HAMM-LIPPSTADT
UNIVERSITY OF APPLIED SCIENCES

TinyML Model Survey And Implementation

Department Lippstadt 2

Bachelor Thesis
for optainment of the degree of
Bachelor of Engineering

submitted by
STEPHEN PATRICK ETENG

Electronic Engineering
Mat.Nr.: 2190654
<stephen-eteng.patrick@stud.hshl.de>

July 2, 2024

First supervisor:	Prof. Dr. Achim Rettberg
Second supervisor:	Prof. Dr. Stefan Henkler

Abstract

This thesis comprehensively evaluates a Convolutional Neural Network (CNN) model developed for Bragg spot detection, optimized for deployment on Tiny Machine Learning (TinyML) platforms. The study outlines the process from data preparation and model training to evaluation using confusion matrices and validation of model accuracy and size. The CNN model was trained on 80 percent of a dataset comprising 1600 samples and tested on the remaining 20 percent (400 samples). The model's performance was assessed across three image resolutions: 720x720, 300x300, and 100x100 pixels. The confusion matrix analysis highlighted the model's proficiency in detecting 'Hit' and 'Maybe' instances, with notable challenges in accurately identifying 'Miss' instances. Misclassifications were detailed, revealing areas for potential improvement. The validation focused on determining the optimal image dimension for training the CNN model, balancing accuracy and model size for effective deployment on TinyML devices. The results demonstrated a trade-off between image resolution and model performance, with the highest accuracy achieved at 720x720 pixels but with a model size unsuitable for resource-constrained environments. The 100x100 pixel resolution provided a feasible model size (63.6 KB) with an accuracy of 79 percent and an F1-score of 0.8, indicating a viable option for TinyML deployment. This evaluation underscores the importance of optimizing model size and computational efficiency without significantly compromising accuracy, essential for applying TinyML in real-world crystallography imaging tasks. The insights gained from the confusion matrix and performance metrics pave the way for future enhancements in model architecture and training techniques to improve accuracy and efficiency on TinyML platforms.

Contents

1 Introduction

The advent of TinyML and its integration with CNNs marks a breakthrough period in the field of edge computing and embedded systems[MS23]. This integration has revolutionized multiple domains, including crystallography, by opening the door for intelligent, real-time decision-making capabilities on systems with limited resources.

In particular, using CNNs and TinyML for image processing in crystallography, especially for identifying Bragg spots in diffraction images, presents a promising and exciting method for studying macromolecular structures[FS97]. Traditionally, locating Bragg spots that show crystal structures within a sample has been a time-consuming and computationally demanding procedure that frequently requires significant computer resources alongside expert human intervention.

For real-time, on-device image processing in crystallography, TinyML presents a viable option. Its small size and low power consumption make it appropriate for implementation on microcontrollers, field-programmable gate arrays (FPGAs), and other resource-constrained platforms[Bie16]. TinyML models can be trained to reliably identify and locate Bragg spots in diffraction images. A crucial first step in identifying the underlying crystal structure is utilizing the pattern recognition powers of CNNs.

However, the adoption of TinyML and CNNs in crystallography is not without its challenges. The use of embedded systems necessitates highly optimized and compressed models to achieve accurate results within limited memory and computing resources[SBT+22]. Moreover, the integration of these models into existing crystallographic data analysis pipelines requires specialized toolchains, hardware-software co-design techniques, and rigorous testing and validation methodologies. Recognizing these challenges is the first step towards overcoming them and harnessing the full potential of this technology in crystallography.

Despite the hurdles, the integration of TinyML and CNNs holds immense transformative potential for crystallography. This technology could revolutionize the analysis of macromolecular structures, enabling real-time, on-device processing of diffraction data. Resulting in accelerated discoveries and a deeper understanding of biological processes at the molecular level[SBT+22]. Moreover, the successful application of TinyML and CNNs in crystallography could pave the way for their adoption in other scientific disciplines, ushering in a new era of real-time, automated analysis of complex information.

1.1 Motivation

The main goal of this thesis is to tackle the significant challenges in the field of crystallography, particularly in detecting Bragg spots in diffraction images. Conventional methods

for identifying Bragg spots are time-consuming and require substantial computational resources, making them impractical for real-time analysis, especially considering the large data volumes produced by advanced techniques such as X-ray free-electron lasers (XFELs). By combining TinyML with CNNs, this research aims to create a solution with enormous potential. This efficient, automated solution, capable of functioning on low-power, resource-constrained devices, promises to significantly improve the speed and accuracy of Bragg spot detection.

1.2 Scope of the Research Work

This research focuses on developing a TinyML model integrated with CNNs designed explicitly for detecting Bragg spots in crystallography images. The study encompasses the model's concept, implementation, and evaluation phases, emphasizing optimizing the model for low-power devices without compromising accuracy or processing speed. The datasets (Protein serial crystallography diffraction) used for training and testing the model are provided by Maia[Mai12].

1.3 Aims and Objectives

The primary objective of this study is to conduct a comprehensive examination and implementation of a TinyML model. The research aims to achieve the following key goals:

1. Development of a CNN model for Bragg spot detection using the Python programming language
2. Optimization of the model for the FPGA platform, tailored explicitly for TinyML applications
3. Assessment of the accuracy of the TinyML model through the utilization of a confusion matrix and the measurement of computation time

1.4 Overview

The thesis is structured to guide the reader through a comprehensive exploration of integrating TinyML with CNNs for Bragg spot detection in crystallography images.

Chapter 1: Introduction not only sets the stage by presenting the background, motivation, aims, and scope of the research but also highlights the novelty of integrating TinyML with CNNs for Bragg spot detection in crystallography images, sparking the reader's curiosity.

Chapter 2: Fundamentals delves into the basic concepts of Machine Learning and TinyML, covering necessary hardware and software platforms and discussing model optimization and power efficiency.

Chapter 3: Related Work surveys the existing literature on TinyML model development and applications, comparing different approaches to identify research gaps.

Chapter 4: Concept and Implementation details methods used in the TinyML model tailored to this study's case, addressing implementation challenges such as model optimization and resource constraints.

Chapter 5: Evaluation focuses on the criteria and metrics for evaluating TinyML models, including performance analysis of the implemented model regarding accuracy and efficiency.

Finally, Chapter 6: Conclusion and Summary not only summarizes the research's key findings and contributions but also underlines the practical implications of integrating TinyML with CNNs for Bragg spot detection in crystallography images.

2 Fundamentals

2.1 Introduction

This chapter is divided into three sections. The first part discusses the basic concepts of machine learning and tinyML. The second part provides an overview of the hardware and software platforms available for tinyML. Finally, we explore power efficiency and model optimization in tinyML. These aspects will help us better understand the approach to identifying the problem at hand and provide insight for obtaining better results.

2.2 Basic concepts in Machine Learning and TinyML

The computing field is undergoing rapid and significant transformation, driven by advances in Machine Learning (ML) and TinyML. These cutting-edge technologies are revolutionizing our interactions with the digital world, providing devices with incredible intelligence, simplifying processes, and harnessing the full potential of data like never before. In this section, we will explore the fundamentals of machine learning and TinyML, establishing the framework for understanding their transformative potential and applications.

2.2.1 Understanding TinyML

TinyML is an emerging field that aims to bring the power of machine learning to resource-constrained devices such as microcontrollers, System-on-Chip (SoC) platforms, and Internet of Things (IoT) devices. TinyML models are specifically designed to operate within embedded systems' severe memory, compute, and power constraints, enabling real-time, on-device data processing without relying on cloud or server-based infrastructure [SBA20]. The significance of TinyML lies in its ability to expand the benefits of machine learning to a wide range of applications and domains where traditional methods may be impractical or unfeasible. TinyML enables intelligent decision-making and data processing capabilities on low-power, edge devices, thereby opening up new possibilities for applications such as:

1. Industrial automation and control systems: TinyML can offer real-time monitoring, predictive maintenance, and automated decision-making in industrial settings, lowering downtime and increasing efficiency.
2. Healthcare and wearable devices: On-device processing of sensor data and biometric signals allows for real-time health monitoring, disease diagnosis, and individualized therapy suggestions.
3. Smart home and consumer electronics: TinyML can power intelligent home automation systems, voice assistants, and context-aware devices that respond to user

preferences and behaviour.

4. Environmental monitoring and agricultural applications: TinyML models on low-power sensors and devices can provide real-time monitoring of environmental variables, crop health, and animal behaviour, allowing for data-driven decision-making in these areas.

5. Scientific instrumentation and data analysis: TinyML, as illustrated in the context of crystallography, can revolutionize complex dataset analysis by enabling real-time, on-device processing, minimizing latency, and addressing privacy concerns associated with transmitting sensitive data to the cloud.

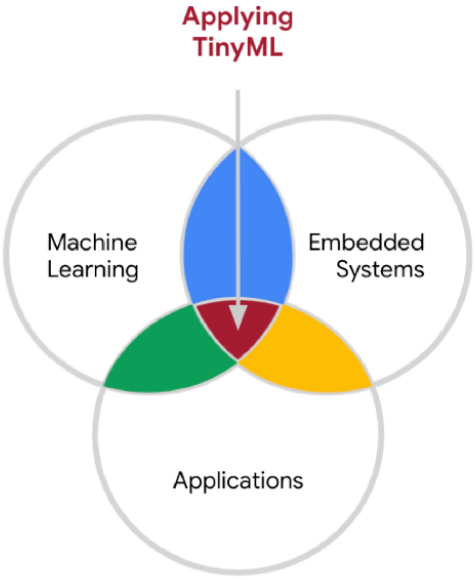

Figure 2.1: Diverse applications of TinyML across various domains [JRPK+22]

TinyML aims to create new applications by integrating machine learning models into embedded systems. Located at the intersection of embedded systems and machine learning, TinyML is dedicated to enabling unique functionality for embedded device applications, as shown in Figure 2.1 above.

Traditional machine learning models, especially those based on deep neural networks, require significant processing resources, such as powerful GPUs or cloud computing infrastructure. These models are often trained on massive datasets and have millions or billions of parameters, rendering them unsuitable for use on resource-constrained devices with limited memory and computing capability[MSS22].

TinyML models, on the other hand, are intended to be compact and efficient, using techniques such as quantization, pruning, and knowledge distillation to minimize model size and computational complexity while maintaining accuracy and performance. These optimized models can run on microcontrollers, FPGAs, and other low-power processors, allowing for real-time, on-device inference and decision-making[MUDK22].

Table 2.1: Traditional machine learning models vs TinyML models

Aspect	**Traditional Machine Learning**	TinyML
Computational Requirements	High (GPUs, TPUs, or powerful CPUs)	Low (microcontrollers, low-power CPUs)
Model Size	Large (MBs to GBs)	Compact (KBs to a few MBs)
Power Consumption	High (tens to hundreds of watts)	Minimal (<1 watt)
Inference Time	Fast (with powerful hardware)	Varies (optimized for efficiency, not speed)
Data Transfer Needs	High (due to cloud or server-based processing)	Low (localized processing reduces transfer)
Privacy and Security Considerations	Varies (higher risk without localized processing)	Enhanced (localized data processing improves privacy)
Typical Applications	Image recognition, Natural Language Processing, Complex data analysis	Wearable devices, IoT sensors, Edge computing

Table 2.1 highlights the critical differences between traditional machine learning and TinyML models regarding their computational requirements, model size, power consumption, inference time, data transfer needs, privacy and security considerations, and typical applications.

While traditional machine learning models perform well in scenarios with abundant computational resources, TinyML models are specifically designed to operate within resource-constrained devices, allowing for real-time, on-device processing with low power consumption and improved privacy and security[WS19].

TinyML model development sometimes involves a trade-off between model accuracy and constraint resources. Researchers and developers must carefully balance these parameters, using various optimization techniques and hardware-software co-design methodologies to achieve the desired performance within the available resource envelope.

2.2.2 Machine Learning Basics

2.2.2.1 *Supervised Learning*

Supervised learning is one of the most used and well-understood machine learning paradigms. This approach trains the algorithm on a labelled dataset, with each input data instance linked with a corresponding output or target value. The primary goal of supervised learning is to learn a mapping function from input data to output labels

that can subsequently be used to make predictions or decisions about new, unknown data. Supervised learning typically involves the following steps:

1. **Data collection and preprocessing**: Create a labelled dataset with well-defined input data (features) and output labels (targets). Data cleansing, normalization, and feature engineering are frequently used to ensure the dataset is training-ready.
2. **Split the Dataset**: The labelled dataset is usually divided into two or more subsets: a training set and a validation or testing set. The training set is used for the model training, whereas the test or validation set evaluates the model's performance on previously unseen data.
3. **Model Selection and Training**: Based on the problem and data characteristics, choose an appropriate machine learning algorithm. Common algorithms for supervised learning include linear regression, logistic regression, decision trees, random forests, support vector machines (SVMs), and neural networks. During training, the algorithm learns to map input features to output labels by modifying its internal parameters or weights in response to training data.
4. **Model Evaluation and Tuning**: Test or validate the trained model's performance on the validation or test set using relevant measures (for example, accuracy, precision, recall, F1-score, mean squared error). If the performance is inadequate, change the model's hyperparameters or try an alternative approach, then repeat the training.
5. **Model Deployment and Prediction**: Once an ideal model is developed, it can be utilized to make predictions based on previously unseen data.

The efficiency of supervised learning algorithms is strongly dependent on the quality and quantity of labelled training data, as well as the method and its hyperparameters being selected and tuned for the specific problem.[WS19].

2.2.2.2 *Unsupervised Learning*

Unsupervised learning is a machine learning paradigm in which the algorithm learns from unlabeled data with no predefined target or output specified. Unlike supervised learning, which aims to learn a mapping from input features to output labels, unsupervised learning algorithms seek to uncover patterns, structures, or relationships within the data[TAG21].

The main objectives of unsupervised learning include:

1. Clustering: Grouping similar data points together based on their inherent characteristics or features. The algorithm detects clusters of data points that are more comparable to those in other clusters.
2. Dimensionality Reduction: Reducing the number of features or dimensions in high-dimensional data while preserving the most critical information or patterns can help visualize and analyze complex data more effectively.
3. Anomaly Detection: Identifying unusual or outlier data points that deviate significantly from average or expected behaviour is helpful in fraud detection, network security, and fault monitoring.

4. Association Rule Learning: Discovering interesting relationships or associations between different attributes or items in large datasets. This is commonly used in market basket analysis and recommendation systems.

Unsupervised learning often involves the following steps:

- Data Collection and Preprocessing: Obtain an unlabelled dataset and perform any necessary data cleaning, normalization, and feature engineering.
- Algorithm Selection and Training: Choose an appropriate unsupervised learning algorithm based on the desired objective (e.g., clustering, dimensionality reduction, anomaly detection). During training, the algorithm learns the underlying patterns or structures in the data without relying on labelled targets.
- Model Evaluation and Tuning: Evaluate the quality of the learned patterns or structures using appropriate metrics or domain-specific criteria. If the results are unsatisfactory, adjust the algorithm's hyperparameters, try a different approach, and repeat the training process.
- Result Interpretation and Application: Interpret the learned patterns, structures, or relationships in the context of the problem domain and apply them to achieve the desired objectives, such as data visualization, anomaly detection, or recommendation systems.

Unsupervised learning is beneficial when labelled data is scarce or difficult to obtain or when the goal is to discover unknown patterns or structures in the data. Applications of unsupervised learning include: While unsupervised learning algorithms can uncover valuable insights from data, interpreting and validating the learned patterns or structures often require domain knowledge and human expertise.[TAG21].

2.2.2.3 *Reinforcement Learning*

Reinforcement Learning is a machine learning paradigm based on how humans and animals learn through trial and error and receive rewards or penalties for their actions. Reinforcement learning teaches an agent to make decisions by interacting with its environment, performing actions, and receiving rewards or penalties based on the results of those activities. The goal is to learn a policy or decision-making strategy that maximizes the cumulative reward over time.

The critical components of a reinforcement learning system include:

1. Agent: The decision-making entity that takes actions and learns from the environment.
2. Environment: The physical or virtual world in which the agent operates and interacts.
3. State: The current condition or situation of the environment, which the agent perceives and uses to make decisions.
4. Actions: The set of possible actions the agent can take in a given state.
5. Reward: The feedback signal produced by the environment, showing the attractiveness or undesirableness of the agent's activities.

6. Policy: The decision-making strategy or mapping from states to actions that the agent learns.

The reinforcement learning process typically consists of the following steps:

- Initialization: Define the environment, the agent's possible actions, and the reward structure.
- Exploration and Exploitation: The agent interacts with the environment by performing actions and seeing the outcomes, such as state transitions and rewards. It investigates various activities to learn more about the surroundings while capitalizing on its current knowledge to maximize rewards.
- Learning and Updating: Based on the observed state transitions, actions, and rewards, the agent updates its policy or decision-making strategy using reinforcement learning algorithms, such as Q-learning, Deep Q-Networks (DQN), Policy Gradients, or Actor-Critic methods.
- Convergence and Deployment: The learning process continues until the agent's policy converges or achieves an acceptable level of performance. The learned policy can then be applied in the intended environment or application.

