VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JNANA SANGAMA",MACHHE, BELAGAVI-590018

Project Work Phase – 2 Report

on

Optimization of CNN for implementing in FPGA/SoC

Submitted in partial fulfillment of the requirements for the VIII semester

Bachelor of Engineering

in

Artificial Intelligence & Machine Learning

of

Visvesvaraya Technological University, Belagavi

by

Akash V (1CD20AI002)

Ayush Kumar Singh (1CD20AI006)

C Siva Prasad (1CD20AI011)

Monish Sai R (1CD20AI035)

Under the Guidance of

Dr. D. Antony Louis Piriyakumar
Dean
Research & Development
Cambridge Institute of Technology

Mr. Gaurav Upadhyay
Scientist/Engg-SE
Baseband Data Handling Section, PDMG
URSC/ISRO

Department of Artificial Intelligence & Machine Learning

CAMBRIDGE INSTITUTE OF TECHNOLOGY, BANGALORE-560 036

2023-2024

CAMBRIDGE INSTITUTE OF TECHNOLOGY

K.R. Puram, Bangalore-560 036

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

CERTIFICATE

Certified that **Akash V** bearing **1CD20AI002**, **Ayush Kumar Singh** bearing **1CD20AI006, C Siva Prasad** bearing **1CD20AI011** and **Monish Sai R** bearing **1CD20AI035** bonafide students of **Cambridge Institute of Technology,** have successfully completed the **Project Work Phase–2** entitled "**Optimization of CNN for implementing in FPGA/SoC"** in partial fulfillment of the requirements for VIII semester **Bachelor of Engineering** in **Artificial Intelligence & Machine Learning** of **Visvesvaraya Technological University, Belagavi** during academic year 2023-24. It is certified that all Corrections/Suggestions indicated for Internal Assessment have been incorporated in the report deposited in the departmental library. The Project Work Phase–2 report has been approved as it satisfies the academic requirements prescribed for the Bachelor of Engineering degree.

Guide, **Dr. D. Antony Louis Piriyakumar, Dean,** **R&D, CITech**	**Coordinator,** **Dr. Buddesab,** **Associate Professor** **Dept. of AI&ML, CITech**	**Head of the Department,** **Dr. Varalatchoumy.M,** **Head-CHOSS & HOD** **Dept. of AI&ML, CITech**

Name of the Examiners **Signature**

1.

2.

DECLARATION

We**, Akash V**, **Ayush Kumar Singh, C Siva Prasad** and **Monish Sai R** students of VIII semester B.E., Department of Artificial Intelligence & Machine Learning, Cambridge Institute of Technology, Bengaluru, hereby declare that the **Project Work Phase–2** entitled "**Optimization of CNN for implementing in FPGA/SoC"** has been carried out by us and submitted in partial fulfilment of the course requirements of VIII semester **Bachelor of Engineering** in **Artificial Intelligence & Machine Learning** as prescribed b**y Visvesvaraya Technological University, Belagavi**, during the academic year 2023-2024.

We also declare that, to the best of our knowledge and belief, the work reported here does not form part of any other report on the basis of which a degree or award was conferred on an earlier occasion on this by any other students.

Place: Bangalore

Date:

Name Signature

1. Akash V (1CD20AI002)
2. Ayush Kumar Singh (1CD20AI006)
3. C Siva Prasad (1CD20AI011)
4. Monish Sai R (1CD20AI035)

ACKNOWLEDGEMENT

We would like to place on record our deep sense of gratitude to **Shri. D. K. Mohan,** Chairman, Cambridge Group of Institutions, Bangalore, India for providing excellent Infrastructure and Academic environment at CITech without which this work would not have been possible.

We are extremely thankful to **Dr. G. Indumathi,** Principal, CITech, Bangalore, for providing us the academic ambience and everlasting motivation to carry out this work and shaping our careers.

We express my sincere gratitude to **Dr. Varalatchoumy M.,** Head-CHOSS & HOD, Dept. of Artificial Intelligence & Machine Learning, CITech, Bangalore, for her stimulating guidance, continuous encouragement, and motivation throughout the course of present work.

Our sincere thanks to Project Coordinator to **Dr. Buddesab,** Associate Professor, Dept. of AI&ML, CITech, Bangalore for the, insightful comments, and timely instruction and support in coordinating the project work.

This work would not have been possible without the opportunity to do an internship given to us by U R Rao Satellite Center. We are especially indebted to **Dr. Jothy soman** (Group Head, PDMG), **Mrs. Srividhya S.** (Division Head, DHD, PDMG) and **Mrs. Shantala S.H.** (Section Head, DHS-I).

We wish to extend our warmest thanks to our mentor **Mr. Gaurav Upadhyay** (Scientist - SE, PDMG) who allowed us to work in the industry hot domain of Deep Learning, and gave continuous support throughout the internship and worked actively to provide us with the protected academic time to pursue the goals.

We would also like to thank **Mr. Srijan Ghoshal** (Scientist - SD, PDMG) for always being with us throughout the project and helping us with the best programming practices, ‘vi’ editor.

We also wish to extend my thanks to project guide, **Dr. D. Antony Louis Piriyakumar,** Dean, Research and Development, CITech, Bangalore for the critical, insightful comments, guidance, and constructive suggestions to improve the quality of this work.

Finally, to all my friends, classmates who always stood by us in difficult situations also helped us in some technical aspects and last but not the least, we wish to express deepest sense of gratitude to our parents who were a constant source of encouragement and stood by our side as pillar of strength for completing this work successfully.

Akash V (1CD20AI002)

Ayush Kumar Singh (1CD20AI006)

C Siva Prasad (1CD20AI011)

Monish Sai R (1CD20AI035)

ABSTRACT

The optimization of Convolutional Neural Networks (CNNs) for implementation in Field-Programmable Gate Arrays (FPGAs) and Systems on Chip (SoCs) is critical for advancing deep learning applications in embedded systems. This study focuses on enhancing the efficiency of CNNs within these platforms to meet the demands of real-time processing tasks such as image classification. FPGAs offer flexibility and efficiency, optimizing the performance of CNN operations, whereas SoCs integrate CPU, GPU, and FPGA resources, providing a comprehensive solution for executing CNNs. With the growth of edge computing in sectors requiring immediate data processing, like autonomous vehicles and healthcare, there is a pressing need for powerful yet efficient deep learning models that address bandwidth limitations and minimize latency. Our research investigates quantization strategies, which convert 32-bit floating-point precision into more compact formats, to improve memory usage and increase processing speed. We evaluate the impact of Post Training Quantization (PTQ) and Quantization-Aware Training (QAT) across different network architectures, achieving up to fourfold increases in inference speed with only minimal reductions in accuracy (1-4%) when tested on the EuroSat dataset. Additionally, utilizing a Xilinx Deep Processing Unit (DPU) demonstrated robust performance with significant model size reduction, enhancing the speed and accuracy of inferencing while optimizing storage capabilities on edge devices. This project underscores the potential of FPGAs and SoCs in streamlining CNN deployment for real-time embedded applications and highlights the benefits of quantization techniques in maximizing the efficiency of deep learning models for edge computing.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

ADAGRAD Adaptive gradient

ADAM Adaptive moment estimation

CNN Convolutional Neural Network

CNN's Convolutional Neural Networks

CPU Central Processing Unit

DCNN's Deep Convolutional Neural Networks

DNA Deep Neural Architecture

DPU Data Processing Unit

EDRAM Embedded Dynamic random-access memory

FC layer Fully Connected Layer

FFT Fast Fourier Transform

FP32 32-bit floating-point

FPGA Field Programmable Gate Array

FPR False Positive Ratio

FPS Frames per Second

GFLOPS GIGAFLOPS

GPU Graphics Processing Unit

HLS High Level Synthesis

LeNet-5 Linear Network-5

MPSoC Multiprocessor System on a Chip

NAS Neural Architecture Search

R-CNNs Region-Based Convolutional Neural Networks

ResNet Residual Network

SGD Stochastic Gradient Descent

SVD Singular Value Decomposition

VGG Visual Geometry Group

CHAPTER-1

INTRODUCTION

1.1 Background

Deep learning technologies have become instrumental in advancing a multitude of sectors, including traffic monitoring, healthcare, and the expansive realm of the Internet of Things (IoT). These systems have traditionally depended on cloud computing for inferencing tasks that involve extensive data processing like image classification and object detection. However, the evolving demands of critical real-time applications such as autonomous vehicles, surveillance systems, and spacecraft have catalyzed a significant shift toward edge computing. This paradigm shift is primarily driven by the necessity to enhance the immediacy and reliability of computational tasks, which are pivotal in scenarios where real-time processing is crucial.

In this transformative landscape, Field-Programmable Gate Arrays (FPGAs) and Systems on Chip (SoCs) emerge as vital platforms, facilitating the swift acceleration of Convolutional Neural Networks (CNNs)—a principal component of deep learning frameworks. FPGAs excel by offering unparalleled flexibility and superior performance, enabling precise and tailored optimization of CNN operations suitable for dynamic computational requirements. On the other hand, SoCs provide a comprehensive solution by integrating CPU, GPU, and FPGA resources into a single system. This integration results in a robust and versatile computational platform capable of handling sophisticated tasks that require high data throughput and complex model computations, seamlessly performing these tasks locally on the device.

The choice of FPGA/SoC platforms for optimizing CNN operations is not merely a technical decision but a strategic one, aligning with the broader objectives of modern computational infrastructure. These platforms support a diverse range of applications, from enhancing operational efficiency to reducing latency and improving the responsiveness of systems operating in bandwidth-constrained environments. The architectural advantages of FPGAs and SoCs, including their adaptability and computational power, make them ideally suited to not only meet but exceed the rigorous demands of real-time embedded systems.

Moreover, the integration of these technologies allows for a scalable approach to deploying deep learning models, accommodating the growing complexity and expanding scale of tasks across industries. This capability is particularly valuable in environments where traditional cloud computing infrastructures fall short. The strategic utilization of FPGA and SoC platforms thus

represents a forward-thinking solution that leverages the strengths of edge computing to deliver enhanced performance, reliability, and efficiency in real-time operations, fundamentally transforming the technological landscape of deep learning applications. The visual representation in Fig.1.1 succinctly illustrates the rationale behind opting for FPGA/SoC platforms in optimizing CNN operations, underscoring their pivotal role in the current and future contexts of technology deployment.

Feature	FPGA	SoC	SoC FPGA
Programmable	Yes	No	Yes
Flexibility	High	Medium	High
Performance	High	Medium	High
Power consumption	Medium	Low	Medium

Fig. 1.1 Reasoning for choice of FPGA/SoC

1.2 Limitations of The Existing System

The traditional cloud-based inferencing systems present several key limitations that impact the efficiency and feasibility of deploying deep learning models in real-time applications. Firstly, cloud computing suffers from bandwidth limitations that can severely restrict the volume of data that can be transferred between the cloud servers and the end devices, resulting in significant latency. This is particularly detrimental in applications requiring immediate response, such as in autonomous driving or real-time surveillance.

Secondly, transmission delays are inherent in cloud systems due to the physical distance data must travel between the cloud servers and the operational devices. These delays make it impractical for applications where millisecond decision-making is crucial. Moreover, network connectivity issues can arise, leading to intermittent data transmission that further exacerbates the response times and reliability of cloud-based systems.

Lastly, the cloud model often falls short in terms of privacy and security, as sensitive data must be transmitted to and from external servers, increasing the risk of data breaches and unauthorized access. These challenges necessitate a shift towards more localized processing solutions such as edge computing, which allows data processing to occur directly on the devices, thereby minimizing latency, enhancing security, and reducing the dependency on continuous network connectivity. These considerations have propelled the development of hardware solutions like FPGAs and SoCs, which support the local execution of computationally intensive deep learning models without the drawbacks associated with cloud computing.

1.3 Problem Statement

In the rapidly evolving fields of autonomous vehicles and space exploration, the implementation of Convolutional Neural Networks (CNNs) on edge devices presents significant computational challenges. Traditional cloud-based inferencing, while powerful, introduces unacceptable levels of latency, bandwidth limitations, and potential security vulnerabilities, which are especially problematic for applications requiring real-time processing. Edge computing addresses these issues by enabling local data processing, yet the challenge remains to adapt the computationally intensive CNN models for efficient execution on constrained hardware platforms such as FPGAs and SoCs.

The specific problem lies in the adaptation and optimization of CNN architectures for implementation on FPGA/SoC platforms using systolic array architectures, which are well-suited for parallel matrix operations inherent in CNNs. The goal is to develop a hardware-friendly CNN model that maintains high accuracy while achieving lower latency and increased inference speeds, crucial for real-time decision-making in autonomous vehicles and space exploration devices. This involves not only the selection and modification of suitable CNN architectures but also optimizing these models for the unique capabilities and limitations of FPGAs and SoCs, thereby ensuring efficient use of hardware resources, minimizing power consumption, and enhancing overall system reliability.

1.4 Objective of The Project

- **Enhance Computational Efficiency**: Utilize FPGA/SoC hardware accelerators to significantly boost the speed and performance of Convolutional Neural Networks (CNNs), enabling rapid processing and real-time data analysis in embedded systems.
- **Overcome Cloud Computing Constraints**: Develop solutions that address bandwidth limitations, transmission delays, and network connectivity issues associated with cloud computing, thereby enhancing the reliability and efficiency of edge computing applications.
- **Minimize Latency and Maximize Local Processing**: Implement CNNs directly on the FPGA/SoC devices to keep data and models local, reducing latency to meet the demands of real-time applications such as autonomous vehicles and immediate-response surveillance systems.
- **Optimize Resource Utilization and System Flexibility**: Design CNN architectures that are optimized for low resource usage and high adaptability on FPGA/SoC platforms. Explore advanced techniques like quantization and pruning to efficiently use hardware resources and facilitate scalable deployments across various sectors.
- **Real-Time Capabilities**: Leverage the parallel processing power of FPGAs to enhance real-time processing capabilities, ensuring the system's effectiveness under diverse operational conditions and datasets.

1.5 Proposed Solution

The project initiates with the construction of CNN models, serving as the foundational step. Subsequent to this, three pathways for model optimization are explored:

1. **Neural Architecture Search (NAS) Process**: An automated search for the optimal CNN architecture is conducted to find the model that offers the best trade-off between performance and complexity.

2. **Transfer Learning Model**: Existing pre-trained CNN models are utilized and fine-tuned with the project-specific dataset to leverage learned features and speed up the convergence time.

3. **Custom Net Knowledge Distillation**: A process where a compact and efficient custom CNN is created by distilling knowledge from a larger, more complex network, thereby capturing its capabilities in a more resource-efficient framework.

Once the CNN has been optimized through one of these three pathways, the model undergoes quantization and pruning. Quantization reduces the precision of the model's parameters to decrease the computational load, while pruning removes redundant or non-significant parameters from the network, thus streamlining it.

The refined and optimized CNN model is then mapped onto a target FPGA/SoC hardware, specifically a Xilinx chip. This hardware-specific tailoring ensures that the model is compatible with the computational resources and architecture of the Xilinx hardware.

The final step in the solution is to validate the model’s performance in terms of accuracy and latency. Ensuring that the model delivers high accuracy and low latency is critical for its feasibility in real-world applications. Upon meeting these performance criteria, the CNN is deployed to the field.

