



Hybrid Simulation of Application Specific Deep Learning Accelerators Graduation Project 2

Supervised By:

- ❖ Prof. Dr. Watheq Elkharashi
- ❖ Dr. Mohamed Abd El Salam
- ❖ Eng. Tasneem Awaad

Prepared By:

Ahmad Ossama Ahmad Aboelkassem	18P6575
Ahmed Amr Ahmed Mohamed	18P5837
Daniel Tarek Lewis Saad Hanna ElShabrawy	18P1185
Mostafa Mohammad Mohammad Shams Eldeen	18P6781
Shehab Mohamed Ibrahim Khalil	18P7213
Yusuf Sameh Fawzi Elshall	18P1399



I. LIST OF CONTENT

I.	LIST OF CONTENT	2
II.	LIST OF FIGURES	5
III.	LIST OF TABLES	5
IV.	ABBREVIATIONS	6
V.	ACKNOWLEDGMENT	8
VI.	ABSTRACT	9
VII.	CHAPTER 1: INTRODUCTION	10
A.	Motivation	10
B.	Problem Definition	11
C.	Objectives	11
D.	Prerequisites	12
E.	Contribution	13
F.	Thesis Organization	14
VIII.	CHAPTER 2: BACKGROUND	16
A.	QT Wizard Overview	16
B.	SystemC Overview	17
1.	Why SystemC?	18
2.	SystemC major components	19
C.	Convolutional Neural Networks Overview	20
1.	Why Convolutional Neural Networks?	20
2.	Convolutional Neural Network Layers	21
Convolution Layer	21	
Padding	22	
Stride	22	
Pooling Layer	23	
Fully Connected Layer	23	
Softmax	24	
3.	Convolutional Neural Networks Architectures	24
LeNet-5	25	
VGG-16	25	
IX.	CHAPTER 3: RELATED WORK	28



A.	NVIDIA's Deep Learning Accelerator.....	28
B.	History.....	30
1.	Cairo University 2020/21 Team.....	30
2.	Cairo University 2021/22 Team.....	31
X.	CHAPTER 4: PROJECT WORKFLOW	33
A.	Overall Flow	33
B.	Template Design Pattern.....	35
XI.	CHAPTER 5: METHODOLOGIES	37
A.	Model Specification & JSON File	37
B.	Code Generation with PyTorch.....	39
1.	Parsing the JSON File	40
2.	Generating CNN Layers and Model Code	40
3.	Generating Training Code.....	40
4.	Integration and Workflow	41
C.	Torch Script & Model Serialization	41
D.	Integration of the C++ Environment	42
XII.	CHAPTER 6: VERIFICATION ENVIRONMENT	44
A.	Description of Master-Slave SystemC Setup	44
B.	Payload & Image Processing.....	47
1.	Payload Structure	47
2.	Image Processing Steps	48
C.	Model Inference & Output	49
XIII.	CHAPTER 7: RESULTS.....	51
A.	Experimental Setup & Dataset.....	51
1.	Hardware Setup.....	51
2.	Software and Libraries	51
3.	Dataset Description.....	51
4.	Dataset Split	52
B.	Training & Testing Procedures	52
1.	Training Procedure	52
2.	Integration into SystemC codebase	53
C.	Performance Metrics.....	53
1.	Accuracy.....	53



2. Execution Time	54
D. Discussion	55
XIV. CHAPTER 8: CONCLUSION.....	57
A. Summary of Findings.....	57
B. Limitations & Future work	58
XV. REFERENCES.....	59



II. LIST OF FIGURES

Figure 1 - Hardware Platforms	10
Figure 2 - SystemC Overview	17
Figure 3 - SystemC Components	20
Figure 4 - CNNs Layers Overview	21
Figure 5 - Convolution Layer	22
Figure 6 - Padding Layer	22
Figure 7 - Stride	23
Figure 8 - Pooling Layer	23
Figure 9 - Fully Connected Layer	24
Figure 10 - Softmax Activation Function	24
Figure 11 - Lenet-5 Architecture	25
Figure 12 - VGG-16 Architecture	26
Figure 13 - NVDLA	28
Figure 14 - Cairo University 2020/21 Team Overview	30
Figure 15 - Cairo University 2021/22 Team Overview	32
Figure 16 - Project Overall Flow	33
Figure 17 - Tool Overall Flow	35
Figure 18 - Template Design Pattern	35
Figure 19 - Qt GUI Wizard (Architecture tab)	37
Figure 20 - Qt GUI Wizard (Miscellaneous Parameters tab)	38
Figure 21 - Sample JSON File (1)	39
Figure 22 - Sample JSON File (2)	39
Figure 23 - Lenet-5 Full Flow	43
Figure 24 - Master-Slave SystemC Setup	44
Figure 25 - Initiator & Target Communication	45
Figure 26 - SystemC Generic Payload	47
Figure 27 - MNIST Dataset	52
Figure 28 - Lenet-5 Accuracy Score	54
Figure 29 - Machine 1 Execution Time	54
Figure 30 - Machine 2 Execution Time	55

III. LIST OF TABLES

Table 1 - Lenet-5 Layers	25
Table 2 - VGG-16 Layers	26
Table 3 - Machines' Execution Time Comparison	55



IV. ABBREVIATIONS

CNN	Convolutional Neural Network
ASDLA	Application Specific Deep Learning Accelerator
DLA	Deep Learning Accelerator
DNN	Deep Neural Network
AI	Artificial Intelligence
FC	Fully Connected Layer
FPGA	Field-Programmable Gate Array
GP	General purpose Processor
IoT	Internet of Things
RNN	Recurrent Neural Network
RTL	Register Translation Level
TLM	Transaction-Level Modeling
VHDL	VHSIC Hardware Description Language
VHSIC	Very High-Speed Integrated Circuit
VP	Virtual Platform
YOLO	You only look once
NVDLA	Nvidia Deep Learning Accelerator
ReLU	Rectified Linear Unit
RGB	Red-Green-Blue
ML	Machine Learning
IC	Integrated Circuit
HDL	Hardware Description Language
GPU	Graphics Processing Unit
Conv	Convolution
GUI	Graphical User Interface
OCR	Optical Character Recognition
NLP	Natural Language Processing
ANN	Artificial Neural Network
DL	Deep Learning
IDE	Integrated Development Environment
RAM	Random Access Memory
API	Application Programming Interface
JSON	JavaScript Object Notation
LUT	LookUp Table
SoC	System-on-a-Chip
VGG	Visual Geometry Group
MNIST	Modified National Institute of Standards and Technology
NRE	Non-Recurring Engineering
ILSVRC	ImageNet Large Scale Visual Recognition Challenge



CSV	Comma Separated Values
MSE	Mean Squared Error
NLL	Negative Log Likelihood
e.g.	For example
RAM	Random Access Memory
SSD	Solid-State-Drive
HDD	Hard-Disk-Drive



V. ACKNOWLEDGMENT

We would like to express our sincere gratitude and appreciation to the following individuals who have played a significant role in the successful completion of this graduation project.

First and foremost, we extend our deepest appreciation to **Professor Dr. Watheq ElKharashi** for his invaluable support, and mentorship throughout this project. We are truly grateful for his insightful feedback and the time he dedicated to reviewing and providing valuable insights to enhance the quality of this work.

We would also like to extend our heartfelt thanks to **Dr. Mohamed AbdElSalam** for his valuable input, expertise, and support throughout the research process. His invaluable guidance and suggestions have been crucial in expanding our understanding of the subject matter and refining the methodologies employed. We are grateful for his time and effort in providing constructive feedback and helping us overcome challenges encountered during the course of this project.

Furthermore, we would like to express our sincere appreciation to **Engineer Tasneem Awaad** for her support and assistance in various aspects of this research. Her expertise and technical insights have been immensely valuable in the development and implementation of the project. We are grateful for her assistance and the collaborative spirit she brought to the team.

The completion of this thesis would not have been possible without the support and contributions of all those mentioned above. Thank you for being an integral part of this journey and for your unwavering commitment to excellence.



VI. ABSTRACT

This thesis presents a comprehensive study on the development of a GUI (QT wizard) based tool for configuring and deploying CNN models in a SystemC-based verification environment. The tool allows users to specify CNN model architectures, including layer configurations and hyperparameters, which are then saved in a JSON file. The Python-based implementation utilizes the PyTorch library to build and train the specified CNN models using datasets (if any). The trained models are then serialized using Torch Script, enabling their utilization in a C++ environment. In this environment, the SystemC framework is employed to validate the functionality of the deployed CNN models.

The verification setup consists of a master-slave architecture, where the master sends test images as payloads to the slave, which processes the images using the deserialized model and returns the predictions back to the master.

The thesis presents a comprehensive analysis of the proposed tool's capabilities and its integration with the SystemC verification environment. The experimental results demonstrate the successful deployment and validation of various CNN models using the developed tool. Additionally, the advantages and limitations of the tool are discussed, highlighting its potential for efficient CNN model configuration and validation.

The outcomes of this research contribute to the field of CNN model deployment and verification, providing a user-friendly tool that facilitates the configuration, training, and deployment of CNN models in a SystemC-based environment. The developed tool opens up avenues for further research in automated model generation and validation techniques.

Keywords: CNN, SystemC, QT wizard, PyTorch, Torch Script, verification environment.

VII. CHAPTER 1: INTRODUCTION

In this chapter, we delve into the exploration of the project. The following sections aim to provide a comprehensive understanding of the motivation behind this research, the problem it addresses, the defined objectives, the necessary prerequisites, and the thesis organization. By establishing this foundation, we lay the groundwork for a thorough analysis and discussion of our project throughout this thesis.

A. Motivation

Hardware acceleration has emerged as a promising approach for optimizing the execution of ML algorithms by leveraging specialized chips. Over the past two decades, hardware acceleration has evolved as an alternative to expensive many-core CPUs with the advent of multi-core GPUs. In the context of our project, we are particularly interested in accelerating the inference time of CNNs.

Training and inference are two distinct processes in neural networks. While training is an unpredictable and computationally intensive process, inference occurs after the network has been trained and is a predictable process with estimated execution time. Our focus lies in accelerating the inference time, as it plays a critical role in real-time applications.

To address the constraints of training and inference processing, parallelism has been introduced. Parallelism involves splitting the computation into smaller tasks that can be distributed among computational blocks. This approach can be implemented at the chip level using schedulers and multi-core processors. Various companies have introduced products to fulfill the tasks of hardware acceleration, which can be categorized into different computing devices: GPUs, FPGAs, ASICs, and Neuromorphic chips.

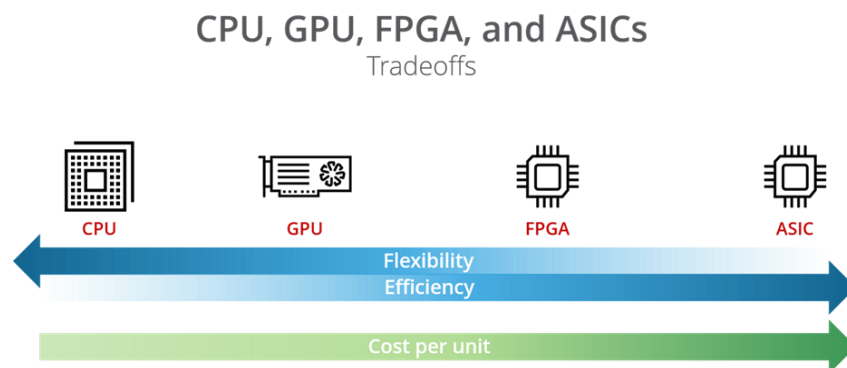


Figure 1 - Hardware Platforms

GPUs, originally designed for graphical processing, have proven to be highly effective for AI tasks. FPGAs are programmable computing elements that offer flexibility through LUTs. ASICs are specialized processors designed to perform specific tasks, such as the multiplication, accumulation, and convolution operations prevalent in neural networks. Neuromorphic computing chips are designed specifically for implementing neural network models at the chip level.

Among the various ML algorithms, we have chosen to focus on CNNs due to their wide range of applications, including image classification, recognition, and face detection. These applications have significant implications in numerous fields, such as autonomous driving, security systems, and various other domains.

Considering the growing demand for efficient and real-time CNN-based applications, our motivation lies in developing a tool that accelerates the inference time of CNN models through hardware acceleration. By leveraging parallelism and integrating the power of specialized chips, we aim to enhance the speed and efficiency of CNN-based inference, enabling seamless integration into various real-world systems and applications.

Hybrid simulation was chosen as a better option than RTL and chips as deep neural networks are now developing rapidly having large numbers of configurations and architectures that are hard to be caught by RTL.

B. Problem Definition

The problem addressed in this thesis is the lack of a streamlined and efficient workflow for designing, training, and deploying CNN models, particularly in the context of integrating them into SystemC-based verification environments. Currently, the process of specifying CNN architectures, configuring hyperparameters, and deploying the models in a C++ environment with SystemC requires manual and time-consuming steps.

Additionally, the existing tools and frameworks often lack flexibility in terms of allowing users to define their own CNN architectures and optimize the deployment process for specific verification tasks. This limitation hinders the efficient utilization of CNN models in the SystemC environment for verification and testing purposes.

Therefore, the primary objective of this thesis is to develop an integrated workflow and tool that addresses these challenges and enables seamless design, training, and deployment of CNN models in a SystemC-based verification environment. This involves providing a user-friendly interface for specifying CNN architectures, automating the process of generating PyTorch code, and integrating the trained models into a SystemC-based verification environment.

By tackling these challenges, the developed solution aims to enhance the efficiency and effectiveness of utilizing CNN models for image processing and verification tasks. This will enable researchers and engineers to leverage the power of CNNs while utilizing the capabilities of the SystemC environment for thorough testing and validation of their designs.

Through the development and evaluation of this integrated workflow and tool, this thesis seeks to contribute to the field of computer vision and verification by facilitating the adoption and utilization of CNN models within the SystemC framework.

C. Objectives

The aim of this project is to develop an integrated workflow and tool that enables seamless design, training, and deployment of CNN models in a SystemC-based verification environment. By addressing the challenges associated with the manual and time-consuming processes of specifying CNN architectures, configuring hyperparameters, and integrating the models into a



C++ environment with SystemC, the project seeks to provide a streamlined and efficient solution for researchers and engineers working in the field of computer vision and verification.

