOR each configuration in ablation study:

- a. INITIALIZE SNN model with:
 - Input size = 16
 - Hidden size = `config["hidden_size"]`

- Output size = 2
- LIF neuron beta = config["beta"]

b. DEFINE loss function (CrossEntropyLoss)

c. DEFINE optimizer (Adam with config["learning_rate"])

d. TRAINING LOOP:

i. For each epoch:

- For each batch:
 - Forward pass through SNN
 - Compute loss using membrane potential
 - Backpropagate and update weights
- Track average loss and print epoch time

e. EVALUATION LOOP:

i. Disable gradient tracking

ii. For each batch:

- Run forward pass
- Predict using argmax of membrane potential
- Collect predictions and true labels

iii. Compute accuracy, precision, recall, F1-score

iv. Store metrics in results list with config details

10. SAVE all ablation results to CSV

END

ALGORITHM A-5: MODEL ARCHITECTURE SUMMARY

To ensure clarity and consistency across the dissertation, this appendix provides a consolidated overview of all model architectures used in the study. Each model is described in terms of its input representation, number and type of layers, node configuration, and learning strategy. This summary removes ambiguity about how hand-crafted features and raw image inputs were integrated into CNNs and SNNs and provides a reference point for replication.

A-5.1 CNN Models Using Raw Image Inputs

- **Input:** Rail surface images resized to $300 \times 300 \times 3$ (RGB).
- **Backbone:** EfficientNet-B3 pretrained on ImageNet.
- **Fine-tuning:** Base layers either frozen or unfrozen depending on configuration.
- **Layers:**
 - Convolutional layers (EfficientNet structure).
 - Global Average Pooling layer.
 - Dense (Fully Connected) layer with 256 nodes, ReLU activation.
 - Dropout (0.2 or 0.5 depending on ablation setting).
 - Output Dense layer with 1 node, sigmoid activation (binary classification).
- **Optimizers tested:** Adam, SGD.

A-5.2 CNN Models Using Hand-Crafted Feature Inputs

- **Input:** 16-element feature vector derived from GLCM, LBP, Sobel, and Canny descriptors.
- **Architecture:**
 - Fully Connected layer with 64 nodes, ReLU activation.
 - Dropout (0.2).
 - Fully Connected layer with 32 nodes, ReLU activation.
 - Output Dense layer with 1 node, sigmoid activation.
- **Note:** The network is a shallow MLP-style classifier adapted to accept feature vectors. It is not a 2D CNN, but for consistency within the dissertation it was grouped under CNN-based models.

A-5.3 SNN Model Using Raw Image Inputs

- **Input:** Images resized to 28×28 grayscale, converted to spike trains using rate coding.
- **Architecture:**

- Input layer encodes pixel intensities as spike frequencies.
- Two hidden layers of Leaky Integrate-and-Fire (LIF) neurons.
- Output layer of two LIF neurons representing binary classes (defect/no defect).
- **Learning:** Spike-Timing-Dependent Plasticity (STDP).

A-5.4 SNN Model Using Hand-Crafted Features

- **Input:** 16-element feature vector derived from GLCM, LBP, Sobel, and Canny.
- **Architecture:**
 - Input layer encodes feature values into spike rates.
 - Hidden layer of LIF neurons, sparsely connected.
 - Output layer of two LIF neurons (defect/no defect).
- **Learning:** STDP, with unsupervised weight updates based on spike timing.

A-5.5 Akida CNN Deployment

- **Input:** Rail surface images converted to Akida-compatible spike streams via MetaTF API.
- **Architecture:** CNN converted into a spiking CNN using event-driven execution.
- **Layers:** Convolutional layers with in-memory compute, pooling layers, fully connected classification layer.
- **Output:** Single spiking output neuron corresponding to defect/no defect prediction.
- **Training:** Performed offline on CPU/GPU, then weights quantised and deployed to Akida hardware.

Clarification on Hand-Crafted Features

The term “features” in this dissertation refers to image descriptors such as GLCM, LBP, Sobel, and Canny, which emphasise specific aspects of the input data without substantially reducing dimensionality. These processed feature maps were either:

- fed directly into shallow CNN-like architectures (in practice, fully connected networks suited to vector inputs), or
- spike-encoded and processed by SNNs.