Reinforcement learning algorithms are grouped into two distinct categories:

1. Model-based: The agent learns a model or representation of the environment's dynamics used to plan and make decisions.
2. Model-free: The agent learns directly from experience without explicitly modelling the environment's dynamics.

While reinforcement learning has achieved remarkable successes, it can be challenging to design practical reward functions, deal with large or continuous state-action spaces, and ensure the stability and convergence of the learning process. Additionally, the exploration-exploitation trade-off and the potential for agents to converge to sub-optimal policies are ongoing research challenges in the field[BBA22].

Table 2.4 highlights the critical differences between supervised, unsupervised, and reinforcement learning in terms of input data requirements, goals, training processes, example applications, feedback signals, popular algorithms, challenges, and practical applications.

Table 2.2: Comparing the three main types of machine learning

Aspect	**Supervised Learning**	**Unsupervised Learning**	**Reinforcement Learning**
Input Data	Labeled data (inputs with corresponding outputs)	Unlabeled data (no target outputs)	Sequential data from interactions with an environment
Goal	Learn a mapping from inputs to outputs	Discover patterns, structures, or relationships in data	Learn a policy to maximize cumulative reward
Training Process	Adjust model parameters to minimize error on labeled data	Identify inherent patterns or clusters in data	Agent takes actions and learns from rewards/penalties
Examples	Image classification, speech recognition, predictive modeling	Clustering, dimensionality reduction, anomaly detection	Game playing, robotics, control systems
Feedback Signal	Ground truth labels provided	No explicit feedback signal	Rewards or penalties from the environment
Popular Algorithms	Linear regression, logistic regression, decision trees, neural networks	K-means clustering, hierarchical clustering, PCA, autoencoders	Q-learning, Deep Q-Networks, Policy Gradients, Actor-Critic
Challenges	Obtaining high-quality labeled data, overfitting, bias	Interpreting learned patterns, validating results	Exploration-exploitation trade-off, reward design, convergence
Applications	Computer vision, natural language processing, predictive analytics	Customer segmentation, topic modeling, recommender systems	Autonomous systems, game AI, personalized medicine

2.2.3 Convolutional Neural Networks (CNNs)

Since CNNs are crucial for image processing tasks, a fundamental understanding of their architecture, operation, and application in image recognition and classification is essential. Convolutional Neural Networks (CNNs) are a type of deep neural network architecture that has proven remarkably effective for various computer vision tasks, including image recognition, object detection, and image segmentation. CNNs are particularly well-suited for analyzing and extracting features from grid-like data, such as images, making them a crucial component in TinyML, especially for applications involving image processing and computer vision on resource-constrained devices[KSH17].

The critical components of a CNN architecture include:

1. **Convolutional Layers**: These layers apply a set of learnable filters (kernels) to the input image, performing convolution operations that capture local patterns and features. Each filter is spatially convoluted across the entire image, producing a feature map that encodes the presence and location of specific features in the input.
2. **Pooling Layers**: Pooling layers downsample the feature maps, reducing their spatial dimensions while retaining the most essential information. This helps to reduce computational complexity and introduces translation invariance, making the network more robust to minor shifts and distortions in the input.
3. **Activation Functions**: Non-linear activation functions, such as Rectified Linear Unit (ReLU), are applied to the output of convolutional and fully connected layers, introducing non-linearity and allowing the network to learn complex patterns and representations.
4. **Fully Connected Layers**: After several rounds of convolution and pooling operations, the feature maps are flattened and fed into fully connected layers, which perform high-level reasoning and classification based on the extracted features.

The architecture of a CNN typically consists of multiple convolutional and pooling layers, followed by one or more fully connected layers and a final output layer that produces the desired predictions or classifications, as shown in figure 2.2 below.

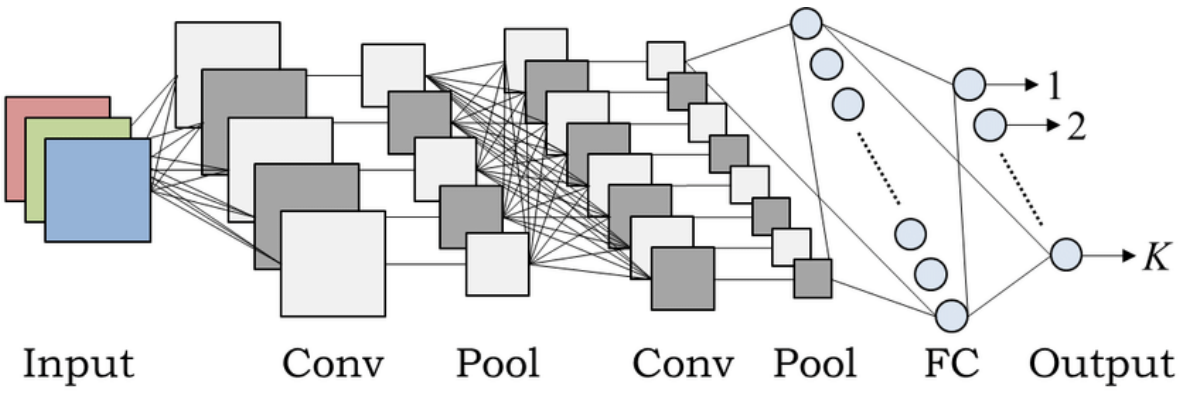

Figure 2.2: convolution operation and feature map generation [HK17]

One of the key advantages of CNNs is their ability to learn and extract relevant features directly from raw input data, without the need for manual feature engineering. This

efficient process makes them highly effective for complex tasks such as image recognition and object detection, where traditional feature extraction methods may struggle to capture all the relevant patterns and nuances.

CNNs have been successfully applied in a wide range of domains, including:

1. **Computer Vision**: Object detection, image classification, facial recognition, and segmentation tasks in various industries, such as automotive, security, and medical imaging.
2. **Natural Language Processing**: Involves processing text data as grid-like sequences to perform text classification, sentiment analysis, and language modelling tasks.
3. **Signal Processing**: Analyzing and classifying audio signals, time-series data, and other grid-like data formats.

In the context of TinyML, optimizing and deploying CNNs on resource-constrained devices presents several challenges, such as:

1. **Model Compression**: Reducing the size and computational complexity of CNNs through techniques like quantization, pruning, and knowledge distillation while maintaining acceptable accuracy.
2. **Hardware Acceleration**: Leveraging specialized hardware architectures, such as digital signal processors (DSPs) or neural processing units (NPUs), to efficiently execute the computationally intensive operations of CNNs.
3. **Energy Efficiency**: Optimizing power consumption and energy usage to enable long-lasting battery life for embedded and mobile devices running CNN-based TinyML models.

Despite these challenges, CNNs' ability to perform efficient and accurate image processing and computer vision tasks on edge devices makes them a crucial component in the TinyML ecosystem. This enables a wide range of applications in domains such as industrial automation, healthcare, and environmental monitoring.[KSH17].

2.2.4 Model Optimization

In the field of Tiny Machine Learning (TinyML), model optimization plays a crucial role in deploying of machine learning models on low-power and resource-constrained devices, such as microcontrollers, FPGAs, and embedded systems. These devices often have strict limitations in terms of memory, computational power, and energy consumption, making it essential to optimize the models to fit within these constraints while maintaining acceptable levels of accuracy and performance[Kri18].

Model optimization techniques aim to reduce machine learning models' size, computational complexity, memory footprint, and intense neural networks without significantly compromising their predictive capabilities[Kri18]. The following are some of the most widely used model optimization techniques in the field of TinyML:

1. **Quantization**: Quantization is the process of reducing the precision of the model's weights and activations from floating-point values (typically 32-bit or 64-bit) to

lower-precision fixed-point or integer values (e.g., 8-bit or 4-bit). This technique can significantly reduce the model's memory footprint and computational complexity, as lower-precision operations require fewer resources and can be executed more efficiently on hardware.

2. **Pruning**: Pruning involves identifying and removing redundant or less essential connections (weights) in a neural network, effectively reducing the number of parameters and computations required. Various pruning techniques exist, such as magnitude-based pruning, which removes weights with small magnitudes, and structured pruning, which removes entire filters or channels.

3. **Knowledge Distillation**: Knowledge distillation is a technique where a smaller, more efficient "student" model is trained to mimic the behaviour of a larger, more accurate "teacher" model. The student model learns to approximate the output distributions of the teacher model, effectively transferring the teacher's knowledge while being more compact and computationally efficient[HVD15].

4. **Low-Rank Factorization**: Low-rank factorization techniques, such as Singular Value Decomposition (SVD) or Matrix Factorization, decompose the weight matrices of a neural network into smaller, low-rank matrices. This approach can significantly reduce the number of parameters and computations required, while preserving the essential information and predictive capabilities of the original model.

5. **Architecture Search**: Architecture search algorithms, such as Neural Architecture Search (NAS) or Efficient Neural Architecture Search (ENAS), aim to automatically discover optimal neural network architectures tailored for specific hardware constraints and performance requirements. These algorithms explore architectural configurations and identify efficient designs that balance accuracy and resource utilization.

Table 2.3 highlights each technique's strengths, limitations, and applicability, providing a foundation for selecting the appropriate optimization strategy for TinyML applications.

The selection and combination of model optimization techniques should be guided by various factors, including the target hardware platform, application requirements, and trade-offs between model accuracy, latency, and energy consumption. It is crucial to consider the potential impact of these optimization techniques on the model's accuracy and robustness, as aggressive optimization may lead to performance degradation. This underscores the need for a balanced approach to model optimization.[HPTD15].

Table 2.3: Advantages and Disadvantages of different model optimization techniques

Technique	Description	Benefits	Challenges	Applications
Quantization	Reduces the precision of weights and activations from floating-point to lower-precision values	Significantly reduces model size and computational complexity	May lead to decreased accuracy due to reduced precision	Real-time processing on edge devices; IoT applications
Pruning	Removes redundant or less important weights, reducing parameters and computations	Reduces model size and enhances computational efficiency	Selecting weights to prune without affecting performance is challenging	Embedded systems; mobile devices requiring efficient processing
Knowledge Distillation	Trains a smaller model to mimic a larger model's output distributions	Creates compact models with preserved knowledge from larger models	Requires careful tuning to ensure the student model effectively learns from the teacher	Deploying efficient models in bandwidth or memory-constrained environments
Low-Rank Factorization	Decomposes weight matrices into smaller, low-rank matrices to reduce parameters	Maintains predictive capabilities while reducing model complexity	Determining the optimal rank for factorization can be complex	Applications with limited computational resources; embedded systems
Architecture Search	Automatically discovers optimal architectures balancing accuracy and resource utilization	Identifies efficient models tailored for specific hardware constraints	Resource-intensive process; may require significant computational power	Optimizing models for specific tasks and hardware configurations

2.2.5 Hardware for TinyML

The successful deployment of TinyML models heavily relies on the hardware platforms on which they are executed. TinyML targets resource-constrained devices with limited memory, computational power, and energy budgets, making the choice of hardware platform a critical factor in achieving optimal performance, efficiency, and cost. This section explores the hardware options suitable for TinyML, focusing on their capabilities, limitations, and selection criteria[JgZOC23].

1. **Microcontrollers (MCUs)**: Microcontrollers are small, low-power integrated circuits designed for embedded systems and Internet of Things (IoT) applications. Due to their low cost, low power consumption, and compact form factors, they are particularly well-suited for TinyML tasks.
    - **Capabilities**: MCUs can execute optimized TinyML models, enabling real-time inference and decision-making on edge devices. Famous MCU families for TinyML include the ARM Cortex-M series, Espressif's ESP32, and Microchip's PIC microcontrollers.
    - **Limitations**: MCUs generally have limited memory and processing power, restricting the complexity and size of TinyML models that can be deployed. Additionally, they may lack dedicated hardware acceleration for efficient neural network computations.
    - **Selection Criteria**: Memory capacity, processing power, power consumption, peripheral interfaces, and software ecosystem should be considered when choosing an MCU for TinyML applications.
2. **Field-Programmable Gate Arrays (FPGAs)**: FPGAs are reconfigurable in integrated circuits that can be programmed to implement custom hardware designs, including specialized architectures for accelerating machine learning workloads.
    - **Capabilities**: FPGAs offer parallel processing capabilities and can be configured to efficiently execute TinyML models with low latency and high energy efficiency. They can be programmed using hardware description languages (HDLs) or high-level synthesis tools.
    - **Limitations**: FPGAs generally have higher development complexity and costs compared to MCUs. They may require specialized toolchains and expertise in hardware design and optimization.
    - **Selection Criteria**: Factors such as logic capacity, memory resources, I/O capabilities, power consumption, and development toolchain support should be considered when selecting an FPGA for TinyML applications.
3. **Application-Specific Integrated Circuits (ASICs)**: ASICs are custom-designed chips optimized for specific applications or algorithms, including neural networks and machine learning tasks.
    - **Capabilities**: ASICs can provide highly optimized and efficient hardware implementations for executing TinyML models, offering unparalleled performance and energy efficiency compared to general-purpose processors.

- **Limitations**: ASICs are expensive to design and manufacture, with high non-recurring engineering costs. They lack the flexibility and reprogrammability of other hardware platforms.
- **Selection Criteria**: ASICs are typically considered for high-volume, cost-sensitive applications where the performance and efficiency benefits outweigh the development and manufacturing costs.

4. **System-on-Chip (SoC) Platforms**: SoC platforms integrate components, such as a CPU, GPU, and dedicated neural processing units (NPUs), onto a single chip, providing a compact and power-efficient solution for TinyML applications.
   - **Capabilities**: SoCs can leverage hardware accelerators like NPUs to efficiently execute TinyML models, while also providing general-purpose processing capabilities for other tasks.
   - **Limitations**: SoCs may have higher power consumption and costs than MCUs or FPGAs, and their performance may be limited by the specific hardware configurations and accelerators available.
   - **Selection Criteria**: Factors such as processing power, hardware acceleration capabilities, power consumption, memory resources, and software ecosystem support should be considered when choosing an SoC platform for TinyML applications.

The selection of the appropriate hardware platform for TinyML deployments depends on various factors, including the specific application requirements, performance and latency constraints, power and energy budgets, cost considerations, and the availability of development tools and software ecosystems. In many cases, a combination of multiple hardware platforms may be used, leveraging their respective strengths and optimizing for different stages of the TinyML workflow, such as training, optimization, and inference[BRT+21].

2.2.6 Software and Toolchains

The development of TinyML applications requires specialized software frameworks and toolchains designed to accommodate the unique challenges and constraints of deploying machine learning models on resource-constrained devices. These toolchains facilitate the entire workflow, from model training and optimization to deployment and inference on target hardware platforms. This section introduces some popular software frameworks and toolchains in the TinyML ecosystem.

1. **TensorFlow Lite for Microcontrollers**: TensorFlow Lite for Microcontrollers is a lightweight, low-latency machine learning framework developed by Google for deploying models on microcontrollers and other low-power devices. It provides tools and libraries for building, optimizing, and executing TensorFlow Lite models on various hardware platforms[MG22].
   - **Key Features**: Model optimization, quantization, operator kernels for efficient execution, support for microcontrollers and integration with the TensorFlow ecosystem.

- **Deployment Workflow**: Train and convert models using TensorFlow, optimize models using TensorFlow Lite Converter and deploy and run optimized models on microcontrollers using TensorFlow Lite for Microcontrollers runtime[MG22].

2. **Edge Impulse**: Edge Impulse is an end-to-end development platform for TinyML applications, offering a comprehensive toolchain for data acquisition, model training, optimization, and deployment on various edge devices.

    - **Key Features**: Web-based development environment, automated data collection and labelling, transfer learning capabilities, model optimization (quantization, pruning), and deployment support for microcontrollers, FPGAs, and custom hardware.

    - **Deployment Workflow**: Collect and label data using Edge Impulse Studio, train and optimize models using transfer learning or custom models, and deploy optimized models on target hardware through Edge Impulse firmware or custom deployment.

3. **Arm Ethos-U NPU**: Arm's Ethos-U NPU (Neural Processing Unit) is a dedicated hardware accelerator designed to execute machine learning workloads on low-power devices efficiently. It is accompanied by a software toolchain for model development and deployment.

    - **Key Features**: Hardware-accelerated inference, optimized for Arm Cortex-M and Cortex-A processors, support for TensorFlow Lite and ONNX models, and tools for model optimization and quantization.

    - **Deployment Workflow**: Train models using TensorFlow or other frameworks, optimize and quantize models using Arm tools, and deploy optimized models on Ethos-U NPU-enabled devices using Arm's software libraries and runtime.

4. **NVIDIA TensorRT**: NVIDIA TensorRT is a high-performance deep learning inference optimizer and runtime for deploying optimized models on NVIDIA GPUs, embedded systems, and IoT devices through the TensorRT Embedded Software.

    - **Key Features**: Model optimization techniques (e.g., layer fusion, kernel auto-tuning), support for various deep learning frameworks (TensorFlow, PyTorch, ONNX), and deployment on NVIDIA Jetson and other embedded platforms.