The main objectives of the project can be summarized as follows:

- Develop a user-friendly graphical interface, based on the existing QT wizard, that allows users to specify the architecture of CNN models. The interface should provide options for selecting layer types and configuring hyperparameters, among other customizable parameters.
- Automate the process of generating PyTorch code based on the user-specified CNN model architecture. The generated code should accurately reflect the specified model, allowing for seamless integration with PyTorch.
- Leverage the Torch Script functionality in PyTorch to serialize the trained Python models into a format compatible with a C++ environment. This serialization process will enable the utilization of the trained models within the SystemC-based verification environment, providing real-time inference capabilities for image processing and verification tasks.
- Develop a master-slave verification environment using SystemC, where the master module sends payload (test images) to the slave module, and the slave module performs model inference on the received payload using the serialized CNN model. The output prediction is then sent back to the master module for further analysis and evaluation.
- Evaluate and validate the effectiveness and efficiency of the developed workflow and tool. This includes assessing the accuracy and performance of the trained models, comparing the results with existing approaches, and conducting experiments to measure the advantages and disadvantages of the tool in terms of design productivity and verification quality.

By achieving these objectives, the project aims to provide a comprehensive solution that empowers researchers and engineers to efficiently design, train, and deploy CNN models in a SystemC-based verification environment. This will facilitate the integration of computer vision capabilities into the verification process, enabling robust testing and validation of complex designs with improved accuracy and efficiency.

D.Prerequisites

To undertake this project, several prerequisites were essential to ensure a solid foundation in the relevant areas of study. The following prerequisites were completed to acquire the necessary knowledge and skills:

- **CNN Course by DeepLearning.AI:** A comprehensive course on CNNs provided by DeepLearning.AI was completed prior to the commencement of this project. This course covered the fundamental concepts of CNNs, including convolutional layers, pooling layers, and fully connected layers. It also delved into advanced topics such as object detection and image segmentation using CNNs. The completion of this course equipped us with a thorough understanding of CNN architectures, training methodologies, and their applications in computer vision tasks [2].

- **SystemC Tutorials by Asic World:** To gain proficiency in SystemC, a series of tutorials provided by Asic World were studied. These tutorials covered the basics of SystemC, including modeling techniques, modules, and inter-process communication. Additionally, the tutorials explored advanced concepts such as temporal decoupling and transaction-level modeling. By completing these tutorials, we acquired a solid foundation in SystemC, which is essential for implementing the verification environment and integrating the CNN models into the SystemC framework [1].
- **CSE425: Software Design Patterns:** This course focused on various design patterns used in software development, including creational, structural, and behavioral patterns. The course provided a comprehensive understanding of design principles and best practices, enabling us to apply appropriate design patterns to the development of our project. The knowledge gained from this course was particularly beneficial in designing and implementing the proposed template design pattern within the project.

The successful completion of these prerequisites ensured a solid understanding of CNNs, SystemC modeling and simulation, and software design patterns. This foundation was crucial for effectively undertaking the project and implementing the proposed solution.

E. Contribution

Our team has made significant contributions to the project, leveraging our expertise in various technologies and domains. The following are the key contributions we have made:

- Developed a user-friendly Qt GUI wizard to collect CNN architecture parameters and training parameters from the user, allowing flexibility in creating custom architectures or working with pre-trained models like VGG-16.
- Implemented the parsing and storage of user-specified options into a JSON file, ensuring easy access and retrieval of configuration data throughout the project workflow.
- Utilized the PyTorch library to train and convert CNN models into serialized Torch Script files (.pt), enabling efficient deployment and integration into the subsequent stages of the project.
- Integrated the serialized Torch Script models into a C++ codebase, utilizing the "torch" library for model loading and de-serialization, preparing the models for usage within the SystemC environment.
- Designed and implemented the SystemC environment, comprising the Master and Slave modules, to enable the testing and evaluation of CNN models using real-time image data.
- Developed the communication protocol between the Master and Slave modules, allowing the transmission of testing images and receiving predictions from the de-serialized CNN models.
- Conducted extensive testing and validation of the entire system, ensuring the accuracy and reliability of the CNN predictions within the SystemC environment.

These contributions collectively demonstrate our team's comprehensive involvement and proficiency across multiple technologies, including Qt, PyTorch, Torch Scripts, and SystemC. By leveraging our diverse skill set and collaborating effectively, we have successfully developed an



automated flow for configuration and generation of CNN-based AI accelerators, encompassing both software and hardware aspects of the project.

F. Thesis Organization

This thesis is organized into several chapters to provide a structured and comprehensive exploration of the research topic. The following sections briefly describe the contents of each chapter, highlighting their respective focus and contribution to the overall study.

Chapter 1: Introduction

The first chapter serves as an introduction to the thesis, presenting the motivation behind the research, defining the problem to be addressed, and outlining the objectives. The prerequisites required for understanding the subsequent chapters are also discussed in this section.

Chapter 2: Background

This chapter delves into the background information necessary to comprehend the subsequent chapters. It covers the fundamentals of QT, CNNs, and SystemC, providing a comprehensive understanding of these key concepts.

Chapter 3: Related Work

Chapter 3 provides an overview of previous projects conducted by Cairo University teams, focusing on their contributions and insights. Additionally, it explores the NVDLA, discussing its significance in the field of deep learning.

Chapter 4: Project Workflow

Chapter 4 presents the overall workflow of the project, including relevant figures and explanations. It discusses the design pattern employed in the project and justifies its selection based on its suitability and advantages.

Chapter 5: Methodologies

In Chapter 5, the methodologies employed in the project are described in detail. This includes the model specification and the use of JSON files, code generation with PyTorch, torch script, and model serialization. Additionally, the integration of the C++ environment and SystemC is discussed.

Chapter 6: Verification Environment

Chapter 6 focuses on the verification environment utilized in the project. It provides a detailed description of the Master-Slave SystemC setup, payload and image processing mechanisms, as well as model inference and output generation.

Chapter 7: Results

This chapter presents the experimental setup and dataset used in the project. It covers the training and testing procedures employed, along with the performance metrics used to evaluate the system. The results obtained are discussed in detail.

Chapter 8: Conclusion

The final chapter summarizes the findings of the thesis, highlighting the main contributions and outcomes. It also acknowledges any limitations encountered during the research and suggests potential areas for future work and improvement.

**Appendix**

The appendix section contains additional material that supplements and enhances the understanding of the main thesis content. It includes relevant details, data, and other supporting information.

References

The references section provides a comprehensive list of papers and citations used throughout the thesis, acknowledging the sources and works consulted during the research process.



VIII. CHAPTER 2: BACKGROUND

In the background chapter, we aim to provide a comprehensive understanding of the key components and concepts relevant to the project. This chapter serves as a foundation for the subsequent sections by presenting an overview of the fundamental technologies involved: Qt, CNNs, and SystemC. By delving into the essential aspects of each technology, we establish the necessary background knowledge to comprehend the project's development and implementation.

A. QT Wizard Overview

A Qt GUI wizard is a tool provided by the Qt framework that assists in the creation of GUIs for applications. It simplifies the process of designing and implementing interactive interfaces by providing a set of pre-designed components, widgets, and templates that can be customized and combined to create visually appealing and functional GUIs.

The Qt GUI wizard allows developers to create intuitive and user-friendly interfaces for their applications. It provides a drag-and-drop interface builder that allows users to visually design the layout of the GUI by placing widgets such as buttons, text fields, checkboxes, and menus onto a form. Developers can then connect these widgets to specific functionality and define the behavior of the application in response to user interactions.

Qt provides a wide range of built-in widgets and controls, as well as support for custom widget creation, enabling developers to create highly customized and tailored interfaces for their specific application requirements. The Qt GUI wizard also offers various style options and themes, allowing developers to achieve a consistent and visually appealing look and feel across different platforms.

In our project, Qt is used as the GUI framework through which users can interact with the CNN architecture. It serves as the interface for users to input the desired parameters and configurations for the CNN model, such as the number of layers, each layer's type (convolution, pooling, or fully connected), the input image size, each layer's hyperparameters, the loss function, the optimizer, and other relevant settings. The Qt GUI wizard simplifies the process of designing and implementing this user interface by providing a set of tools and components that can be easily integrated into our tool.

By leveraging the Qt GUI wizard, we can create an intuitive and efficient user experience for configuring and visualizing CNN architectures. The flexibility and extensibility of the Qt framework enable us to design a user interface that meets the specific needs of our CNN-based project.

Qt is a powerful and versatile GUI framework that empowers developers to create visually appealing and interactive applications, and with the added advantage of the Qt wizard being free and open-source, it provides an accessible and cost-effective solution for designing intuitive user interfaces.

B. SystemC Overview

SystemC is a platform-independent, open-source hardware description language that enables the modeling and simulation of digital systems. It serves as a high-level language for capturing the behavior of digital systems, providing an abstract representation of their functionality. By utilizing SystemC, designers can verify the functionality of their designs before proceeding with the physical hardware implementation. The language offers an environment for the design and verification of digital systems, allowing designers to develop and test their designs in a virtual setting.

Compared to traditional hardware description languages like VHDL or Verilog, SystemC offers several advantages that make it a preferred choice for digital system design:

- **High-Level Representation:** SystemC's high-level representation of system behavior simplifies the understanding and development of complex digital systems. Designers can express the system's behavior in a more intuitive and concise manner.
- **Support for Concurrent Processes:** SystemC facilitates the modeling of systems with multiple components that interact concurrently. This feature is essential for capturing the complex interactions and dependencies within digital systems.
- **Integration of Digital and Analog Components:** SystemC provides an environment for integrating digital and analog components, enabling the modeling and verification of mixed-signal systems. This capability is crucial for designing systems that incorporate both digital and analog functionalities.

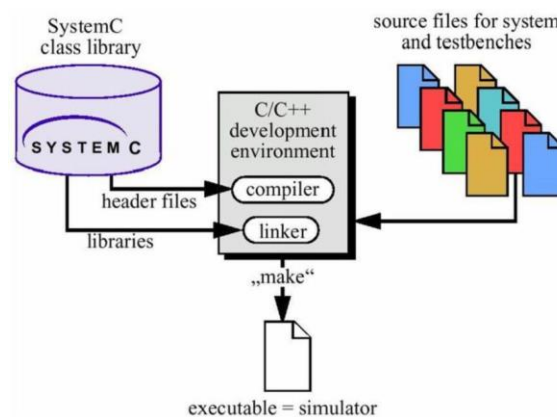


Figure 2 - SystemC Overview

SystemC further enhances the design process through the following features:

- **Standardized Interface:** SystemC offers a standardized interface for communication between different modules and components in the system. This facilitates the design and implementation of reusable components, which can be utilized across multiple projects, promoting modularity and efficiency.
- **Modeling and Simulation at Different Abstraction Levels:** SystemC supports modeling and simulation at various levels of abstraction. Designers can initially create a high-level model of the system and gradually refine it as needed. This flexibility enables verification at different stages of development, ensuring that the design meets the required specifications.

SystemC serves as a powerful hardware description language for designing and verifying digital systems. Its platform independence, high-level representation of system behavior, support for concurrent processes, and standardized interface make it an ideal choice for developing complex digital systems. Moreover, SystemC's capability to integrate digital and analog components and its flexibility in modeling at different levels of abstraction contribute to its effectiveness and efficiency in the design and verification process [6].

1. Why SystemC?

SystemC is a C++ hardware libraries class standardized by the IEEE, offering a platform for modeling concurrency in digital systems. When developing DLAs for embedded systems, the choice between implementing the design at the RTL using an HDL like VHDL or using SystemC at the behavioral level arises as an important consideration. Evaluating both levels of abstraction for ANN architecture, it was found that RTL offers benefits such as performance, precision, and familiarity.

However, it results in a larger codebase, increased design time, and a greater gap between algorithm design and RTL implementation. On the other hand, SystemC at the behavioral level allows for faster design time, a relatively smaller gap between algorithm design and hardware-level system design, and ease of use. Yet, it may have limitations in parameterization mechanisms and flexibility for hardware synthesis.

One notable advantage of SystemC over RTL is its ability to expedite and improve design verification. By utilizing C++ as its modeling language, SystemC allows designers to leverage a wide range of software development tools, including debuggers and testbenches, for verifying design behavior. This significantly accelerates the verification process and reduces the risk of errors.

Additionally, SystemC provides a higher level of abstraction compared to RTL, enabling designers to focus on the functionality of their designs rather than implementation details. This enhances the design process's intuitiveness and reduces the likelihood of errors. Furthermore, SystemC models can be easily translated into RTL code using automated tools, minimizing the effort required for hardware implementation.

Moreover, SystemC enables hardware/software co-development, enabling designers to model and verify both hardware and software components of a SoC design. This facilitates the assurance of correct interaction between the hardware and software components, contributing to an improved overall design process. As an HDL, SystemC is often utilized as a high-level wrapper for hardware designs, including those implementing deep learning models.

There are several compelling reasons for employing SystemC as a wrapping language for CNN models:

- **Abstraction:** SystemC provides a high level of abstraction, simplifying the design and simulation of complex hardware systems. This abstraction is particularly advantageous when working with CNNs, which exhibit intricate layering and interconnections. Utilizing SystemC allows designers to concentrate on the CNN's high-level functionality without being burdened by underlying hardware details.
- **Portability:** Being a standard language supported by a wide range of tools and simulators, SystemC facilitates easy portability of designs across different hardware

platforms, such as FPGAs or ASICs. This characteristic proves especially valuable when dealing with computationally intensive CNNs that may require specialized hardware for efficient execution.

- **Reusability:** SystemC enables the creation of reusable modules and components that can be applied across diverse designs. This reusability is particularly advantageous in CNN applications, where numerous layers and functions can be shared across different contexts. By employing SystemC, designers can develop reusable modules applicable to multiple CNN designs, leading to significant time and resource savings.
- **Verification and Validation:** SystemC allows for the simulation and verification of designs before their hardware implementation. This capability is crucial when working with CNNs, as they entail complex interactions and necessitate comprehensive testing for correct functionality. Employing SystemC permits designers to simulate and verify CNN designs early in the development process, facilitating the identification and rectification of any errors.
- **Co-Design:** SystemC facilitates the co-design of hardware and software components, which proves advantageous in CNN scenarios. Through co-design, designers can optimize the overall system performance by refining hardware-software interactions. For instance, SystemC can be used to optimize memory usage and data flow within a CNN, thereby enhancing its performance.
- **Interoperability:** SystemC allows for seamless connectivity between different hardware components and models. This characteristic is particularly valuable in CNN applications, as these models can be implemented on various hardware platforms, such as FPGAs, GPUs, or ASICs. Employing SystemC enables designers to integrate different hardware components effectively, ultimately improving system performance.

Using SystemC as a wrapping language for CNN models offers numerous advantages, including abstraction, portability, reusability, verification and validation, co-design, and interoperability. By leveraging SystemC, designers can focus on the CNN's high-level functionality, abstracting away underlying hardware details. SystemC's standardization, wide tool support, and portability enable designs to be readily deployed on different hardware platforms. The reusability of SystemC modules and components enhances design efficiency, while simulation and verification capabilities aid in error detection. Co-designing hardware and software components optimizes overall system performance, and SystemC's interoperability features ensure seamless integration between diverse hardware components and models, resulting in improved system performance [6].