APPENDIX B: COMPARISON OF CNN ARCHITECTURES

Model	Accuracy & Metrics	Inference Speed (CPU)	Model Size & Efficiency	Small vs. Large Defect Detection
ResNet-50 (Residual Network)	~76% top 1 ImageNet accuracy. Strong defect classification performance when well-trained. High F1 and recall scores [47].	~20 images/sec unoptimized. Requires substantial compute (~3.8 billion FLOPs) [47].	~25 million parameters. High capacity but less efficient than newer models [77].	Strong large-defect detection. May struggle with very fine features at low input resolution [78].
Inception-V3	~78% top 1 on ImageNet accuracy. Performs well on balanced defect datasets [47].	Moderate speed on CPU. Better than ResNet but not optimised for real-time without tuning [79].	~22M parameters. Efficient vs. accuracy ratio is good, but complexity is high [77].	Excellent multi-scale performance via parallel filters (1x1, 3x3, 5x5). Effective for both small texture anomalies and large visual disruptions [79].
DenseNet-121	~75-76% accuracy. High precision/recall with compact design [47].	Performs well. Densely connected layers introduce slight computation overhead [80].	~7M parameters. Encourages feature reuse, improving learning efficiency [77, 80].	Passes early features to deeper layers, preserving fine defect cues and broader context simultaneously [80].
MobileNet-V2	~72% top 1 accuracy. ~85-90% task accuracy possible with tuning [47, 81]. Slightly lower recall on subtle defects.	Extremely fast and efficient. Real-time achievable on CPU [81].	~2-4M params. Depthwise separable convolutions minimise compute and memory use [81].	Performs well on moderate/ obvious defects. May miss very small defects at lower resolutions due to limited feature capacity [81].
EfficientNet-B3 (Selected model)	~81.6% top 1 ImageNet accuracy. Higher than ResNet-50 [47]. Demonstrated high F1-	Faster than ResNet-50 despite higher accuracy. Achieves ~1.8 billion FLOPs (vs. ResNet's	~12 million parameters. Balanced compound scaling leads to high	High-resolution input (300x300), SE attention, and compound scaling enable robust detection of small and

Model	Accuracy & Metrics	Inference Speed (CPU)	Model Size & Efficiency	Small vs. Large Defect Detection
	scores in defect classification experiments (Chapter 3 – Results).	3.8B) [47], (Chapter 3 – Results). Near real-time on CPU with optimisation (Chapter 3 – Results).	accuracy-per-parameter ratio [47], (Chapter 3 – Results). Efficient in memory and compute.	large defects [47], (Chapter 3 – Results). Superior multi-scale performance for industrial tasks.

APPENDIX C: CNN AND SNN DESIGN PROCESS

C-1 CNN STEP-BY-STEP DESIGN PROCESS

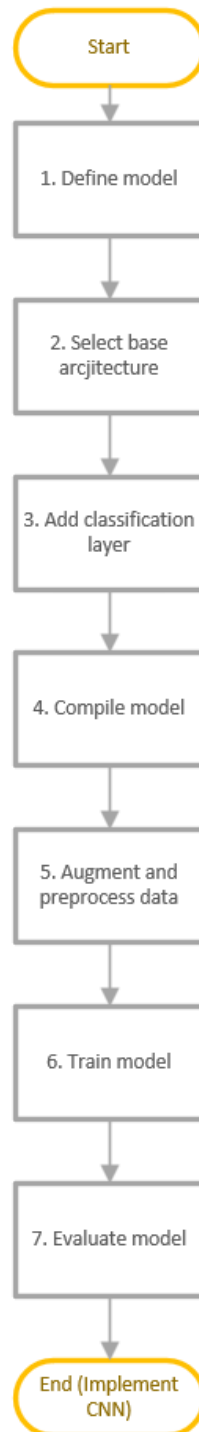


Figure C-1: CNN design process

Figure C-1 illustrates the flow chart for the step-by-step design of the CNN model. The following provides a detailed explanation of each stage in the model development process implemented on the CPU, GPU, and Akida platforms.