1.6 Advantages of The Proposed Solution

1. **Optimized Performance**: The use of NAS, transfer learning, and knowledge distillation ensures that the CNN is highly optimized for performance, balancing computational efficiency with accuracy.
2. **Resource Efficiency**: Quantization and pruning significantly reduce the model size and computational requirements, making the CNN suitable for deployment on resource-constrained devices like FPGAs and SoCs.
3. **Hardware-Specific Optimization**: By targeting the Xilinx chip, the solution takes full advantage of the specific features of the hardware, such as parallel processing capabilities, which further enhances performance.
4. **Rapid Deployment**: The streamlined model enables rapid deployment and integration into real-world applications due to its reduced complexity and resource demands.
5. **Real-Time Processing**: Ensuring good accuracy and low latency means that the CNN can support real-time processing needs, which is crucial for applications that require immediate data analysis and decision-making, such as autonomous vehicles or medical diagnostics.

This proposed solution combines the latest in deep learning model optimization techniques with advanced hardware targeting strategies to create a highly efficient and deployable CNN suitable for real-time applications on FPGA/SoC platforms.

1.7 Organization of The Project

The project report is organized as follows:

Chapter 2: **Literature Review –** Gives a brief overview of the survey papers and the research sources

that have been studied to establish a thorough understanding of the project under consideration.

Chapter 3: **Theoretical Background –** Establishes groundwork for the proposed project by giving a detailed analysis of the project topic, existing research relevant to the project, arguments in favor and against the existing solutions and finally explores the motivation behind the proposed solution.

Chapter 4: **System Architecture and Design –** Gives the architecture and design description of the project, conceptual and detailed design well supported with architecture diagrams.

Chapter 5: **Methodology and Implementation –** Discusses the methodology followed, implementation details of the project and reasons the use of methods and designs provided in architecture and development environment.

Chapter 6: **Experimental Result and Analysis –** Briefs the testing methods used for different modules in the project and provides graphs and tables of the results obtained.

Chapter 7: **Conclusion and Future Scope –** Gives the concluding remark of the project, throwing light on its future aspects.

References Lists the websites and references referred during the project work.

CHAPTER-2

LITERATURE SURVEY

2.1 Base Paper:

Title: - ***An FPGA-Based Hardware Accelerator for CNNs Using On-Chip Memories Only: Design and Benchmarking with Intel Movidius Neural Compute Stick***

Link: - https://www.hindawi.com/journals/ijrc/2019/7218758/

Description: - The high computational complexity of deep neural networks has long been a challenge in the field of artificial intelligence. As these networks grow in depth and complexity, the demand for techniques to enhance energy efficiency and throughput becomes increasingly crucial. The computational demands of training and deploying deep neural networks are not only resource-intensive but can also hinder their practical application in real-world scenarios. This overview delves into the landscape of deep neural networks, shedding light on their structure, functioning, and the challenges they pose. It explores various hardware platforms and architectures that can support deep neural networks, emphasizing the critical role of hardware in enabling the efficient execution of these models. This includes dedicated hardware accelerators, which have gained significant prominence for their ability to offload and expedite deep learning computations, ultimately enhancing energy efficiency and throughput. Moreover, the overview delves into the dynamic landscape of trends aimed at reducing the computation cost of deep neural networks. These trends encompass the utilization of specialized hardware accelerators, such as GPUs and TPUs, specifically designed to handle the intensive matrix operations common in deep learning. Overall, the multifaceted approach of hardware advancements and algorithmic innovations is essential for addressing the computational complexities of deep neural networks and ensuring their practicality in diverse domains, from healthcare to autonomous vehicles.

Pros: -

- **Efficiency in Resource Use:** Utilizes on-chip memory effectively, minimizing external data transfers, which enhances speed and reduces power consumption.
- **Performance:** Offers better performance metrics in terms of inference time when compared to the Intel Movidius Stick, particularly for keyword spotting tasks

Cons: -

- **Complexity of Design:** The implementation of CNNs on FPGAs requires intricate design techniques to optimize resource allocation and processing speed.
- **Scalability Issues:** The paper indicates potential challenges in scaling the solution for larger neural networks or different applications without significant redesigns.

2.2 Other Reference Paper:

[1] Title: A Survey of Quantization Methods for Efficient Neural Network Inference

Link: - https://arxiv.org/abs/2103.13630

Description: - The paper provides a comprehensive survey of quantization methods for efficient neural network inference. It discusses the importance of quantization in reducing memory usage and improving computational efficiency in neural network models. The authors explore various techniques and approaches used in quantization, highlighting the challenges and opportunities in this area of research. They also discuss the impact of quantization on real-world applications and the potential benefits it offers in terms of memory footprint and latency reduction. Overall, the paper serves as a valuable resource for understanding the role of quantization in optimizing neural network performance.

Pros: -

- The paper provides a thorough survey of quantization methods for efficient neural network inference, covering a wide range of techniques and approaches in this field.
- The paper is well-structured, with sections dedicated to different aspects of quantization, making it easy for readers to navigate and understand the content.
- The discussion on the impact of quantization on real-world applications adds practical value to the research, showing the significance of this topic in modern neural network implementations.
- The acknowledgment section highlights the support received from various organizations and individuals, recognizing their contributions to the research.

Cons: -

- The paper primarily focuses on inference in neural networks, with less emphasis on training aspects. Expanding the coverage to include training implications of quantization could provide a more holistic view of the topic.
- While the paper discusses various quantization methods, a comparative analysis of these methods in terms of effectiveness, efficiency, and applicability could enhance the depth of the research.
- The acknowledgment disclaimer stating that the conclusions do not necessarily reflect the position or policy of the sponsors raises questions about potential bias in the research findings.
- The authors acknowledge the challenge of not being able to discuss every relevant work in the field due to space limitations. Including a more extensive reference list or supplementary materials could address this limitation.

[2] Title: A White Paper on Neural Network Quantization

link: - https://arxiv.org/abs/2106.08295

Description: - The White Paper on Neural Network Quantization discusses techniques for reducing the computational cost of neural networks while preserving accuracy. It covers the quantization of weights and activations, introducing scale factors for flexibility and error reduction. The paper mentions Learned Step Size Quantization and Bayesian Bits as relevant works in the field. Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT) are highlighted as approaches for quantization. The paper emphasizes the importance of quantization for deploying neural networks on edge devices efficiently.

Pros: -

- The paper provides a detailed overview of state-of-the-art algorithms for mitigating the impact of quantization noise on neural network performance.
- It offers practical considerations and tested pipelines based on existing literature and extensive experimentation for achieving state-of-the-art performance in common deep learning models and tasks.

- The paper starts with an introduction to quantization, discusses hardware considerations, and delves into two main quantization regimes, making the content accessible to a wide audience.

- Given the increasing integration of deep learning into electronic devices, the paper addresses a timely and important topic in the field of machine learning.

Cons: -

- The paper may not cover all possible quantization techniques or considerations, potentially leaving out some relevant approaches or perspectives.
- While the paper discusses PTQ and QAT, it may benefit from a more in-depth comparison of these methods and their respective strengths and weaknesses.
- The paper may not address specific challenges or nuances that could arise in different application domains or scenarios where neural network quantization is applied.
- It could provide more insights into future research directions or emerging trends in neural network quantization to guide further exploration in the field.

[3] Title: A Survey on Deep Neural Network Pruning-Taxonomy, Comparison, Analysis, and Recommendations.

Link: - https://arxiv.org/abs/2308.06767

Description: - The survey is structured into sections covering terminology, taxonomy of pruning, speedup overview, timing of pruning, methods of pruning, comparative analysis of pruning techniques, integration of pruning with other compression methods, practical guidance for selecting pruning methods, and future research directions. The paper concludes by providing a comprehensive resource for neural network pruning papers and open-source codes on GitHub to support ongoing research in the domain.

Pros: -

- The paper offers a detailed overview of modern deep neural network pruning techniques, consolidating insights from a wide range of academic papers.
- It establishes a new taxonomy of pruning methods, providing a structured framework for understanding different approaches to neural network pruning.
- The paper conducts comparative experiments and analysis of various pruning settings, offering valuable insights into the performance of different pruning methods.

- Summarizing diverse pruning applications and providing benchmark datasets and evaluations can be beneficial for researchers and practitioners in the field.

- The paper offers valuable recommendations for selecting pruning methods based on application requirements and highlights promising research directions for further exploration.

Cons: -

- The paper primarily focuses on deep neural network pruning, potentially overlooking other aspects of neural network compression techniques.
- While the paper provides a comparative analysis of pruning methods, a more in-depth analysis of the performance metrics and trade-offs could enhance the understanding of the results.
- The paper may benefit from a more detailed discussion on the practical implementation aspects of the pruning methods discussed.
- Given the rapidly evolving nature of deep learning research, the paper may need frequent updates to incorporate the latest advancements in the field.
- While providing a repository of resources is valuable, ensuring the accessibility and usability of these resources for a wider audience could be improved.

[4] Title: Knowledge Distillation: A Survey

Link: - https://arxiv.org/abs/2006.05525

Description: - The paper focuses on compressing and accelerating large models for deployment on resource-constrained devices. It provides an overview of knowledge distillation, including different knowledge types, distillation strategies, and teacher-student architectures. The survey reviews recent progress in knowledge distillation algorithms and applications in various real-world scenarios. It discusses challenges in knowledge distillation, such as extracting rich knowledge from teachers and transferring it to guide student training. The paper also highlights the importance of knowledge transfer in tasks like adversarial attacks, data augmentation, data privacy, and security. Additionally, it explores the concept of dataset distillation to reduce training loads of deep models. The survey concludes with insights on future research directions and applications for knowledge distillation in deep learning.

Pros: -

- The paper provides a thorough overview of knowledge distillation, covering various aspects such as knowledge types, distillation strategies, teacher-student architectures, algorithms, performance comparisons, and applications.

- The paper is well-structured, with sections dedicated to different aspects of knowledge distillation, making it easy for readers to navigate and understand the content.
- The paper offers insights into the challenges of knowledge distillation and discusses potential future research directions, providing valuable information for researchers and practitioners in the field.

Cons: -

- While the paper provides a comprehensive survey of existing knowledge distillation techniques, it may not introduce significantly novel concepts or approaches in the field.
- The paper primarily focuses on knowledge distillation in the context of model compression and acceleration, potentially overlooking other important aspects of deep learning research.
- The paper could benefit from more practical examples or case studies to illustrate the application of knowledge distillation in real-world scenarios, enhancing its practical relevance for readers.

[5] Title: - A Survey on Neural Architecture Search

Link: https://arxiv.org/abs/1905.01392

Description: - The paper is a comprehensive survey on Neural Architecture Search (NAS), It aims to unify the landscape of existing methods in NAS, critically examine different approaches, and highlight popular misconceptions and pitfalls. The survey is divided into several sections, covering topics such as architecture search spaces, formal problem definitions, optimization methods (reinforcement learning, evolutionary algorithms, surrogate model-based optimization, and one-shot architecture search), constraints, multiple objective functions, and model compression techniques. The authors discuss the importance of considering meta-features of datasets in NAS and provide examples of transfer learning for automating fine-tuning of architectures across tasks. They also mention the use of Bayesian optimization and optimal transport in NAS. Additionally, the paper references works on robust architectures against adversarial attacks and presents insights from the Annual Conference on Neural Information Processing Systems.

Pros: -

- The paper provides a thorough overview of Neural Architecture Search (NAS), covering various aspects such as architecture search spaces, optimization methods, constraints, and model compression techniques.

- The authors offer a critical examination of different approaches in NAS, highlighting popular misconceptions and pitfalls in the field.
- The paper references recent works and studies in NAS, providing readers with valuable insights into the latest developments and trends.
- The survey is well-structured, with sections dedicated to different aspects of NAS, making it easy for readers to navigate and understand the content.

Cons: -

- While the paper covers existing methods and approaches in NAS, it may not introduce many new concepts or groundbreaking ideas in the field.
- The survey primarily focuses on existing techniques and may not delve deeply into potential future directions or emerging trends in NAS.
- The paper mentions conducting experiments to supplement criticism but does not provide detailed experimental results or comparisons between different methods.
- The paper briefly touches on the interpretability of deep learning models but does not delve deeply into this aspect, which could be important for understanding the inner workings of NAS algorithms.

[6] Title: - Efficient Processing of Deep Neural Networks: A Tutorial and Survey

Link: - https://ieeexplore.ieee.org/document/8114708

Description: - The paper, serves as a comprehensive guide to recent advancements aimed at achieving efficient processing of Deep Neural Networks (DNNs). Addressing the computational complexity challenges of DNNs, the tutorial explores techniques to enhance energy efficiency and throughput without compromising accuracy or increasing hardware costs. It covers DNN fundamentals, various supporting hardware platforms and architectures, and trends in reducing computation costs through hardware and algorithmic changes.

Pros: -

- Provides a comprehensive tutorial and survey on efficient DNN processing, covering hardware platforms, trends, and development resources.
- Discusses benchmarking metrics and design considerations for evaluating DNN hardware designs.

Cons: -

- Lacks novel method proposal; serves more as a survey and tutorial.
- May not capture all recent advances due to the rapid pace of research; does not discuss potential limitations.

[7] Title: - Hardware Considerations for Tensor Implementation and Analysis Using the Field Programmable Gate Array

Link:- https://www.mdpi.com/2079-9292/7/11/320

Description: - The paper discusses the use of Field Programmable Gate Arrays (FPGAs) for implementing tensor operations, which are fundamental to many machine learning algorithms. The authors propose a hardware design that utilizes the available resources within the FPGA. The paper presents the design architecture, simulation results, and physical prototype test results.

Pros: -

- Presents a hardware design leveraging FPGA resources for efficient data processing.
- Addresses the need for embedded DSP algorithms in complex embedded systems for IoT applications.

Cons: -

- Lacks comparison with other state-of-the-art methods and potential generalizability concerns.
- Specificity to certain problems and lacks discussion on potential limitations or drawbacks.

[8] Title: - Optimizing the Convolution Operation to Accelerate Deep Neural Networks on FPGA

Link: - https://ieeexplore.ieee.org/document/8330049

Description: - The paper introduces a novel method for optimizing the convolution operation in Convolutional Neural Networks (CNNs) on Field-Programmable Gate Arrays (FPGAs). It proposes a specific dataflow and architecture to enhance resource utilization and minimize data

communication. By addressing challenges in convolution optimization, the paper provides a systematic and comprehensive approach, demonstrating its effectiveness through the implementation of various CNNs on Intel FPGAs.

Pros: -

- Quantitative analysis and optimization of CNN accelerator design.
- Efficient dataflow and architecture for minimizing data communication.
- Notable throughput results for VGG-16 on FPGAs (348 GOPS and 715 GOPS).

Cons: -

- Lack of comparison with other state-of-the-art methods.
- Potential specificity to certain CNN types may limit generalization.
- No discussion of potential limitations or drawbacks.

[9] Title: - Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation

Link: - https://arxiv.org/abs/2004.09602

Description: - This paper explores how quantization techniques can shrink DNNs, speed up inference, and boost processor efficiency. They analyze different quantization parameters and test them on diverse models (vision, speech, language). Most importantly, they propose an 8-bit quantization workflow that preserves accuracy (within 1%) even for challenging models like Mobile Net's and BERT-large.

Pros: -

- Quantitative analysis and optimization of CNN accelerator design for efficient inference.
- Efficient dataflow minimizes data communication, maximizing resource utilization.
- Demonstrates successful implementation of end-to-end CNNs with high throughputs on FPGAs.

Cons: -

- Lack of comparison with other state-of-the-art methods and generalization concerns.
- Limited discussion on potential drawbacks or limitations of the proposed method.

[10] Title: - Transfer Learning

Link: - https://ieeexplore.ieee.org/document/8114708

Description: - The paper introduces and surveys the field of transfer learning, focusing on leveraging knowledge from related tasks to enhance learning in new contexts. Covering both inductive and reinforcement learning, the paper outlines current research, highlighting challenges and the dependence on underlying machine learning algorithms. This work serves as a valuable resource for understanding the goals and complexities of transfer learning.