2. SystemC major components

SystemC major components consist of:

- **Modules:** SystemC modules represent the components of the digital system and define their behavior. These modules can contain multiple processes and signals, facilitating communication and interaction with other modules through ports.
- **Processes:** SystemC processes are responsible for modeling the behavior of components in the digital system. Concurrent statements within processes allow designers to simulate complex systems with multiple interacting components.

- **Signals:** SystemC signals represent communication channels between different modules within the system. By providing a standardized interface for communication, signals enable the development of reusable components that can be utilized across multiple projects.
- **Ports:** SystemC ports serve as connections between modules in the system. They enable data transfer between modules and control the flow of data throughout the system.
- **Clocks:** SystemC clocks are utilized to model the timing of signals in the digital system. Clocks ensure synchronization of component behavior and accurately represent the timing characteristics of signals.
- **TLM:** SystemC supports TLM, which allows designers to model system behavior at a higher level of abstraction. TLM provides a means to describe system behavior without the need to delve into low-level hardware details.

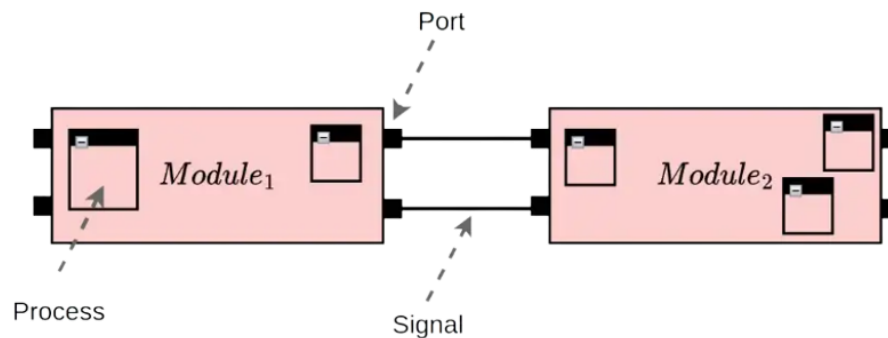


Figure 3 - SystemC Components

In conclusion, SystemC encompasses various key components that collectively provide a comprehensive environment for designing and verifying digital systems. Modules, processes, signals, ports, clocks, and TLM functionality enable designers to model and test digital system behavior in a virtual environment before moving on to hardware implementation. This approach ensures the thorough evaluation and refinement of designs, leading to more efficient and reliable hardware outcomes [6].

C. Convolutional Neural Networks Overview

In this section, we delve into the fundamental concepts and components of CNNs, a powerful class of deep learning models. By exploring their structure and functionality, we lay the groundwork for understanding the significance and application of CNNs in various domains, including image classification, object detection, and semantic segmentation.

1. Why Convolutional Neural Networks?

CNNs excel in processing image data and offer several advantages for object detection tasks. One key advantage is their ability to automatically learn and extract relevant features from images. They can identify edges, shapes, textures, and other visual features.

CNNs also demonstrate robustness to translation, enabling them to detect objects in varying positions and orientations within an image. The scalability of CNNs is another strength, as they can be trained on large datasets, making them suitable for complex object detection challenges. Real-time performance is a critical factor, and CNNs are optimized to efficiently run on GPUs. This makes them well-suited for real-time object detection applications, such as autonomous vehicles. Furthermore, CNNs support end-to-end learning, enabling them to learn object detection directly from image data without relying on manual feature extraction or hand-engineered algorithms. This streamlines the training process and enhances the efficiency of object detection tasks [7].

2. Convolutional Neural Network Layers

A CNN has multiple hidden layers responsible for extracting data from an image. There are three main layers in a CNN that will be discussed later on, they are:

- Convolution layers.
- Pooling layers.
- Fully connected layers.

Also, padding and stride will be demonstrated.

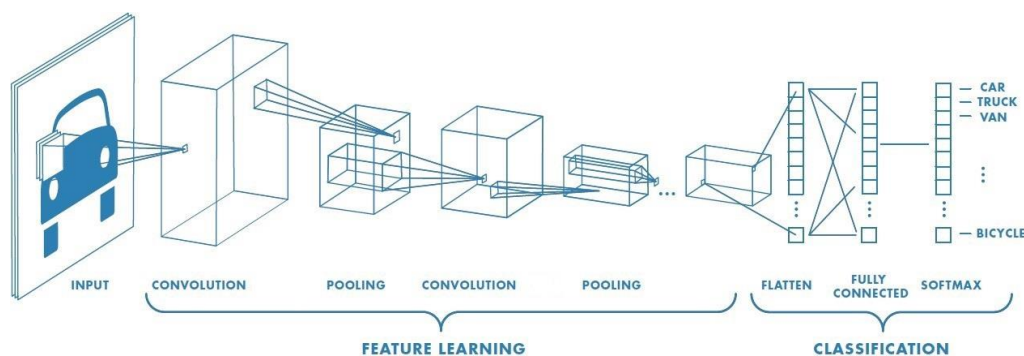


Figure 4 - CNNs Layers Overview

Convolution Layer

The convolution layer is a crucial component of CNNs responsible for feature extraction. It applies a set of learnable filters to the input data, generating output feature maps. The filters scan the input data, calculating dot products at each position and storing the results in the corresponding position of the output feature map. Shared weights are employed, reducing parameters and enhancing computational efficiency.

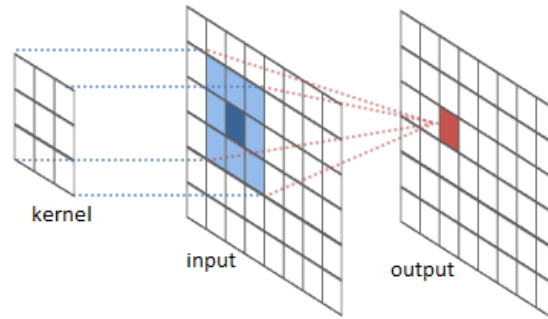


Figure 5 - Convolution Layer

Additional operations like pooling and activation functions are applied to the feature maps to down-sample and introduce non-linearity. Hyperparameters such as the number and size of filters, stride, and padding impact the depth and spatial resolution of the feature maps.

Padding

Padding is a technique employed in CNNs to manipulate the spatial size of the output feature maps. During convolution, the filter only considers a subset of the input data at each position, resulting in smaller feature maps. Padding addresses this by adding extra pixels around the input data before the convolution operation.



Figure 6 - Padding Layer

Two types of padding exist: "same" padding maintains the output size equal to the input, while "valid" padding reduces the output size. Padding offers benefits such as improved accuracy by considering edge pixels, preserving spatial resolution for image data, and facilitating the stacking of multiple convolutional layers.

Stride

Stride is a hyperparameter in CNNs that determines the distance the filter moves over the input data. The stride affects the size of the output feature map and the number of parameters in the model. A larger stride reduces the output size and computational complexity, but may lose fine details. Conversely, a smaller stride increases the output size and model accuracy but increases computational demands.

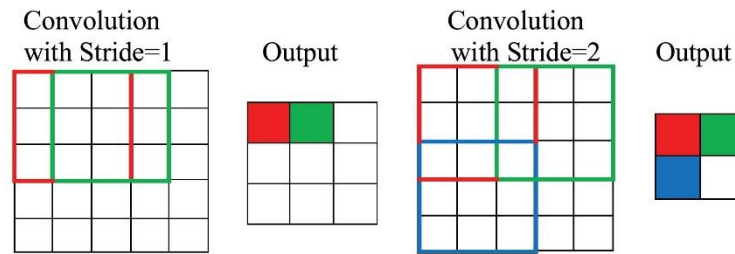


Figure 7 - Stride

Typical choices for stride are 1 or 2, where a stride of 1 captures more details and a stride of 2 downsamples the data without pooling layers. The selection of stride balances the trade-off between model complexity and information preservation.

Pooling Layer

The pooling layer is commonly used in CNNs to reduce the spatial dimensionality of feature maps generated by convolutional layers. It helps in downsampling while preserving important information. Two types of pooling, max pooling and average pooling, are often employed. Max pooling selects the maximum value from each window, retaining crucial features and reducing the feature map size. On the other hand, average pooling takes the average value, smoothing the map and reducing noise.

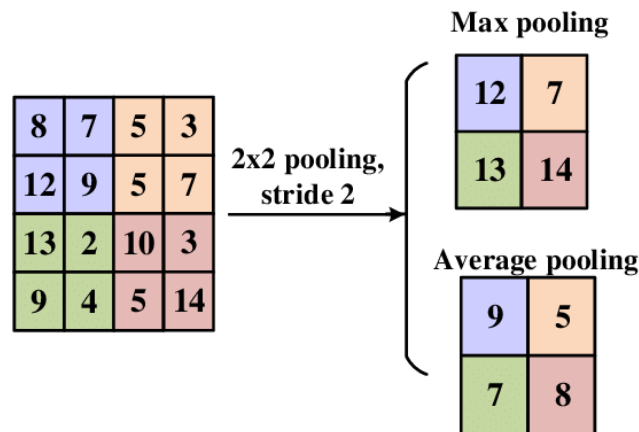


Figure 8 - Pooling Layer

The pooling layer aids in computational efficiency, robustness to translations, and reduction of spatial dimensionality. Typically, the pooling window size and stride are the same, with a small window size like 2x2 or 3x3 moving one pixel at a time.

Fully Connected Layer

The FC layer, also known as a dense layer, is a commonly used layer in CNNs and other deep learning architectures. It is typically placed at the end of a CNN to produce the final network output, such as classification or regression results. The FC layer connects to all neurons in the previous layer, with each neuron in the FC layer receiving input from all neurons in the previous layer and generating a single output.

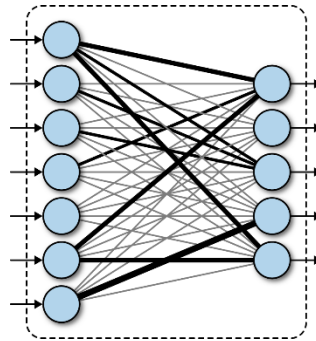


Figure 9 - Fully Connected Layer

This layer can also be used within the network to learn complex features from the input data. During training, the weights of the FC layer are adjusted to learn important features from the input data. However, FC layers can be computationally expensive due to a large number of parameters and can lead to overfitting. Techniques like dropout or weight decay can be used to address these issues.

Softmax

The softmax function is commonly used in CNNs for image classification tasks. It takes a vector of real numbers as input and produces a probability distribution over different classes as output.

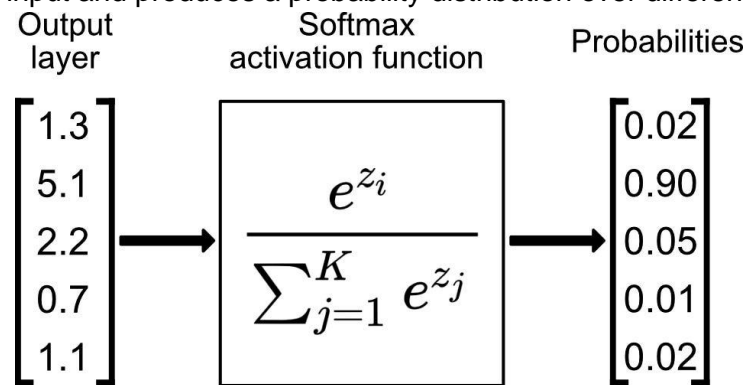


Figure 10 - Softmax Activation Function

The output vector represents the confidence of the network for each class. This allows the CNN to predict the class of an image by converting the output values into probabilities using the softmax function.

3. Convolutional Neural Networks Architectures

Now let's delve into the exploration of prominent CNN architectures that have played a pivotal role in shaping the field of computer vision. These architectures have been meticulously designed and optimized to tackle specific challenges and achieve state-of-the-art performance in various image analysis tasks. By examining these architectures, we gain insights into the innovative techniques and architectural choices that have propelled the field forward.

LeNet-5

LeNet-5 is a CNN architecture that was proposed by Yann LeCun in 1998. It was one of the first successful CNNs and is considered a classic in the field of deep learning. LeNet-5 was designed to recognize handwritten digits, but it has also been used for other image classification tasks.

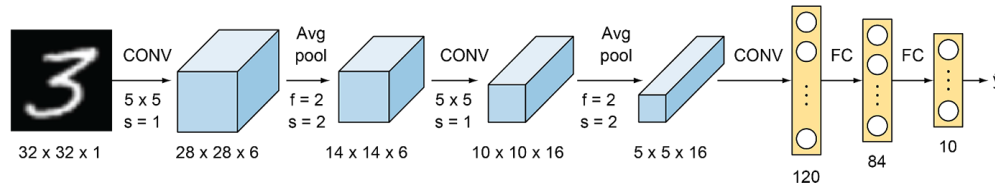


Figure 11 - Lenet-5 Architecture

The LeNet-5 architecture consists of 7 layers:

Layer	Feature Map	Size	Kernel Size	Stride	Activation
Input	1	32x32	-	-	-
Convolution	6	28x28	5x5	1	tanh
Average Pooling	6	14x14	2x2	2	tanh
Convolution	16	10x10	5x5	1	tanh
Average Pooling	16	5x5	2x2	2	tanh
Convolution	120	1x1	5x5	1	tanh
FC	-	84	-	-	tanh
Output	-	10	-	-	Softmax

Table 1 - Lenet-5 Layers

LeNet-5 uses the tanh function as the activation function for its layers. The architecture also includes a backpropagation algorithm for training the network. LeNet-5 was trained on the MNIST dataset, which contains 60,000 training images and 10,000 test images of handwritten digits. The architecture was able to achieve an accuracy of 99.05% on the test set, which was state-of-the-art at the time [8].

VGG-16

The VGG-16 model is a CNN architecture that was introduced by the VGG at the University of Oxford. It gained popularity and widespread use in the field of computer vision after its performance in the ILSVRC in 2014.

The VGG-16 model is renowned for its depth and simplicity. It consists of 12 convolutional layers followed by three FC layers. The convolutional layers are primarily composed of 3x3 filters with a stride of 1 and padding of 1. The pooling layers utilize 2x2 filters with a stride of 2. VGG-16 is known for its uniform architecture, where the number of filters and the spatial resolution of feature maps remain the same throughout the network.