1. Define model:

The CNN model is defined as a supervised deep learning algorithm used for classifying into two categories: "defect" and "no defect". It consists of multiple learnable layers—primarily convolutional and pooling layers—that automatically extract spatial features from input data.

2. Select base architecture:

An appropriate backbone architecture is selected based on the target platform. EfficientNetB3 is chosen for CPU and GPU platforms due to its balance between accuracy and computational efficiency. For Akida deployment, a smaller and hardware-compatible CNN model is selected to enable conversion using the `cnn2snn` toolchain.

3. Add classification layer:

After the convolutional base, a classification head is appended. This typically includes a Global Average Pooling layer, a Dropout layer (to reduce overfitting), and a Dense output layer with a sigmoid activation function for binary classification. On the Akida platform, the classification layer is kept minimal and fully compatible with Akida's neuromorphic processing requirements.

4. Compile model:

The model is compiled by specifying the loss function (`binary_crossentropy`), the optimizer (such as Adam or SGD), and evaluation metrics (e.g., accuracy, precision, recall, F1-score). For GPU implementations, compilation is optimised to accelerate convergence. Prior to Akida deployment, the model is compiled on the host system and then converted into an event-driven format suitable for spike-based inference.

5. Augment and preprocess data:

Before training, the input images are resized to a consistent shape (e.g., 300×300 or 64×64 pixels), normalized, and converted to grayscale if necessary. Data augmentation is applied to increase variability, including transformations such as rotation, flipping, brightness shifts, and zoom. In hand-crafted feature setups, additional preprocessing steps extract descriptors like GLCM, LBP, Sobel, and Canny features.

6. Train model:

The CNN is trained on labeled data using mini-batch gradient descent over multiple epochs. During training, the network learns to optimize its filter weights to minimize classification errors. The GPU accelerates training through parallel processing, while Akida models are trained on conventional hardware before being converted. Training parameters—such as learning rate, dropout rate, batch size, and epoch count—are tuned based on ablation study results.

7. Evaluate model:

After training, the model is evaluated on a test dataset. Performance is assessed using metrics such as accuracy, precision, recall, F1-score, and AUC. Confusion matrices and ROC curves are generated to analyse classification quality. For the Akida platform, additional evaluation includes inference latency and energy consumption, providing insight into real-time feasibility.

8. Implement CNN

The final trained and validated CNN model is deployed to the respective platform. On CPU and GPU, it is executed using the TensorFlow/Keras runtime. On Akida, the model is converted into .akida format and deployed to the PCIe board for ultra-low-power real-time classification.