Pros: -

- Offers a comprehensive tutorial and survey on efficient DNN processing with insights into hardware platforms and development resources.
- Highlights key trends and important metrics for evaluating DNN hardware designs.

Cons: -

- Primarily a survey and tutorial, lacking new method proposals.
- Potential gaps in coverage of recent advances due to the fast-paced nature of research.
- Does not address the limitations or drawbacks of surveyed methods.

2.3 Related Work

The field of FPGA-based CNN accelerators has seen substantial research due to the computational intensity of CNNs and the desire to deploy them on embedded systems. One such study presented a CNN core architecture called mNet2FPGA, which mapped a trained CNN onto an SoC FPGA. This approach focused on the processing system being responsible for the convolution and fully connected core configuration, with the programmable logic holding cores for convolution and fully connected layers. The system, tested on a Z-7020 FPGA, showcased the advantages of using fixed-point arithmetic and kernel binarization to reduce resource utilization while maintaining good performance [1].

Another study aimed at integrating a CNN accelerator for both standard and depthwise separable convolution, which is particularly challenging due to the limited resources of FPGAs and the complexity of CNNs. This research leveraged the ZYNQ heterogeneous platform and used a roofline model to balance resources and bandwidth. The accelerator was capable of handling different network layers through parameter configuration and maximized bandwidth utilization by employing a data stream interface and ping-pong on-chip cache, resulting in a performance of 17.11GOPS for 32-bit floating-point operations [2].

These works highlight a trend towards making CNN implementations more efficient and effective on FPGAs and SoCs, focusing on optimizing resource usage and computational efficiency without sacrificing performance. The research suggests that such specialized accelerators can achieve significant performance gains and energy efficiency compared to traditional computing systems, which is crucial for real-time applications in embedded devices.

CHAPTER-3

THEORETICAL BACKGROUND

The theoretical foundation of deploying Convolutional Neural Networks (CNNs) on FPGA/SoC platforms involves a deep understanding of both the architecture of CNNs and the critical optimization strategies for efficient implementation. At the heart of this process are CNNs, a powerful class of deep neural networks highly effective in areas such as image and video recognition, powered by their ability to capture hierarchical patterns within data. The deployment of these sophisticated models to the constraints of FPGA/SoC platforms, such as those produced by Xilinx, requires the CNN to be meticulous optimization to ensure resource efficiency and reduced operational latency. This optimization is achieved through advanced methodologies including Neural Architecture Search (NAS), which automates the search of optimal network architectures; Transfer Learning to enhance adaptability and accelerate the learning process by using pre-existing models; Knowledge Distillation which focuses on distilling the expansive knowledge of large-scale networks into more compact forms. Optimization is further advanced by employing techniques like quantization and pruning, which streamline the model by reducing its variable precision and stripping away redundancies, thereby decreasing both the model size and computational load. Each of these techniques not only enhances the performance of the networks but also ensures that they meet the stringent real-time processing requirements necessary for practical applications. This section will explore the theoretical underpinnings of these techniques, providing a framework for understanding how they contribute to the overarching goal of efficient CNN deployment on specialized hardware platforms

3.1 Overview of Convolution Neural Networks

Convolutional Neural Networks (CNNs) have revolutionized the field of computer vision by enabling machines to understand and interpret visual data. One key aspect of CNNs is their ability to automatically learn hierarchical representations of features from raw data

CNNs, which are composed of multiple CONV layers as shown in Fig.3.1. In such networks, each layer generates a successively higher-level abstraction of the input data, called a feature map (fmap), which preserves essential yet unique information. The output of these layers is then passed through non-linear activation functions, such as ReLU, to introduce nonlinearity and enable the network to learn complex patterns in the data.

Another important component of CNNs is the pooling layer, which reduces the spatial dimensions of the feature maps while retaining important information. Pooling helps to make the network more robust to variations in the input data, such as changes in scale and orientation. Additionally, CNNs often include fully connected layers at the end of the network, which combine the features learned by the convolutional layers to make predictions about the input data.

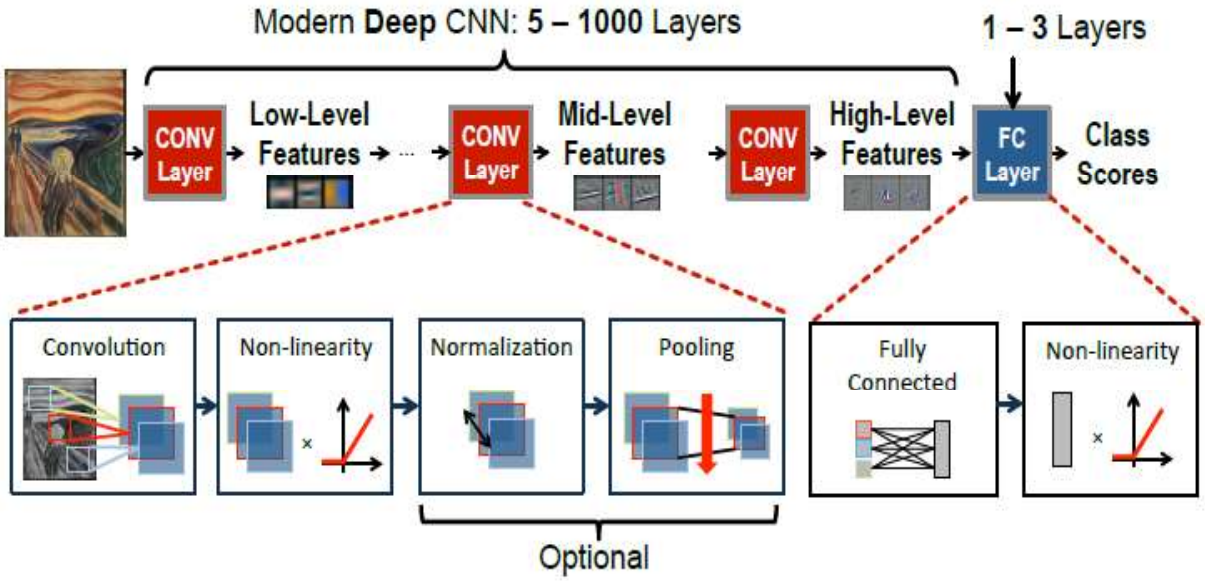

Fig. 3.1 Structure of typical CNN

The combination of convolutional, pooling, and fully connected layers allows CNNs to achieve state-of-the-art performance on a wide range of computer vision tasks, including image classification, object detection, and image segmentation.

3.2 Transfer Learning

Transfer learning is a technique in machine learning where a model trained on one task is repurposed on a second related task. This approach leverages the knowledge learned from the first task to improve performance on the second task, especially when the second task has limited labelled data. In the context of CNNs, transfer learning involves using a pre-trained CNN model on a large dataset, such as ImageNet, and fine-tuning it on a smaller dataset specific to the target task. This allows the model to quickly adapt to the new task and achieve good performance with less training data. The Fig 3.2 shows how transfer learning works with CNN.

One popular transfer learning model is the VGG (Visual Geometry Group) network, which was developed by the University of Oxford. The VGG network is known for its simplicity and

effectiveness, consisting of a series of convolutional layers followed by max-pooling layers, and ending with fully connected layers. The VGG network is pretrained on the ImageNet dataset, which contains millions of labelled images across thousands of categories. By fine-tuning the VGG network on a specific task, such as image classification or object detection, we can achieve state-of-the-art results with minimal effort.

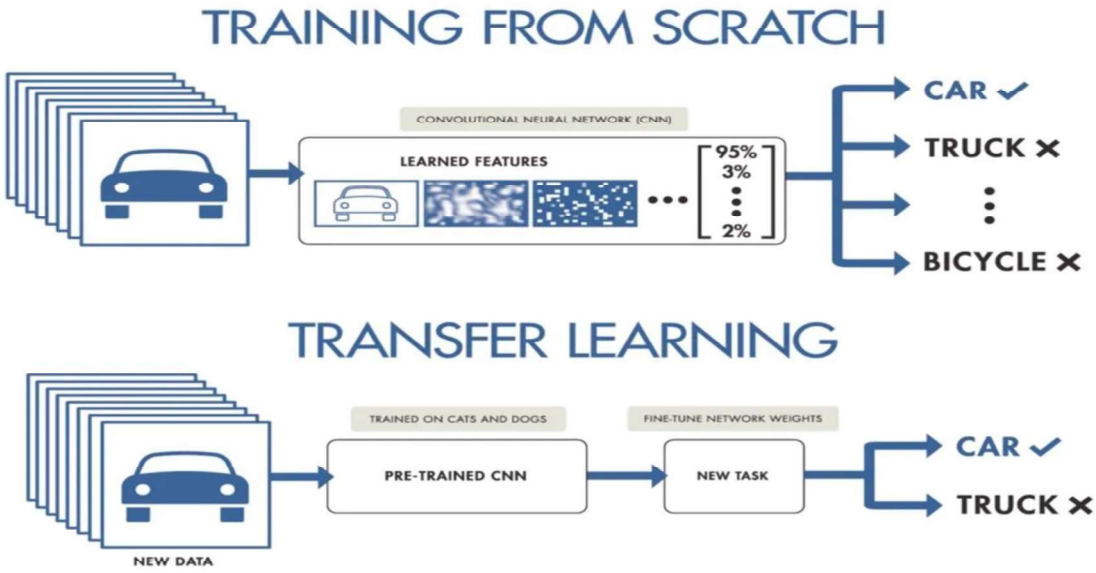

Fig. 3.2 Transfer Learning

In addition to VGG, other popular transfer learning models include Inception, ResNet, MobileNet, and DenseNet, each with its own strengths and applications. These models have been pretrained on large datasets and fine-tuned on specific tasks to achieve state-of-the-art results in various domains of computer vision. By using transfer learning and leveraging the knowledge learned from these pretrained models, researchers and practitioners can accelerate the development of new computer vision applications and achieve high performance with limited labelled data.

3.3 Quantization

Large models running on cloud environments have huge compute demand resulting in high cloud cost for developers, posing a major barrier to profitability and scalability. For edge deployments, edge devices are resource-constrained and therefore are limited in their ability to support large and complex models. Whether the model is deployed on the cloud or at the edge, AI developers are often confronted with the challenge of reducing their model size without compromising model

accuracy. Quantization is a common technique used to reduce the model size, though it can sometimes result in reduced accuracy.

Quantization is a model size reduction technique that converts model weights from high-precision floating-point representation to low-precision floating-point (FP) or integer (INT) representations, such as 16-bit or 8-bit. By converting the weights of a model from high-precision floating-point representation to lower-precision, the model size and inference speed can improve by a significant factor without sacrificing too much accuracy. Additionally, quantization will improve the performance of a model by reducing memory bandwidth requirements and increasing cache utilization.

Quantization can also introduce some challenges, particularly when using low-precision integer formats such as INT8. One major challenge is the limited dynamic range of these formats, which can lead to a loss of accuracy when converting from higher-precision floating-point representations. While FP16 can be used instead of FP32 with only a small loss in accuracy of the representation in the context of deep neural networks inference, the smaller dynamic range formats like INT8 pose a challenge. During quantization, we have to squeeze a very high dynamic range of FP32 into only 255 values of INT8. Fig 3.3 **is** a representation of the various datatypes used in CNNs.

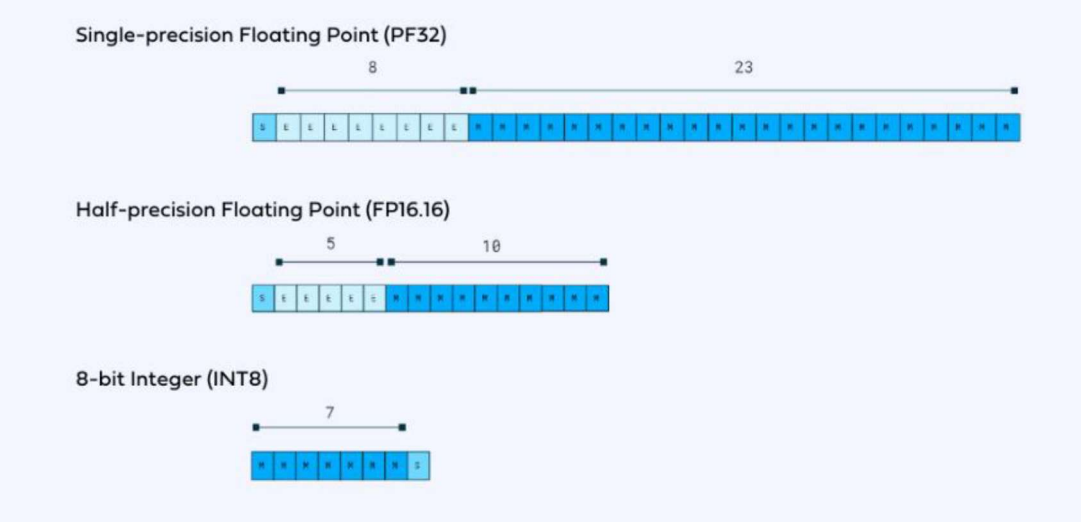

Fig.3.3 Different Data types used in Deep Learning

To mitigate this challenge, various techniques have been developed for quantizing models, such as per-channel or per-layer scaling, which adjust the scale and zero-point values of the weight and activation tensors to better fit the quantized format as shown in Fig 3.3. Other techniques, such as quantization-aware training, can also help to prepare a model for quantization by simulating the quantization process during training.

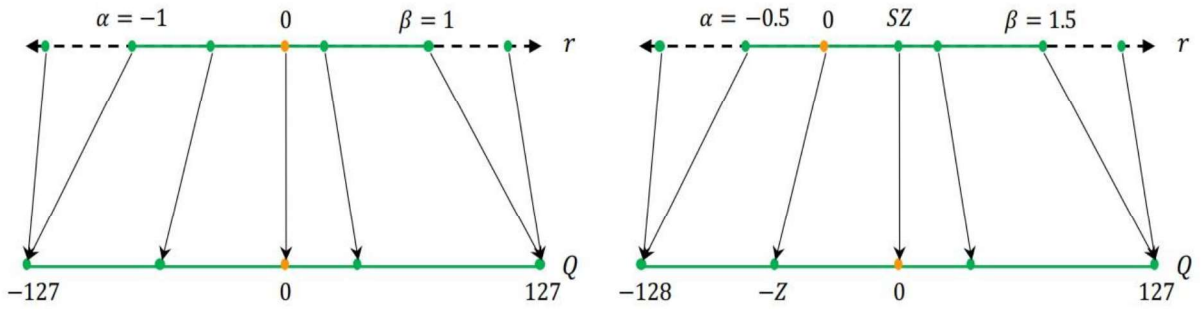

Fig.3.3 Symmetric and Asymmetric Quantization

Post-training quantization (PTQ) is a quantization technique where the model is quantized after it has been trained. Quantization-aware training (QAT) is a fine-tuning of the PTQ model, where the model is further trained with quantization in mind.Fig.3.4 shows the overview of PTQ and QAT. The quantization process (scaling, clipping, and rounding) is incorporated into the training process, allowing the model to be trained to retain its accuracy even after quantization, leading to benefits during deployment (lower latency, smaller model size, lower memory footprint).

3.4 Pruning

Convolutional Neural Networks (CNNs) operate by applying a set of filter channels to an input tensor, where each filter channel performs element-wise multiplication with the input values, resulting in a single output value at each location. This process is repeated across the entire input tensor, producing a new tensor. The output tensor's shape is determined by the size and movement of the filter channels, with the number of filter channels determining the depth of the output tensor, creating a single CNN layer.