Layer	Feature Map	Size	Kernel Size	Stride	Activation
Input	1	224x224x3	-	-	-
2x Convolution	64	224x224x64	3x3	1	relu
Max Pooling	64	112x112x64	2x2	2	relu
2x Convolution	128	112x112x128	3x3	1	relu
Max Pooling	128	56x56x128	2x2	2	relu
2x Convolution	256	56x56x256	3x3	1	relu
Max Pooling	256	28x28x256	2x2	2	relu
3x Convolution	512	28x28x512	3x3	1	relu
Max Pooling	512	14x14x512	2x2	2	relu
3x Convolution	512	14x14x512	3x3	1	relu
Max Pooling	512	7x7x512	2x2	2	relu
FC	-	25088	-	-	relu
FC	-	4096	-	-	relu
FC	-	4096	-	-	relu
Output	-	1000	-	-	Softmax

Table 2 - VGG-16 Layers

One significant characteristic of the VGG-16 model is its depth. With 12 convolutional layers, it has a larger number of layers compared to earlier CNN architectures like AlexNet. This depth allows the model to capture intricate and hierarchical patterns within images, leading to improved performance in object recognition tasks.

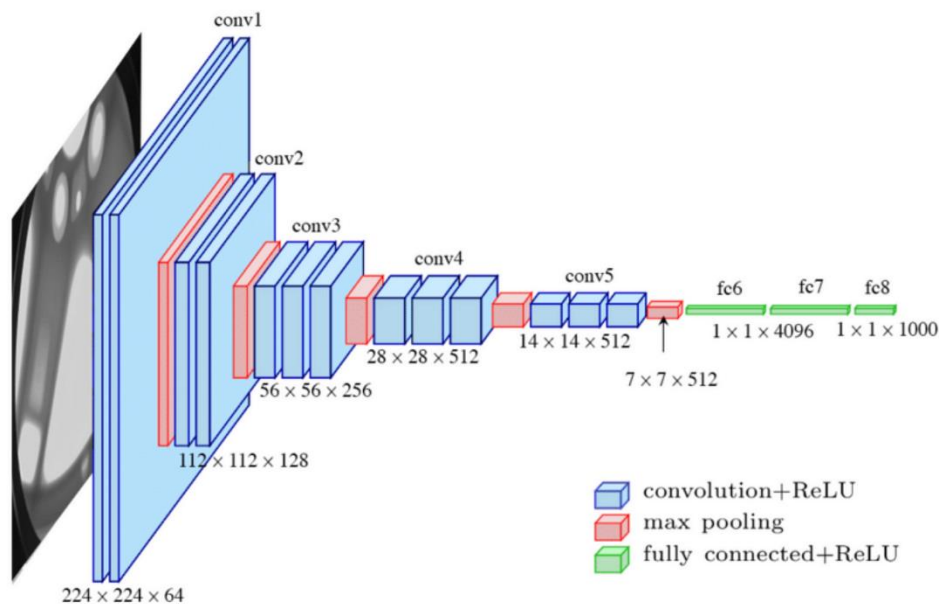


Figure 12 - VGG-16 Architecture

Another notable aspect of the VGG-16 model is its simplicity. The consistent use of small 3x3 filters and 2x2 pooling filters throughout the network contributes to its straightforward design.



This simplicity, combined with its depth, makes the model easy to understand, implement, and modify for various computer vision applications.

The VGG-16 model achieved remarkable results in the ILSVRC 2014 competition. It outperformed previous models by reducing the top-5 error rate significantly. Despite its higher complexity in terms of the number of parameters, the VGG-16 model showcased the effectiveness of deeper networks in learning and recognizing diverse visual concepts.

The VGG-16 model's success and its straightforward architecture have made it a popular choice for various computer vision tasks, including image classification, object detection, and feature extraction. Researchers often use pre-trained versions of the VGG-16 model as a starting point for transfer learning, where they fine-tune the model's weights on new datasets specific to their applications.

Overall, the VGG-16 model stands as a significant milestone in the evolution of CNN architectures, demonstrating the impact of depth and simplicity on improving the accuracy of computer vision tasks [9].

IX. CHAPTER 3: RELATED WORK

In the following chapter, we explore the related work that has contributed to the advancements in deep learning and its practical implementations. The chapter begins by examining NVIDIA's deep learning accelerator, which has played a significant role in accelerating the training and inference processes of neural networks. Additionally, we delve into the historical context of the past two teams' projects from Cairo University over the preceding years, shedding light on their valuable contributions and achievements in the field. This comprehensive overview sets the stage for a deeper understanding of the current research landscape and provides insights into the motivation behind our own work.

A. NVIDIA's Deep Learning Accelerator

The process of designing and implementing a deep learning algorithm involves two main stages: training and inference. Training is a computationally intensive task that requires substantial computing power and complex algorithms. On the other hand, inference can be performed on advanced embedded edge devices.

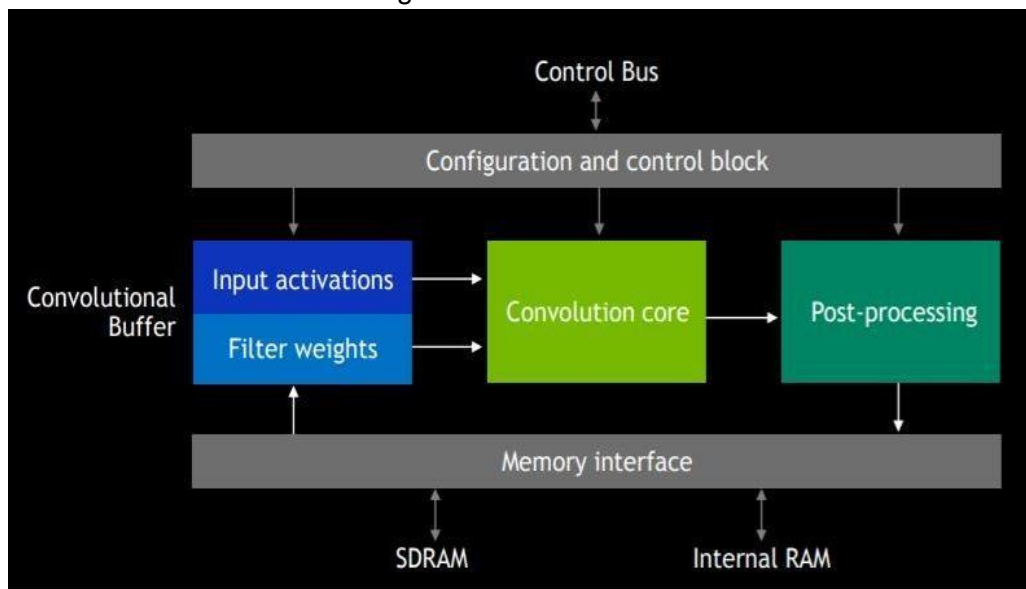


Figure 13 - NVDLA

To address the challenges of training on embedded edge devices with limited resources, extensive research has focused on designing and implementing deep learning algorithm architectures for these systems.

Some notable aspects of this research include:

1. Training and Inference:

- **Training:** A computationally intensive process that demands significant computing power and complex algorithms.
- **Inference:** Execution of trained models on embedded edge devices, which is more feasible due to recent advancements in hardware and software.

2. NVDLA Overview:

NVDLA (Nvidia Deep Learning Accelerator) is a deep learning accelerator developed by Nvidia Corporation. It is an open-source hardware platform specifically designed to deliver high-performance, low-power deep learning processing capabilities for edge devices. Key points about NVDLA include:

- **Purpose:** NVDLA aims to provide efficient deep learning processing for IoT devices, autonomous vehicles, and other embedded systems.
- **Scalability:** The platform is designed to scale and adapt to various deep learning applications, including CNNs, RNNs, and fully connected networks.
- **Flexibility:** NVDLA is highly programmable, enabling customization to meet specific application requirements.
- **Architecture:** NVDLA employs a highly parallel and modular architecture comprising multiple processing elements, memory units, and interconnects.
- **Processing Elements:** These elements are optimized for matrix multiplications and activations, the fundamental operations in deep learning algorithms.
- **Memory Units:** NVDLA includes memory units that provide storage and high-bandwidth access to support the processing elements.
- **Interconnects:** The accelerator incorporates interconnects to enable efficient communication between components, facilitating data movement between processing elements and memory units.
- **Low-Power Consumption:** NVDLA is designed to operate at high performance levels while consuming low power, making it suitable for energy-constrained edge environments.

3. Project Inspiration:

Our project draws inspiration from NVDLA's architecture and functionalities. The following components of NVDLA align with our project's implementation:

- **Input Activations and Filter Weights:** Correspond to the parameters input through the tool's GUI wizard.
- **Convolution Core:** Simulated by the PyTorch scripts, which generate C++ code.
- **Post-Processing:** Covered by wrapping the final component in SystemC.

4. Scalability and Flexibility:

Similar to NVDLA, our project emphasizes scalability and flexibility in deep learning applications. We offer users a wizard GUI where they can input different layers and parameters to generate the corresponding CNN model, allowing scalability for various network sizes and configurations.

In conclusion, NVDLA is a powerful deep learning accelerator designed to provide high-performance, low-power processing capabilities for edge devices. Its parallel and modular architecture, along with its scalability, flexibility, and low-power consumption, makes it an optimal choice for integrating deep learning capabilities into products. By drawing inspiration from NVDLA, our project aims to deliver similar benefits and performance while catering to the specific requirements of our implementation [3].

B. History

In this section, we delve into the rich history and progression of previous projects, highlighting the significant milestones and advancements that have shaped their development. By tracing the historical context, we gain a deeper appreciation for the evolution and significance of our work.

1. Cairo University 2020/21 Team

Cairo University's 2020/21 team project focused on developing an automated flow for generating configurable AI accelerators for various image classification-based CNNs like LeNet-5. They aimed to address the challenges of accelerating CNNs due to their high computational and memory requirements. The team implemented a flow using Perl scripts to generate a high-throughput, configurable, and scalable RTL design. The design was verified using the Veloce emulator, known for its high capacity and fast simulation speed.

The team recognized that autonomous cars heavily rely on CNNs, which demand significant computational power. GPUs and CPUs are commonly used but suffer from high power consumption and relatively slow performance. Alternatively, dedicated AI accelerators offer low power consumption and better performance, but they are typically designed for specific CNNs, leading to high NRE costs for migration to different CNNs. Furthermore, these accelerators lack the ability to be retrained after deployment. Hence, there is a need for generalized and configurable accelerators.

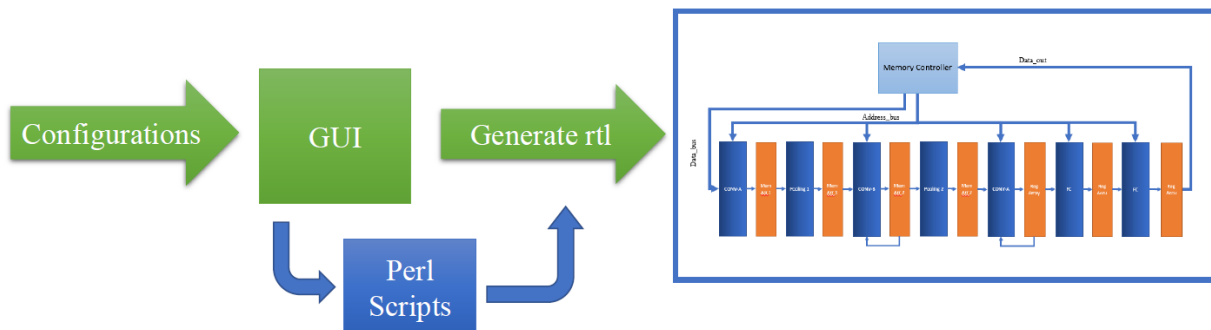


Figure 14 - Cairo University 2020/21 Team Overview

The team's main objective was to develop a working, configurable, and scalable AI accelerator through an automated flow using Perl scripts. Their target was to generate a high-throughput, configurable, and scalable RTL design. The team utilized FPGA and Veloce emulator for designing and verifying the generated accelerators.

This team's project involved the following key aspects:

1. **Automated Flow Development:**

- Implemented an automated flow using Perl scripts.
- The flow focused on generating AI accelerators for various image classification-based CNN models, including LeNet-5 and AlexNet.

2. **Configurable and Scalable RTL Design:**

- Targeted a high-throughput, configurable, and scalable RTL design.



- Utilized the generated RTL design for the AI accelerators.

3. Verification Using FPGA and Veloce Emulator:

- Design verification was conducted using FPGA and Veloce emulator.
- FPGA provided a hardware platform for testing and evaluating the generated accelerators.
- Veloce emulator, known for its high capacity and fast simulation speed, was employed for efficient verification of the RTL design.

4. Performance Evaluation:

- Evaluated the performance of the generated accelerators in terms of throughput and configurability.
- Analyzed the trade-offs between power consumption, performance, and configurability.

The first team's project successfully demonstrated the feasibility of generating configurable AI accelerators for various image classification-based CNN models. Their work provided valuable insights into addressing the challenges of accelerating CNNs while achieving high throughput, configurability, and scalability. The automated flow using Perl scripts and the verification using FPGA and Veloce emulator contributed to the overall success of the project [4].

2. Cairo University 2021/22 Team

The Cairo University's 2021/22 team project aimed to explore the modifications of ML algorithms for fully automated and complex tasks. They focused on CNNs, a class of deep learning neural networks that have made significant breakthroughs in image recognition. The team recognized the challenges associated with accelerating CNN algorithms due to their high computational and memory requirements. To address these challenges, they implemented an automated flow using Perl scripts to generate AI accelerators for various image classification-based CNNs such as Drone Agent, LeNet-5, and VGG16. Their objective was to develop a high-throughput, configurable, and scalable RTL design using the generated accelerators. The team utilized the Veloce emulator for designing and verifying the RTL design due to its high capacity and fast simulation speed.