    - **Deployment Workflow**: Train models using supported frameworks, optimize and convert models using TensorRT, and deploy optimized models on NVIDIA GPUs or embedded platforms using TensorRT Inference Library.

In addition to these widely adopted toolchains, there are open-source and domain-specific frameworks tailored for TinyML applications in areas such as computer vision, natural language processing, and signal processing. The choice of software framework or toolchain depends on factors such as the target hardware platform, application requirements, existing software ecosystem, and the level of customization and optimization needed.

2.2.7 Data Collection and Preprocessing

Data is the fuel that drives machine learning models, and the quality and significance of training data can substantially impact model performance and reliability. In the context of TinyML, when models are deployed on resource-constrained devices, it is critical to ensure that the training data is representative, diverse, and customized to the target application. This section investigates methods for obtaining and prepping data for TinyML models, emphasizing the significance of dataset quality and relevance.

1. **Data Collection Strategies**: The first step in building a successful TinyML application is to collect high-quality, relevant data. The following strategies can be employed for data collection:
    - **Real-world Data Collection**: Data is captured directly from the target environment or application domain using sensors, cameras, or other data acquisition devices. This ensures that the data accurately represents the real-world conditions in which the TinyML model will operate.
    - **Synthetic Data Generation**: In cases where real-world data collection is challenging or insufficient, synthetic data can be generated using simulations, computational models, or domain-specific algorithms. This approach can augment and diversify the training dataset.
    - **Crowdsourcing and Collaborative Data Collection**: Leveraging crowdsourcing platforms or collaborative efforts within a community to collect and annotate data can help build diverse and large-scale datasets.
2. **Data Preprocessing**: Once the data is collected, it often requires preprocessing to ensure its suitability for training TinyML models. The following preprocessing steps are commonly employed:
    - **Data Cleaning**: Removing or handling missing, corrupted, or inconsistent data points to improve data quality and integrity.
    - **Data Normalization**: Scaling or transforming the data to a standard range or distribution can improve model convergence and stability during training.
    - **Data Augmentation**: Applying various transformations (e.g., rotations, flips, noise addition) to the existing data to artificially increase the diversity and size of the training dataset, enhancing the model's robustness and generalization capabilities.
    - **Data Labeling and Annotation**: Labeling or annotating the data with ground truth labels or annotations is crucial for supervised learning tasks. This process can be manual, automated, or a combination of both, depending on the complexity of the task and the available resources.
    - **Data Splitting**: Dividing the preprocessed data into training, validation, and testing sets to enable proper model training, evaluation, and testing.

    This table summarizes essential data preprocessing steps tailored for TinyML applications, highlighting their significance in preparing data for efficient and accurate model training and deployment on resource-constrained devices.

Table 2.4: Data preprocessing steps for TinyML applications

Preprocessing Step	Description	Applications in TinyML
Data Cleaning	Involves removing or handling missing, corrupted, or inconsistent data points to enhance data quality and integrity	Critical in TinyML for ensuring that models are trained on accurate and reliable data, especially important for edge devices where data quality can directly impact performance
Data Normalization	Scales or transforms data to a common range or distribution, improving model convergence and stability during training	Essential for TinyML applications to minimize computational complexity and ensure efficient use of limited processing power and memory
Data Augmentation	Applies transformations like rotations, flips, and noise addition to existing data, artificially increasing diversity and size of the dataset	Enhances the robustness and generalization of TinyML models by providing a more varied training dataset, crucial for applications with limited original data
Data Labeling and Annotation	Involves labeling or annotating data with ground truth labels for supervised learning tasks, which can be manual, automated, or a mix	Fundamental for TinyML to ensure models learn the correct patterns and predictions, particularly in applications like image recognition and sensor data analysis
Data Splitting	Divides the preprocessed data into training, validation, and testing sets for proper model training, evaluation, and testing	Enables effective model development and evaluation in TinyML by ensuring that the model is tested on unseen data, simulating real-world performance

3. **Data Quality and Relevance**: The quality and relevance of the training data are critical factors that determine the performance and reliability of TinyML models. Some key considerations include:

    - **Representativeness**: The training data should accurately represent the diversity and variation in the target application domain, including edge cases and corner conditions.
    - **Class Balance**: For classification tasks, the training data should have a balanced distribution of classes needed to prevent bias and ensure fair performance across all classes.
    - **Domain Shift**: Ensure that the training data represents the deployment environment, as domain shifts or distributional shifts between the training and deployment domains can lead to performance degradation.
    - **Data Privacy and Security**: When dealing with sensitive or personal data, appropriate measures should be taken to protect privacy and ensure data security during collection, preprocessing, and model training.

By employing effective data collection strategies and preprocessing techniques and ensuring the quality and relevance of the training data, TinyML models can achieve high levels of accuracy and robustness while operating within the constraints of resource-limited devices[DV22].

2.2.8 Deployment Strategies

Deploying TinyML models onto resource-constrained devices presents unique challenges and considerations. Unlike traditional machine learning deployments on powerful servers or cloud platforms, TinyML models must be optimized and integrated into the target hardware and software environment, ensuring efficient execution while adhering to strict resource constraints. This section explores various deployment strategies for TinyML models, including compilation techniques, firmware integration, and model updating strategies[JGZOC24].

1. **Model Compilation and Optimization**: Before deploying a model onto a target device, it is crucial to optimize and compile the model for efficient execution on the specific hardware platform. This process often involves the following steps:

    - **Model Quantization**: Reducing the precision of the model's weights and activations from floating-point to lower-precision fixed-point or integer values, significantly reducing the memory footprint and computational complexity.
    - **Model Pruning**: It involves identifying and removing redundant or less important connections (weights) in the neural network, reducing the model size and computational requirements.
    - **Operator Optimization**: Optimizing the implementation of specific operators or layers within the neural network to leverage hardware-specific acceleration or specialized instruction sets.

- **Code Generation and Compilation**: Converting the optimized model into executable code tailored for the target hardware architecture, often involving specialized compilers or cross-compilation toolchains.

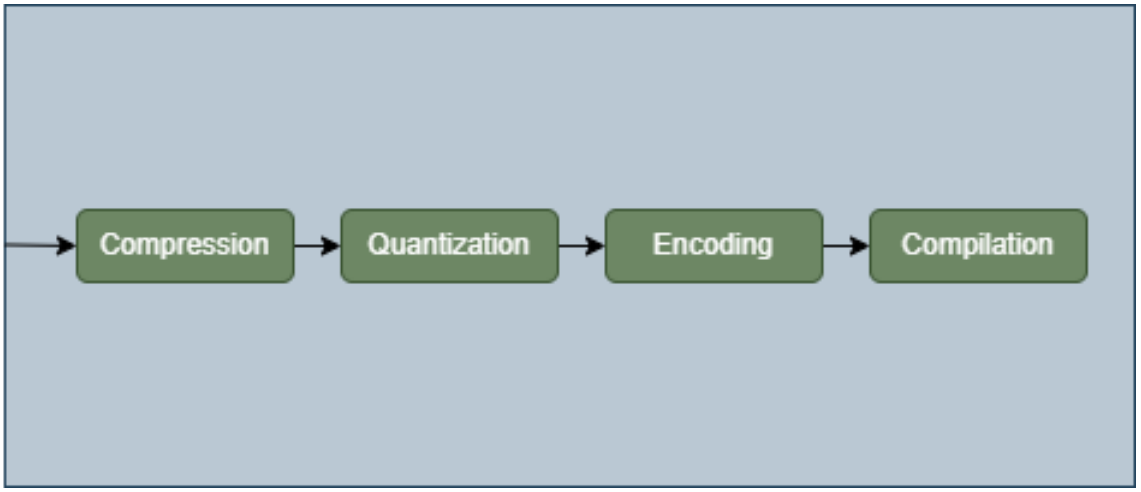

Figure 2.3: model compilation and optimization process for TinyML deployment [JgZOC23]

2. **Firmware Integration**: TinyML models are typically deployed as part of an embedded firmware or software running on the target device. The integration process involves embedding the optimized model and its associated runtime libraries into the device's firmware or application code. Depending on the hardware platform and software ecosystem, this may involve:
    - **Static Linking**: Statically linking the model and runtime libraries into the firmware binary, resulting in a self-contained executable file.
    - **Dynamic Linking**: Dynamically loading the model and runtime libraries at runtime, allowing for more flexible updates and modularity.
    - **Memory Management**: Ensuring efficient memory management and allocation strategies to accommodate the model's memory requirements within the device's limited resources.
    - **Peripheral Integration**: Integrating the TinyML model with relevant peripherals, such as sensors, cameras, or actuators, for data acquisition and processing.
3. **Model Updating and Deployment Strategies**: In many TinyML applications, the deployed model may need to be updated or replaced to accommodate changes in the target environment, new data distributions, or improved model versions. Several strategies can be employed for model updating and deployment:
    - **Over-the-Air (OTA) Updates**: Providing mechanisms for securely transferring and updating the TinyML model on the target device remotely, without physical access.
    - **Incremental Updates**: Instead of replacing the entire model, implement techniques for efficiently updating or fine-tuning the model parameters based on new data or changes in the environment.

- **Model Swapping**: Allowing for the seamless swapping of different TinyML models on the device, enabling dynamic adaptation to changing application requirements or deployment scenarios.

4. **Security and Privacy Considerations**: When deploying TinyML models on edge devices, it is crucial to consider security and privacy aspects, primarily if the devices handle sensitive data or operate in critical environments. Strategies to address these concerns may include:
    - **Secure Booting**: Ensuring that the device boots and loads only trusted firmware and models through secure booting mechanisms and cryptographic verification.
    - **Encrypted Model Transfer**: Employing secure communication protocols and encryption techniques when transferring or updating TinyML models over the air or from external sources.
    - **Privacy-Preserving Techniques**: Implementing privacy-preserving techniques, such as federated learning or differential privacy, to protect user data and prevent unauthorized access or inference.

By employing appropriate deployment strategies, optimizing models for the target hardware, and considering security and privacy aspects, TinyML models can be effectively integrated into resource-constrained devices, enabling intelligent decision-making and real-time processing capabilities at the edge[JGZOC24].

2.2.9 Performance Evaluation

Evaluating the performance of TinyML models is crucial to ensure their reliability, efficiency, and suitability for deployment on resource-constrained devices. Unlike traditional machine learning models deployed on high-performance computing infrastructures, TinyML models must balance accuracy, latency, power consumption, and model size. This section explores various metrics and methods for evaluating the performance of TinyML models, providing a comprehensive understanding of the trade-offs and considerations involved[JgZOC23].

1. **Accuracy Metrics**: Accuracy is a fundamental metric for assessing the predictive performance of machine learning models, including TinyML models. Standard accuracy metrics include:
    - **Classification Accuracy**: The proportion of correctly classified instances for classification tasks.
    - **Precision, Recall, and F1-Score**: Metrics that evaluate the model's performance in identifying positive instances, considering false positives and negatives.
    - **Mean Squared Error (MSE) or Root Mean Squared Error (RMSE)**: Measures the average squared deviation between predicted and actual values for regression tasks.
    - **Intersection over Union (IoU)**: Evaluate the overlap between predicted Moreover, object detection and segmentation tasks use ground truth bounding boxes or segmentation masks.

2. **Latency and Throughput**: In real-time applications, the latency and throughput of TinyML models are critical performance factors. Latency refers to the time it takes for the model to process a single input and produce an output, while throughput measures the number of inputs the model can process per unit of time.
    - **Inference Latency**: The time required for the model to perform inference on a single input, typically measured in milliseconds or microseconds.
    - **End-to-End Latency**: The total time from data acquisition to inference output, including any necessary preprocessing steps.
    - **Throughput**: The number of inputs the model can process per second, often measured in frames per second (FPS) for video or image processing tasks.
3. **Power Consumption and Energy Efficiency**: Power consumption and energy efficiency are critical considerations for TinyML models deployed on battery-powered or energy-constrained devices. Relevant metrics include:
    - **Average Power Consumption**: The average power consumed by the device during model execution, typically measured in milliwatts or microwatts.
    - **Peak Power Consumption**: The maximum power consumed by the device during model execution is essential for thermal management and battery life considerations.
    - **Energy Efficiency**: Measures the amount of energy required to perform a specific task or computation is often expressed as energy per inference or operation.
4. **Model Size and Memory Footprint**: The size of TinyML models and their memory footprints are essential factors determining their suitability for deployment on resource-constrained devices with limited memory and storage capabilities.
    - **Model Size**: The total size of the model file or binary, typically measured in kilobytes (KB) or megabytes (MB).
    - **RAM Footprint**: The amount of random-access memory (RAM) required to load and execute the model during inference.
    - **Flash or Storage Requirements**: The amount of non-volatile storage (e.g., flash memory) required to store the model and any associated data or libraries.
5. **Evaluation Methodologies**: Evaluating the performance of TinyML models often involves a combination of benchmarking, profiling, and real-world testing. Standard evaluation methodologies include:
    - **Benchmarking Suites**: Standardized benchmark suites, such as MLPerf Tiny or AIoT Bench, provide a consistent and reproducible way to evaluate and compare the performance of TinyML models across different hardware platforms and applications.
    - **Profiling Tools**: Hardware and software profiling tools can be used to analyze the execution of TinyML models, identifying performance bottlenecks, memory usage patterns, and potential optimization opportunities.

- **Real-World Testing**: Deploying and testing TinyML models in real-world environments and scenarios are essential to assess their performance under realistic conditions, including varying environmental factors, sensor data quality, and edge case scenarios.

By carefully evaluating the accuracy, latency, power consumption, model size, and other relevant performance metrics, developers can make informed decisions about the suitability of TinyML models for specific applications and hardware platforms, ensuring optimal performance and resource utilization[JgZOC23].

2.3 Introduction to power efficiency and model optimization in TinyML

In the burgeoning field of Tiny Machine Learning (TinyML), power efficiency and model optimization stand as pivotal concerns, directly influencing the feasibility and effectiveness of deploying machine learning models on resource-constrained devices. These devices, ranging from microcontrollers to low-power IoT sensors, operate under strict energy budgets, often relying on batteries or energy harvesting methods for power. As such, the dual objectives of minimizing power consumption while maximizing computational efficiency and model accuracy pose unique challenges and opportunities for innovation in TinyML.

2.3.1 Power Efficiency in TinyML

Power efficiency is paramount in TinyML, as it dictates deployed devices' operational lifespan and real-world usability. Efficient power use extends battery life, reduces the need for frequent recharging or replacement, and enables sustainable, continuous operation of devices in remote or inaccessible locations. Techniques to enhance power efficiency include dynamic voltage and frequency scaling, low-power operating modes, and energy-efficient hardware components.

2.3.2 Model Optimization for TinyML

Model optimization in TinyML focuses on reducing model size, computational complexity, and memory footprint without significantly compromising performance. This involves techniques such as model pruning, which removes unnecessary parameters; quantization, which reduces the precision of the model's parameters; and knowledge distillation, which transfers knowledge from a larger model to a smaller, more efficient one. These strategies are crucial for fitting sophisticated machine learning models into the limited resources available on tiny devices.

2.3.3 Balancing Act

The interplay between power efficiency and model optimization requires a careful balancing act. Optimizing a model for size and speed often comes at the cost of increased design

complexity and potentially lower prediction accuracy. Conversely, minimizing power consumption must uphold the model's ability to perform its intended function. Achieving an optimal balance necessitates a deep understanding of the hardware capabilities and the application-specific requirements of TinyML projects.

2.3.4 Future Directions

Advances in semiconductor technology, novel algorithmic approaches, and the development of specialized hardware accelerators continue to push the boundaries of what is possible in TinyML. Future directions may include

- more sophisticated on-device learning capabilities,
- ultra-low-power communication methods for data transmission and
- Energy harvesting techniques must be integrated to create self-sustaining TinyML devices.

This introduction to power efficiency and model optimization in TinyML underscores the critical importance of these aspects in unlocking the full potential of machine learning on resource-constrained devices. As the field progresses, ongoing research and innovation in these areas will be instrumental in driving the widespread adoption and success of TinyML applications across a myriad of industries and use cases.

3 Related Work

3.1 Survey of literature on TinyML model development and applications

This literature review critically examines the fusion of TinyML and CNNs for improving Bragg spot detection in crystallography, spotlighting the significant strides and challenges in the field. It highlights how these advancements guide the methodologies to enhance the precision and efficiency of on-device crystallographic data analysis.