Pruning is a technique used to remove unnecessary parameters from a trained model, improving its efficiency. In the context of CNNs, pruning involves removing entire feature maps from convolutional layers. By eliminating redundant or less informative feature maps, the pruned network can achieve faster inference times without significant loss of accuracy. Fig.3.5 illustrates the impact of pruning on a neural network, showing the network looks before and after the pruning process.

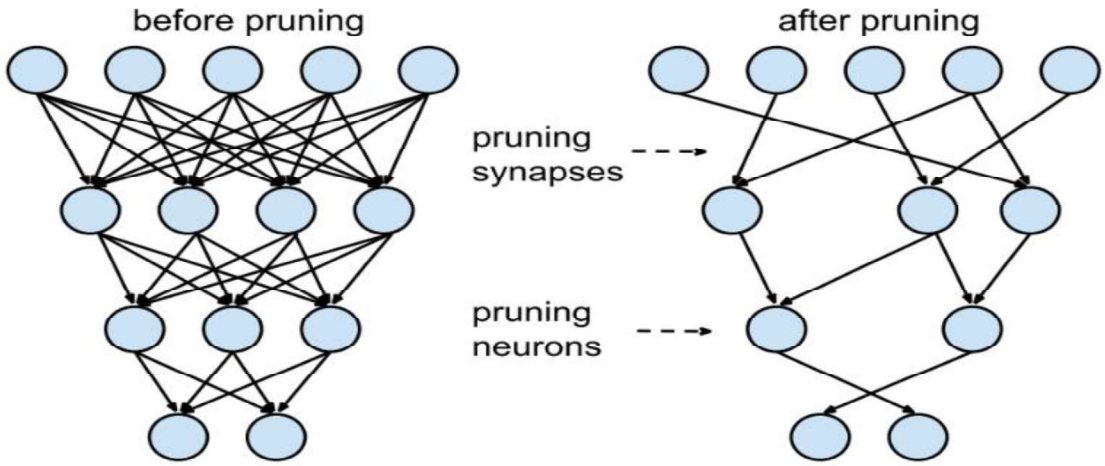

Fig.3.4 Pruning

If you could rank the neurons in the network according to how much they contribute to the output, you could then remove the low-ranking neurons from the network, resulting in a **smaller** and **faster** network.

3.5 Knowledge Distillation

As ML models have become larger, significant challenges have arisen regarding their deployment, especially on edge devices. These large models cannot be deployed on devices with limited resources, such as mobile phones and IoT devices. Additionally, the majority of data science modeling work focuses on training a single large model (or several different models) to perform well on a validation set which is often not representative of the real-world data. This tension between training and deployment objectives leads to the development of models that may be highly accurate, but often fail to meet performance, latency, and throughput benchmarks at the time of inference on real-world test data.

Knowledge distillation is a training technique that trains small models to be as accurate as larger models by transferring knowledge. In the domain of knowledge distillation, the larger model is referred to as the "teacher network," while the smaller network is known as the "student network." The teacher model can be a single large model or an ensemble of separate models, usually trained on complex data sources, including low-resolution data, multi-domain and multi-task data, or cross-modality data. After the teacher model is trained, the distillation process can capture and transfer the trained knowledge to the smaller student model, and then train the student network.

In knowledge distillation, the small student model learns to mimic the large model to achieve similar or even superior performance accuracy as show in Fig.3.6. Once the large deep neural network is appropriately compressed, it can be deployed on low-grade hardware devices to run real-world inferences.

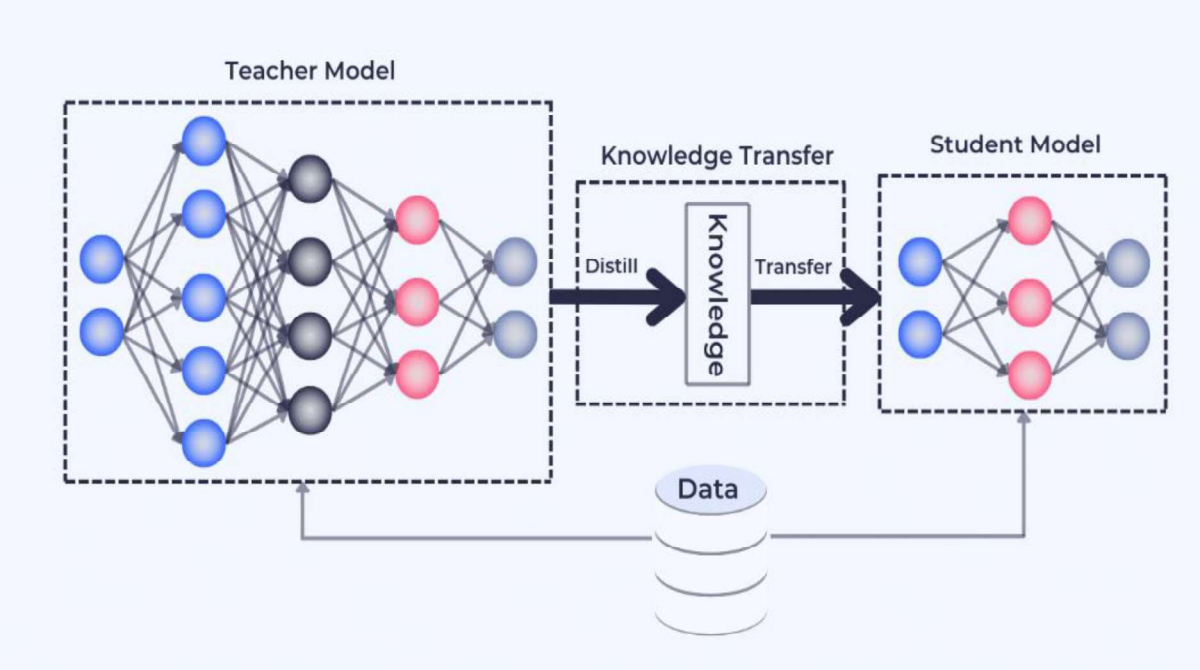

Fig.3.5.1 Knowledge Distillation

A knowledge distillation mechanism has three main components: knowledge, the distillation algorithm, and the teacher-student architecture. The teacher-student architecture design determines the quality of knowledge transfer between teacher and student models.

The teacher and student models share a relationship between their network structures that enable knowledge sharing. The student network can be one of the following:

- A simplified version of the teacher network
- A quantized version that preserves the structure of the teacher network
- A similar network as the teacher model
- A network with basic operations
- A network with an optimized and condensed structure

as shown in Fig.3.7.

To minimize the model capacity gap, it is best to minimize the difference between the teacher-student models using various distillations algorithms and optimization techniques. In simple knowledge distillation, the distilled model (student) is trained on the subset (transfer set) of data on which the teacher network is trained.

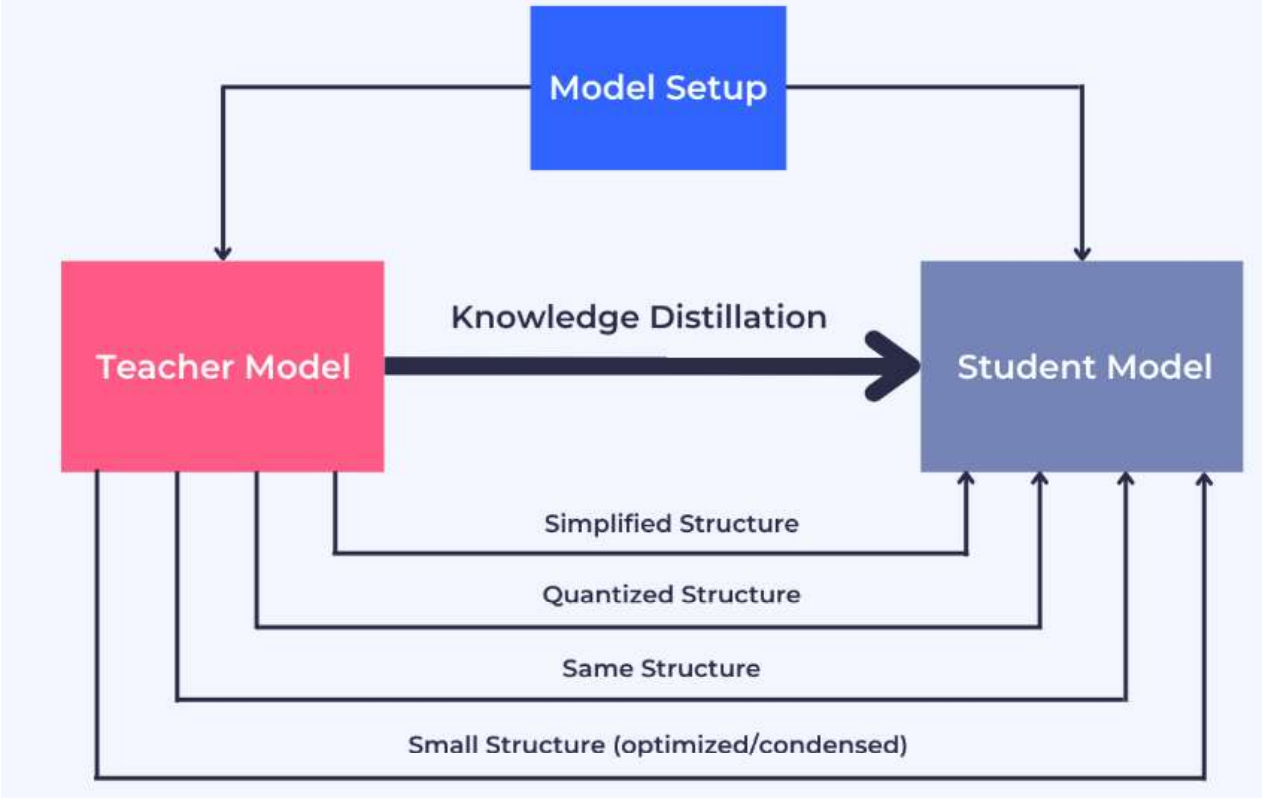

Fig 3.5.2 Types of Knowledge Distillation

3.6 Neural Architecture Search

Deep Neural Networks (DNNs) are powerful architectures but they are hard and time consuming to develop. There are numerous ways to structure and modify a neural network. In order to achieve the ultimate performance, there are various elements to consider including layer types, operations, and activation functions as well as the training data and deployment considerations (runtime, memory, and the inference hardware and its computational constraints).

Finding the most suitable deep learning architecture is a process that involves many trial and error iterations. Neural Architecture Search (NAS) provides an alternative to the manual designing of DNNs. The general idea behind NAS is to select the optimal architecture from a space of allowable architectures. The selection algorithm relies on a search strategy, which in turn depends on an objective evaluation scheme. The popular convolutional neural network EfficientNet is an example of an architecture generated by NAS.

Neural Architecture Search has three main building blocks that can be categorized in terms of -

- Search Space
- Search Strategy
- Performance Estimation Strategy

The NAS search space determines what type of architecture can be discovered by the NAS algorithm. It is defined by a set of operations that specify the overall structure of the network, the type of units or blocks that define the layers, as well as the allowable connectivity between layers to create architectures. The more elements the search space has, the more complex and versatile it gets. Types of operations used in defining the search space include sequential layer-wise operations, cell-based representation, hierarchical structure, and more.

NAS allows to:

- Automate the architecture selection and testing process
- Easily achieve better accuracy and inference speed results
- Shorten the iterative and cumbersome process of getting deep learning models to production

3.7 Vitis-AI and DPU

Vitis AI is an advanced development platform from AMD, specifically engineered to maximize AI acceleration on Xilinx FPGAs and Adaptive Compute Acceleration Platforms (ACAPs). This comprehensive toolkit includes optimized IP cores, practical tools, extensive libraries, pre-trained models, and sample designs, all curated to enhance efficiency and simplify the use for developers. Aimed at democratizing AI hardware acceleration, Vitis AI simplifies the development of deep learning inference applications by abstracting the complexities associated with the underlying FPGA and ACAP technologies. This allows even those developers with minimal FPGA expertise to easily harness the power of accelerated computing, thereby unlocking new possibilities in AI-driven applications.

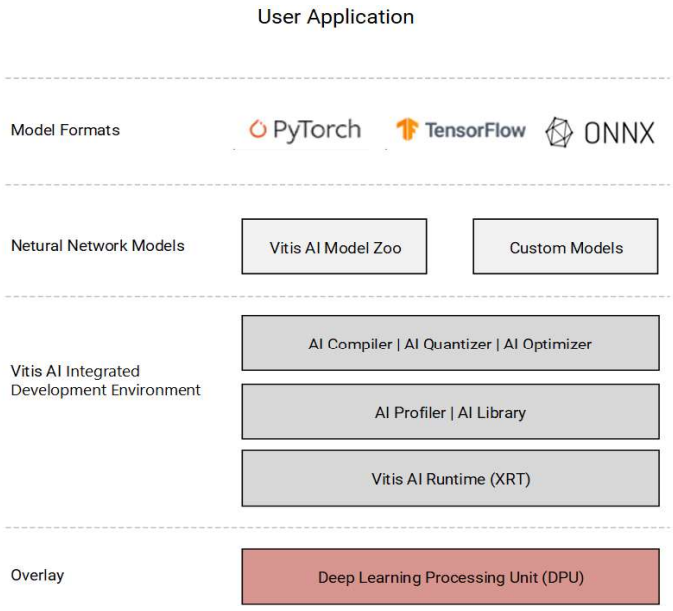

Fig 3.7 Vitis AI Stack

Vitis AI includes the following features:

- **Framework Support:** Compatible with mainstream deep learning frameworks and incorporates the latest model architectures for diverse AI tasks.
- **Pre-Optimized Models:** Offers a collection of ready-to-deploy models that are pre-optimized for Xilinx devices.
- **Advanced Quantization:** Features a powerful quantizer for model quantization, calibration, and fine-tuning. An optional AI optimizer is also available for advanced users, capable of pruning models by up to 90% with minimal accuracy loss.
- **Performance Analysis:** Provides layer-by-layer analysis tools to identify and address bottlenecks in model performance.
- **Unified APIs:** Includes high-level C++ and Python APIs to ensure maximum portability from Edge computing devices to Cloud platforms.
- **Customizable IP Cores:** Enables customization of efficient and scalable IP cores tailored to specific application needs, optimizing throughput, latency, and power efficiency.

The **Deep-Learning Processor Unit (DPU)** is a programmable engine specifically optimized for executing deep neural networks. It consists of parameterizable IP cores that are pre-implemented on hardware, simplifying the deployment process as no additional place and route steps are required. Designed primarily to enhance deep learning inference tasks, the DPU supports a wide array of computer vision applications including image and video classification, semantic segmentation, and object detection/tracking.

Scalability is a key feature of the DPU, making it adaptable across various Xilinx hardware platforms, from Zynq UltraScale+ MPSoCs to Kria KV260, Versal cards, and Alveo boards. This flexibility allows the DPU to meet diverse application requirements ranging from edge devices to cloud solutions.

Configuration and compilation of models for the DPU are handled through a specific JSON file (arch.json) generated during the Vitis workflow. Changes to the DPU's configuration necessitate regeneration of this file to ensure that model compilations align with the new settings.

Overall, Vitis AI's range of DPUs provides a versatile and powerful solution that can be tailored to specific needs regarding throughput, latency, scalability, and power efficiency, enhancing the capability to deploy advanced neural networks across various platforms.