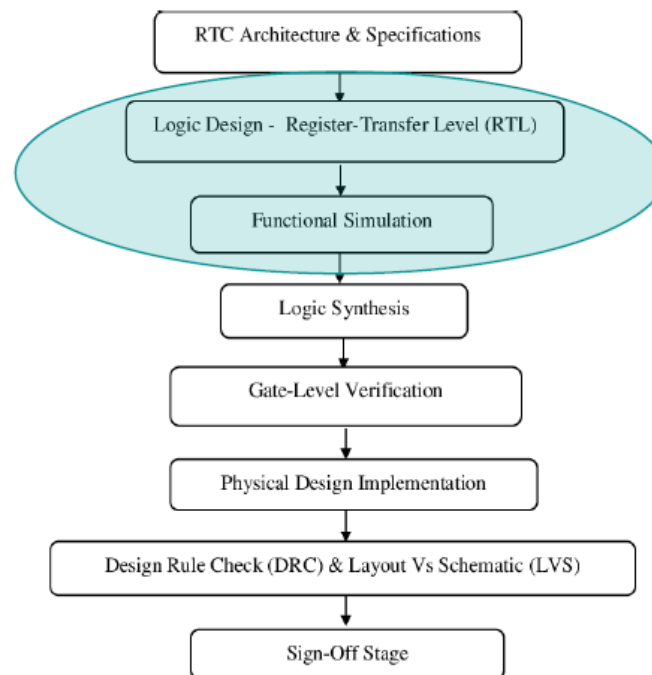


Figure 15 - Cairo University 2021/22 Team Overview

The team identified the heavy computational power required by CNNs in autonomous cars. While GPUs and CPUs are commonly used, they suffer from high power consumption and relatively slow performance. Alternatively, dedicated AI accelerators offer low power consumption and better performance. However, these accelerators are typically designed for specific CNNs, resulting in high NRE costs when migrating to different CNNs. Furthermore, they lack the ability to be retrained after deployment. Therefore, the team recognized the need for generalized and configurable accelerators.

This team aimed to develop a working, configurable, and scalable AI accelerator through an automated flow using Perl scripts. Their objective was to generate AI accelerators for different CNN models, including LeNet-5, AlexNet, VGG16, etc. They sought to achieve a high throughput, configurable, and scalable RTL design using the generated accelerators. The team employed FPGA and Veloce emulator for designing and verifying the accelerators.

Their project successfully demonstrated the feasibility of generating configurable AI accelerators for different convolutional neural network models. Their work emphasized the importance of addressing the challenges associated with accelerating CNN algorithms while achieving high throughput, configurability, and scalability. The automated flow using Perl scripts and the verification using FPGA and Veloce emulator contributed to the overall success of the project [5].

X. CHAPTER 4: PROJECT WORKFLOW

In this chapter, we provide a comprehensive overview of the sequential steps and processes involved in the development and implementation of our project. This chapter serves as a guide to understanding the systematic approach and methodologies employed to achieve our objectives successfully.

A. Overall Flow

In this section, we provide a detailed overview of the overall flow of the project, starting from the Qt GUI wizard and concluding with the SystemC environment. The project's flow encompasses various stages, including user input, data processing, model training, serialization, and integration into the SystemC environment. Each step contributes to the seamless operation of the CNN based AI accelerator.

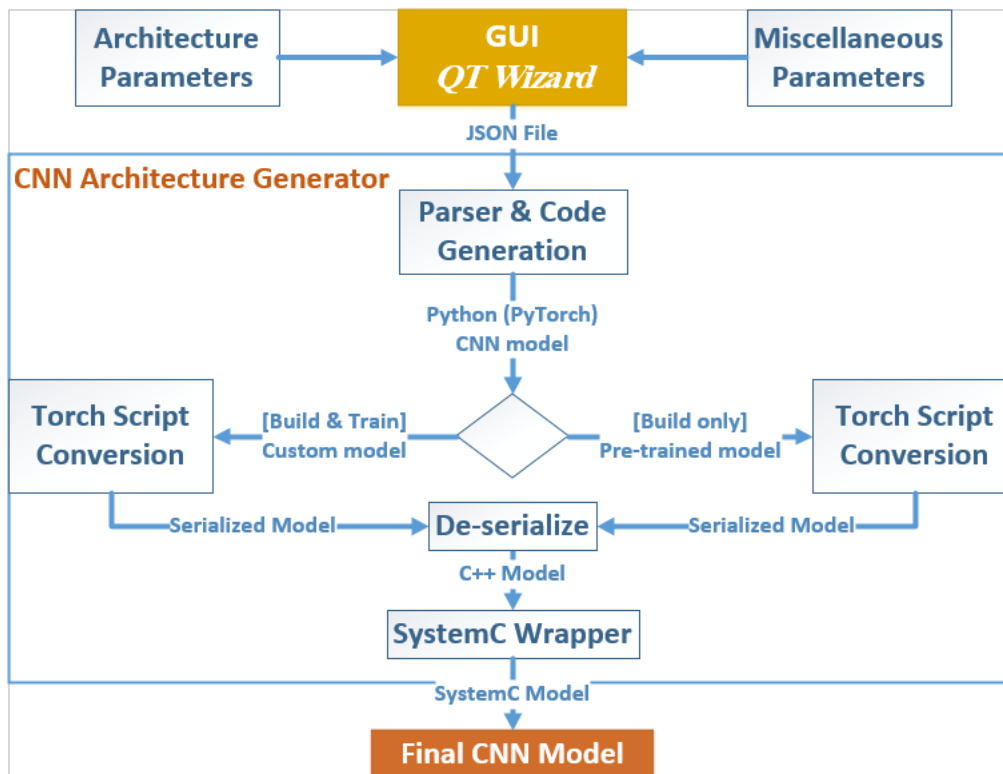


Figure 16 - Project Overall Flow

The flow can be summarized as follows:

1. Qt Wizard: Gathering User Input:

The project begins by obtaining the CNN architecture parameters and training parameters from the user through an intuitive GUI powered by Qt. The Qt wizard allows users to define the following:

- Architecture's layers
- Layers' types (e.g., standard convolution, depthwise convolution, max pooling, average pooling, fully connected)

- Stride
- Padding
- Number of filters
- Kernel size
- Input height
- Input width
- Batch size
- Number of epochs
- Optimizer
- Loss function

Users have the flexibility to either build a custom CNN architecture from scratch or utilize pre-trained models such as VGG-16.

2. JSON File Storage:

Once the user completes the selection process in the Qt wizard, all the specified options and parameters are stored in a JSON file. This file serves as a structured repository for the configuration information, making it easily accessible for subsequent stages of the project.

3. JSON Parsing and Template Design Pattern:

The JSON file is parsed, and the information contained within is utilized to instantiate the appropriate CNN model in PyTorch using the Template Design Pattern. The design pattern employs a super-class (DNN) and a sub-class (CNN) to achieve flexibility and modularity in handling various CNN architectures.

4. Model Training and Serialization:

The instantiated PyTorch CNN model is then trained using the specified training parameters. In the case of building a custom CNN, the trainable parameters (weights and biases) are updated during the training process. However, when working with a pre-trained model like VGG-16, the existing weights and biases are retained. After training, the model is serialized using Torch Scripts, resulting in a serialized model file (.pt). This serialized model encapsulates the learned information and the model's architecture.

5. Integration with C++ Code:

The serialized model file is integrated into the C++ codebase. The "torch" library is utilized to load the serialized model, which in turn converts it into a deserialized model. This deserialized model is a representation of the trained CNN, ready for utilization within the SystemC environment.

6. SystemC Integration: Master-Slave Interaction:

In the SystemC environment, the deserialized model is sent to the Slave Module. The Master Module initiates the process by providing a testing image in the form of a payload to the Slave Module. The Slave Module then passes the testing image through the deserialized model, performing the necessary computations. Finally, the Slave Module sends the model's prediction (output class label) back to the Master Module, completing the flow.

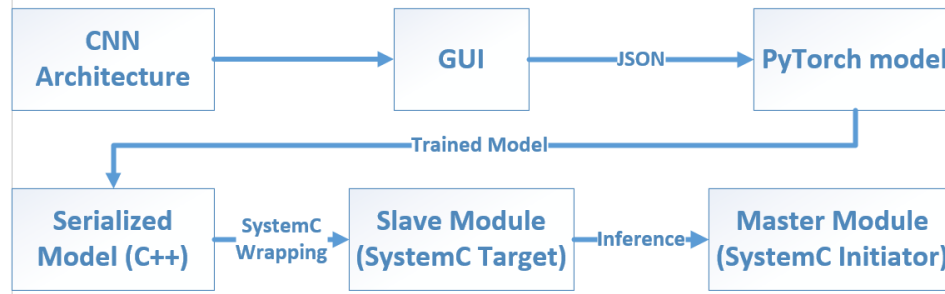


Figure 17 - Tool Overall Flow

This comprehensive flow ensures a seamless integration between user-defined CNN architectures, model training, serialization, and the deployment of the accelerator within the SystemC environment. By leveraging the capabilities of Qt, Torch Scripts, and SystemC, the project enables efficient development and evaluation of CNN-based AI accelerators, catering to both custom-built architectures and pre-trained models like VGG-16.

B. Template Design Pattern

In the development of our project, we have employed the Template Design Pattern to enhance modularity and extensibility in our codebase. The Template Design Pattern provides a flexible and scalable approach for designing classes that share a common structure but have varying implementation details.

In our case, we have utilized the Template Design Pattern to establish a hierarchical relationship between the "DNN" super-class and the "CNN" sub-class, enabling us to define a standardized workflow for building and training CNNs.

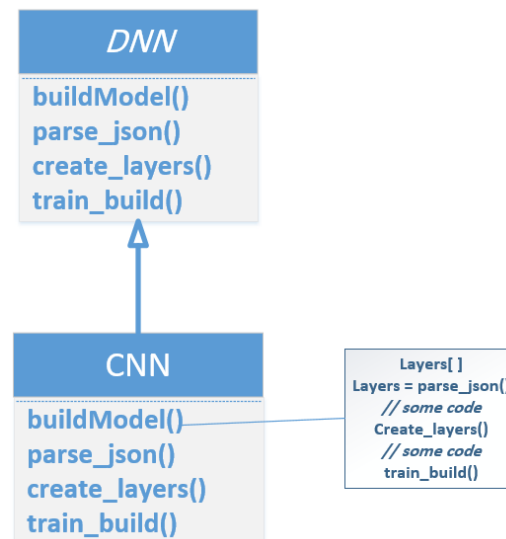


Figure 18 - Template Design Pattern

We chose the Template Design Pattern for several compelling reasons. Firstly, it promotes code reuse and encapsulation by abstracting common functionality into the super-class, "DNN," while allowing specific implementations to reside in the sub-class, "CNN." This separation of concerns



ensures that the core structure and workflow remain consistent across different CNN architectures while enabling flexibility in customizing the individual implementations. Secondly, the Template Design Pattern facilitates code maintenance and scalability. By adhering to a standardized structure, it becomes easier to introduce and integrate new CNN architectures into our system. The defined abstract functions, such as "buildModel," "parse_json," "create_layers," and "train_build," provide a clear guideline for implementing specific functionalities within the "CNN" sub-class. This consistency ensures a unified and cohesive development process, minimizing errors and promoting efficient collaboration within our team.

Lastly, the Template Design Pattern aligns well with the principles of object-oriented programming and design. It allows us to leverage the power of inheritance, abstraction, and polymorphism, enabling modular and reusable code. This design pattern also promotes code readability and maintainability by providing a logical and intuitive structure for other developers to understand and extend our project in the future.

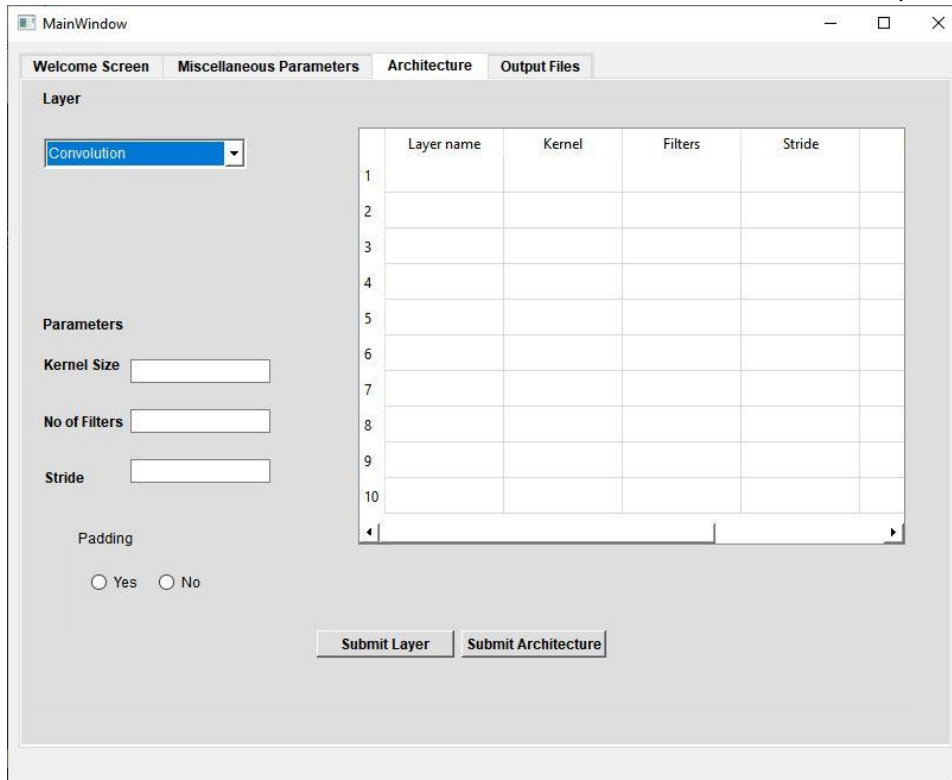
By employing the Template Design Pattern in our project, we have achieved a well-organized and flexible architecture that facilitates the development of various CNN architectures while maintaining a cohesive and standardized workflow.

XI. CHAPTER 5: METHODOLOGIES

The methodologies employed in this project encompass a series of well-defined steps and techniques aimed at achieving the desired objectives. Each phase contributes to the overall flow and ensures the seamless integration of various components.

A. Model Specification & JSON File

In this phase of the project, we focus on capturing and organizing the specifications of the CNN architecture and other miscellaneous parameters through the user-friendly Qt GUI wizard. The Qt wizard consists of two tabs: the CNN architecture tab and the miscellaneous parameters tab.



The screenshot shows the 'Architecture' tab of the Qt GUI Wizard. It features a 'Layer' dropdown menu set to 'Convolution'. Below it are input fields for 'Kernel Size', 'No of Filters', and 'Stride', along with a 'Padding' section with 'Yes' and 'No' radio buttons. To the right is a table with columns 'Layer name', 'Kernel', 'Filters', and 'Stride', containing 10 rows for layer configuration. At the bottom are 'Submit Layer' and 'Submit Architecture' buttons.

	Layer name	Kernel	Filters	Stride
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				

Figure 19 - Qt GUI Wizard (Architecture tab)

CNN Architecture Tab:

The CNN architecture tab allows the user to define the layers of the CNN according to their requirements. The following specifications can be set for each layer:

- **Layer Type:** The user selects the type of layer they want to add, such as Convolution, Max Pooling, Average Pooling, Depth-wise Convolution, Flatten, or Fully-Connected.
- **Convolution Parameters:** If the user selects Convolution or Depth-wise Convolution, they need to specify the kernel size, number of filters, stride, and padding for the layer.
- **Max Pooling / Average Pooling Parameters:** For Max Pooling or Average Pooling layers, the user specifies the kernel size and stride.
- **Flatten Layer:** The Flatten layer requires no additional parameters.