Han et al. (2015) introduced a groundbreaking method in the "Learning Both Weights and Connections for Efficient Neural Networks" research. The method significantly reduced neural networks' computational and memory requirements without compromising accuracy. Addressing the challenge of deploying neural networks on embedded systems, the authors proposed a novel pruning technique that identifies and eliminates redundant connections within the network. By training the network to recognize essential connections, pruning the unimportant ones, and then retraining to fine-tune the weights of the remaining connections, their method achieves a remarkable reduction in the size of the network. Specifically, they demonstrate that their approach can reduce the number of parameters in AlexNet by 9x and in VGG-16 by 13x, effectively compressing these networks without a loss in performance on the ImageNet dataset. The paper outlines a three-step The process begins with initial training to identify crucial connections, followed by pruning less significant connections. It concludes with a retraining phase to optimize the performance of the sparsely connected network. This method cuts down the computational load and memory usage significantly and hints at a deeper understanding of the network's architecture, potentially mirroring biological processes of synaptic pruning in the human brain. The experiments showcase the practicality of their method in reducing network size and complexity and its effectiveness in maintaining and sometimes even enhancing the model's accuracy. This technique presents a compelling approach to making neural networks more efficient, paving the way for their broader deployment in resource-constrained environments[HPTD15].

Ke et al. (2018) introduced an innovative CNN-based approach to analyzing X-ray serial crystallography data. Their method focused on the automated screening of diffraction images, distinguishing between those with significant Bragg spots ("hits") and those without("misses"). This work responds to the challenges posed by the massive datasets generated by X-ray free-electron lasers (XFELs), characterized by their noisy nature and experimental artefacts. With XFEL technology advancing rapidly, enabling the capture of high-resolution diffraction patterns crucial for macromolecular structure determination. Deep learning was proposed, hence, leveraging the CNN to refine the data analysis pipeline, making a case for the method's potential to significantly streamline the processing of serial crystallography data. The methodology employed involves a detailed construction

and training of a CNN designed to identify and learn from the complex visual patterns associated with Bragg spots in crystallography images. The approach is data-driven, relying on a labelled dataset for training the network, and incorporates preprocessing techniques like Local Contrast Normalization (LCN) and data augmentation to enhance learning efficacy. The employed technique achieved high accuracy in image classification across different datasets, evidencing the model's adaptability to various experimental setups and its superiority over traditional image processing methods in terms of efficiency and accuracy. Moreover, the study explores the model's capability to generalize across datasets, a feature vital for its application in diverse XFEL experimental conditions. Finally, the paper underlined the transformative potential of applying CNN-based tools in XFEL research, highlighting the prospects for reduced data processing times and broader applications in crystallography, thereby marking a significant advancement in the field and demonstrating the value of integrating deep learning techniques in scientific investigations[KBY+18].

Abdelouahab et al. (2018) presented a survey on "Accelerating CNN inference on FPGAs: A Survey", which provides a comprehensive overview of the current state-of-the-art techniques for accelerating CNN inference on FPGAs. The study delves into the computational challenges and opportunities presented by FPGAs in the context of CNN inference, highlighting their suitability for handling the streaming nature and high computational workload of CNNs. The acceleration strategies were categorized into algorithmic optimizations, data-path optimizations, and approximate computing techniques, each aimed at exploiting FPGAs' unique reconfigurability and parallelism capabilities. GEMM transformation, Winograd, and Fast Fourier Transform were used for algorithmic optimization, and systolic arrays and SIMD accelerators were analyzed to optimize data paths. The emerging trend of approximate computing, including techniques like fixed and dynamic fixed-point arithmetic, was discussed further to enhance the efficiency of CNN inference on FPGAs. Furthermore, the impact of these acceleration strategies on the performance and efficiency of CNN inference was evaluated by comparing state-of-the-art FPGA-based CNN accelerators. A critical analysis of the trade-offs between computational accuracy, throughput, and power consumption underscores the potential of FPGA-based accelerators in achieving high performance and energy efficiency. The survey serves as a valuable resource for researchers and practitioners in deep learning and hardware acceleration and identifies future directions for advancing FPGA-based CNN inference. This includes exploring more sophisticated data-path optimizations, leveraging the full potential of approximate computing, and developing comprehensive frameworks and tools to facilitate the deployment of optimized CNN models on FPGAs[APSB18].

Mabaso et al. (2018) presented a study titled "Spot Detection in Microscopy Images using Convolutional Neural Network with Sliding-Window Approach" to tackle the challenge of accurately detecting spots within microscopy images—a task pivotal for various biomedical applications. Recognizing the limitations of existing methodologies, particularly in handling noise and background inhomogeneity, a novel approach was introduced, leveraging a convolutional neural network (CNN) in tandem with a sliding-window technique. This method not only automates the detection process but also promises enhanced precision by training a supervised CNN to discern spots within image patches, subsequently employing a sliding window to process testing images. The proposed solution was rigorously compared against two other state-of-the-art CNNs, GoogleNet and AlexNet, using synthetic images to

validate its efficacy. The study articulates a detailed framework for the CNN architecture, comprising three convolutional layers and two fully connected layers, aimed at classifying image patches as containing a spot or not. This initiative marks a significant stride in employing deep learning for refining biological data analysis, underscoring the potential of fine-tuned, pre-trained CNN models in spot detection tasks. The results presented in the paper demonstrate that the proposed methodology not only performs comparably with but, in some instances, surpasses the capabilities of the benchmark CNN models in terms of precision, recall, and F-score values. This finding is particularly noteworthy given the challenges of varying signal-to-noise ratios (SNR) in the synthetic images used for testing. Moreover, the study highlights the practicality of leveraging pre-trained models for biological-image analysis, suggesting that extensive retraining from scratch may not be necessary to achieve high levels of accuracy in spot detection. By effectively navigating the hurdles of imbalance in positive and negative patch distribution through strategic dataset augmentation and implementing measures like overlap suppression, the research sets a new precedent for the automated analysis of microscopy images, paving the way for future advancements in biomedical imaging and diagnostics[MWT18].

Li et al. (2019) explored the potential of overclocking FPGAs to enhance the performance of CNN accelerators in the paper "Squeezing the Last MHz for CNN Acceleration on FPGAs". The research aimed to improve computational speed and efficiency for deep learning tasks by pushing the operational limits of FPGAs beyond their standard clock frequencies. However, this approach introduces a trade-off between increased performance and the risk of timing errors, which could lead to incorrect computations and reduced prediction accuracy. To counterbalance this, the research leverages the inherent fault tolerance of neural networks, proposing a novel methodology that incorporates on-accelerator training to adapt models to the minor inaccuracies induced by overclocking. The paper thoroughly analyses the impacts of overclocking on FPGA-based CNN accelerators, demonstrating that it is possible to boost performance significantly without substantial loss in accuracy with careful management. By utilizing techniques such as runtime accuracy monitoring and dynamic reconfiguration, the authors show that the adverse effects of overclocking, such as timing errors and system instability, can be mitigated. Their experiments, conducted on various neural network architectures, including LeNet, AlexNet, and VGG, illustrate that overclocking can lead to marked improvements in both computational speed and energy efficiency, albeit with minor and manageable decreases in prediction accuracy. This study not only underscores the potential of overclocking as a viable strategy for enhancing the capabilities of FPGA-based neural network accelerators but also highlights the importance of error management strategies in maintaining the balance between performance and reliability[LXX+19].

Abdel et al. (2019) presented a study on an IOT-based image classification titled "Image Classification on IoT Edge Devices: Profiling and Modeling". The approach explores the feasibility and performance of executing image classification algorithms on IoT devices. This research focused on understanding how various factors such as dataset size, image resolution, algorithm type, phase, and device hardware influence energy consumption. Utilizing popular IoT devices like the Raspberry Pi 3 and BeagleBone Black Wireless, the study provides a comprehensive analysis of energy usage across different configurations and machine learning algorithms, including Support Vector Machines (SVM), k-Nearest Neighbors (k-NN), and logistic regression. Extensive experiments were conducted,

manipulating image characteristics and algorithmic factors to observe their impacts on energy efficiency, processing time, and classification accuracy. Key findings indicate that image resolution and dataset size significantly affect energy consumption, with lower resolutions and smaller dataset sizes proving more energy-efficient without substantially compromising accuracy. Additionally, the choice of machine learning algorithm plays a crucial role in energy usage, with each algorithm exhibiting distinct energy profiles based on the task complexity and data characteristics. To aid in the prediction of energy consumption for image classification tasks on IoT devices, the study introduces a machine learning-based model, primarily focusing on a random forest regression approach. This model, trained on the experimental data, aims to predict energy consumption based on various input features, offering valuable insights for optimizing energy efficiency in IoT edge computing scenarios[AMPD20].

Pandey et al. (2020) explored the frontier of time-resolved crystallography (TRX) through the lens of pump-probe experiments conducted at X-ray Free Electron Laser (XFEL) facilities in the work titled "Pump-Probe Time-Resolved Serial Femtosecond Crystallography at X-Ray Free Electron Lasers,". The study underlines the unique capabilities. Of XFELs to capture the femtosecond-timescale dynamics of biological macromolecules, a domain previously inaccessible to conventional X-ray sources. The pump-probe methodology, pivotal in their research, initiates reactions in crystallized proteins with a light pulse (pump) followed by an X-ray pulse (probe) to record the ensuing structural changes. This technique has provided unparalleled insights into the dynamic processes governing biological functions, notably capturing transient states along reaction coordinates in real time. The authors detail the technological advancements and experimental strategies that have propelled TRX to its current prominence. XFEL's ability to deliver intense, femtosecond X-ray pulses enables the capture of diffraction images before radiation damage ensues, embodying the "diffraction before destruction" principle. The paper discusses the development of sophisticated injector systems and detectors that have evolved in tandem with XFEL capabilities, enhancing the resolution and efficiency of data collection. Moreover, it elaborates on the data processing challenges inherent to serial femtosecond crystallography, including the indexing and merging diffraction patterns from numerous microcrystals. Through their comprehensive review, scientific breakthroughs were facilitated. The pump probe highlighted TRX, and this rapidly advancing field's remaining hurdles and prospects were outlined[PPM20].

Je et al. (2021) presented "An Adaptive Row-based Weight Reuse Scheme for FPGA Implementation of CNNs". The challenge of optimizing CNN deployment on FPGAs was addressed by introducing an adaptive row-based weight reuse scheme. The scheme is rooted in the understanding that previous FPGA implementations of CNNs suffered from inefficiencies due to fixed dataflow designs. This led to high on-chip buffer utilization and frequent memory access, particularly when handling layers with small feature map sizes. By proposing a dynamic approach that adjusts the level of row reuse based on the characteristics of each layer, the authors claim significant improvements in buffer utilization and throughput, with their design achieving 994.74 Giga Operations Per Second (GOPS) on U-Net and 1080.9 GOPS on VGG16, while also reducing buffer size by 1.7 times compared to prior works. The work presented a detailed examination of the limitations inherent in traditional no-weight reuse and row-based weight-reuse strategies, highlighting how these can lead to suboptimal use of resources and increased power consumption.

By allowing the weight block sliding range to encompass multiple rows — the number determined by the input and output buffer sizes and the layer's characteristics — the proposed method effectively increases weight reuse and decreases external memory access, critical factors for power efficiency. Implemented on a Xilinx KCU1500 board, the approach demonstrates enhanced performance in terms of GOPS and robustness across different network types and layer sizes, suggesting a versatile and efficient solution for FPGA-based CNN acceleration. This work not only contributes to the technical field by improving the computational efficiency and resource utilization of FPGA-based CNN implementations but also opens avenues for further research on adaptive schemes in hardware acceleration of deep learning models[JNLL21].

Banbury et al. (2021) presented a review titled Benchmarking TinyML Systems: Challenges and Direction, which addresses the crucial gaps in the emerging field of TinyML. As ultra-low-power machine learning hardware begins to unlock new classes of intelligent applications, the absence of a widely accepted benchmark poses a significant hindrance to progress. Benchmarking was highlighted in measuring, comparing, and enhancing the performance of TinyML systems, which is foundational for the field's maturity. The current TinyML landscape was presented, emphasizing the necessity for a fair and comprehensive hardware benchmark that accurately reflects the unique constraints and applications of TinyML workloads. They discuss the collective insights of the TinyMLPerf working group, which comprises over 30 organizations dedicated to this endeavour. The challenges of developing a TinyML benchmark were discussed, including the diverse and dynamic nature of the field, characterized by its low power consumption, limited memory, hardware heterogeneity, and software diversity. They propose a benchmarking suite tailored to TinyML's unique requirements to foster innovation and guide the field's development. This suite accommodates various devices through open and closed divisions, ensuring inclusivity and comparability. The selection of benchmarks, which includes audio wake words, visual wake words, image classification, and anomaly detection, reflects a careful balance between diversity, feasibility, and industry relevance. The challenges in benchmarking TinyML systems were highlighted, and a clear direction was set for establishing a common framework to accelerate the development of TinyML applications, making it a pivotal contribution to the field[BRL+20].

Siang et al. (2021) presented a review titled "Anomaly Detection Based on Tiny Machine Learning" by delving into the burgeoning field of applying Tiny Machine Learning (TinyML) techniques for anomaly detection across various domains. TinyML, characterized by its low power consumption and ability to run on miniature hardware, presents a promising avenue for embedding intelligence directly into devices at the network's edge. This review systematically categorizes and evaluates existing methodologies, highlighting the progress and potential of TinyML in detecting anomalies in time-series data, images, and more. Through a comprehensive state-of-the-art examination, the authors identify key challenges such as the trade-offs between model complexity and computational efficiency, the need for labelled data in unsupervised learning scenarios, and integrating TinyML models into existing IoT ecosystems. Furthermore, the paper emphasizes the significance of anomaly detection in critical applications ranging from healthcare monitoring to industrial maintenance and environmental monitoring, where TinyML can enable real-time, on-device processing, bypassing the latency and privacy issues associated with cloud computing. The review not only underscores the advancements in algorithmic techniques tailored for

constrained environments but also points to the importance of hardware innovations that support the deployment of TinyML. As the paper outlines future research directions, it becomes evident that fostering a symbiotic relationship between algorithmic efficiency and hardware capabilities is a crucial step towards realizing the full potential of anomaly detection through TinyML. The research recommends continued exploration into model compression, energy-efficient computing, and self-supervised learning models to overcome the hurdles in scaling TinyML for widespread adoption[SAA21].

Ray (2022) conducted a comprehensive review titled "A review on TinyML: State-of-the-art and prospects" in the area of the burgeoning field of Tiny Machine Learning (TinyML), emphasizing its critical role in integrating machine learning techniques with embedded devices at the network edge. This work addresses the challenge of deploying machine learning algorithms on devices constrained by power, processing capacity, and memory, aligning with the increasing demand for edge computing and the Internet of Things (IoT). This research offers an insightful exploration of the foundational elements of TinyML, including its basics, constraints, and critical toolsets, thus setting the stage for understanding its applications and the technological innovations driving its development. The review systematically categorizes the enablers of TinyML technology, including hardware platforms and software frameworks, and identifies key strategies for enhancing TinyML's performance and applicability, such as model compression, quantization, and neural architecture search. Furthermore, it presents a range of TinyML applications, from speech and image recognition to anomaly detection, highlighting the technology's versatility and potential impact across various sectors. The research concludes with a discussion of the open challenges facing TinyML, such as energy efficiency and model accuracy. It provides a future roadmap for research, suggesting areas like algorithm optimization and hardware advancements as key to overcoming existing limitations. This review offers a comprehensive overview of the state-of-the-art in TinyML and its prospects for future development[Ray22].

Koziol et al. (2023) presented a novel approach for integrating artificial intelligence titled "Artificial neural network-on-chip and in-pixel implementation towards pulse amplitude measurement," which introduced direct readout channels of X-ray detectors into the pixel. This method leverages an artificial neural network (ANN) to accurately determine the amplitude of pulses generated by incoming photons, a critical parameter for energy measurement in various scientific and industrial applications. Utilizing a multi-layer perceptron architecture with six layers and 420 parameters, the ANN demonstrates an impressive measurement accuracy of 0.34 per cent. The system, designed to process signals from a 6-bit ADC at a 5 MHz sampling rate, shows promising results when tested on an FPGA platform, highlighting the feasibility of this advanced signal processing technique within a standard front-end electronics setup optimized for low-noise pulses. The design methodology combines electronics and computer science expertise to transition the ANN from a high-level Python implementation using TensorFlow to a transistor-level design suitable for ASIC fabrication. The innovation lies in the model's ability to act as a denoising filter and an interpolation operator, potentially adjusting to signal profile changes without hardware modifications. The paper discusses the challenges and solutions related to quantifying and pruning the ANN model for efficient hardware implementation. Tested using FPGA, the design showcases the potential for reducing latency and the amount of data transferred by executing complex signal processing algorithms directly on

the detector chip. This research aligns with the trend of moving computational efforts from centralized servers to edge devices, offering a significant advancement in the field of X-ray photon energy estimation technology[KSK+23].

3.2 Comparison of different approaches and identification of research gaps

The studies above collectively advance the domain of machine learning in scientific and technological applications, each addressing unique challenges and proposing innovative solutions. The 2018 paper by Tsung-Wei Ke et al. showcases a CNN-based method for X-ray crystallography, focusing on efficiently processing massive datasets generated by XFELs. This approach significantly enhances the classification accuracy and efficiency of diffraction image analysis, demonstrating the potential of deep learning to streamline complex scientific data processing tasks. However, despite its advancements, there remains a need for further research into the scalability of such models to accommodate the ever-growing size and complexity of XFEL datasets and the adaptability of these models to new, uncharacterized types of crystallographic data.