3.7.1 DPU Naming

Different fields of DPU name is used to indicate different features or purposes, and the naming scheme is shown in the following figure 3.9:

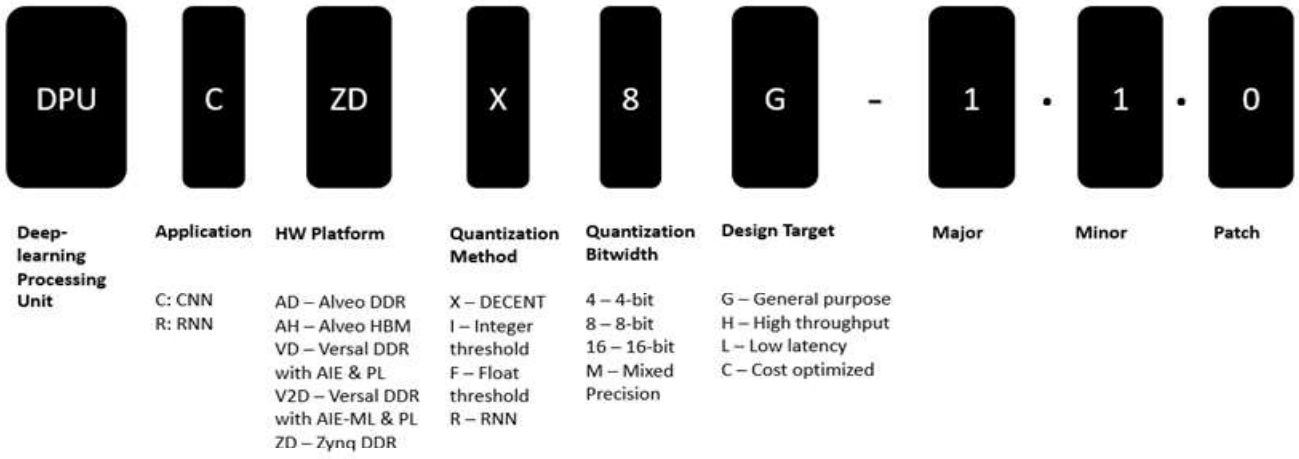

Fig 3.7.1 DPU Naming Scheme

3.7.2 DPUCZDX8G Architecture

The DPUCZDX8G IP core has been selected for its specific optimization with Zynq UltraScale+ MPSoCs, allowing for seamless integration into the programmable logic (PL) layer of these systems. This Deep-Learning Processor Unit (DPU) is central to our architecture, providing direct connectivity to the processing system (PS) and offering a user-configurable platform where parameters can be finely adjusted to optimize programmable logic resources or customize features based on the requirements of our project.

This DPU for its robust computational capabilities, which are essential for efficiently processing complex deep neural network models, including convolutional neural networks employed in a variety of computer vision tasks.

For integration and configuration details, we rely on guidance provided by Xilinx, which facilitates the incorporation of the DPUCZDX8G into our custom projects. Developers can access and download the necessary configuration files from the Xilinx website, enabling us to tailor the DPU's capabilities to our specific application needs.

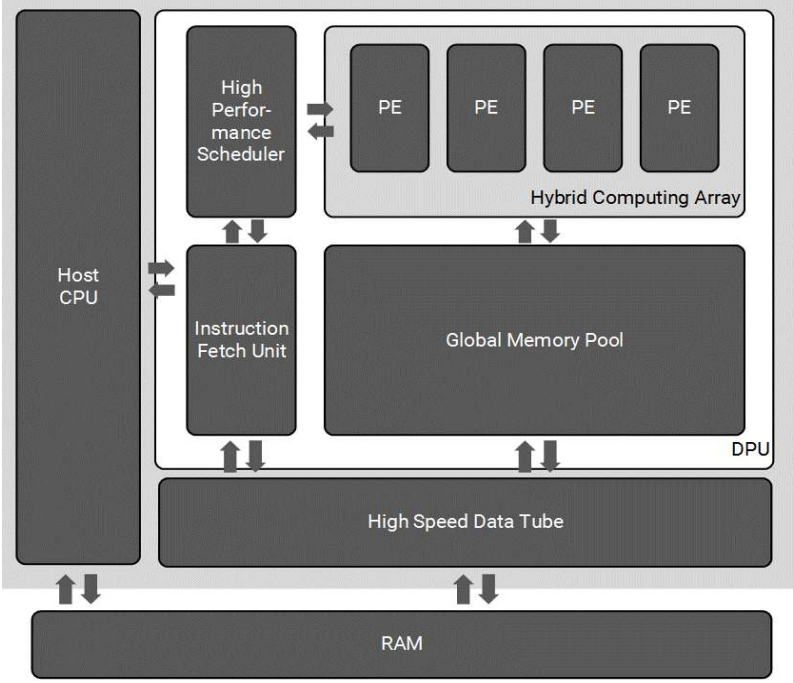

Fig3.7.2 DPUCZDX8G Architecture

CHAPTER-4

SYSTEM ARCHITECTURE AND DESIGN

4.1 Purpose

This chapter gives an overview of the design of the proposed system. The design covers the overall architecture of the system, starting with conceptual design and details added during subsequent phases of design. The static as well as dynamic behavior of the individual entities is detailed. The implementation and testing phases of the project are influenced by this documentation. The details are expected to evolve during the entire design process.

4.2 System Architecture

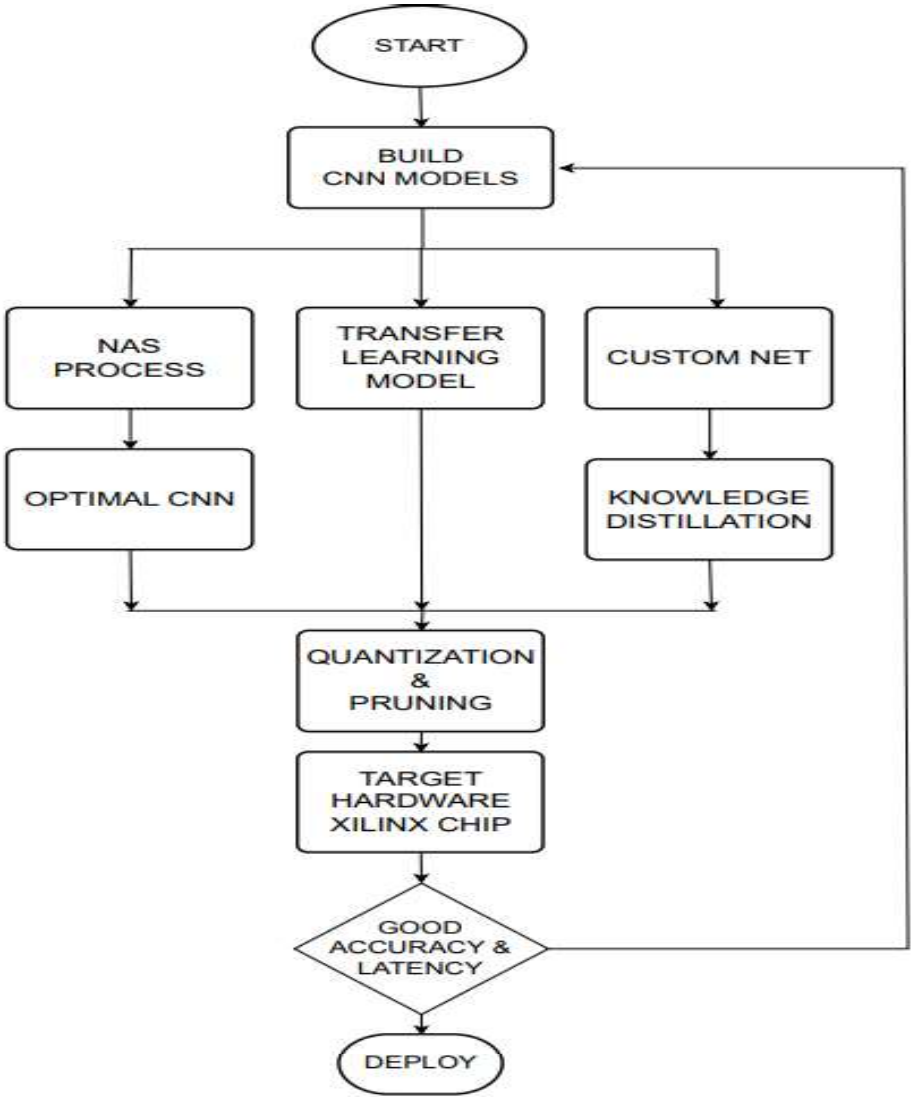

FIG 4.1 SYSTEM ARCHITECTURE

4.3 Design Approach

4.3.1 Model Generation:

We utilize transfer learning as a foundation. Pre-trained models provide a strong starting point, reducing development time and leveraging knowledge from previous tasks.

We employ Neural Architecture Search (NAS) to automatically explore different model architectures. This allows us to potentially discover even higher performing models compared to just transfer learning.

For specific use cases requiring custom architectures, knowledge distillation is used. This technique transfers knowledge from a larger, complex model to a smaller, custom-designed model, making it more efficient for the DPU.

4.3.2 Training and Evaluation:

During training, we will closely monitor the models' accuracy on a validation set. The validation set is a separate portion of our data specifically used for evaluating model performance without influencing the training process. By tracking accuracy on the validation set, we can identify when a model is starting to overfit the training data and losing its ability to generalize to unseen examples.

We will employ techniques like early stopping to prevent overfitting. Early stopping monitors the validation accuracy and stops training when it plateaus or starts to decrease, indicating overfitting. This helps us achieve the best possible accuracy on unseen data.

4.3.3 Deployment:

After training and optimization, the models are evaluated for their accuracy and latency on the DPU. This ensures they meet the performance requirements for the target application.

Vitis-ai compiler is then used to convert the optimized models into the XModel format, which is compatible with the DPUCZDX8G_ISA1_B4096 hardware. Finally, the XModel files are deployed on the target DPU for real-world performance testing.

4.3.4 Software and Hardware

Hardware: -

- **Processor:** Intel(R) Xeon(R) CPU @ 2.00GHz
- **Graphics Processing Unit (GPU):** NVIDIA Tesla T4 with 4GB VRAM
- **Memory (RAM):** 30 GB
- **Target Deep Processing Unit (DPU):** DPUCZDX8G_ISA1_B4096

Software: -

- **Operating System:** Ubuntu 22.04.3 LTS
- **Deep Learning Framework (Training):** PyTorch 2.2.0
- **Programming Language:** Python 3
- **DPU Development Framework (Deployment):** Vitis-AI 2.5
- **Online Machine Learning Platform (Optional):** Kaggle (For training and quantizing the model)

CHAPTER 5

METHODOLOGY AND IMPLEMENTATION

In this section, we outline the methodology and implementation details underpinning our research. This includes a systematic exploration of various advanced techniques aimed at optimizing convolutional neural network models for effective deployment on Deep Processing Units (DPU) via the Vitis-AI platform. The methodologies adopted encompass a series of strategic enhancements and optimizations such as Transfer Learning, Knowledge Distillation, Quantization, Pruning, and Neural Architecture Search (NAS). Each subsection below details the specific processes and datasets employed, ensuring a robust and transparent account of the experimental setup and execution phases of our study.

5.1 Methodology

5.1.1 Transfer Learning Models

In this study, we employed transfer learning to enhance the performance of convolutional neural networks on the EuroSat dataset, which comprises satellite images representing various land covers and uses. Transfer learning is utilized here as a strategic approach where a model developed for a specific task is repurposed to serve as the starting point for a model on a related task. This method is particularly advantageous under conditions of limited data availability or computational resources.

Through the strategic application of transfer learning, we have significantly improved the model's capability to accurately classify complex satellite images, while also optimizing the computational efficiency of the training process. This methodology has been pivotal in effectively leveraging deep learning technologies for enhanced performance on remote sensing data, demonstrating substantial benefits in practical applications.

5.1.2 Custom Models

Custom models are crucial in machine learning for addressing unique challenges specific to particular datasets, such as the EuroSat dataset's satellite images. Tailored specifically to the task at hand, custom models optimize performance by focusing solely on relevant features, which enhances their efficiency and accuracy. This specialization not only results in a smaller model

size and faster inference times, essential for real-time applications on edge devices, but also reduces computational demands.

Consequently, custom models operate effectively within the stringent resource limitations of edge devices, ensuring robust functionality with minimal resource use. Furthermore, their streamlined nature allows for greater scalability and flexibility, with reduced maintenance needs compared to generic models. Ultimately, developing custom models is key to efficiently processing specialized data, providing a practical solution that aligns advanced machine learning techniques with the operational realities of deploying technology in field applications.

5.1.3 Knowledge Distillation

Knowledge distillation is a technique used to compress the information of a large, complex model (the "teacher") into a smaller, more efficient model (the "student"). This approach addresses the challenge of deploying high-performance machine learning models on edge devices, which typically lack the computational resources to run large models directly.

For the EuroSat dataset, which encompasses a complex array of satellite images depicting various land uses, deploying large-scale convolutional networks directly on edge devices is not feasible. Knowledge distillation is employed to create custom models that maintain high accuracy while being tailored to the constraints of hardware typically found on edge devices:

By implementing knowledge distillation, we ensure that our custom models are not only practical for real-world applications but also maintain the integrity and accuracy needed for detailed satellite image analysis. This methodology thus bridges the gap between advanced machine learning techniques and their applicability in operational environments.

5.1.4 NAS

In our methodology, we employ **Neural Architecture Search (NAS)** to optimize neural network architectures specifically for analyzing the EuroSat dataset. This automated approach allows us to efficiently explore a wide range of architectural configurations, identifying models that balance accuracy with computational efficiency—essential for deployment on resource-constrained edge devices.

By leveraging NAS, we enhance the performance, adaptability, and efficiency of our models, making them ideally suited for real-world applications where computational resources are

limited. This process ensures our custom models are not only effective in high-dimensional data analysis but also practical for use in diverse operational environments.

5.1.5 Quantization

In our methodology, the optimization and deployment workflow begin with the initial training of models to accurately handle the complex EuroSat dataset. Post-training, knowledge distillation is employed to refine these models by transferring the capabilities of larger, complex "teacher" models to smaller, more efficient "student" models. Subsequently, these distilled models are further optimized using Neural Architecture Search (NAS), which hyper-tunes various architectural parameters to discover the most effective model structures.

The next step involves quantization, where the models' floating-point parameters are converted to lower precision formats—FP16 and INT8—significantly reducing the model's size and enhancing computational speed. This reduction is crucial for deployment on edge devices with limited resources, achieving a balance between maintaining high accuracy and operational efficiency with minimal impact on performance. Collectively, these processes—knowledge distillation, NAS, and quantization—create a comprehensive approach to developing high-performance, resource-efficient models suitable for real-world applications on constrained devices.

5.1.6 Pruning

In our comprehensive methodology for optimizing neural network models for deployment on resource-constrained edge devices, the final step after training, knowledge distillation, and Neural Architecture Search (NAS) is model pruning. Pruning effectively reduces the model size by systematically removing neurons that are not activated, i.e., those that do not significantly contribute to model outputs in various scenarios.

This technique streamlines the model by eliminating redundancy without compromising the model’s integrity. In practice, pruning achieves a substantial reduction in model size while typically incurring only minimal losses in accuracy, generally within the range of 1-2%. This

careful trimming enhances the model's efficiency by reducing the computational overhead required during inference, making it ideally suited for real-time applications on devices with limited computational capabilities.

Through this sequence of methodology robustly prepares neural networks to deliver high performance under the stringent operational constraints of edge computing environments.