- **Fully-Connected Layer Parameters:** If the user chooses the Fully-Connected layer, they need to specify the number of filters.

Parameter	Value
1 Input Width	28
2 Input Height	28
3 Input Type	Gray Scale
4 Convolution Type	Standard
5 Data CSV Path	E:\courses\Grad...
6 Batch Size	256
7 Learning Rate	0.001
8 Number of Epochs	10
9 Optimizer	Adadelta
10 Loss Function	L1Loss

Figure 20 - Qt GUI Wizard (Miscellaneous Parameters tab)

Miscellaneous Parameters Tab:

In the miscellaneous parameters tab, the user provides additional specifications necessary for training the CNN. These parameters include:

- **Input Width and Height:** The dimensions of the input images.
- **Dataset CSV Path:** The path to the dataset in CSV format.
- **Batch Size:** The number of training examples processed in each iteration.
- **Learning Rate:** The rate at which the model learns during training.
- **Number of Epochs:** The number of times the model goes through the entire dataset during training.
- **Optimizer:** The optimization algorithm used for updating the model's weights and biases. The available options are Adam, AdaDelta, and AdaGrad.
- **Loss Function:** The loss function used to measure the model's performance. Options include L1Loss, MSE Loss, CrossEntropyLoss, and NLL-Loss.
- **Input Type:** The color representation of the input images, either RGB or Greyscale.
- **Convolution Type:** The type of convolution used, either standard convolution or Winograd convolution.

Once the user has specified all the desired CNN architecture and miscellaneous parameters, the Qt wizard generates a JSON file containing the collected information. This JSON file serves as a structured representation of the model specifications, enabling easy access, storage, and future reference.

As an example, consider a generated JSON file for a Lenet-5 architecture with specific parameter values. The JSON file contains an array of "layers" specifying the layer configurations and an array of "params" containing the miscellaneous parameters. Each layer entry includes details such as the number of channels, filters, height, width, kernel size, padding, and stride.

By utilizing the Qt wizard and generating the JSON file, we ensure an efficient and standardized approach to capturing the CNN architecture and miscellaneous parameters, which are essential for further stages of the project.

Sample JSON File:

```

1  {
2    "layers": [
3      {
4        "channels": 1,
5        "filters": 6,
6        "height": 32,
7        "kernel_size": 5,
8        "name": "conv1",
9        "padding": 0,
10       "stride": 1,
11       "units": 0,
12       "width": 32
13     },
14     {
15       "channels": 6,
16       "filters": 6,
17       "height": 28,
18       "kernel_size": 2,
19       "name": "maxpool1",
20       "padding": 0,
21       "stride": 2,
22       "units": 0,
23       "width": 28
24     },
25     {
26       "channels": 6,
27       "filters": 25,
28       "height": 14,
29       "kernel_size": 5,
30       "name": "conv2",
31       "padding": 0,
32       "stride": 1,
33       "units": 0,
34       "width": 14
35     },
36     {
37       "channels": 25,
38       "filters": 16,
39       "height": 10,
40       "kernel_size": 2,
41       "name": "maxpool2",
42       "padding": 0,
43       "stride": 2,
44       "units": 0,
45       "width": 10
46     }
47   ]

```

Figure 21 - Sample JSON File (1)

```

44  {
45    "channels": 400,
46    "filters": 120,
47    "height": 1,
48    "kernel_size": 1,
49    "name": "FC",
50    "padding": 0,
51    "stride": 1,
52    "width": 1
53  },
54  {
55    "channels": 120,
56    "filters": 84,
57    "height": 1,
58    "kernel_size": 1,
59    "name": "FC",
60    "padding": 0,
61    "stride": 1,
62    "units": 120,
63    "width": 1
64  },
65  {
66    "channels": 84,
67    "filters": 10,
68    "height": 1,
69    "kernel_size": 1,
70    "name": "FC",
71    "padding": 0,
72    "stride": 1,
73    "width": 1
74  }
75  ],
76  "params": [
77    {
78      "batch_size": "256",
79      "csv_path": "E:\\courses\\Graduation Project\\Git_Graduation\\CSV_Mnist",
80      "learning_rate": "1e-1",
81      "loss_fun": "CrossEntropyLoss",
82      "num_epochs": "2",
83      "optimizer": "Adam"
84    }
85  ]
86  }

```

Figure 22 – Sample JSON File (2)

In the next stages of the project, this JSON file serves as a crucial input for generating the CNN model code and further integrating it into the overall workflow.

B. Code Generation with PyTorch

In order to automate the process of generating code for the CNN architecture, a code generation approach using PyTorch was adopted. This approach involved parsing a JSON file that contains the specifications of the CNN layers and other parameters. The code generation process consisted of three main functions: "parse_json", "create_layers", and "train_build".

1. Parsing the JSON File

The “parse_json” function is responsible for loading the JSON file and extracting the necessary information from it. The JSON file contains the specifications of each layer in the CNN architecture, such as the layer type, filters, channels, height, width, kernel size, stride, padding, and units. Additionally, the JSON file includes miscellaneous parameters for training the CNN model, such as batch size, learning rate, optimizer, loss function, CSV file path, and the number of epochs.

The “parse_json” function reads the JSON file and extracts the layer specifications, creating Layer objects for each layer. These Layer objects encapsulate the layer properties. The miscellaneous parameters for training are also extracted and stored in variables. The extracted layer objects and training parameters are then used in subsequent steps of the code generation process.

2. Generating CNN Layers and Model Code

The “create_layers” function iterates over each layer extracted from the JSON file and generates the corresponding code for each layer. Depending on the layer type, the function dynamically generates the appropriate PyTorch code for that layer. The generated code includes the initialization of the layer and its parameters within the CNN model.

To facilitate the code generation process, a “model.py” file is created, which will contain the complete code for the CNN architecture. Inside the “model.py” file, a CNN class is defined. This class inherits from the PyTorch “nn.Module” class and represents the CNN architecture. The “__init__” function of the CNN class initializes the CNN layers based on the extracted layer specifications. The forward function is also defined within the CNN class, representing the forward pass of the CNN model.

The generated code for each layer is appended to the “torch_code” string, which stores the overall PyTorch code for the CNN architecture. The forward function is constructed by sequentially applying each layer to the input data, using the appropriate layer name and generated layer code. Finally, the “torch_code” string is written to the “model.py” file.

3. Generating Training Code

The “train_build” function is responsible for generating the code required to train the CNN model using the extracted miscellaneous training parameters. The generated code includes importing the necessary modules from PyTorch, setting the training parameters such as learning rate, batch size, and number of epochs, and defining the training loop.

The training code includes creating instances of the dataset, defining data loaders, instantiating the CNN model defined in the “model.py” file, specifying the optimizer and loss function based on the extracted parameters, and performing the training loop. The training loop iterates over the training dataset, computes the forward pass, calculates the loss, performs backpropagation, and updates the model's parameters. Additionally, evaluation on validation and test datasets can be included in the training loop to monitor the model's performance.

The generated training code is written to the “train.py” file, which will be responsible for training the built CNN architecture.

4. Integration and Workflow

The three main functions, “parse_json”, “create_layers”, and “train_build”, are integrated within the “BuildModel” method of the “Cnn” class. This method encapsulates the overall process of code generation and training model construction. Upon calling the “BuildModel” method, the JSON file is parsed, the CNN layers and model code are generated, and the training code is created if the CSV file path is specified. The resulting code files, “model.py” and “train.py”, contain the complete code for the CNN architecture and its training.

The generated code files can be executed using the appropriate tools and libraries. Running the “train.py” file initiates the training process for the constructed CNN architecture, utilizing the specified training parameters.

The code generation approach using PyTorch provides an automated and systematic way to generate CNN architectures and associated training code based on the provided specifications in the JSON file. This approach simplifies the process of designing and training CNN models, enabling efficient experimentation and exploration of different architectures and training configurations.

C. Torch Script & Model Serialization

To integrate the trained PyTorch model into a C++ codebase, Torch Script and model serialization were utilized. Torch Script allows for the conversion of PyTorch models into a serialized format that can be loaded and executed efficiently in different programming languages, including C++. The serialization process converts the model into a serialized file (.pt) that can be easily loaded and utilized in the C++ code.

The “torch.jit.trace” function is used to trace the trained PyTorch model. The trace function takes two arguments: the model and an example input tensor (torch.rand(1, 1, 28, 28) for instance when working with Lenet-5). The model represents the built and trained PyTorch model, while the input tensor serves as a sample input to the model. The trace function analyzes the model and traces the execution, capturing the operations performed on the input tensor.

The “torch.jit.trace” function returns a script module that encapsulates the traced model. This script module represents the serialized version of the PyTorch model, ready to be used in the C++ codebase. It provides a compact and optimized representation of the model, enabling efficient execution and integration.

Finally, the serialized model is saved to a file using the save method of the traced model (traced_model.save(“trained_model.pt”). The saved file, named “trained_model.pt” in this example, contains the serialized model that can be loaded and utilized in the C++ codebase.

The serialized model file (.pt) can be easily loaded in a C++ program using the appropriate libraries and functions. The loaded model can then be used for inference and other tasks within the C++ codebase, providing seamless integration between the trained PyTorch model and the C++ environment.

The utilization of Torch Script and model serialization facilitates the deployment and utilization of PyTorch models in production systems. It enables the seamless integration of machine learning models trained in PyTorch with existing C++ codebases, allowing for efficient and scalable inference in various application domains.

D.Integration of the C++ Environment

To seamlessly integrate the trained PyTorch model into the C++ codebase, the serialized model file (.pt) obtained from the Python codebase is deserialized and utilized in the C++ environment. This section outlines the process of integrating the Python and C++ codebases, allowing for efficient execution and inference of the trained model within the C++ environment.

The “torch::jit::Module” class is used to represent the serialized model. The module object is created to load and store the serialized model obtained from the .pt file. The model is deserialized using the “torch::jit::load” function, which takes the path to the serialized model file as input.

To utilize the deserialized model efficiently, the appropriate device is set based on the availability of CUDA. If CUDA is available, the device is set to “torch::kCUDA”, indicating the usage of GPU acceleration. Otherwise, the device is set to “torch::kCPU” for CPU-based execution.

The deserialized model is then moved to the selected device using the “module.to(device)” call. This ensures that the model is appropriately allocated and optimized for execution on the chosen device.

By calling “module.eval()”, the model is switched to the evaluation mode, preparing it for inference tasks within the C++ codebase.

To perform inference using the deserialized model, an input tensor is prepared. The input tensor is created by reshaping the “input_data” using the dimensions specified by “batch_size”, “num_channels”, “height”, and “width”. The “input_data” represents the input data for which inference is desired.

The input tensor is then added to the “ivalueVector”, which is a container used to store inputs for model execution. Additional inputs can be added to the “ivalueVector” if the model requires multiple input tensors or additional arguments.

Once the deserialization and input preparation steps are complete, the C++ codebase is ready to utilize the deserialized model for inference or other tasks within the C++ environment.

The integration of the C++ environment with the serialized PyTorch model facilitates the seamless execution of the trained model in production systems or applications that require C++ compatibility. This integration allows for efficient utilization of the trained model's capabilities in diverse domains, providing the benefits of PyTorch's machine learning capabilities within the C++ codebase.

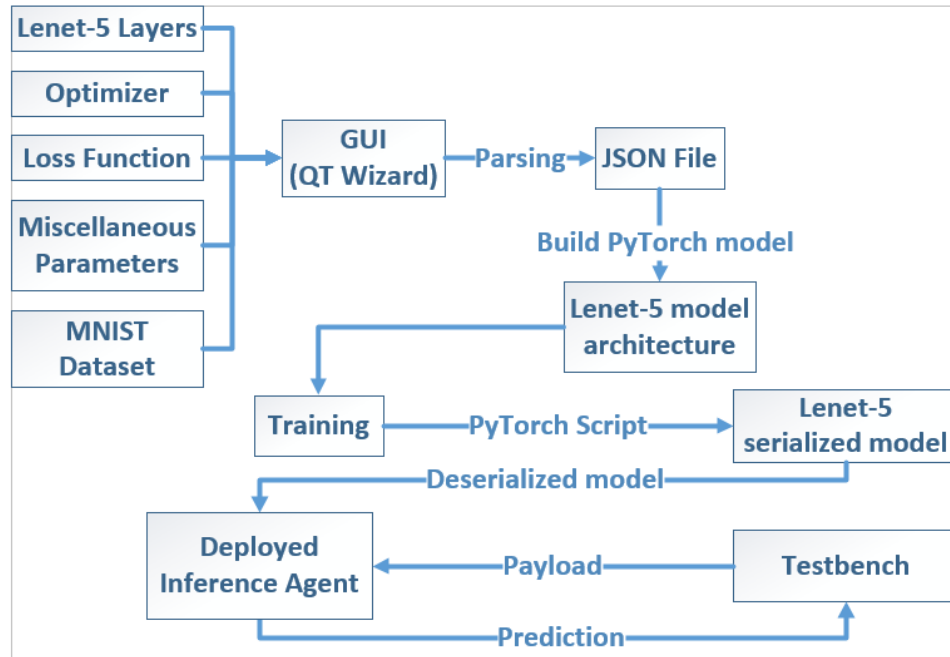


Figure 23 - Lenet-5 Full Flow

XII. CHAPTER 6: VERIFICATION ENVIRONMENT

In this chapter, we present the verification environment designed to validate the functionality and performance of the implemented system. The verification process plays a crucial role in ensuring the correctness and reliability of the system, allowing us to gain confidence in its behavior and make informed decisions regarding its deployment.

The verification environment encompasses various aspects of the system, including the master-slave SystemC setup, payload and image processing, and model inference and output. Each of these sub-topics contributes to the comprehensive evaluation of the system's behavior and performance under different scenarios.

A. Description of Master-Slave SystemC Setup

The master-slave SystemC setup is a key component of the verification environment, enabling the communication and interaction between the initiator (master) and the target (slave) modules. This setup facilitates the exchange of data and control signals necessary for the evaluation of the system's functionality and performance.