On the hardware acceleration front, the survey by Kamel Abdelouahab et al. (2018) and Hyeonseung Je et al. (2021) explore the optimization of CNN inference on FPGAs. Abdelouahab et al. provide a comprehensive overview of algorithmic, data-path, and approximate computing techniques for enhancing CNN performance on FPGAs, highlighting the balance between computational accuracy and efficiency. In contrast, Je et al. propose a specific adaptive row-based weight reuse scheme that directly tackles the inefficiencies in on-chip buffer utilization and memory access. These works underline the critical importance of hardware optimization in deploying efficient neural network models. However, they also underscore the gap in comprehensive, application-specific frameworks that seamlessly integrate these optimizations into end-to-end system designs, especially for real-world applications requiring dynamic adaptability.

The study on TinyML systems by Colby R. Banbury et al. (2021) and the review on TinyML by Partha Pratim Ray (2022) highlight the burgeoning field of machine learning on edge devices. While Banbury et al. focus on the need for benchmarks to propel the development of TinyML, Ray offers a broad overview of the state-of-the-art challenges and prospects in TinyML. These contributions are pivotal in setting the foundation for future TinyML research but also pinpoint the necessity for a unified, comprehensive benchmarking suite that can address the vast diversity of hardware platforms and application scenarios within TinyML, alongside developing strategies to enhance model accuracy and energy efficiency in severely resource-constrained environments.

In the realm of image processing and neural network pruning, the 2018 paper by Matsilele Mabaso et al. and the 2015 study by Song Han et al. address the efficiency of neural networks in microscopy image analysis and general neural network deployment, respectively. Mabaso, et al.'s method, emphasizes the utility of CNNs with a sliding-window approach for spot detection in microscopy images, offering a promising direction for automated biological image analysis. Han et al. introduce a pruning technique to reduce neural network computational requirements, a crucial advancement for deploying deep learning

models in embedded systems. Both studies underscore the ongoing need for methodologies that improve the operational efficiency of neural networks and maintain, or even enhance, their accuracy and reliability, particularly in specialized applications such as biomedical imaging, where precision is paramount.

Lastly, the innovative work by A. Koziol et al. (2023) on integrating artificial neural networks for on-chip and in-pixel pulse amplitude measurement opens new avenues for applying machine learning in hardware. This approach exemplifies the potential for embedded AI to revolutionize signal processing in detector technologies, suggesting. An emerging research avenue in designing and implementing hardware-optimized neural networks for a broader range of sensor technologies.

Across these studies, a common theme emerges. While significant strides have been made in applying machine learning to various domains, from scientific data processing to hardware acceleration and edge computing, each area harbours unique challenges that necessitate further investigation. Future research should aim to bridge these gaps, focusing on scalability, system integration, benchmarking, energy efficiency, and the precision of machine learning models, thereby advancing the field toward more versatile, efficient, and application-specific solutions.

4 Concept and Implementation

4.1 Introduction

This chapter outlines the comprehensive development process of a convolutional neural network (CNN) model tailored for detecting Bragg spots in X-ray crystallography images. Bragg spots, critical for assessing the crystallographic structure of materials, are visible patterns in images that can reveal the arrangement of atoms within a crystal. The methodology detailed here includes the steps from initial data handling to optimizing the model for deployment on resource-constrained TinyML platforms. The process is structured to ensure the model achieves high accuracy in detecting and classifying Bragg spots typical for TinyML environments.

4.2 Material

Software and tools used in the development process are;

1. TensorFlow and Keras: Used for designing and training the CNN model.
2. TensorFlow Lite: Employed for model conversion and optimization post-training, ensuring the model is suitable for deployment on low-power devices.
3. Python is the primary programming language used for developing the model. It is supported by libraries such as NumPy and Matplotlib for data manipulation and visualization.
4. SciKit-Learn: Utilized to split the dataset and evaluate the model performance.

4.3 Methodology

The following methodology was adopted in this research to develop a CNN model tailored for Bragg spot detection in X-ray crystallography images, its optimization for FPGA platforms under TinyML constraints, and the evaluation of its performance through computational efficiency metrics. These steps are detailed as follows:

1. Development of CNN Model for Bragg Spot Detection using Python IDE:
    - Development of a convolutional neural network (CNN) model in a Python-integrated development environment (IDE). The architecture was designed to process high-resolution images and effectively detect Bragg spots.

- Prepare a SLAC Computing Cluster environment with a GPU to handle the computational load effectively.
- Implementation of image preprocessing functions, such as Local Contrast Normalization (LCN) and Global Normalization (GN), directly in the data preparation pipeline to standardize the input data and enhance the model's ability to extract relevant features.

2. Optimization of Developed Model for FPGA Platform (TinyML):
   - Model Reduction Techniques, such as pruning and quantization, are applied to reduce the model size and computational requirements. This ensures that the model fits within the limited memory and processing capabilities of FPGA platforms used in TinyML applications.
   - FPGA Deployment: Adapt and optimize the model to run efficiently on an FPGA platform, focusing on maximizing computational efficiency and minimizing power consumption.
3. Evaluation of Models Using Confusion Matrix and Measure Computation Time:
   - Performance Metrics: Evaluate the model using a confusion matrix to analyze its accuracy in classifying Bragg spots such as 'Hit', 'Miss', or 'Maybe'. This evaluation will help determine the model's precision and recall rates.

4.4 Development of CNN model for Bragg spot detection using Python

This section provides a comprehensive account of the development process for a CNN model tailored to detect Bragg spots in crystallographic images. Bragg spots are critical indicators in materials science, used for assessing crystal structures through diffraction patterns.

4.4.1 Problem Definition

The primary objective of the TinyML model is to detect and classify Bragg spots accurately within X-ray crystallography images. Bragg spots, visible as distinct bright spots or patterns on the images, are essential for determining the crystallographic structure of materials.

1. Inputs: The model inputs are 2000 unprocessed (raw) images from crystallography experiments saved in dot h5 format (.h5) with dimensions 1765 x 1765, as shown in 4.1.

   Each image is a grayscale image in which the intensity of each pixel is crucial, as these intensities represent potential Bragg spots. The typical input format is a high-resolution 2D array in which each element represents pixel intensity.

2. Outputs: The model outputs a classification for each detected spot within the image. The classifications include:

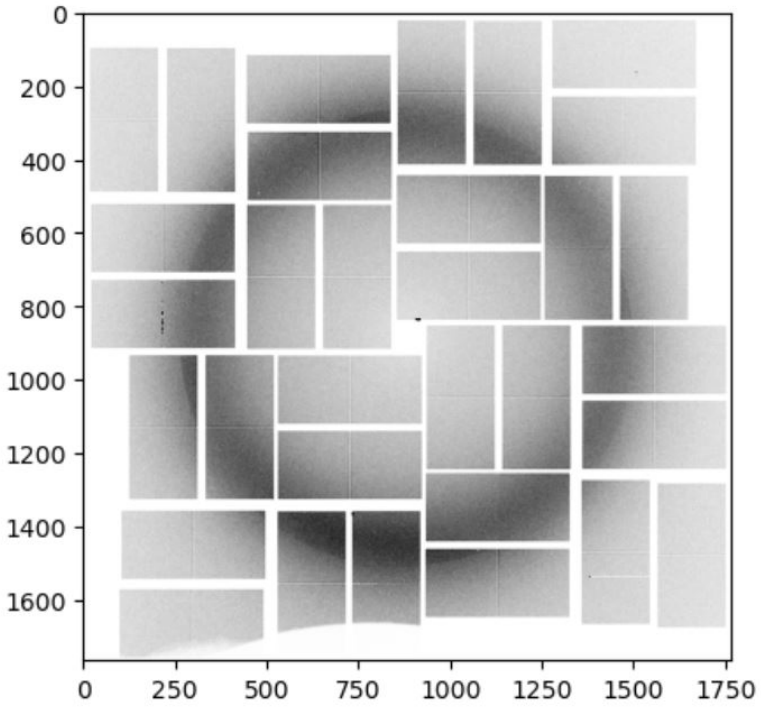

Figure 4.1: Raw Image

- 'Hit': Indicates a high-confidence Bragg spot corresponding to a crystallographic feature.
- 'Miss': Denotes areas where no relevant Bragg spots are detected.
- 'Maybe': Suggests spots where there is some uncertainty—these may require further analysis or verification by a human expert.

This model thus serves a dual purpose, enhancing the efficiency of crystallographic studies by automating the detection of Bragg spots and improving the reliability of material analysis by ensuring consistent and accurate spot classification. By automating Bragg spot detection, researchers can save significant time and reduce the subjectivity involved in manually identifying these spots.

4.4.2 Data Collection

The data collection process for developing a TinyML model to detect Bragg spots involves a series of meticulous steps to ensure the integrity and usability of the data for machine learning applications. The methods employed for data acquisition, the specific types of data handled, the volume of data required for effective training, and the necessary preprocessing techniques are discussed.

1. Methods for Data Acquisition: Data for this project is primarily obtained from X-ray crystallography experiments conducted in controlled laboratory settings. These experiments generate high-resolution images captured in .h5 format, which is known for its ability to store complex and large datasets efficiently.
2. Data Types and Volume: The datasets consist of 2000 high-resolution images in .h5

format, containing detailed diffraction patterns from various crystalline structures. These images are crucial for training the model to accurately recognize and classify Bragg spots.

Below are the several preprocessing steps undertaken for the preparation of the images for the TinyML model;

1. Extraction and Conversion:
   - The images are first extracted from the .h5 files, which involves reading the datasets from the hierarchical data format designed to store large amounts of scientific data.
   - After extraction, the images are saved in a more accessible format (PNG) for subsequent processing steps. This conversion simplifies the handling and manipulation of the images in the later stages of the model development process.
2. Downsampling: For the purpose of comparison, three cases of resampling were considered for this research work. The raw images for the CSPAD detector (LG36, 87) were first resampled from 1765x1765 to 960 x 960, 480 x 480 and 300 x 300 with respective cropping of 720x720, 300x300 and 100 x 100 before it was fed for model train as illustrated in the image 4.2.

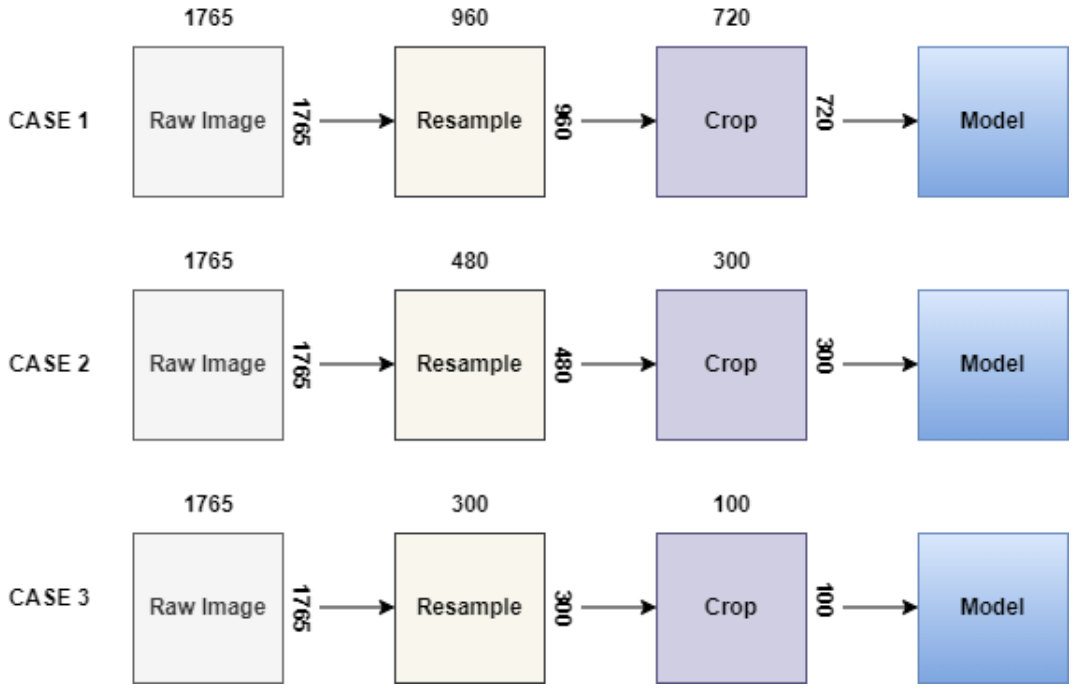

Figure 4.2: Preprocessing

The second and third cases were created to optimize the model, i.e., to manage the computational load and adapt the images for the TinyML application. In this case, the original images, typically sized at 1765x1765 pixels, are resampled to 480x480 pixels and later cropped to 300x300, as illustrated in cases 2 and 3 of image 4.2. This downsampling uses the 'Image.Resampling.BILINEAR' method from the Pillow library, which employs a bilinear interpolation algorithm.

- Bilinear Interpolation Algorithm: Bilinear Interpolation Algorithm: Bilinear interpolation is a resampling technique that uses a weighted average of the four nearest pixel values to estimate a new pixel intensity. It provides a smoother image than the nearest neighbour methods shown in (4.1). The mathematical representation of bilinear interpolation for image resizing is given by:

$$f(x,y) = f(Q_{11})\cdot(1-\Delta x)\cdot(1-\Delta y)+f(Q_{21})\cdot\Delta x\cdot(1-\Delta y)+f(Q_{12})\cdot(1-\Delta x)\cdot\Delta y+f(Q_{22})\cdot\Delta x\cdot\Delta y \tag{4.1}$$

Where:

f(x,y) is the interpolated pixel value

$(Q_{11}), (Q_{12}), (Q_{13}), and (Q_{14})$ are the pixel values of the top-left, top-right, bottom-left, and bottom-right corners of the pixel square surrounding the new pixel position.

Δx and Δy are the distances from the unknown point to the left and top sides of the pixel square, respectively.

This approach reduces the image size and helps maintain the essential features necessary for detecting Bragg spots, making it particularly suitable for the constrained resources of TinyML devices.

4.4.3 Model Selection

The selection of an appropriate model architecture is crucial in ensuring the effectiveness of a TinyML application for detecting Bragg spots in X-ray crystallography images. The structured approach to model selection is outlined by focusing on the computational and operational constraints of TinyML devices and the unique demands of processing high-resolution imaging data.

1. **Criteria for Model Selection**

   When selecting a model for TinyML applications, especially for tasks like Bragg spot detection in X-ray crystallography images, the CNN is often chosen due to its strong performance in image processing and analysis tasks. CNNs are particularly adept at reducing the spatial volume of input images through convolutional and pooling layers, which inherently improves computational efficiency—crucial for the limited processing capabilities of TinyML devices.

2. **Why CNNs Maximize Computational Resource Efficiency**

   - Sparse Connectivity: Unlike fully connected networks, CNNs leverage local connectivity through convolutional layers. This means that each neuron in a layer is connected only to a small region of the layer before it, reducing the number of computations needed.
   - Shared Weights: In CNNs, all neurons in a single feature map share the same filter or kernel, unlike in dense layers where each output neuron has its own set

of weights. This significantly reduces the number of parameters, leading to less memory usage and faster computation.

- Hierarchical Feature Learning: CNNs are structured to extract low-level features in the initial layers (e.g., edges and textures) and progressively build up to high-level features in deeper layers. This hierarchical feature extraction method allows CNNs to learn complex patterns more efficiently. Efficiency in Depth: With each subsequent pooling layer, the spatial dimensions of the feature maps are reduced, decreasing the computational load on the network in deeper layers.

3. **Real-Time Analysis Optimization**

   Ensuring the model runs efficiently with low latency is critical for real-time analysis on edge devices. Here are specific techniques to further enhance a CNN's efficiency for TinyML:

   - Quantization: This reduces the precision of the model's weights from floating points to integers, decreasing the model size and speeding up inference by lowering the computational complexity.
   - Pruning: Removes redundant or non-informative weights from a network, which can dramatically reduce the number of necessary computations and the model size without significantly impacting performance.

   Optimizing a CNN for computational efficiency involves selecting the exemplary architecture and applying techniques like quantization and pruning. These methods align with the primary objective of deploying models on TinyML devices, where the balance between accuracy, computational speed, and power efficiency dictates the feasibility of real-time edge applications.

   Mathematical Foundation of Convolutional Layers:

   Convolutional layers in CNNs use a mathematical operation called convolution, defined by the equation:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{4.2}$$

   where I is the input image,

   K is the kernel/filter, and

   S is the feature map produced

   This operation captures local dependencies in the image, essential for identifying Bragg spots.

4. **Model Architecture:**

   The CNN model consists of multiple convolutional layers, followed by batch normalization and rectified linear unit (ReLU) activation functions to introduce non-linearity. The design incorporates multiple pooling layers to reduce dimensionality and focus on the most relevant features. The final layers include a dense output layer with

softmax activation to classify each image based on the presence and type of Bragg spots, as shown in 4.3.