5.1.7 Implementing on Hardware Using Vitis AI

As part of our methodology, we utilize the DPUCZDX8G IP core, specifically optimized for integration with Zynq UltraScale+. This Deep-Learning Processor Unit (DPU) is integral to enhancing the computational capabilities required for processing complex neural network models within our projects.

Configurable Convolution Architectures: The DPUCZDX8G IP supports multiple convolution architectures, which differ primarily in their degree of parallelism, directly impacting computational throughput and efficiency. The available configurations for this IP are as follows: B512, B800, B1024, B1152, B1600, B2304, B3136, and B4096. These configurations allow for tailored setup depending on the specific demands of the application, balancing between resource usage and performance.

Dimensions of Parallelism: The convolution unit within the DPUCZDX8G leverages three distinct dimensions of parallelism:

Pixel Parallelism (PP): This dimension dictates how many pixels are processed simultaneously, enhancing the speed of convolution operations.

Input Channel Parallelism (ICP): Parallel processing across different input channels. The DPUCZDX8G maintains equality between input and output channel parallelism, optimizing data flow and processing efficiency.

Output Channel Parallelism (OCP): Similar to ICP, this involves parallel processing across output channels and is directly correlated to the number of kernels employed during convolution computations.

Figure 5.1 illustrates these concepts, depicting a scenario where both input and output channel parallelisms are set to 3, and pixel parallelism is set to 2. This setup effectively demonstrates the concurrent processing capabilities of the DPU, which are critical for accelerating deep learning inference tasks.

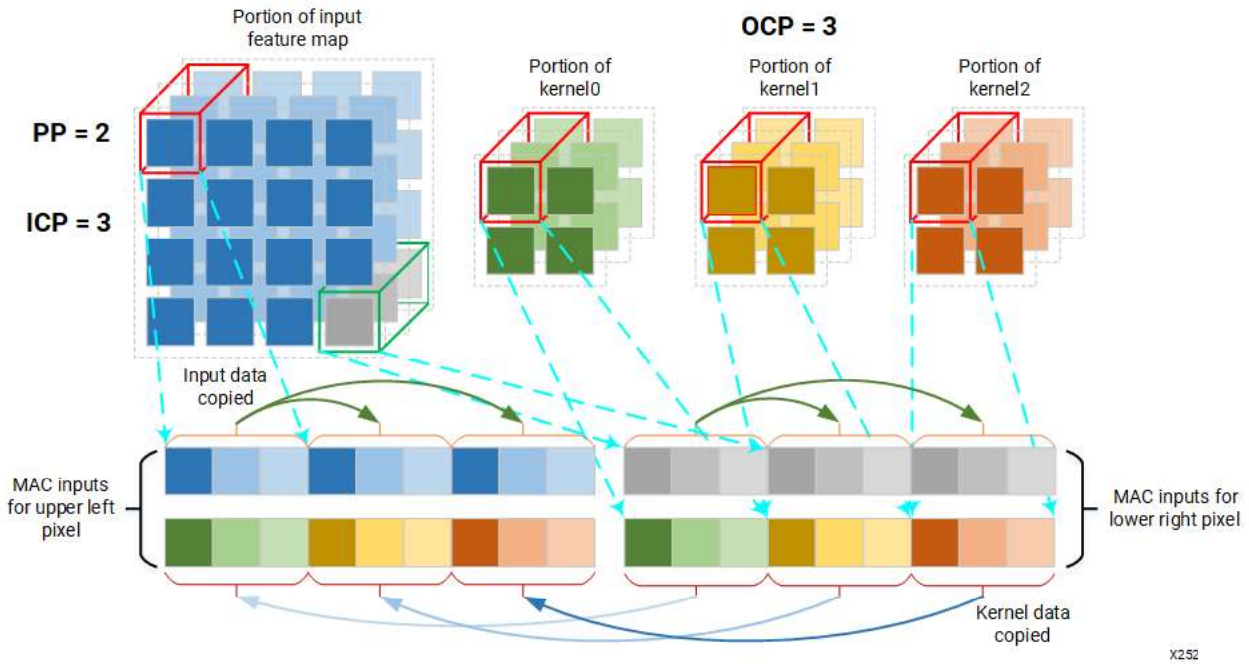

Fig 5.1.7 Visualizing the three dimensions of parallelism

Through the strategic application of these configurations and understanding the underlying hardware parallelism, our methodology ensures that the AI models deployed are not only efficient but also scalable across various platforms.

5.2 Implementation

5.2.1 Dataset

The study utilizes a subset of the Eurosat dataset, an assemblage of remote sensing imagery. Specifically, the Eurosat dataset encompasses an extensive collection of satellite imagery with high resolution, systematically categorized into **10 distinct landscape classes**. The imagery showcases in Figure 5.2 shows diverse landscapes across 34 European nations, providing a comprehensive geographical coverage. The dataset is comprised of 27,000 labeled images, each with a resolution of 64x64 pixels, offering a fine-grained view that is conducive for detailed analysis and classification tasks.

In addition to the Eurosat dataset, the study incorporates the Flower dataset to augment the analysis and validate the efficacy of the quantization process. The Flower dataset includes a diverse array of floral images, encompassing **14 distinct classes**. A selection of these images is depicted in Figure 5.3, illustrating the dataset's variety. The Flower dataset constitutes a total of 13,700 images, which were employed to bolster the research findings, particularly in the context of quantization's impact on model performance.

Fig 5.2.1(a) Sample of Eurosat Dataset

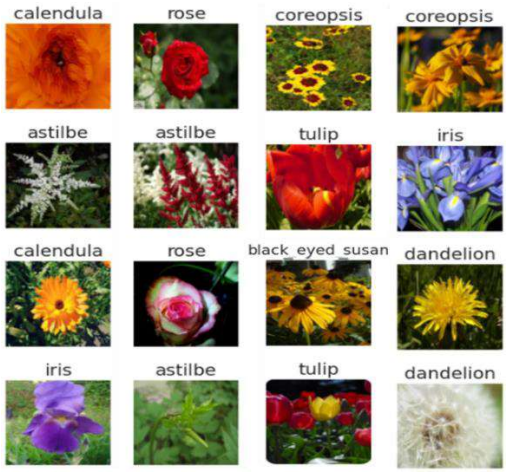

Fig 5.2.1(b) Sample of Flower Dataset

5.2.2 Transfer Learning Models

 # Transfer Learning on Pre-trained Network

```
# Load a pre-trained network
model = models.alexnet(pretrained=True)
model.name = "alexnet"
model

# Freeze parameters so we don't backprop through them
```

```
for param in model.parameters():
    param.requires_grad = False

# Define loss and optimizer
criterion = nn.NLLLoss()
optimizer = optim.Adam(model.classifier.parameters(), lr=0.001)

# Define deep learning method
epochs = 5
print_every = 30 # Prints every 30 images out of batch of 50 images
steps = 0

# Implement a function for the validation pass
def validation(model, testloader, criterion):
    test_loss = 0
    accuracy = 0

    for ii, (inputs, labels) in enumerate(testloader):

        # Uncomment below line if gpu is available
        inputs, labels = inputs.to(device), labels.to(device)

        output = model.forward(inputs)
        test_loss += criterion(output, labels).item()

        ps = torch.exp(output)
        equality = (labels.data == ps.max(dim=1)[1])
        accuracy += equality.type(torch.FloatTensor).mean()

    return test_loss, accuracy

# Implement a function for the Model training
def train(model,steps,print_every,epochs):
    print("Training process initializing .....\n")
        for e in range(epochs):
        running_loss = 0
        model.train()

        for ii, (inputs, labels) in enumerate(trainloader):
            steps += 1
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
```

```
            # Forward and backward passes
            outputs = model.forward(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

            if steps % print_every == 0:
                model.eval()

                with torch.no_grad():
                    valid_loss, accuracy = validation(model, validloader, criterion)

                print("Epoch: {}/{} | ".format(e+1, epochs),
                      "Training Loss: {:.4f} | ".format(running_loss/print_every),
                      "Validation Loss: {:.4f} | ".format(valid_loss/len(validloader)),
                      "Validation Accuracy: {:.4f}".format(accuracy/len(validloader)))

                running_loss = 0
                model.train()

    print("\nTraining process is now complete!!")
    return model
model = train(model,steps,print_every,epochs)
```

5.2.3 Custom Models

```
# Defining Custom Model Structure
img_inputs = tf.keras.Input(shape=(224, 224, 3))
x0 = tf.keras.layers.experimental.preprocessing.Rescaling(1./255)(img_inputs)

x1 = tf.keras.layers.Conv2D(32, (5,5), padding='same',activation='relu')(img_inputs)
x1 = tf.keras.layers.BatchNormalization()(x1)

x2 = tf.keras.layers.MaxPooling2D((4,4))(x1)

x3 = tf.keras.layers.Conv2D(64, (5,5), padding='same',activation='relu')(x2)
x3 = tf.keras.layers.BatchNormalization()(x3)

x4 = tf.keras.layers.MaxPooling2D((4,4))(x3)

x5 = tf.keras.layers.Conv2D(128, (5,5), padding='same',activation='relu')(x4)
```

```
x5 = tf.keras.layers.BatchNormalization()(x5)

x6 = tf.keras.layers.MaxPooling2D((4,4))(x5)

x7 = tf.keras.layers.Flatten()(x6)
x8 = tf.keras.layers.Dense(128,activation='relu')(x7)
x8 = tf.keras.layers.BatchNormalization()(x8)

x9 = tf.keras.layers.Dropout(0.5)(x8)
x10 = tf.keras.layers.Dense(10)(x8)

outputs = tf.keras.layers.Activation('softmax')(x10)

model = tf.keras.Model(inputs=img_inputs, outputs=outputs, name="custom_rs_classifier")
model.summary()
```

 # Training a neural network, using Keras (a high-level neural networks API)

```
model_path = "/kaggle/working/models/eurosat_custom_model.h5"

checkpoint = ModelCheckpoint(filepath=model_path, monitor="val_loss",
save_best_only=True)
reduce_lr = ReduceLROnPlateau(monitor="val_loss", factor=0.1, patience=5)
early_stopping = EarlyStopping(monitor="val_loss", patience=15, restore_best_weights=True,
verbose=True)

callback_list = [checkpoint, early_stopping, reduce_lr]

model.compile(loss="categorical_crossentropy",
              optimizer=Adam(learning_rate=1e-4),
              metrics=["accuracy"])

history = model.fit(
    training_set,
    validation_data=validation_set,
    callbacks=callback_list,
    epochs=100,
    verbose=1,)
```

#Loss and Accuracy Plots

```
plt.figure(figsize=(18, 5))

# Loss
```

```
plt.subplot(1, 2, 1)
plt.plot(history.history["loss"])
plt.plot(history.history["val_loss"])
plt.title("Loss")
plt.ylabel("loss")
plt.xlabel("epoch")
plt.legend(["train", "test"], loc="upper left")

# Accuracy
plt.subplot(1, 2, 2)
plt.plot(history.history["accuracy"])
plt.plot(history.history["val_accuracy"])
plt.title("Accuracy")
plt.ylabel("accuracy")
plt.xlabel("epoch")
plt.legend(["train", "test"], loc="upper left")

plt.show()
```

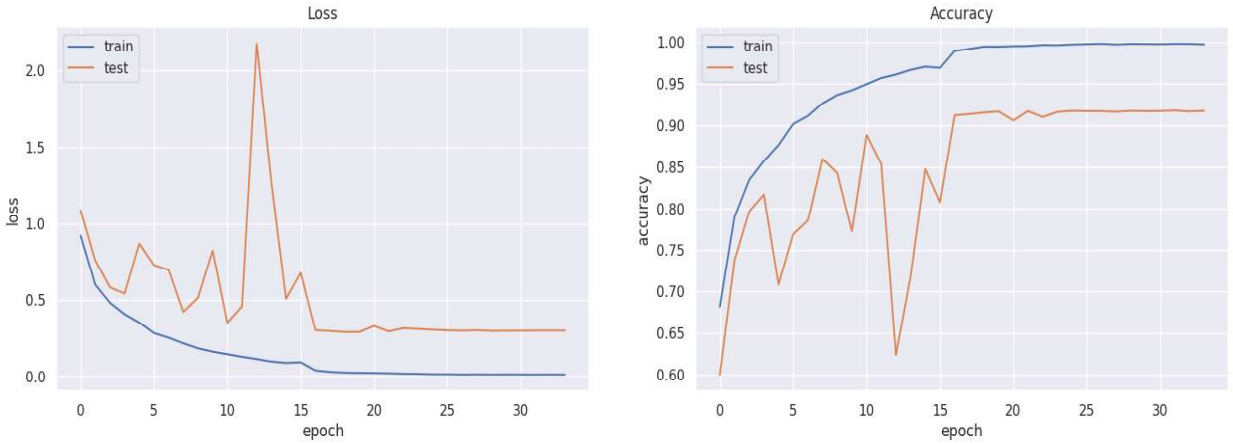

Fig 5.2.3 Loss and Accuracy Plots

5.2.4 Knowledge Distillation

#Training Student model

```
def train_knowledge_distillation(teacher, student, train_loader, epochs, learning_rate, T,
soft_target_loss_weight, ce_loss_weight, device):
    ce_loss = nn.CrossEntropyLoss()
    optimizer = optim.Adam(student.parameters(), lr=learning_rate)
    # Teacher set to evaluation mode
    teacher.eval()
    # Student to train mode
    student.train()
```

```
    for epoch in range(epochs):
        running_loss = 0.0
        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)

            optimizer.zero_grad()

            # Forward pass with the teacher model - do not save gradients here as we do not
change the teacher's weights
            with torch.no_grad():
                teacher_logits = teacher(inputs)

            # Forward pass with the student model
            student_logits = student(inputs)

            #Soften the student logits by applying softmax first and log() second
            soft_targets = nn.functional.softmax(teacher_logits / T, dim=-1)
            soft_prob = nn.functional.log_softmax(student_logits / T, dim=-1)

            soft_targets_loss = -torch.sum(soft_targets * soft_prob) / soft_prob.size()[0] * (T**2)

            # Calculate the true label loss
            label_loss = ce_loss(student_logits, labels)

            # Weighted sum of the two losses
            loss = soft_target_loss_weight * soft_targets_loss + ce_loss_weight * label_loss

            loss.backward()
            optimizer.step()

            running_loss += loss.item()

        print(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss / len(train_loader)}")
#Calling Train function
train_knowledge_distillation(teacher=model_teacher, student=new_m_student,
train_loader=trainloader,
                             epochs=15, learning_rate=0.001, T=2, soft_target_loss_weight=0.25,
                             ce_loss_weight=0.75, device=device)
```

5.2.5 NAS (Neural Architecture Search)

Define the Search Space

A json file with the search space defined

```
{
  "num_units_layer1": {
    "_type": "choice",
    "_value": [128, 256, 512, 1024]
  },
  "activation_layer1": {
    "_type": "choice",
    "_value": ["relu", "tanh", "sigmoid"]
  },
  "dropout_rate": {
    "_type": "uniform",
    "_value": [0.1, 0.5]
}
```

Modify the Model Script for NNI

```
def create_model(input_shape, num_classes, params):
  # Start modifying the model here
  base_model = load_model(model_path)
  base_model.trainable = False  # Freeze the layers

  # Add custom layers on top of the loaded base model
  x = base_model.output
  x = Dense(params['num_units'], activation='relu')(x)
  x = Dropout(params['dropout_rate'])(x)
  predictions = Dense(num_classes, activation='softmax')(x)

  model = tf.keras.Model(inputs=base_model.input, outputs=predictions)
  model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
  return model

def main():
  # NNI provides the parameters based on the search space
  params = nni.get_next_parameter()
  model = create_model(input_shape, num_classes, params)
  history = model.fit(training_data, training_labels, validation_data=(val_data, val_labels),
epochs=50, verbose=0)

  # Report the results to NNI
  nni.report_final_result(history.history['val_accuracy'][-1])
```

```
if __name__ == '__main__':
    main()
```