C-2 SNN STEP-BY-STEP DESIGN PROCESS

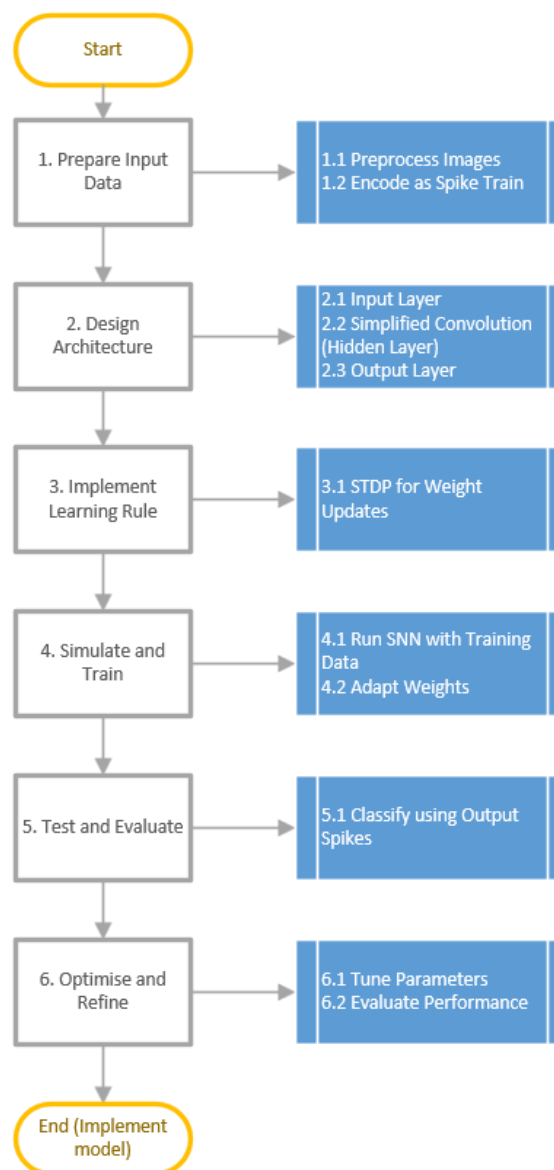


Figure C-0-1: SNN design process

Figure C-0-1 illustrates the flow chart for the design of the SNN model using a simplified version of the CNN. Below is a comprehensive overview of the design process for the SNN model implemented on the CPU.

1. **Input data preparation:** The input data consisted of railway images, resized to a uniform dimension (e.g., 128 x 128 pixels), converted to grayscale, and normalized to the range [0,1]. These pre-processed images were encoded into spike trains using rate encoding, where the intensity of each pixel determined the spike frequency. This step enabled the SNN to process discrete spiking events instead of continuous values.
2. **Network architecture design:** The SNN architecture mirrored a simplified CNN structure:
 - **Input layer:** Encodes pre-processed image data into spike trains.
 - **Hidden layer (simplified convolution):** Instead of learnable convolutional filters, the hidden layer utilised predefined receptive fields and sparsely connected neurons following the LIF model. The LIF model simulates biological neurons by accumulating input spikes, firing when a threshold is reached, and resetting afterward.
 - **Output layer:** Two output neurons represented the “defect” and “no defect” classes. The neuron with the highest spike count determined the classification.
3. **Synaptic learning and weighted updates:** The SNN employed STDP, a biologically inspired learning rule that adjusts synaptic weights based on the relative timing of spikes. Unlike CNNs that rely on backpropagation, the STDP rule allows localised learning without requiring gradient computations.
4. **Simulation and training:** Using the Brian2 simulator, the SNN was trained by presenting spike-encoded images sequentially. Over multiple iterations, the synaptic weights adapted through the STDP mechanism to recognise patterns associated with defects.
5. **Testing and classification:** During testing, spike-encoded images were passed through the trained network. The spike counts of the output neurons determined the predicted class: the neuron with the highest spike count signalled the classification.
6. **Optimisation and refinement:** The network parameters, such as firing thresholds, synaptic weights and STDP learning rates, were tuned iteratively to achieve better accuracy and energy efficiency. Additionally, the network was evaluated on its computational performance on a CPU.

Table C - 1: Comparison between SNN and CNN

Aspect	CNN	SNN
Data representation	Continuous values (real numbers)	Discrete spikes (binary events)
Computation	Convolution operations with learnable filters	Event-driven computations with predefined connectivity

Learning algorithm	Backpropagation using gradient descent	STDP
Energy efficiency	High energy consumption	Energy-efficient, suitable for neuromorphic hardware
Latency	Static, fixed computational flow	Real-time, event-driven processing
Hardware dependency	Optimised for GPUs	Suitable for CPUs and neuromorphic chips
Interpretability	Limited biological plausibility	High biological plausibility

Table C - 1 shows the comparisons between SNN and CNN.