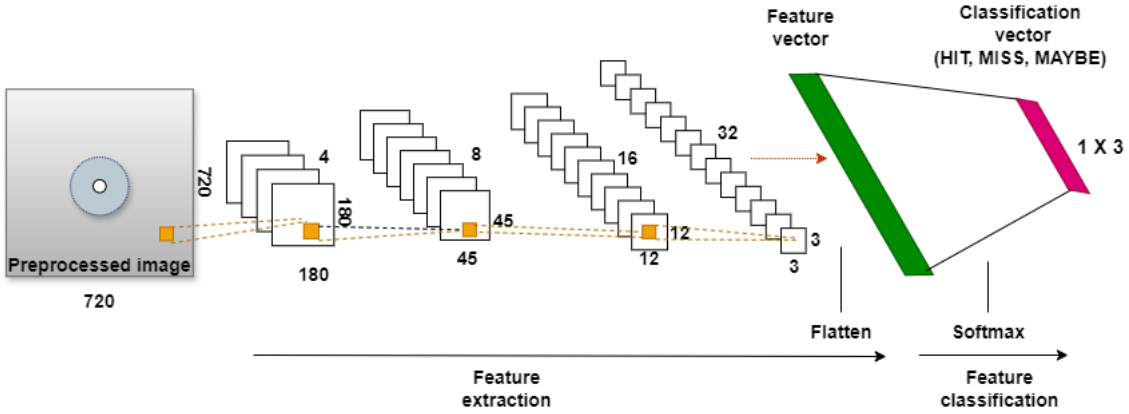

Figure 4.3: feature extraction and classification

The model structure is optimized for performance in a resource-constrained environment, considering the computational and memory constraints typical in TinyML applications.

5. **Feature Engineering**

Feature engineering is a critical phase in the development of the CNN model for Bragg spot detection, especially when dealing with high-resolution X-ray crystallography images. This process involves transforming raw data into a format better suited for the model, enhancing its ability to learn meaningful patterns from the images.

Techniques and Methods includes;

- Local Contrast Normalization (LCN):

Local Contrast Normalization (LCN) enhances the contrast in local regions in each image, helping to highlight Bragg spots against varied background intensities. LCN operates by normalizing the pixel values in a local neighbourhood around each pixel.

$$\text{LCN}(I_{x,y}) = \frac{I_{x,y} - \mu_{x,y}}{\sigma_{x,y} + \epsilon} \tag{4.3}$$

Where, $I_{x,y}$ is the intensity of the pixel at position (x, y)

$\mu_{x,y}$ is the mean intensity of pixels in a local neighbourhood around (x, y),

$\sigma_{x,y}$ is the standard deviation of intensities in the same neighborhood, and

ϵ is a small constant to prevent division by zero.

- Global Normalization (GN):

In addition to local normalization, global normalization is employed to standardize the entire image. This step involves adjusting the pixel values across the entire image to have a standardized range and distribution, facilitating consistent processing across multiple images.

$$\mathrm{GN}(I) = \frac{I - \mu}{\sigma} \tag{4.4}$$

where,

I represent the pixel intensities of the entire image,

μ is the global mean of pixel intensities and

σ is the global standard deviation.

After applying the global and local normalization on all the images (image 1751 as a case study), the features become more pronounced, as shown in Figure 4.4 below.

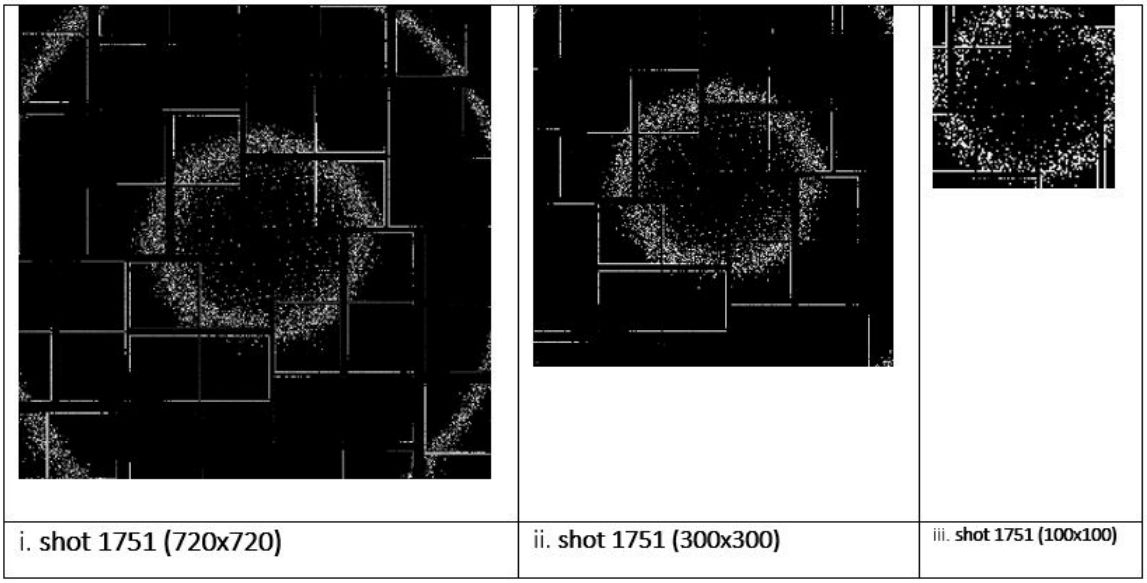

Figure 4.4: Local Contrast and Global Normalization

6. **Model Training**

Model training is a fundamental phase in the development of the CNN for Bragg spot detection. It ensures that the model accurately interprets X-ray crystallography images and effectively identifies Bragg spots. This part details the setup and methodology employed during the training process, including the environment, data handling, and validation strategies.

a. Training Environment Setup

- Hardware Configuration: The model is trained on a local machine equipped with a Graphics Processing Unit (GPU), accelerating the computations involved in deep learning training processes. A GPU is crucial for handling large volumes of high-resolution images and complex computations efficiently.
- Software and Libraries: TensorFlow and Keras: The model is developed using TensorFlow, particularly its Keras API, which provides a high-level interface for neural network design and training. These libraries support seamless integration with GPU hardware for optimized performance. Data Handling Utilities:

Libraries such as NumPy for numerical operations and Pillow for image preprocessing prepare the dataset for training. Data Handling Utilities: Libraries such as NumPy for numerical operations and Pillow for image preprocessing are used to prepare the dataset for training.

- Development Environment: A Python development environment is used, which is conducive to data science and machine learning due to its extensive ecosystem of libraries and frameworks. The environment is set up to ensure reproducibility and ease of experimentation.

b. Model Summary and Data Preparation

- Model summary: The first step involves passing the input image through convolutional layers, which apply filters to extract low-level features like edges and shapes. The extracted features are then passed through pooling layers that reduce the spatial dimensionality while retaining vital information. Finally, the flattened feature maps are fed into fully connected layers, which perform the classification task by mapping the features to the desired output labels. Figure4.5 shows a table of the model.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 96, 96, 3)	78
activation (Activation)	(None, 96, 96, 3)	0
max_pooling2d (MaxPooling2D)	(None, 48, 48, 3)	0
conv2d_1 (Conv2D)	(None, 44, 44, 3)	228
activation_1 (Activation)	(None, 44, 44, 3)	0
max_pooling2d_1 (MaxPooling2D)	(None, 22, 22, 3)	0
conv2d_2 (Conv2D)	(None, 18, 18, 3)	228
activation_2 (Activation)	(None, 18, 18, 3)	0
max_pooling2d_2 (MaxPooling2D)	(None, 9, 9, 3)	0
conv2d_3 (Conv2D)	(None, 5, 5, 3)	228
activation_3 (Activation)	(None, 5, 5, 3)	0
flatten (Flatten)	(None, 75)	0
dense (Dense)	(None, 3)	228
activation_4 (Activation)	(None, 3)	0

Total params: 990 (3.87 KB)
Trainable params: 990 (3.87 KB)
Non-trainable params: 0 (0.00 B)

Figure 4.5: TinyML Model Summary

- Dataset Splitting: The preprocessed image dataset is divided into training,

validation, and test sets. Typically, 80 per cent of the training data is used for training, and 20 per cent is used for validation and testing. This split is crucial for evaluating the model's performance and generalizability on unseen data.

c. Model Training Process

- Batch Processing: Batch Processing: Training is performed in batches, typically with a batch size of 64 images. Batch processing efficiently uses memory and helps stabilize the gradient updates during training.
- Optimization Algorithm: An Adam optimizer is chosen for its effectiveness in handling sparse gradients and adaptive learning rate capabilities. The learning rate is initially set to 0.0001, balancing training speed and convergence stability.
- Loss Function: The model uses sparse categorical cross-entropy as the loss function, suitable for multi-class classification tasks where each class is mutually exclusive.
- Training Epochs: The model is trained for sufficient epochs to ensure convergence, typically around 20. Each epoch involves a complete pass over the entire training dataset.

d. Validation and Performance Metrics

- Validation During Training: The validation set monitors the model's performance at the end of each epoch. This ongoing validation helps detect overfitting early and adjust training parameters dynamically.
- Metrics: Accuracy is the primary metric for assessing performance. Confusion matrices and classification reports provide detailed insights into the model's performance across different classes (Hit, Miss, Maybe).

7. **Model optimization:** Model optimization in the context of TinyML, particularly for a CNN designed for Bragg spot detection, is critical due to the stringent resource constraints of edge devices. Optimization can be broadly categorized into two phases: pretraining Optimization and Post-Training Optimization. Each phase has distinct strategies and techniques to enhance the model's compactness, speed, and efficiency without significantly compromising accuracy.

   Pretraining Optimization involves modifying the model architecture and its parameters before the training begins. This is crucial for ensuring the model is inherently efficient and manageable on low-power devices.

   - Pruning: Pruning involves systematically removing weights (or connections) in the neural network that contribute the least to the output. This can be achieved by setting the smallest weights to zero, effectively reducing the complexity of the model. If W represents the weights matrix, pruning might involve setting weights w in W to zero if $|w| < \epsilon$, where ϵ is a threshold defining the "smallness" of weights. Pruning reduces the number of active parameters, thereby decreasing memory usage and computational cost during inference.
   - Quantization: Quantization involves reducing the precision of the numerical parameters (weights, activations) from floating-point to lower bit-width integers,

such as 16-bit, 8-bit, or even binary/ternary values. This reduction in precision leads to smaller model sizes and faster computation, as integer operations are less computationally intensive than floating-point operations.

- Knowledge Distillation: A technique where a smaller "student" model is trained to emulate the behaviour of a more prominent "teacher" model. The student learns to replicate the teacher's output distribution (soft targets), effectively capturing its generalization ability. This enables the deployment of smaller, more efficient models that perform comparably to their larger counterparts.
- Low-Power Operations: Choosing efficient operations such as ReLU and max pooling helps minimize computational demands. ReLU provides a non-linear activation function that involves simple thresholding at zero, which is computationally inexpensive. Max Pooling reduces the spatial dimensions of the input feature maps, thereby decreasing the amount of computation needed for subsequent layers.
- Adaptation to Memory Constraints: The final structure of the CNN is carefully designed to fit within the memory constraints of the intended TinyML hardware. This includes optimizing the number of layers, the size of each layer, and the overall architecture to ensure that the model does not exceed the available memory while maintaining sufficient capacity to learn and perform effectively.

4.5 Implementation of CNN for TinyML-Based Model development

The following parameters were defined during the implementation, as shown in 4.1. which plays a crucial role in defining the architecture, functionality, and performance of the CNN model developed for Bragg spot detection. The details include various layers and their configurations, training setup, and preprocessing methods which collectively ensure that the model is both accurate and efficient.

These parameters were used to define the algorithm used to implement this work. The algorithm outlined in "4.6: Pseudocode for the Development of the TinyML Model" is a comprehensive guide for the systematic construction of a CNN aimed at detecting Bragg spots in X-ray crystallography images. This algorithm encapsulates the entire lifecycle of a machine learning model, from initialization and data preparation to training optimizations specifically tailored for deployment on resource-constrained TinyML platforms.

1. Overview of the Algorithm Steps:

- Initialization: The algorithm begins by setting the working environment, specifying directories for image data and label storage. This foundational step ensures that all subsequent operations are performed relative to a defined base directory, streamlining file access and management.
- Data Handling: Loading and Preprocessing: Labels are loaded and categorized into numeric formats, facilitating more straightforward computation during the training process. Images are preprocessed using Local Contrast Normalization and

Table 4.1: Parametric definition of training variables

Parameter	Value	Description
Convolutional Layers	4 layers (3, 3, 3, 3 filters)	Number of convolutional layers with filter sizes 3 on all layers
Kernel Sizes	(5x5)	Sizes of the kernels used in convolutional layers
Strides	(2, 2)	Strides for convolution operations
Activation Function	ReLU	Activation function used in convolutional layers
Pooling Layers	MaxPooling (2x2)	Types and sizes of pooling layers used
Padding	'valid', 'same'	Padding methods used in convolutional and pooling layers
Output Layer	Softmax	Activation function used in the output layer for classification
Number of Classes	3	Number of output classes ('HIT', 'MISS', 'MAYBE')
Optimization Algorithm	Adam	Optimization algorithm used during training
Learning Rate	4e-4	Initial learning rate for the optimizer
Loss Function	Sparse Categorical Crossentropy	Loss function used for multi-class classification
Batch Size	64	Number of training examples utilized in one iteration
Epochs	10	Total number of times the training data is used to update the model
Image Preprocessing	LCN, GLN	Local Contrast Normalization and Global Normalization applied to images
Input Image Format	Grayscale, 1765x1765 resized/cropped to 300x300/100x100	Initial and preprocessing size of input images

Global normalised techniques to enhance feature visibility and consistency across the dataset.

- Resizing: Images are resized using bilinear interpolation to a standard dimension, optimizing them for efficient processing while maintaining essential details necessary for accurate analysis.
- Data Augmentation: To enhance the model's robustness against overfitting and improve its ability to generalize, the algorithm applies transformations such as random cropping and rotation to the images. This step diversifies the training data and simulates real-world variations the model might encounter in deployment.
- Model Construction (architecture): The CNN architecture is defined by multiple layers, including convolutional, max pooling, and dense layers, structured to effectively capture and classify complex patterns.
- Model Training: The model uses the prepared datasets to optimize accuracy and reduce loss through a well-chosen optimizer and loss function.
- Quantization: The model undergoes quantization, which simplifies the model by reducing the precision of its calculations, thus making it lighter and faster—qualities essential for deployment on TinyML devices.

This algorithm is not just a procedural guide but also a strategic framework that ensures each step is optimized for efficiency and effectiveness. The flowchart in Figure 4.7 also provides the procedure by which the code is implemented. By meticulously detailing each step, from data handling to model optimization, the algorithm provides a clear path to achieving high performance in a constrained environment, which is crucial for real-time applications like Bragg spot detection in X-ray crystallography.

4.6 Evaluation of Models Using Confusion Matrix and Measurement of Computation Time

Evaluating the performance of the CNN model developed for detecting Bragg spots is critical to ensuring its effectiveness and suitability for deployment on TinyML platforms. The evaluation process helps us understand how well the model performs on the task at hand and highlights areas for improvement.

4.6.1 Evaluation Metrics

1. Accuracy Metrics:
    - Precision: Precision is the ratio of correctly predicted positive observations to the total predicted positives. It is a measure of a classifier's exactness. The higher the precision, the more accurate the model is in classifying a sample as positive.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{4.5}$$

Where,

```
BEGIN
// Initialize and set the working directory
SET working_directory to '/Users/jaeyounglee/Downloads/LG36'
CHANGE directory to working_directory

// Define paths
DEFINE hdf5_file as 'r0087_2000.h5'
DEFINE raw_image_path as 'raw_images'
DEFINE preprocessed_image_path as 'LG36_Preprocessed'
DEFINE label_file as 'LG36.txt'

// Convert HDF5 images to PNG format
FUNCTION convert_hdf5_to_png(hdf5_file, raw_image_path)
LOAD HDF5 file
FOR each image in HDF5 file
CONVERT and SAVE image to raw_image_path
END FOR
END FUNCTION

// Preprocess images
FUNCTION preprocess_image(path)
OPEN and NORMALIZE image
RESIZE and CROP image
RETURN preprocessed image
END FUNCTION

// Apply preprocessing to all images
FUNCTION preprocess_all_images(raw_image_path, preprocessed_image_path)
FOR each file in raw_image_path
PREPROCESS and SAVE image to preprocessed_image_path
END FOR
END FUNCTION

// Load preprocessed images and labels
FUNCTION load_preprocessed_images(preprocessed_image_path)
LOAD images from preprocessed_image_path
RETURN images as numpy array
END FUNCTION

FUNCTION load_and_map_labels(label_file)
LOAD and MAP labels to numeric values
RETURN numeric_labels
END FUNCTION

// Split data into training and test sets
FUNCTION split_data(images, labels)
SPLIT images and labels into train and test sets
RETURN X_train, X_test, y_train, y_test
END FUNCTION

// Create CNN model
FUNCTION create_cnn_model(input_shape, num_classes)
INITIALIZE Sequential model
ADD Conv2D, ReLU, MaxPooling layers
ADD Flatten and Dense layers with softmax activation
RETURN model
END FUNCTION

// Train the model
FUNCTION train_model(model, X_train, y_train, X_test, y_test)
COMPILE and FIT model on X_train and y_train
EVALUATE model on X_test and y_test
PRINT test accuracy
RETURN trained_model
END FUNCTION

// Main Execution Block
convert_hdf5_to_png(hdf5_file, raw_image_path)
LOAD numeric_labels using load_and_map_labels(label_file)
preprocess_all_images(raw_image_path, preprocessed_image_path)
preprocessed_images = load_preprocessed_images(preprocessed_image_path)
X_TRAIN, X_TEST, Y_TRAIN, Y_TEST = split_data(preprocessed_images, numeric_labels)
MODEL = create_cnn_model((100, 100, 1), 3)
TRAINED_MODEL = train_model(MODEL, X_TRAIN, Y_TRAIN, X_TEST, Y_TEST)
SAVE TRAINED_MODEL to 'TinyM.h5'
PREDICT and GENERATE confusion matrix
PRINT classification report
END
```

Figure 4.6: Pseudocode for the development of the TinyML model

TP is the number of true positives, and FP is the number of false positives.