Create the NNI Configuration File and training

```
authorName: default
experimentName: NAS_Custom_Model
trialConcurrency: 2
maxExecDuration: 1h
maxTrialNum: 20
trainingServicePlatform: local

trial:
  command: python3 your_script.py
  codeDir: .
  gpuNum: 1

searchSpacePath: search_space.json
useAnnotation: false
tuner:
  builtinTunerName: TPE

nnictl create --config config.yml --port 8080
```

5.2.5 Quantization

 # Quantization to fp16 (loaded model is fp32 by default)

```
model_fp16 = copy.deepcopy(model)

model_fp16.half()

weights_half = model_fp16.state_dict()
print(weights_half['features.0.weight'].dtype)

model_fp16.to(device)
```

#Quantization to fp64 (loaded model is fp32 by default)

```
model_64 = copy.deepcopy(model)
model_64.double()

weights_64 = model_64.state_dict()
print(weights_64['features.0.weight'].dtype)
model_64.to(device)
```

#Quantization to Int8 (loaded model is fp32 by default)

```
from torch.ao.quantization import QuantStub, DeQuantStub
import torch
from torch.ao.quantization import (
  get_default_qconfig_mapping,
  get_default_qat_qconfig_mapping,
  QConfigMapping,
)
import torch.ao.quantization.quantize_fx as quantize_fx
```

#PTQ (Post Training Quantization)

```
model_fp32 = copy.deepcopy(model)
model_fp32.to('cpu')
input_data = next(iter(trainloader))[0]
calibrate_data = input_data.to("cpu")
model_int8 = copy.deepcopy(model_fp32)
#model_int8.to(device)
qconfig_mapping = get_default_qconfig_mapping("qnnpack")
model_int8.eval()
# prepare
model_prepared = quantize_fx.prepare_fx(model_int8, qconfig_mapping, calibrate_data)
# calibrate
with torch.no_grad():
    for i in range(20):
        batch = next(iter(trainloader))[0]
        output = model_prepared(batch.to('cpu'))
model_quantized_static = quantize_fx.convert_fx(model_prepared)
model_quantized_static.to('cpu')
```

#QAT (Quantize Aware Training)

```
input_data = next(iter(trainloader))[0]
calibrate_data = input_data.to(device)
model.eval()
model.qconfig = torch.ao.quantization.get_default_qat_qconfig('x86')
model.train()
qconfig_mapping = get_default_qat_qconfig_mapping("x86")
model_prepared = quantize_fx.prepare_qat_fx(model, qconfig_mapping, calibrate_data)
model_prepared.to(device)
model_prepared = train(model_prepared,steps,print_every,epochs)
model_prepared.to('cpu')
model_quantized_trained = quantize_fx.convert_fx(model_prepared)
model_quantized_trained.to('cpu')
```

5.2.6 Pruning

Performing Pruning

```
# Define the pruning schedule, e.g., from 0% to 50% sparsity over 5 epochs
pruning_schedule = tfmot.sparsity.keras.PolynomialDecay(initial_sparsity=0.0,
                                                        final_sparsity=0.5,
                                                        begin_step=0,
                                                        end_step=5)

# Apply pruning to the whole model
pruning_params = {
   'pruning_schedule': pruning_schedule,
   'block_size': (1, 1), # The dimensions for the block of weights to prune
   'pruning_policy': tfmot.sparsity.keras.PruningPolicy.PERMANENT
}
pruned_model = tfmot.sparsity.keras.prune_low_magnitude(model, **pruning_params)

# Compile the pruned model
pruned_model.compile(optimizer='adam',
                     loss='categorical_crossentropy',
                     metrics=['accuracy'])
#traing the model

logdir = 'logs'

callbacks = [
   tfmot.sparsity.keras.UpdatePruningStep(),
   tfmot.sparsity.keras.PruningSummaries(log_dir=logdir),
   ModelCheckpoint(filepath='pruned_model.h5', save_best_only=True)
]
# Assuming training_data and validation_data are preloaded datasets
history = pruned_model.fit(training_data, epochs=5, validation_data=validation_data,
callbacks=callbacks)

#Evaluation
loss, accuracy = final_model.evaluate(validation_data)
print(f'Final model accuracy: {accuracy*100:.2f}%')
```

5.2.7 Utilize the DPUCZDX8G IP core with Vitis AI

Model Quantization

```
# Setup the Vitis AI environment
source <path_to_vitis_ai>/setup/alveo/u200_u250/overlaybins/setup.sh
# Run quantization for a sample model
python quantize.py --model <path_to_model>/model.pb --calib_dataset
<path_to_calibration_dataset> --output_dir ./quantized_model
# Compile the quantized model
vai_c_xir -x ./quantized_model/deploy_model.xmodel -a <path_to_arch>/arch.json -o
./compiled_model -n model_name
```

Application Code

Write the application code to load and run the model on the DPU. Using Python and the Vitis AI Runtime (VART)

```
# Load the compiled model
runner = vart.Runner.create_runner('compiled_model/model_name.xmodel', 'run')
# Prepare input data
input_data = np.random.rand(1, 224, 224, 3)
# Execute the model
job_id = runner.execute_async([input_data], [])
runner.wait(job_id)
# Retrieve the output
output_data = runner.get_output_tensors()[0].data
print("Model output:", output_data)
```

Inspecting the Converted Model

```
inspector = vitis_inspect.VitisInspector(target = "DPUCADF8H_ISA0")
#change model name before executing for different models
inspector.inspect_model(model, input_shape = [224, 224, 3],
                        plot = True, plot_file = r"./Inspect_results/custom_eurosat_model_224.svg",
                        dump_results = True,
                        dump_results_file = r"./Inspect_results/custom_eurosat_model_224.txt",
                        verbose = 1)
```

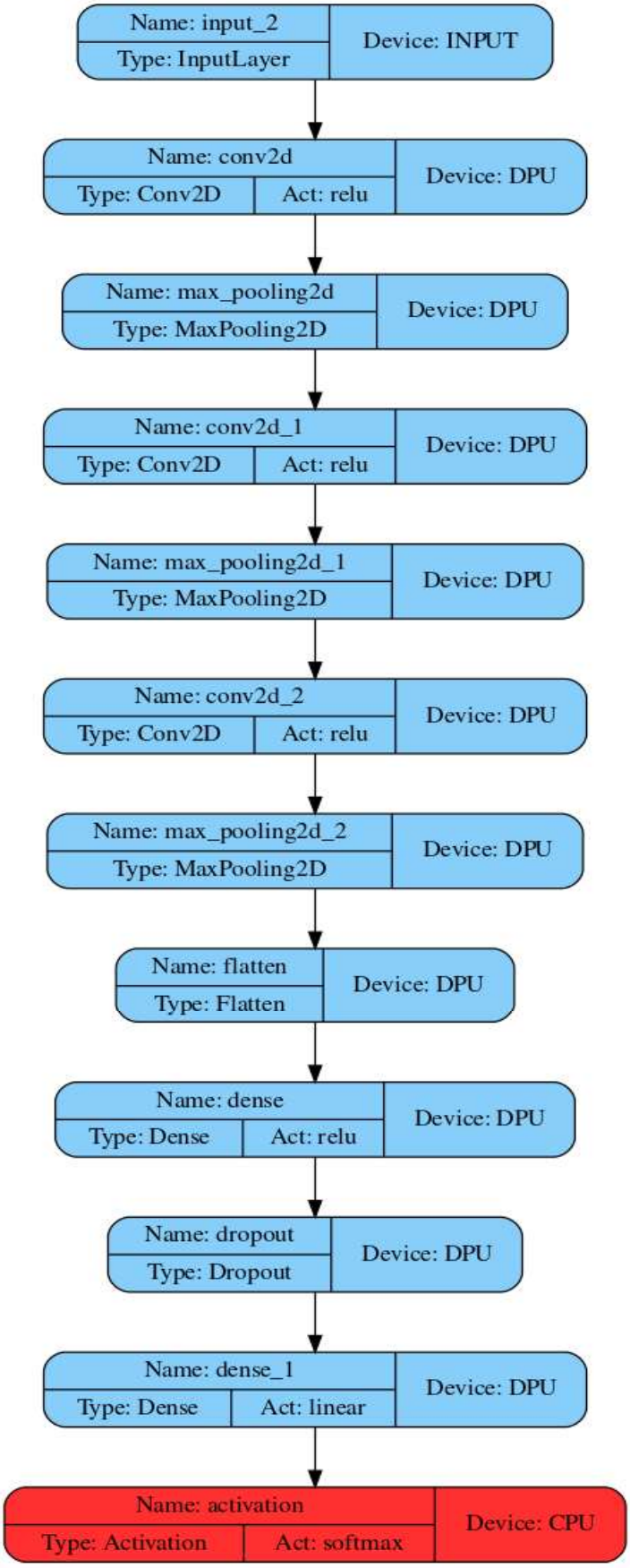

Fig 5.2.7 Inspection results for a custom model on Vitis-AI

CHAPTER 6

EXPERIMENTAL RESULT AND ANALYSIS

This section presents the evaluation results of renowned convolutional neural network models on the EuroSat and Flower Datasets. In addition to these popular models, the CustomNet models, which were specifically developed for inference on Deep Processing Units (DPUs), were exclusively tested on the EuroSat Dataset. This comprehensive assessment covered various computational precisions, including FP-64, FP-32, and INT-8—with implementations using both Post-Training Quantization (PTQ) and Quantization Aware Training (QAT). The analysis meticulously highlights the effects on model accuracy and size as a function of precision changes. Our findings underscore the superior performance of QAT over PTQ when models are quantized to INT-8, confirming its effectiveness in preserving accuracy at reduced precision levels. Additionally, the results demonstrate the robustness of the CustomNet models, which consistently outperformed their counterparts in these tests. This evaluation is essential for understanding the critical trade-offs between model accuracy and computational efficiency, which vary significantly across different hardware configurations.

6.1 Knowledge Distillation Results

In this section, we present the results of applying Knowledge Distillation to our Custom neural network models trained on the EuroSat dataset. Knowledge Distillation involves transferring the learned knowledge from a larger, complex "teacher" model to a smaller, more computationally efficient "student" model. This technique not only simplifies the model architecture without significantly compromising on accuracy but also ensures the models are suitable for deployment on resource-constrained edge devices.

6.1.1 Parameter Size

```
Teacher parameters: 11,341,252
Student parameters: 1,662,076
```

6.1.2 Model Size

```
Teacher: 45.44 MB
Student: 6.66 MB
```

6.2 Accuracy

The accuracy of various CNN models at different computational precisions is detailed in Table I and Table II. These precisions include full precision (FP-64), single precision (FP-32), and quantized integer precision (INT-8) employing both PTQ and QAT.

Our evaluation demonstrates that the transition from FP-64 to FP-32, and subsequently to INT-8, results in an accuracy drop within the expected range of 1-5%. Notably, at the INT-8 precision level, models trained with QAT significantly outperform those adjusted via PTQ,

Among the tested models, the CustomNet architecture exhibited the highest accuracy on the EuroSat Dataset, outperforming other renowned models. This superior performance underscores the advantages of the CustomNet design, which is optimized for specific hardware configurations such as the DPU.

In contrast, on the Flower classification dataset, VGG-16 demonstrated superior performance. This supports the well-known principle in deep learning that more complex features require deeper networks, hence reinforcing the famous dictum: “We need to go deeper!".

TABLE I

MODEL ACCURACY FOR DIFFERENT PRECISIONS ON FLOWER DATASET

MODELS	FP-64	FP-32	FP-16	INT-8	
				PTO	QAT
VGG-16	**86%**	**84%**	**84%**	**82%**	**83%**
Alex Net	84%	83%	82%	81%	82%
ResNet-18	83%	82%	82%	81%	81%
MobileNetV2	81%	80%	79%	77%	78%

TABLE II

MODEL ACCURACY FOR DIFFERENT PRECISIONS ON EUROSAT DATASET

MODELS	FP-64	FP-32	FP-16	INT-8	
				PTQ	QAT
VGG-16	85%	86%	86%	83%	83%
LeNet	85%	85%	85%	80%	82%
ResNet-18	83%	82%	81%	67%	80%
MobileNetV2	87%	86%	85%	56%	82%
AlexNet	84%	84%	84%	83%	84%
VGG-13	86%	85%	85%	84%	85%
SqueezeNet	86%	86%	86%	82%	84%
VGG-19	87%	86%	86%	83%	86%
ConvNeXT	89%	88%	88%	64%	72%
CustomNet	**93.24%**	**92.18%**	**91.83%**	**91.56%**	**91.64%**