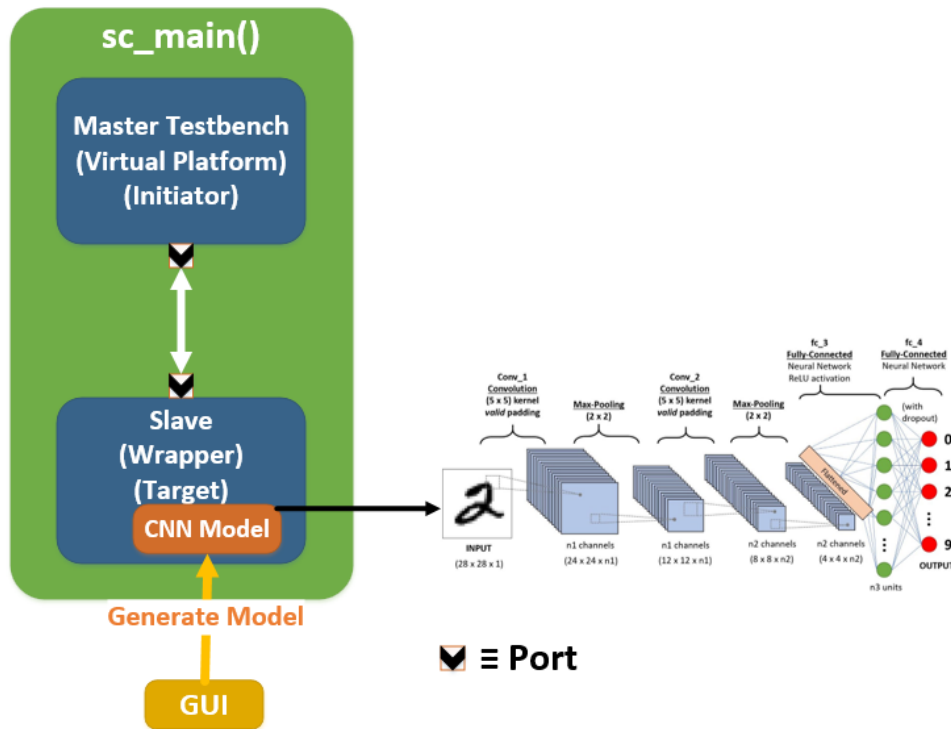


Figure 24 - Master-Slave SystemC Setup

The initiator module, represented by the SC_MODULE named "Initiator," is responsible for initiating the data transfer process. Within this module, a testing image is generated and converted into a payload consisting of floating-point numbers representing the pixels of the image. To facilitate this process, the MNIST dataset is utilized, specifically the "kTest" mode, to obtain a specific image for evaluation. The image is extracted from the dataset, and an additional dimension is added to conform to the expected input format of the model.

Once the payload is prepared, it is encapsulated within a transaction payload object, "tlm_generic_payload," which serves as the carrier of data during communication. The payload is then sent through the "init_socket," a simple initiator socket, to initiate the data transfer to the target module.

On the other side of the communication, the target module, represented by the SC_MODULE named "Target," receives the payload transmitted by the initiator. The target module plays a vital role in the inference process, utilizing a serialized PyTorch model to process the received image and generate predictions.

Within the target module, the received payload is extracted, and the input data, in the form of a floating-point array, is obtained. The PyTorch model, represented by the "torch::jit::Module" class, is deserialized from a file using the "torch::jit::load()" function. The model is then moved to the appropriate device, either the CPU or GPU, based on availability. Additionally, the model is set to evaluation mode using the "module.eval()" function call.

To prepare the input tensor for the model inference, the input data array is reshaped into a torch tensor with dimensions specified by the batch size, number of channels, height, and width. The reshaped input tensor is then added to a vector called "ivalueVector" which holds the input values to be passed to the model's forward function.

The model inference is performed by calling the "module.forward()" function with the input vector as an argument, which returns the output tensor. The output tensor is then processed to obtain the predictions. The softmax function is applied to the output tensor to obtain the probabilities for each class, and the argmax function is used to determine the predicted class label.

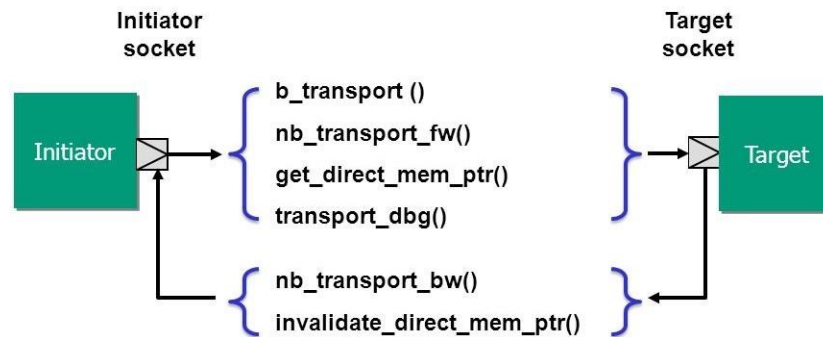


Figure 25 - Initiator & Target Communication

Furthermore, the target module extracts the elements from the output tensor and stores them in a C++ vector for further analysis and processing. The flattened array is then set as the data of the payload, and the appropriate data length is assigned.

Throughout the entire process, the target module monitors the type of transaction being performed, whether it is a read or write transaction, and sets the response status accordingly. The response status is crucial in determining the success or failure of the transaction and is used to provide appropriate feedback to the initiator.

The master-slave SystemC setup, with the initiator and target modules, establishes a bidirectional communication channel for the exchange of data between the initiator module and the target module as shown in Figure 25.

When the initiator module invokes the “b_transport()” function on the target module, it initiates a transaction by passing a transaction payload object as an argument. This payload object carries the necessary information to perform the data transfer, such as the data itself, data length, transaction type (read or write), and other control signals.

Upon receiving the “b_transport()” call, the target module's implementation of the function is invoked. Inside the target module's “b_transport()” function, several steps are typically executed:

- **Extracting Payload Information:** The target module retrieves the relevant information from the payload object, such as the data pointer, data length, and transaction type (read or write).
- **Processing the Transaction:** Depending on the transaction type, the target module performs specific actions. For a read transaction, the target module may read data from internal storage or perform computations and provide the results in the payload. For a write transaction, the target module may update internal states or perform other operations based on the received data.
- **Setting Response Status:** After processing the transaction, the target module sets the response status in the payload object to indicate the success or failure of the transaction. This response status is essential for the initiator module to determine the outcome of the data transfer.
- **Sending Response:** Once the response status is set, the target module returns from the “b_transport()” function, effectively signaling the end of the transaction. The payload object, including any updated data or response status, is passed back to the initiator module.

Back in the initiator module, after the “b_transport()” call, the initiator can examine the payload object to retrieve the response status, any updated data, or other relevant information provided by the target module. Based on this information, the initiator can proceed with further operations or make decisions accordingly.

Overall, the “b_transport()” function serves as the primary means of communication between the initiator and target modules in a SystemC simulation. It enables the exchange of data, control signals, and response status, facilitating the coordination and synchronization of the master-slave SystemC setup.

This integration allows for the seamless integration of the PyTorch model into the SystemC environment, enabling real-time image processing and prediction.

By utilizing this master-slave SystemC setup, the verification environment can validate the functionality and performance of the system. The initiator module generates test images, encapsulates them as payloads, and sends them to the target module. The target module receives the payload, processes the image using the deserialized PyTorch model, and returns the predictions back to the initiator.

The master-slave SystemC setup, coupled with the payload-based communication mechanism, provides a robust framework for testing and verifying the image processing and inference capabilities of the implemented system. The insights gained from this setup contribute to a comprehensive evaluation of the system, ensuring its reliability and aligning its behavior with the intended functionality.

B. Payload & Image Processing

In the master-slave SystemC setup, the payload plays a crucial role in carrying the image data from the initiator module to the target module and receiving the processed results back. It encapsulates the necessary information for data transfer, including the image payload itself, data length, transaction type, and response status.

1. Payload Structure

The payload used in the SystemC code is an instance of the “tlm_generic_payload” class. It is a generic payload type provided by the TLM library in SystemC. The payload structure consists of the following key components:

- **Data Pointer:** The payload contains a data pointer that holds the memory address of the image payload. It allows efficient data transfer between the initiator and target modules.
- **Data Length:** The payload specifies the length of the image payload in bytes. This information is vital for correctly interpreting and processing the image data.
- **Transaction Type:** The payload indicates the type of transaction, which can be either a read or a write. A read transaction indicates that the initiator requests data from the target, while a write transaction implies that the initiator is sending data to the target.
- **Response Status:** After the target module processes the transaction, it sets the response status in the payload to indicate the success or failure of the operation. The response status enables the initiator module to determine the outcome of the data transfer.

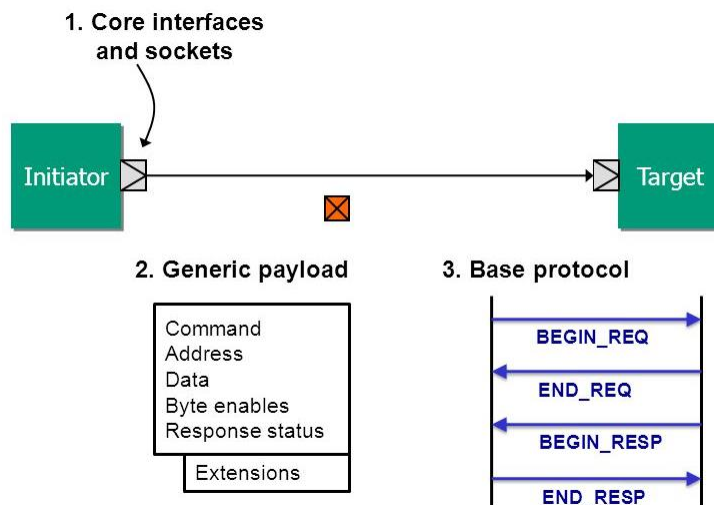


Figure 26 - SystemC Generic Payload

Figure 26 shows a simple TLM transaction between an Initiator and a Target through TLM generic payload. The payload supports two commands: read and write. The specific command, whether it is a read or write, is randomly determined and assigned to the command attribute. The address attribute represents the lowest memory address to which data is to be read from or written to.

The data pointer attribute points to a data buffer within the initiator module, while the data length attribute specifies the length of the data array in bytes. When a write command is issued, the data from the data array is copied to the target module. Conversely, for a read command, the data is copied from the target module to the data array. The actual copying process takes place within the target module.

When the byte enable pointer is set to 0, it indicates that byte enables are not used. There is also a byte enable length attribute.

To ensure proper initialization, the response status should always be set to “TLM_INCOMPLETE_RESPONSE” initially. The target module is responsible for updating the response status accordingly.

2. Image Processing Steps

The target module in the SystemC code performs image processing using a trained deep learning model. The image processing steps involve the following operations:

1. **Data Extraction:** Upon receiving the image payload from the initiator module, the target module extracts the image data from the payload. It interprets the data pointer and data length information to access the image pixels.
2. **Model Loading:** The target module loads a trained deep learning model using the “torch::jit::load()” function from the torch C++ library. The model is deserialized from the stored .pt file, which was trained and saved in PyTorch.
3. **Input Tensor Creation:** To pass the image data through the model, the target module creates an input tensor from the extracted image data. The input tensor is reshaped to match the expected input shape of the model, which is typically in the form of [batch_size, num_channels, height, width].
4. **Model Inference:** The target module performs model inference by invoking the “module.forward()” function, passing the input tensor. This step applies the trained model to the input data, producing an output tensor containing the predicted values.
5. **Prediction Post-processing:** After obtaining the output tensor, the target module applies post-processing steps to obtain the final predictions. In the SystemC code, softmax, sigmoid, and argmax operations are performed on the output tensor to derive probability distributions, activation values, and argmax predictions, respectively.
6. **Response Generation:** The target module populates the payload with the processed results, such as the predicted values or probability distributions. The flattened results are stored in a vector and then assigned to the payload's data pointer. The payload's data length is updated accordingly to reflect the size of the processed data.
7. **Response Transmission:** Finally, the target module sets the response status in the payload to indicate the success of the operation. The payload, containing the processed results and response status, is returned to the initiator module through the target socket.

By following these image processing steps, the target module applies the trained deep learning model to the received image payload and generates predictions or other desired results. The payload acts as a container for the image data, facilitating its transfer and retrieval between the initiator and target modules, while the image processing steps utilize the power of the deep learning model to extract meaningful information from the image payload.

This combination of payload utilization and image processing steps enables efficient communication and data processing within the master-slave SystemC setup, making it a robust framework for image analysis and prediction tasks.

C. Model Inference & Output

This section describes the process of performing model inference at the Target (Slave SystemC module) and the generation of output after the inference is complete in the SystemC code. This section provides detailed insights into the steps involved in running the model and obtaining the predictions.

1. **Model Loading and Setup:**

- The Target module loads the trained model using the “torch::jit::load()” function from the serialized model file.
- The loaded model is then moved to the appropriate device, either “torch::kCPU” or “torch::kCUDA”, based on availability.
- The model is set to evaluation mode using the “module.eval()” function call.

2. **Input Preparation:**

- The input image data received from the Initiator module is reshaped to match the expected input dimensions of the model.
- A tensor is created from the reshaped input data using the “torch::from_blob()” function.

3. **Post-processing:**

- The output tensor may undergo further processing or analysis, depending on the specific requirements.
- Common post-processing steps include applying softmax or sigmoid functions to obtain probability distributions or applying argmax to get the predicted class label.
- These post-processing operations can be performed using functions from the torch library, such as “torch::softmax()”, “torch::sigmoid()”, or “torch::argmax()”.

4. **Output Generation:**

- The predictions or processed output from the model are obtained as tensors.
- The values from the output tensor can be accessed and displayed using appropriate tensor accessor methods or by converting them to C++ arrays.
- The predictions are printed to the console using cout statements.

5. **Timing and Performance:**

- Timing information can be collected to measure the inference time taken by the model. This can be done by recording the start and end times using “clock()” or a similar mechanism.
- The duration of the inference process is calculated by subtracting the start time from the end time.
- The inference time can be outputted to provide insights into the model's performance.



By following these steps, the Target module successfully performs model inference on the received input data and generates output predictions or results. The accuracy and effectiveness of the predictions depend on the quality of the trained model and the input data provided.

XIII. CHAPTER 7: RESULTS

This chapter presents the findings and outcomes of the conducted experiments and evaluations. The chapter provides an in-depth analysis of the experimental setup, training, and testing procedures employed during the project. The performance metrics used to assess the effectiveness of the implemented models are also discussed. Additionally, this chapter includes a comprehensive discussion on the advantages and disadvantages of the tool utilized in the verification environment. By presenting these results and insights, this chapter aims to shed light on the performance, capabilities, and potential limitations of the developed system.

A.Experimental Setup & Dataset

This part describes the experimental setup and the dataset (e.g., MNIST) used for training and evaluating the model (e.g., Lenet-5). The hardware and software configurations, along with the specifics of the MNIST dataset, are presented in this section of the thesis.