- Recall (Sensitivity): Recall is the ratio of correctly predicted positive observations to all observations in actual class. It is a measure of a classifier's completeness.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{4.6}$$

where;

TP is the number of true positives, and FN is the number of false negatives.

- F1-Score: The F1 Score is the weighted average of Precision and Recall. Therefore, this score takes both false positives and false negatives into account. It is

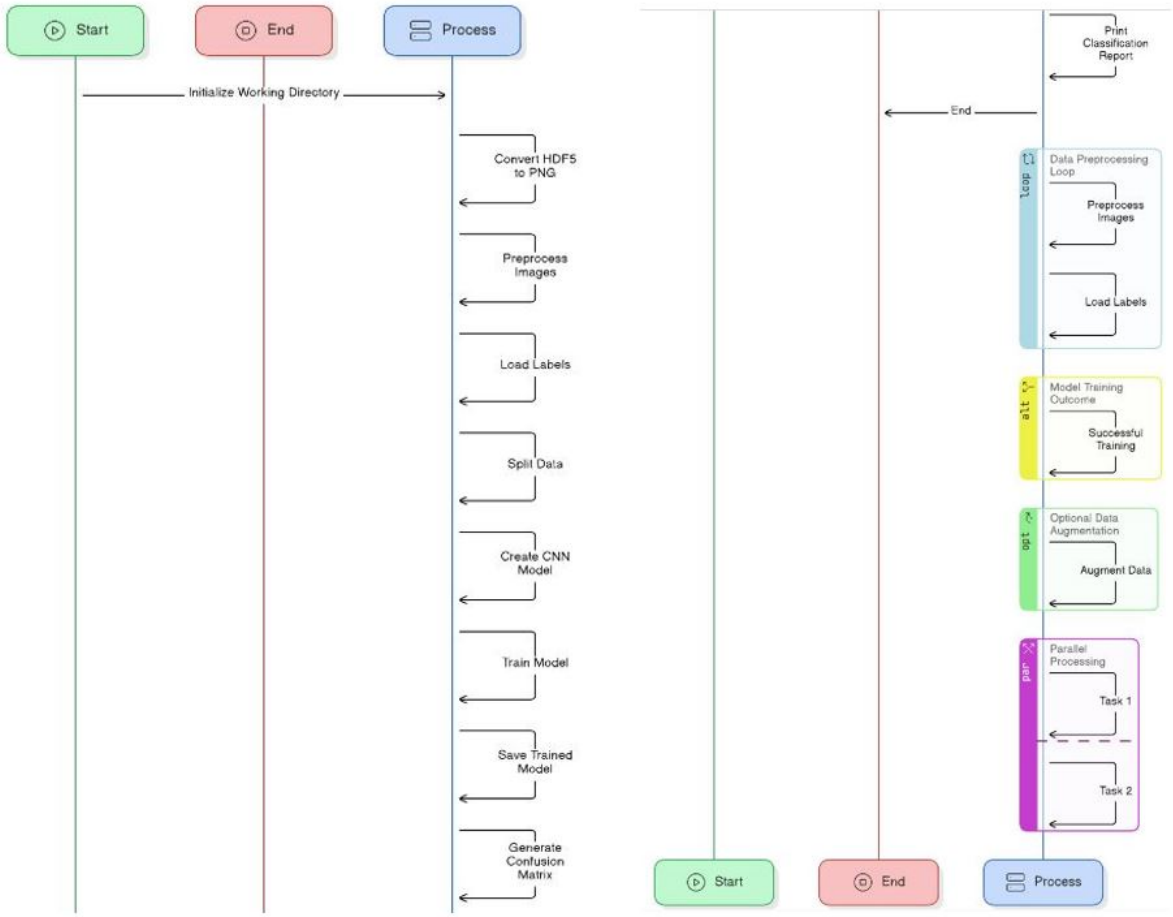

Figure 4.7: TinyML Model Flowchart

beneficial when false positives and negatives have different costs.

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.7}$$

2. Confusion Matrix: A confusion matrix is a table often used to describe the performance of a classification model on a set of test data for which the true values are known. It provides insights into the types of errors the model makes. The matrix compares the actual target values with those the machine learning model predicted, allowing for a detailed analysis of the true positive, negative, false positive, and false pessimistic predictions.

3. Efficiency Metrics:

   - Computation Time: The model's total Time to make predictions on a given dataset. It is a critical factor for real-time applications, where decisions need to be made rapidly.

$$\text{Computation Time} = \text{End Time} - \text{Start Time} \tag{4.8}$$

4.6.2 Significance of Metrics

These metrics provide a comprehensive understanding of the model's performance from various angles:

1. Accuracy metrics ensure the model's predictions are precise and reliable, reducing the likelihood of false detections, which could be costly in real-world applications.
2. The confusion matrix intuitively explains the error distribution across classes, helping to fine-tune the model and improve its accuracy.
3. Efficiency metrics: Ensure that the model operates within the acceptable limits of computation time and power usage, which are critical for sustainable deployment on TinyML devices.

By meticulously optimizing and evaluating these strategies, the CNN model for Bragg spot detection achieves a balanced trade-off between accuracy, efficiency, and operational constraints. This makes it well-suited for deployment in resource-constrained environments typical of TinyML platforms. This thorough evaluation helps validate the model's effectiveness in real-world scenarios, ensuring it meets the rigorous demands of scientific applications.

5 Evaluation

5.1 Introduction

Section 5.1 of this chapter presents the evaluation of our work using confusion matrixes. Section 5.2 presents a comparison evaluation in tabular form in size and accuracy.

5.2 evaluation using confusion matrix

The confusion matrix from the evaluation of the model provided a comprehensive overview of the performance of the Convolutional Neural Network (CNN) model trained for Braggspot detection. The model was trained on 80 percent of the dataset, totalling 1600 samples, and subsequently evaluated using the remaining 20 percent (400) of the data as test data. The confusion matrix presents 3 cases shown in figure 5.1.

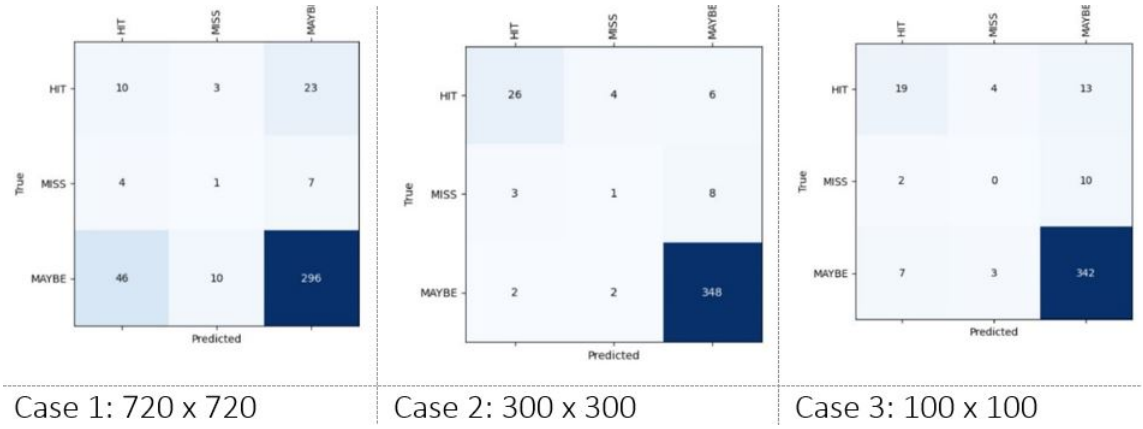

Figure 5.1: Confusion matrix showing model performance

The model evaluation based on the confusion matrix reveals several key insights: From the matrix for Cases 1, 2, and 3, respectively, it is observed that the model correctly predicted 10, 26, and 19 instances of Hit. 296, 348, 342 instances of Maybe, but only 1, 1, 0 instances of Miss. This suggests that the model performs well in classifying Hit and Maybe instances in all three cases but struggles with accurately identifying missed instances. The confusion matrix also highlights instances of misclassification by the models. These misclassifications indicate areas where the model may require further refinement to improve its accuracy. Despite the observed misclassifications, the model's overall performance can be assessed by calculating metrics such as precision, recall, and F1-score for each class. These metrics provide a more nuanced understanding of the model's performance beyond simple accuracy calculations. The confusion matrix analysis provides valuable insights into the strengths and weaknesses of the CNN model trained for Braggspot detection. While the model

Table 5.1: Image dimension (Pixel)

Dimension	**720x720**	**300x300**	**100x100**
Accuracy (percent)	94.2	90	79
Size (kb)	787	155	63.6

demonstrates proficiency in certain classes, particularly Hit and Maybe, there is room for improvement, particularly in accurately identifying Miss instances. Future work could focus on refining the model architecture, addressing class imbalances, and implementing advanced techniques to enhance overall classification performance.

5.3 Validation

The validation helps select an optimal image dimension for the CNN model training. Table 5.1, presented above, shows the variation in accuracy and model size with respect to the model dataset dimension.

The results indicate that decreasing image dimensions leads to a reduction in accuracy and model size. Although Case 1 achieves the highest accuracy, the model size of 787 KB is too large for deployment on tinyML devices due to resource constraints. Case 3 offers an optimal model size of 63.6 KB with an accuracy of 79 percent and an F1-score of 0.8. This analysis underscores the trade-offs between model size and accuracy, guiding the selection of an appropriate model for deployment in resource-constrained environments.

6 Summary and Outlook

This section encapsulates the essential findings and contributions of the research on utilizing TinyML for Bragg spot detection in X-ray crystallography images. It also explores TinyML's future potential in various applications and outlines areas for further research to enhance its effectiveness and deployment in resource-constrained environments.

6.1 Summary

This thesis investigates the application of TinyML to the detection of Bragg spots in X-ray crystallography images, a critical task in materials science. The primary aim was to develop a CNN optimized for FPGA platforms to enable real-time, energy-efficient crystallography data analysis.

Key Findings and Contributions:

1. Model Development: A CNN was developed using Python, specifically designed to handle high-resolution crystallography images and accurately detect Bragg spots. This model leverages image preprocessing techniques (LCN and GN) to enhance feature extraction.
2. Optimization for FPGA: The CNN model was optimized using model reduction techniques such as pruning and quantization to reduce the model, ensuring it fits within the memory and processing constraints of FPGA platforms. This step is crucial for deploying the model in low-power environments typical of TinyML applications.
3. Performance Evaluation: The optimized model was evaluated using a confusion matrix to measure accuracy, precision, and recall in classifying Bragg spots. The results demonstrated that the TinyML approach could significantly enhance the efficiency and accuracy of Bragg spot detection compared to traditional methods.

6.2 Outlook

The successful deployment of TinyML for Bragg spot detection opens numerous avenues for future research. Potential applications include:

1. Expanded Use in Other Scientific Imaging: Extending the TinyML approach to other types of scientific imaging, such as medical diagnostics or environmental monitoring, where real-time analysis is critical.
2. Advanced Model Optimization: Further research into advanced model optimization techniques could improve the performance and efficiency of TinyML applications.

This includes exploring new pruning and quantization methods or developing novel algorithms explicitly tailored for resource-constrained environments.

3. Collaborative Data Collection: Leveraging crowdsourcing and collaborative platforms for data collection and annotation could significantly enhance the training datasets, leading to more robust and generalizable models.

In conclusion, this research demonstrates the feasibility and advantages of using TinyML for Bragg spot detection in X-ray crystallography, paving the way for more efficient and accessible scientific data analysis tools.

Bibliography

[AMPD20] Salma Abdel Magid, Francesco Petrini, and Behnam Dezfouli. Image classification on iot edge devices: profiling and modeling. *Cluster Computing*, 23(2):1025–1043, 2020.

[APSB18] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Serot, and François Berry. Accelerating cnn inference on fpgas: A survey. *arXiv preprint arXiv:1806.01683*, 2018.

[BBA22] N Bhavatarini, Syed Muzamil Basha, and Syed Thouheed Ahmed. *Deep learning: Practical approach*. MileStone Research Publications, 2022.

[Bie16] Grzegorz Bieszczad. Soc-fpga embedded system for real-time thermal image processing. In *2016 MIXDES - 23rd International Conference Mixed Design of Integrated Circuits and Systems*. IEEE, June 2016.

[BRL+20] Colby R Banbury, Vijay Janapa Reddi, Max Lam, William Fu, Amin Fazel, Jeremy Holleman, Xinyuan Huang, Robert Hurtado, David Kanter, Anton Lokhmotov, et al. Benchmarking tinyml systems: Challenges and direction. *arXiv preprint arXiv:2003.04821*, 2020.

[BRT+21] Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau, et al. Mlperf tiny benchmark. *arXiv preprint arXiv:2106.07597*, 2021.

[DV22] Dasu Dasari and P.Suresh Varma. Employing various data cleaning techniques to achieve better data quality using python. In *2022 6th International Conference on Electronics, Communication and Aerospace Technology*. IEEE, December 2022.

[FS97] W. Furey and S. Swaminathan. *[31] PHASES-95: A program package for processing and analyzing diffraction data from macromolecules*, page 590–620. Elsevier, 1997.

[HK17] Akinori Hidaka and Takio Kurita. Consecutive dimensionality reduction by canonical correlation analysis for visualization of convolutional neural networks. volume 2017, pages 160–167, 12 2017.

[HPTD15] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28, 2015.

[HVD15] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[JgZOC23] Fabian Just, Chiara ghinami, Jan Zbinden, and Max Ortiz-Catalan. Deployment of machine learning algorithms on resource-constrained hardware platforms for prosthetics. December 2023.

[JGZOC24] Fabian Just, Chiara Ghinami, Jan Zbinden, and Max Ortiz-Catalan. Deployment of machine learning algorithms on resource-constrained hardware platforms for prosthetics. *IEEE Access*, 2024.

[JNLL21] Hyeonseung Je, Duy Thanh Nguyen, Kyujoong Lee, and Hyuk-Jae Lee. An adaptive row-based weight reuse scheme for fpga implementation of convolutional neural networks. In *2021 36th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC)*, pages 1–4. IEEE, 2021.

[JRPK+22] Vijay Janapa Reddi, Brian Plancher, Susan Kennedy, Laurence Moroney, Pete Warden, Lara Suzuki, Anant Agarwal, Colby Banbury, Massimo Banzi, Matthew Bennett, Benjamin Brown, Sharad Chitlangia, Radhika Ghosal, Sarah Grafman, Rupert Jaeger, Srivatsan Krishnan, Maximilian Lam, Daniel Leiker, Cara Mann, and Dustin Tingley. Widening access to applied machine learning with tinyml. *Harvard Data Science Review*, 01 2022.

[KBY+18] T-W Ke, Aaron S Brewster, Stella X Yu, Daniela Ushizima, Chao Yang, and Nicholas K Sauter. A convolutional neural network-based screening tool for x-ray serial crystallography. *Journal of synchrotron radiation*, 25(3):655–670, 2018.

[Kri18] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

[KSH17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, May 2017.

[KSK+23] A Koziol, R Szczygiel, D Kuznar, K Strzalka, M Wielgosz, P Otfinowski, and P Maj. Artificial neural network on-chip and in-pixel implementation towards pulse amplitude measurement. *Journal of Instrumentation*, 18(02):C02048, 2023.

[LXX+19] Li Li, Dawen Xu, Kouzi Xing, Cheng Liu, Ying Wang, Huawei Li, and Xiaowei Li. Squeezing the last mhz for cnn acceleration on fpgas. In *2019 IEEE International Test Conference in Asia (ITC-Asia)*, pages 151–156. IEEE, 2019.

[Mai12] Filipe R N C Maia. The coherent x-ray imaging data bank. *Nature Methods*, 9(9):854–855, August 2012.