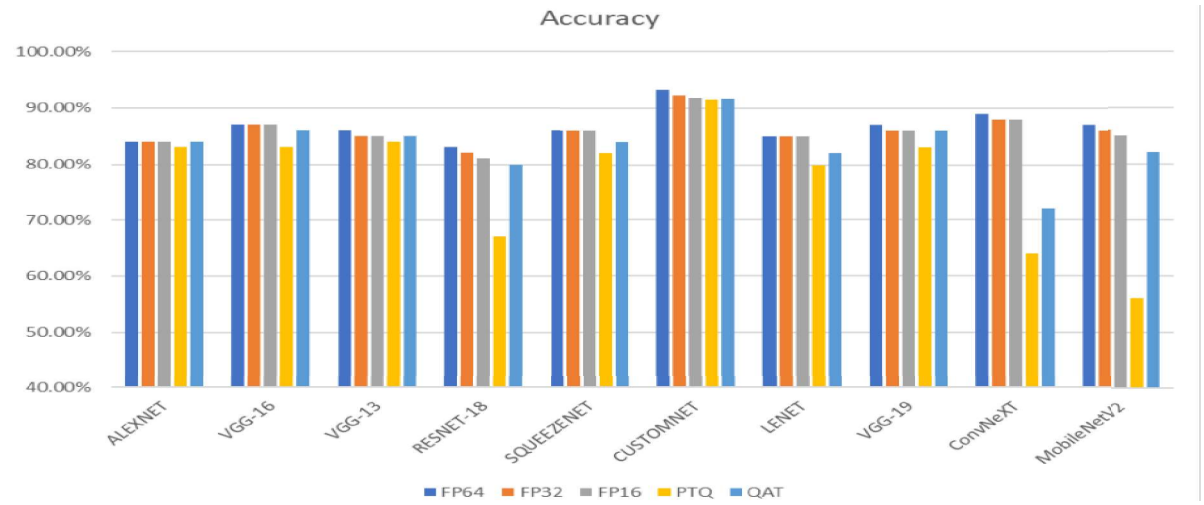

Fig 6.2 Comparison of Accuracy Across Different Precision

6.2.1 Confusion matrix of Custom model before Quantization

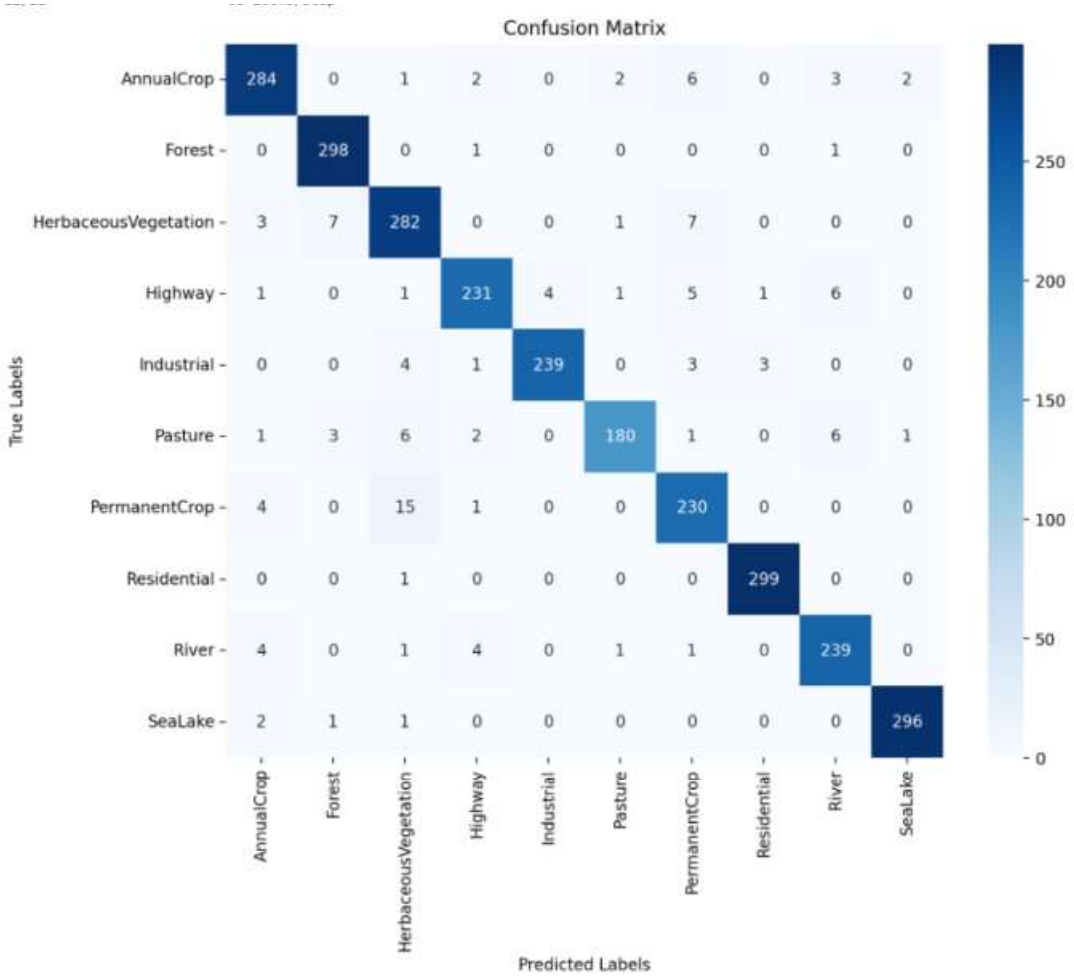

Fig 6.2.1: Confusion Matrix before Quantization

6.2.2 Confusion matrix of Custom model after Quantization

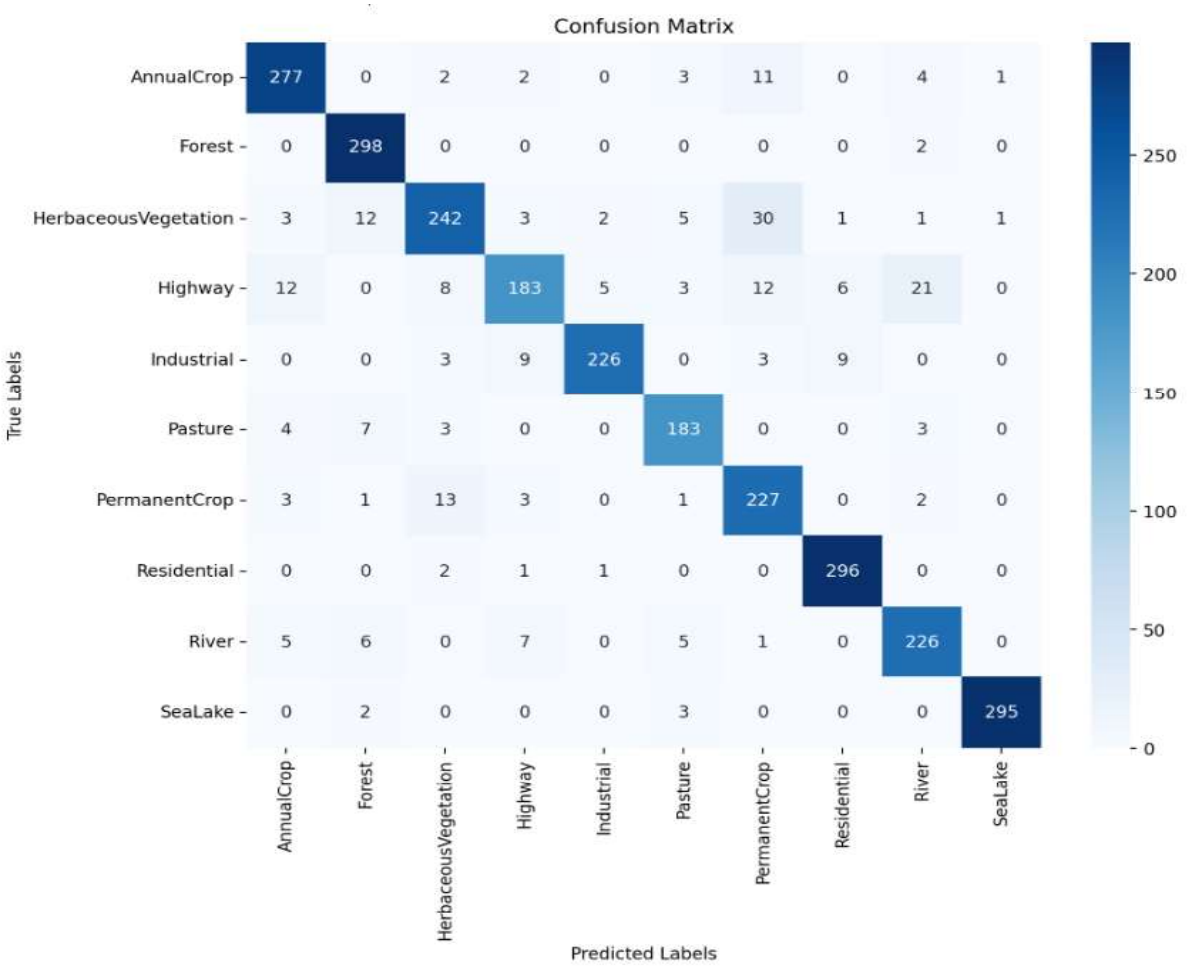

Fig 6.2.2: Confusion Matrix after Quantization

6.3 Size

The impact of quantization on model size is illustrated in Figure 6.2. When the models were quantized to FP16, their size was reduced by half, and further quantization to INT8 resulted in a reduction to one-quarter of the original size. Conversely, increasing the computational precision to FP64 doubled the size of the models. Despite these significant changes in model size, there were no substantial losses in accuracy; the models largely maintained their performance levels. These outcomes are consistent with expectations and lend further support to the underlying motivations of this study.

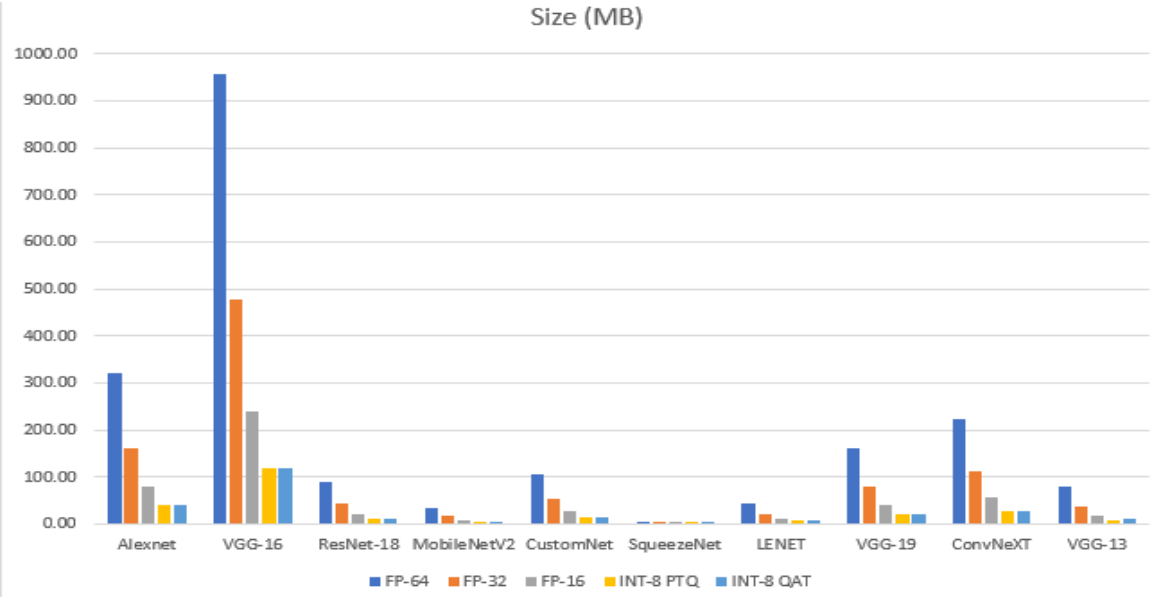

Fig 6.3 Size Drop Across Different Precision

6.4 Inferencing on DPU

The quantized models were subsequently processed and converted into the .xmodel format, which is optimized for execution on Deep Processing Units (DPU). This conversion process resulted in a modest accuracy reduction of 1-2%, as detailed in Table III. However, this slight decrement did not significantly impact the overall performance of the models, aligning well with the anticipated outcomes of this research.

TABLE III

MODEL ACCURACY ON DPU

Metric	Custom Model 1	Custom Model 2	Custom Model 3	Custom Model 4	Custom Model 5
Input Size	224x224	224x224	224x224	224x224	224x224
# of Conv	6	6	3	4	4
# of FC	1	2	3	5	3
Total Weights	204k	335k	408k	10M	5M
FP32 Accuracy	95.24%	93.21%	92.35%	94.12%	92.26%
INT8 Accuracy	93.64%	91.85%	88.93%	91.86%	88.98%
DPU Accuracy	91.73%	91.05%	89.04%	90.35%	88.13%
FPS	604	519	563	480	513

CONCLUSION & FUTURE SCOPE

7.1 Conclusion

In conclusion, our project has been an exploration into optimizing Convolutional Neural Networks (CNNs) for deployment on deep processing units (DPUs), for the purpose of improving the inference speed and latency for applications in fields that suffer from slow cloud inference, such as Autonomous driving, healthcare. etc. Utilizing techniques such as, Transfer Learning, Quantization, NAS, Knowledge Distillation, and Pruning, explored many different network architectures, downsized models in different precision while maintaining accuracy, and even deployed them on an Edge Device such as a DPU to demonstrate accelerated inferencing.

Transfer learning was extensively used, allowing us to take advantage of pre-trained models and fine-tune them for our specific task, so that we can get a better grasp on quantization, in order to understand and quantize the CustomNet Models for our main application.

Quantization proved to be another crucial aspect of our optimization efforts. By quantizing our model, we were able to significantly reduce its size without sacrificing much in terms of accuracy. This reduction in model size not only conserves memory but also facilitates faster inference speeds, which enabled us to deploy these models on our DPU.

NAS further contributed to our project by enabling us to automatically discover network architectures tailored to our dataset and computational constraints. We used NAS on our CustomNet model for the Remote Sensing application. This automated approach allowed us to explore a vast search space efficiently and discover architectures that strike a balance between accuracy and efficiency, thereby maximizing the performance of our CNN on DPUs.

Overall, our project has demonstrated the effectiveness of these optimization techniques in enhancing the performance and efficiency of CNNs for deployment on DPUs. By leveraging transfer learning, quantization, neural architecture search, knowledge distillation, and pruning, we have achieved fast inference speeds and minimal accuracy drop, making our optimized CNNs well-suited for real world edge deployment.

7.2 Future Scope

While in our project, we have explored deployment on edge devices, in the future, we hope to explore a hybrid, Edge-Cloud model for inferencing, to integrate into systems that cannot work on edge alone. Edge-Cloud model for inferencing represents a natural progression from our focus on deployment solely on edge devices in the current project. While edge computing offers advantages such as low latency and reduced bandwidth usage, there are scenarios in which, relying solely on edge devices may not be feasible due to resource constraints or the need for extensive computational power. In such cases, integrating edge and cloud computing into a hybrid model can offer a compelling solution that leverages the strengths of both paradigms. By dynamically allocating workloads between edge devices and cloud servers based on factors such as device capabilities, network conditions, and application requirements, we can optimize resource usage and minimize operational costs while ensuring efficient inferencing across diverse deployment scenario

REFERENCES

[1] Francesco Daghero, Daniele Jahier Pagliari, and Massimo Poncino. Energy-efficient deep learning inference on edge devices, chapter 8,pages 247–301. ELSEVIER, 09 2020.

[2] Thomas Elsken, Jan Metzen, and Frank Hutter. Neural architecture search: A survey. Proceedings of the IEEE, 08 2018.

[3] Xijie Huang, Zechun Liu, Shih-Yang Liu, and Kwang-Ting Cheng. Efficient quantization-aware training with adaptive coreset selection. Proceedings of the IEEE, 2023.

[4] Sledevič, T.; Serackis, A. mNet2FPGA: A Design Flow for Mapping a Fixed-Point CNN to Zynq SoC FPGA. *Electronics* **2020**, *9*, 1823. https://doi.org/10.3390/electronics9111823

[5] Liu, B.; Zou, D.; Feng, L.; Feng, S.; Fu, P.; Li, J. An FPGA-Based CNN Accelerator Integrating Depthwise Separable Convolution. *Electronics* **2019**, *8*, 281. https://doi.org/10.3390/electronics8030281

[6] Thomas Elsken, Jan Metzen, and Frank Hutter. Neural architecture search: A survey. Proceedings of the IEEE, 08 2018.

[7] Junghyup Lee, Dohyung Kim, and Bumsub Ham. Network quantization with element-wise gradient scaling. Proceedings of the IEEE, pages 6444–6453, 06 2021.

[8] Amy Mcgovern and Kiri Wagstaff. Machine learning in space: Extending our reach. Machine Learning, 84:335–340, 09 2011.

[9] The future of ml deployment: Trends and predictions. https://www.modelbit.com/blog/future-of-ml-model-deployment

[10] Kritika Rana, Parampreet Kaur, and Research Publications. Review on machine learning based algorithms used in autonomous cars. SSRN Electronic Journal, 5:114–118, 10 2018.

[11] Gil Shomron, Freddy Gabbay, Samer Kurzum, and Uri Weiser. Post training sparsity-aware quantization. Proceedings of the IEEE, 2021.

[12] Raghubir Singh and Sukhpal Singh Gill. Edge ai: A survey. Internet of Things and Cyber-Physical Systems, 3:1–22, 03 2023.

[13] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. Efficient processing of deep neural networks: A tutorial and survey. Proceedings of the IEEE, 105, 03 2017.

[14] Chen Tang, Kai Ouyang, Zhi Wang, Yifei Zhu, Yaowei Wang, Wen Ji, and Wenwu Zhu. Mixed-precision neural network quantization via learned layer-wise importance. Proceedings of the IEEE, 2023.

[15] Virendra Verma and Savita Verma. Machine learning applications in healthcare sector: An overview. Materials Today: Proceedings, 57, 12 2021.

[16] Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. Integer quantization for deep learning inference: Principles and empirical evaluation. Proceedings of the IEEE, pages –, 04 2020.