APPENDIX D: T-TABLE

t Table

cum. prob	$t_{.50}$	$t_{.75}$	$t_{.80}$	$t_{.85}$	$t_{.90}$	$t_{.95}$	$t_{.975}$	$t_{.99}$	$t_{.995}$	$t_{.999}$	$t_{.9995}$
one-tail	0.50	0.25	0.20	0.15	0.10	0.05	0.025	0.01	0.005	0.001	0.0005
two-tails	1.00	0.50	0.40	0.30	0.20	0.10	0.05	0.02	0.01	0.002	0.001
df											
1	0.000	1.000	1.376	1.963	3.078	6.314	12.71	31.82	63.66	318.31	636.62
2	0.000	0.816	1.061	1.386	1.886	2.920	4.303	6.965	9.925	22.327	31.599
3	0.000	0.765	0.978	1.250	1.638	2.353	3.182	4.541	5.841	10.215	12.924
4	0.000	0.741	0.941	1.190	1.533	2.132	2.776	3.747	4.604	7.173	8.610
5	0.000	0.727	0.920	1.156	1.476	2.015	2.571	3.365	4.032	5.893	6.869
6	0.000	0.718	0.906	1.134	1.440	1.943	2.447	3.143	3.707	5.208	5.959
7	0.000	0.711	0.896	1.119	1.415	1.895	2.365	2.998	3.499	4.785	5.408
8	0.000	0.706	0.889	1.108	1.397	1.860	2.306	2.896	3.355	4.501	5.041
9	0.000	0.703	0.883	1.100	1.383	1.833	2.262	2.821	3.250	4.297	4.781
10	0.000	0.700	0.879	1.093	1.372	1.812	2.228	2.764	3.169	4.144	4.587
11	0.000	0.697	0.876	1.088	1.363	1.796	2.201	2.718	3.106	4.025	4.437
12	0.000	0.695	0.873	1.083	1.356	1.782	2.179	2.681	3.055	3.930	4.318
13	0.000	0.694	0.870	1.079	1.350	1.771	2.160	2.650	3.012	3.852	4.221
14	0.000	0.692	0.868	1.076	1.345	1.761	2.145	2.624	2.977	3.787	4.140
15	0.000	0.691	0.866	1.074	1.341	1.753	2.131	2.602	2.947	3.733	4.073
16	0.000	0.690	0.865	1.071	1.337	1.746	2.120	2.583	2.921	3.686	4.015
17	0.000	0.689	0.863	1.069	1.333	1.740	2.110	2.567	2.898	3.646	3.965
18	0.000	0.688	0.862	1.067	1.330	1.734	2.101	2.552	2.878	3.610	3.922
19	0.000	0.688	0.861	1.066	1.328	1.729	2.093	2.539	2.861	3.579	3.883
20	0.000	0.687	0.860	1.064	1.325	1.725	2.086	2.528	2.845	3.552	3.850
21	0.000	0.686	0.859	1.063	1.323	1.721	2.080	2.518	2.831	3.527	3.819
22	0.000	0.686	0.858	1.061	1.321	1.717	2.074	2.508	2.819	3.505	3.792
23	0.000	0.685	0.858	1.060	1.319	1.714	2.069	2.500	2.807	3.485	3.768
24	0.000	0.685	0.857	1.059	1.318	1.711	2.064	2.492	2.797	3.467	3.745
25	0.000	0.684	0.856	1.058	1.316	1.708	2.060	2.485	2.787	3.450	3.725
26	0.000	0.684	0.856	1.058	1.315	1.706	2.056	2.479	2.779	3.435	3.707
27	0.000	0.684	0.855	1.057	1.314	1.703	2.052	2.473	2.771	3.421	3.690
28	0.000	0.683	0.855	1.056	1.313	1.701	2.048	2.467	2.763	3.408	3.674
29	0.000	0.683	0.854	1.055	1.311	1.699	2.045	2.462	2.756	3.396	3.659
30	0.000	0.683	0.854	1.055	1.310	1.697	2.042	2.457	2.750	3.385	3.646
40	0.000	0.681	0.851	1.050	1.303	1.684	2.021	2.423	2.704	3.307	3.551
60	0.000	0.679	0.848	1.045	1.296	1.671	2.000	2.390	2.660	3.232	3.460
80	0.000	0.678	0.846	1.043	1.292	1.664	1.990	2.374	2.639	3.195	3.416
100	0.000	0.677	0.845	1.042	1.290	1.660	1.984	2.364	2.626	3.174	3.390
1000	0.000	0.675	0.842	1.037	1.282	1.646	1.962	2.330	2.581	3.098	3.300
Z	0.000	0.674	0.842	1.036	1.282	1.645	1.960	2.326	2.576	3.090	3.291
	0%	50%	60%	70%	80%	90%	95%	98%	99%	99.8%	99.9%
	Confidence Level										