[MG22] Erez Manor and Shlomo Greenberg. Custom hardware inference accelerator for tensorflow lite for microcontrollers. *IEEE Access*, 10:73484–73493, 2022.

[MS23] Hiroki Matsutani and Keisuke Sugiura. *Efficient Neural Networks and Their Acceleration Techniques for Embedded Machine Learning*, page 429–452. Springer Nature Switzerland, October 2023.

[MSS22] Palak Meena, Palak Sharma, and Kapil Sharma. Optimizing control of iot device using traditional machine learning models and deep neural networks. In *2022 6th International Conference on Computing Methodologies and Communication (ICCMC)*. IEEE, March 2022.

[MUDK22] Arijit Mukherjee, Arijit Ukil, Swarnava Dey, and Gitesh Kulkarni. Tinyml techniques for running machine learning models on edge devices. In *Proceedings of the Second International Conference on AI-ML Systems*, AIMLSystems 2022. ACM, October 2022.

[MWT18] Matsilele A Mabaso, Daniel J Withey, and Bhekisipho Twala. Spot detection in microscopy images using convolutional neural network with sliding-window approach. 2018.

[PPM20] Suraj Pandey, Ishwor Poudyal, and Tek Narsingh Malla. Pump-probe time-resolved serial femtosecond crystallography at x-ray free electron lasers. *Crystals*, 10(7):628, 2020.

[Ray22] Partha Pratim Ray. A review on tinyml: State-of-the-art and prospects. *Journal of King Saud University-Computer and Information Sciences*, 34(4):1595–1623, 2022.

[SAA21] Yap Yan Siang, Mohd Ridzuan Ahamd, and Mastura Shafinaz Zainal Abidin. Anomaly detection based on tiny machine learning: A review. *Open International Journal of Informatics*, 9(Special Issue 2):67–78, 2021.

[SBA20] Bharath Sudharsan, John G. Breslin, and Muhammad Intizar Ali. Edge2train: a framework to train machine learning models (svms) on resource-constrained iot edge devices. In *Proceedings of the 10th International Conference on the Internet of Things*, IoT '20. ACM, October 2020.

[SBT+22] Bharath Sudharsan, John G. Breslin, Mehreen Tahir, Muhammad Intizar Ali, Omer Rana, Schahram Dustdar, and Rajiv Ranjan. Ota-tinyml: Over the air deployment of tinyml models and execution on iot devices. *IEEE Internet Computing*, 26(3):69–78, May 2022.

[TAG21] B. K. Tripathy, S Anveshrithaa, and Shrusti Ghela. *Principal Component Analysis (PCA)*, page 5–16. CRC Press, July 2021.

[WS19] Pete Warden and Daniel Situnayake. *Tinyml: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers*. O'Reilly Media, 2019.

List of Figures

List of Tables

Affidavit

I Stephen Patrick Eteng herewith declare that I have composed the present paper and work by myself and without use of any other than the cited sources and aids. Sentences or parts of sentences quoted literally are marked as such; other references with regard to the statement and scope are indicated by full details of the publications concerned. The paper and work in the same or similar form has not been submitted to any examination body and has not been published. This paper was not yet, even in part, used in another examination or as a course performance. .
Lippstadt, July 2, 2024

Stephen Patrick Eteng

A Appendix

A.1 Python Code for the TinyML Model

```
{
 "cells": [
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "82d225e3-9429-418f-915b-552c01d17e33",
   "metadata": {},
   "outputs": [],
   "source": [
    "import h5py\n",
    "import os\n",
    "import numpy as np\n",
    "from PIL import Image\n",
    "from matplotlib import pyplot as plt\n",
    "from scipy.ndimage import convolve\n",
    "import tensorflow as tf\n",
    "from tensorflow.keras import layers, models\n",
    "from keras.models import Sequential\n",
    "from keras.layers import Conv2D, Dense, MaxPool2D, Flatten,
    ↪  Activation\n",
    "from sklearn.model_selection import train_test_split\n",
    "from tensorflow.keras.models import load_model\n",
    "from sklearn.metrics import confusion_matrix\n",
    "from sklearn.metrics import classification_report"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "4a8ff480-bc45-4834-a07a-accc8e09a80f",
   "metadata": {},
   "outputs": [],
   "source": [
    "# h5 to png images\n",
    "db_path = '/Users/Patrick/Downloads/LG36/r0087_2000.h5' # Path to
    ↪  the HDF5 file\n",
```

```
    "db = h5py.File(db_path, 'r')                                    # Open the
     ↪  HDF5 file in read mode\n",
    "\n",
    "id = 0\n",
    "for key in db['data'].keys():                                    # Iterating
     ↪  over the keys in 'data' group\n",
    "  img_datas = db['data'][key]['images']                          # Accessing
     ↪  the 'images' dataset for the current key\n",
    "  for i in range(img_datas.shape[0]):                            # Iterating
     ↪  over each image in the dataset\n",
    "    id += 1                                                      #
     ↪  Incrementing the image ID\n",
    "    img = img_datas[i]                                           #
     ↪  Extracting the current image\n",
    "    img[img < 0] = 0                                             # Replacing
     ↪  negative values with 0\n",
    "    pil_img = Image.fromarray(img, mode='I')                     # Creating
     ↪  a PIL image from the array\n",
    "    pil_img = pil_img.resize((960, 960),
     ↪  Image.Resampling.BILINEAR) # # Resizing the image (Needed for
     ↪  Pillow 10)\n",
    "    img_name = '{:05d}.png'.format(id)                           #
     ↪  Formatting the image name\n",
    "    img_path =
     ↪  os.path.join('/Users/Patrick/Downloads/LG36/raw_images',
     ↪  img_name) # Creating the image path\n",
    "    pil_img.save(img_path)                                       # Saving
     ↪  the image"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "849d2526-7249-40cf-abe5-85d3896108c0",
   "metadata": {},
   "outputs": [],
   "source": [
    "# preprocessing function for global normalization and then local
     ↪  normalization and then resampling to 300x300\n",
    "# and then cropping to center 100x100\n",
    "def preprocess(path):\n",
    "    image = Image.open(path)            # Opening the image from
     ↪  the given path\n",
    "    image_array = np.array(image)       # Converting the image to
     ↪  a numpy array\n",
    "    mean_value = np.mean(image_array)   # Calculating the global
     ↪  mean\n",
```

```
    "    std_value = np.std(image_array)     # Calculating the global standard deviation\n",
    "    normalized_image_global = (image_array - mean_value) / std_value # Applying global normalization\n",
    "    kernel = np.ones((3, 3)) / (3 ** 2) # Creating a kernel for local normalization\n",
    "    local_mean = convolve(normalized_image_global, kernel, mode='constant') # Local mean calculation\n",
    "    local_std = np.sqrt(convolve(np.square(normalized_image_global - local_mean), kernel, mode='constant') - local_mean ** 2) # Local std calculation\n",
    "    normalized_image_local = (normalized_image_global - local_mean) / local_std # Applying local normalization\n",
    "    normalized_image_local_display = np.clip(normalized_image_local, 0, 1) # Clipping values to [0, 1] range\n",
    "    normalized_image_local_display = Image.fromarray((normalized_image_local_display * 255)) # Converting array to PIL image\n",
    "    def crop_to_middle(img):\n",
    "        width, height = img.size    # Getting the image dimensions\n",
    "        left = (width - 100) / 2    # Calculating left coordinate for cropping\n",
    "        top = (height - 100) / 2    # Calculating top coordinate for cropping\n",
    "        right = (width + 100) / 2   # Calculating right coordinate for cropping\n",
    "        bottom = (height + 100) / 2 # Calculating bottom coordinate for cropping\n",
    "        cropped_img = img.crop((left, top, right, bottom)) # Cropping the image\n",
    "        return cropped_img\n",
    "    return crop_to_middle(normalized_image_local_display.resize((300, 300), Image.Resampling.NEAREST)).convert(\"L\") # Resizing, cropping, converting to grayscale"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "2b6cb399-16e1-4b97-9133-725c7769c4bf",
   "metadata": {},
   "outputs": [],
   "source": [
```

```
    "# applying preprocessing to all images\n",
    "file_names = [f\"C:\\\\Users\\\\user\\\\Desktop\\\\Mnist_project\\\\dataset\\\\LG36_images\\\\{i:05d}.png\" for i in range(1, 2001)] # Creating file names for images\n",
    "for i in range(1, 2001):                                    # Iterating over image indices\n",
    "    preprocessed_image = preprocess(file_names[i-1])      # Preprocessing each image\n",
    "    preprocessed_image.save(f\"C:\\\\Users\\\\user\\\\Desktop\\\\Mnist_project\\\\dataset\\\\LG36_Preprocessed\\\\{i:05d}.png\") # Saving the preprocessed image"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "40a3aad1-4d2f-4560-8f95-f89ed8382564",
   "metadata": {},
   "outputs": [],
   "source": [
    "# cnn model creation\n",
    "def create_cnn_model(input_shape, num_classes):      # Defining the function to create the CNN model\n",
    "    model = Sequential()                             # Initializing a Sequential model\n",
    "    model.add(Conv2D(filters=3, kernel_size=(5,5), padding='valid', input_shape=(100, 100, 1))) # Adding a Conv2D layer\n",
    "    model.add(Activation('relu'))                    # Adding a ReLU activation layer\n",
    "    model.add(MaxPool2D(strides=2, padding='same'))  # Adding a MaxPooling layer\n",
    "\n",
    "    model.add(Conv2D(filters=3, kernel_size=(5,5), padding='valid'))  # Adding another Conv2D layer\n",
    "    model.add(Activation('relu'))                    # Adding another ReLU activation layer\n",
    "    model.add(MaxPool2D(strides=2, padding='same'))  # Adding another MaxPooling layer\n",
    "\n",
    "    model.add(Conv2D(filters=3, kernel_size=(5,5), padding='valid'))  # Adding another Conv2D layer\n",
    "    model.add(Activation('relu'))                    # Adding another ReLU activation layer\n",
    "    model.add(MaxPool2D(strides=2, padding='same'))  # Adding another MaxPooling layer\n",
```

```
    "    \n",
    "    model.add(Conv2D(filters=3, kernel_size=(5,5),
    ↪  padding='valid'))  # Adding another Conv2D layer\n",
    "    model.add(Activation('relu'))
    ↪     # Adding another ReLU activation layer\n",
    "\n",
    "    model.add(Flatten())
    ↪     # Adding a Flatten layer to convert 2D to 1D\n",
    "    model.add(Dense(3))
    ↪     # Adding a Dense layer with 3 units\n",
    "    model.add(Activation('softmax'))
    ↪     # Adding a softmax activation layer for classification\n",
    "    return model"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "b8294957-0e50-40b5-8014-d72958d28dcf",
   "metadata": {},
   "outputs": [],
   "source": [
    "# bring preprocessed images into memory\n",
    "preprocessed_images = [np.array(Image.open(path)) for path in
    ↪  [f\"C:\\\\Users\\\\user\\\\Desktop\\\\Mnist_project\\\\dataset\
    ↪  \\\LG36_Preprocessed\\\\{i:05d}.png\" for i in range(1, 2001)]]
    ↪  # Loading preprocessed images\n",
    "preprocessed_images = np.array(preprocessed_images)
    ↪    # Converting list of images to numpy array"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "feb495cd-3dca-4888-994b-75adaf6fa453",
   "metadata": {},
   "outputs": [],
   "source": [
    "image_to_display = np.squeeze(preprocessed_images[1750])
    ↪    # Selecting an image to display\n",
    "plt.figure(figsize=(12, 6))
    ↪    # Setting the figure size for the plot\n",
    "plt.imshow(image_to_display, cmap='gray')
    ↪    # Displaying the image in grayscale\n",
    "plt.show()
    ↪    # Showing the plot"
```

```
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "3c5e09a5-6b66-47b2-b896-3ecee40b1019",
   "metadata": {},
   "outputs": [],
   "source": [
    "# load in label file\n",
    "label_file='C:\\\\Users\\\\user\\\\Desktop\\\\Mnist_project\\\\dataset\\\\LG36.txt'   # Path to the label file\n",
    "labels = np.loadtxt(label_file, dtype='str', usecols=(2,), skiprows=0)    # Loading labels from the file\n",
    "label_mapping = {'HIT': 0, 'MAYBE': 1, 'MISS': 2}         # Defining the label mapping\n",
    "numeric_labels = np.array([label_mapping[label] for label in labels])     # Converting labels to numeric form"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "b672f641-82de-4247-a2e1-ddbaf28d2f70",
   "metadata": {},
   "outputs": [],
   "source": [
    "X_train, X_test, y_train, y_test = train_test_split(preprocessed_images, numeric_labels, test_size=0.2, random_state=42) # Splitting the data into train and test sets"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "877ca841-abbd-4745-b883-c12bad5711d3",
   "metadata": {},
   "outputs": [],
   "source": [
    "input_shape = (100, 100, 1)                              # Defining the input shape for the model\n",
    "num_classes = 3                                          # Defining the number of classes\n",
    "model = create_cnn_model(input_shape, num_classes)  # Creating the CNN model\n",
```

```
    "adam = tf.keras.optimizers.Adam(learning_rate=4e-4) # Initializing the Adam optimizer\n",
    "model.compile(loss='sparse_categorical_crossentropy', metrics=['accuracy'], optimizer=adam) # Compiling the model\n",
    "model.summary()"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "7bb4cee7-26bd-470e-95a3-d5a61fbb11e4",
   "metadata": {},
   "outputs": [],
   "source": [
    "model.fit(X_train, y_train, epochs=10, batch_size=64)  # Training the model"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "9dbbd6b9-2011-429b-8b6f-12596e86f7bd",
   "metadata": {},
   "outputs": [],
   "source": [
    "# validate the model on test dataset to determine generalization\n",
    "_, acc = model.evaluate(X_test, y_test, batch_size=64, verbose=0)\n",
    "print(\"\\nTest accuracy: %.1f%%\" % (100.0 * acc))"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "77bffa8f-a374-4230-a346-0d5a618bf292",
   "metadata": {},
   "outputs": [],
   "source": [
    "model.save('C:\\\\Users\\\\user\\\\Desktop\\\\Mnist_project\\\\dataset\\\\TinyM.h5') # Saving the model"
   ]
  },
  {
   "cell_type": "code",
```

```
   "execution_count": null,
   "id": "89188359-9bec-4b11-afe2-e280d67c8d4e",
   "metadata": {},
   "outputs": [],
   "source": [
    "y_predict = model.predict(X_test)        \n",
    "y_predict = y_predict.argmax(axis=1)     \n",
    "# Generate the confusion matrix\n",
    "cm = confusion_matrix(y_test, y_predict)\n",
    "\n",
    "# Create the plot\n",
    "fig, ax = plt.subplots()\n",
    "cax = ax.matshow(cm, cmap=plt.cm.Blues)  # Use a color map that is visually distinct\n",
    "plt.title('Confusion Matrix')\n",
    "plt.colorbar(cax)\n",
    "\n",
    "# Set labels for axes\n",
    "plt.xlabel('Predicted')\n",
    "plt.ylabel('True')\n",
    "\n",
    "# Add class labels if necessary\n",
    "classes = ['HIT', 'MISS', 'MAYBE']  # Adjust as per your classes\n",
    "ax.set_xticklabels([''] + classes)\n",
    "ax.set_yticklabels([''] + classes)\n",
    "\n",
    "# Rotate the tick labels for aesthetics\n",
    "plt.xticks(rotation=90)\n",
    "\n",
    "# Annotate each cell with the numeric value using white or black depending on the background\n",
    "thresh = cm.max() / 2.\n",
    "for i in range(cm.shape[0]):\n",
    "    for j in range(cm.shape[1]):\n",
    "        ax.text(j, i, format(cm[i, j], 'd'),\n",
    "                ha=\"center\", va=\"center\",\n",
    "                color=\"white\" if cm[i, j] > thresh else \"black\")\n",
    "\n",
    "# Save the figure to a file\n",
    "plt.savefig('confusion_matrix.png')\n",
    "plt.close()\n"
   ]
  },
  {
```

```
   "cell_type": "code",
   "execution_count": null,
   "id": "c524a9c4-1c30-4b12-bcea-41942371a202",
   "metadata": {},
   "outputs": [],
   "source": [
    "plt.figure(figsize=(15, 6))\n",
    "plt.imshow(Image.open('confusion_matrix.png'))"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "01d6f619-b122-4d38-b3c5-19a0dc0d6ad1",
   "metadata": {},
   "outputs": [],
   "source": [
    "print(classification_report(y_test, y_predict))"
   ]
  }
 ],
 "metadata": {
  "kernelspec": {
   "display_name": "Python 3 (ipykernel)",
   "language": "python",
   "name": "python3"
  },
  "language_info": {
   "codemirror_mode": {
    "name": "ipython",
    "version": 3
   },
   "file_extension": ".py",
   "mimetype": "text/x-python",
   "name": "python",
   "nbconvert_exporter": "python",
   "pygments_lexer": "ipython3",
   "version": "3.11.9"
  }
 },
 "nbformat": 4,
 "nbformat_minor": 5
}
```