1. Hardware Setup

The experiments were conducted on a computer system equipped with an Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz processor, 8.00 GB of installed RAM, and a 64-bit operating system. The system was used for executing the model training and testing procedures.

2. Software and Libraries

The implementation of the system was carried out using PyCharm as the primary development environment. The following libraries were utilized in the project:

- **torch**: A popular deep learning framework for building and training neural networks.
- **torchvision**: A package providing datasets, models, and transformations for computer vision tasks.
- **torch.utils.data**: A module providing tools for data loading and manipulation during training.
- **nn (torch.nn)**: A sub-module of PyTorch that offers a wide range of neural network layers and loss functions.
- **Adadelta and Adam optimizers from torch.optim**: Optimization algorithms for updating model parameters during training.

3. Dataset Description

The MNIST dataset was employed for training and evaluating the model. It consists of a collection of hand-written digit images, each represented as a grayscale image with dimensions of 32x32 pixels. The dataset encompasses a total of 70,000 samples, with each sample labeled as one of the ten digits (0 to 9).

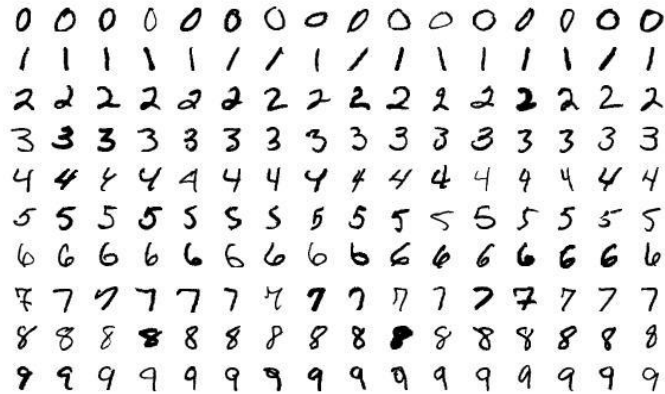


Figure 27 - MNIST Dataset

4. Dataset Split

The MNIST dataset was split into training and testing sets. The training set was comprised of 60,000 images, which were used to train the model. The remaining 10,000 images formed the testing set, utilized for evaluating the performance of the trained model.

B. Training & Testing Procedures

This section outlines the procedures followed for training a model using PyCharm and PyTorch and subsequently testing it within the SystemC codebase. The training process involves data loading, model initialization, training iterations, and model evaluation. The trained model, serialized using Torch Scripts, is then incorporated into the SystemC code for testing using image payloads.

1. Training Procedure

The training procedure involves the following steps:

1. **Data Loading:** The MNIST dataset is loaded using the “mnist.MNIST” class from the “torchvision.datasets” module. The dataset is split into training and testing sets, with the training set containing 60,000 images.
2. **Model Initialization:** The neural network model, named CNN, is initialized using the architecture defined in the “model.py” file (which contains the model that was input from the Qt wizard). This model is used to perform the training process.
3. **Data Preparation:** The training dataset is wrapped into a “DataLoader” object from the “torch.utils.data” module. This object allows for efficient batch-wise processing of the data during training. The data is shuffled within the “DataLoader” to introduce randomness.
4. **Model Training:** The model is trained using the training dataset in a loop for a specified number of epochs. Within each epoch, the model is set to training mode, and the training data is iterated over in batches. For each batch, the model performs a forward pass, computes the loss using the specified loss function (e.g., L1 loss), computes the

gradients, and updates the model's weights using the chosen optimizer (e.g., Adadelta with a learning rate of 0.001).

5. **Model Evaluation:** After completing the training iterations, the trained model's performance is evaluated using the testing dataset. The testing dataset, loaded separately from the training dataset, consists of 10,000 images. The model is set to evaluation mode, and predictions are made on the test data. The accuracy of the model's predictions is calculated by comparing the predicted labels with the ground truth labels.

2. Integration into SystemC codebase

Once the model is trained and saved as a serialized file (.pt file) using Torch Scripts, it is integrated into the SystemC codebase for testing. The serialized model file is deserialized and loaded into the Slave Module. The Master Module then sends image payloads to the Slave Module, which performs inference using the trained model.

1. **Model Deserialization:** The serialized model file (.pt file) is deserialized within the SystemC code to reconstruct the trained model. This allows the SystemC module to access the model's architecture and weights.
2. **Image Payloads:** The Master Module generates image payloads using the MNIST dataset or custom images. These payloads are sent to the Slave Module for inference.
3. **Inference:** The Slave Module receives the image payloads and performs inference using the loaded model. The image data is preprocessed as necessary, and the model predicts the corresponding labels for the images.
4. **Output Generation:** After inference, the Slave Module generates the output, which may include the predicted labels, confidence scores, or any other relevant information. The output can be further processed or used for subsequent tasks.

By integrating the trained model into the SystemC codebase, the testing procedure allows for evaluating the model's performance in a hardware simulation environment.

C. Performance Metrics

This section discusses the performance metrics used to evaluate the trained model's effectiveness in image classification tasks. The performance metrics provide insights into the model's accuracy and the execution time.

1. Accuracy

Accuracy is a commonly used performance metric for classification tasks. It measures the percentage of correctly classified images out of the total number of images. In the context of the MNIST dataset, accuracy represents the model's ability to accurately identify the handwritten digits, our Lenet-5 model achieved a 96.58% accuracy score.

```
Run: train
C:\Users\DELL\AppData\Local\Programs\Python\Python38\python.exe
Epoch [1/10], Loss: 1.5812
Epoch [2/10], Loss: 0.1106
Epoch [3/10], Loss: 0.3625
Epoch [4/10], Loss: 0.1855
Epoch [5/10], Loss: 0.2664
Epoch [6/10], Loss: 0.2807
Epoch [7/10], Loss: 0.0972
Epoch [8/10], Loss: 0.0646
Epoch [9/10], Loss: 0.0126
Epoch [10/10], Loss: 0.1955
Accuracy: 96.58%

Process finished with exit code 0
```

Figure 28 - Lenet-5 Accuracy Score

2. Execution Time

Execution time is another important performance metric, especially in real-time applications or resource-constrained environments. It measures the time taken by the model to perform inference on a given image or a batch of images. Lower execution time is desirable for efficient and responsive image classification systems.

The execution time can be measured at the system level, considering both the software (PyTorch-based model execution) and hardware (SystemC-based simulation) components. We will now conduct a quick comparison between running a Lenet-5 model verification between two different machines.

```
Microsoft Visual Studio Debug Console
SystemC 2.3.4 pub. rev. 20191203-Accellera --- Oct 17 2022 01:15:43
Copyright (c) 1996-2019 by all Contributors,
ALL RIGHTS RESERVED

address: 41
Image shape: [1, 1, 28, 28]
Image Label: 5
[ CPULongType{ } ]

Info: Initiator: Doing a WRITE transaction
Predictions: -0.4409 -2.0079 -0.5651 1.6755 -1.7178 5.7113 -1.3800 -2.6258 2.9005 -1.7978
[ CPUFloatType(1,10) ]
argMax Predictions: 5
[ CPULongType(1) ]
SoftMax Predictions: 0.0020 0.0004 0.0017 0.0163 0.0005 0.9221 0.0008 0.0002 0.0555 0.0005
[ CPUFloatType(1,10) ]
Sigmoid Predictions: 0.3915 0.1184 0.3624 0.8423 0.1522 0.9967 0.2010 0.0675 0.9479 0.1421
[ CPUFloatType(1,10) ]

Flattened array: -0.440924 -2.00785 -0.565133 1.67553 -1.71782 5.71127 -1.38001 -2.62578 2.90055 -1.79782

Inference Time taken: 0.639 seconds

Info: Target: Doing a WRITE transaction

Inputs received in initiator module: -0.440924 -2.00785 -0.565133 1.67553 -1.71782 5.71127 -1.38001 -2.62578 2.90055 -1.79782

Info: Initiator: Received correct reply.

Transaction Time taken: 0.656 seconds
```

Figure 29 - Machine 1 Execution Time

```

Microsoft Visual Studio Debug Console
Info: Initiator: Doing a WRITE transaction
Predictions: -0.4409 -2.0079 -0.5651 1.6755 -1.7178 5.7113 -1.3800 -2.6258 2.9005 -1.7978
[ CPUFloatType(1,10) ]
argMax Predictions: 5
[ CPULongType(1) ]
SoftMax Predictions: 0.0020 0.0004 0.0017 0.0163 0.0005 0.9221 0.0008 0.0002 0.0555 0.0005
[ CPUFloatType(1,10) ]
Sigmoid Predictions: 0.3915 0.1184 0.3624 0.8423 0.1522 0.9967 0.2010 0.0675 0.9479 0.1421
[ CPUFloatType(1,10) ]

Flattened array: -0.440924 -2.00785 -0.565133 1.67553 -1.71782 5.71127 -1.38001 -2.62578 2.90055 -1.79782

Inference Time taken: 0.78 seconds

Info: Target: Doing a WRITE transaction

-----

Inputs received in initiator module: -0.440924 -2.00785 -0.565133 1.67553 -1.71782 5.71127 -1.38001 -2.62578 2.90055 -1.79782

Info: Initiator: Received correct reply.

Transaction Time taken: 0.796 seconds

C:\Users\yshal\Downloads\GP\HWSW\x64\Debug\HWSW.exe (process 18944) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

```

Figure 30 - Machine 2 Execution Time

Machine 1 outperforms Machine 2 in terms of both transaction time and inference time. Machine 1 has a higher clock speed and more RAM, which likely contributes to its faster execution times. The difference in storage types (SSD vs. HDD) may also have an impact on overall performance, although it's not directly reflected in the execution time at the SystemC verification environment. It's important to note that other factors, such as software optimizations, can also influence execution time.

	CPU	RAM (GB)	Storage Drive	Transaction Time (seconds)	Inference Time (seconds)	Communication Delay (seconds)
Machine 1	Intel(R) Core(TM) i7-7660U @ 2.50GHz	16.0	512 SSD	0.656	0.639	0.017
Machine 2	Intel(R) Core(TM) i7-8550U @ 1.80GHz	8.0	1.0 TB HDD	0.796	0.78	0.016

Table 3 - Machines' Execution Time Comparison

D.Discussion

We will now discuss all the advantages and disadvantages of our tool.

Advantages:

- **Accuracy:** The project demonstrates high accuracy in image classification tasks. It achieves accurate predictions on the MNIST dataset, correctly identifying hand-written numbers with a high success rate.
- **Scalability:** The tool can handle large datasets efficiently. It is capable of training and testing on a dataset of 70,000 images without significant performance degradation.

- **Versatility:** The tool supports a range of hardware setups. It can be executed on different machines with varying specifications, making it adaptable to different computing environments.
- **Easy Integration:** The tool seamlessly integrates the PyTorch deep learning framework with the SystemC verification environment. This allows for efficient testing of the trained model using real-world payloads.
- **Serialization:** The project utilizes Torch Scripts to serialize the trained model into a .pt file. This serialized model can be easily transferred and deployed in the SystemC codebase, ensuring consistency and reproducibility of results.

Disadvantages:

- **Hardware Dependency:** The project's performance is influenced by the underlying hardware. Machines with lower specifications may experience longer execution times and reduced overall performance compared to higher-end machines.
- **Resource Requirements:** Training deep learning models can be computationally intensive and memory-consuming. The project may require substantial computational resources, such as RAM and storage, to handle large datasets and complex neural networks effectively.
- **Limited Dataset Augmentation:** The project does not incorporate data augmentation techniques during the training phase. Data augmentation, such as rotation, scaling, and flipping, can enhance model generalization by introducing additional variations in the training dataset.

XIV. CHAPTER 8: CONCLUSION

This chapter presents the conclusion of the project, summarizing the findings and highlighting the limitations and potential areas for future work. The project aimed to develop a tool that integrates the PyTorch deep learning framework with the SystemC verification environment for image classification tasks. The tool demonstrated high accuracy, scalability, and versatility, providing a seamless integration between the two environments. However, it is important to acknowledge the limitations and identify opportunities for future improvements and expansions.

A. Summary of Findings

During the course of the project, several significant findings were made, highlighting the advantages and accomplishments of the developed tool. Here is a comprehensive overview of the project's outcomes:

- **Seamless Integration of PyTorch and SystemC:** The tool successfully bridged the gap between PyTorch, a popular deep learning framework, and SystemC, a widely-used verification environment. This integration allowed for the seamless transfer of trained deep learning models into the SystemC codebase, enabling their testing and validation in a real-world hardware verification context.
- **Efficient Handling of Image Classification Tasks:** The tool demonstrated its effectiveness in handling image classification tasks using the MNIST dataset for instance. It exhibited high accuracy in predicting hand-written numbers, showcasing the potential of the integrated approach in real-world applications.
- **Hardware Adaptability:** The developed tool exhibited adaptability to different hardware setups and computing environments. It leveraged the underlying hardware resources efficiently, accommodating variations in processor types, RAM capacity, and storage devices. This flexibility enables the tool to be utilized in diverse hardware configurations, expanding its applicability in different domains.
- **Serialization and Model Deployment:** The serialization of trained models using Torch Scripts facilitated the smooth deployment of models within the SystemC codebase. This serialization process ensured the preservation of model architecture, weights, and parameters, enabling consistent and reproducible results during the testing phase.
- **Scalability and Dataset Size:** The tool demonstrated its scalability by efficiently handling large datasets. It was able to process the MNIST dataset, consisting of 70,000 samples, with ease. This capability opens the door for applying the tool to more extensive datasets and complex deep learning models in the future.
- **Versatility for Verification Environments:** The developed tool showcased versatility by providing a seamless integration between PyTorch and SystemC. It offered a comprehensive solution for testing deep learning models in hardware verification environments, enabling thorough validation and verification processes.
- **Interdisciplinary Collaboration:** The project necessitated collaboration between the deep learning field and hardware verification domain. The successful integration of

PyTorch and SystemC required experience from both fields, fostering interdisciplinary collaboration and knowledge exchange.

- **Demonstration of Proof-of-Concept:** The project served as a proof-of-concept for integrating deep learning frameworks and hardware verification environments. It provided a foundation for future research and development in this domain, encouraging further exploration and innovation in the intersection of deep learning and hardware verification.

These findings collectively emphasize the strengths and advantages of the developed tool, showcasing its potential to revolutionize the testing and verification of deep learning models in hardware environments.

B.Limitations & Future work

Despite the success of the project, several limitations and opportunities for future work exist:

- **Dataset Augmentation:** The project did not incorporate data augmentation techniques during the training phase. Future work could explore the integration of augmentation methods to further enhance the model's ability to generalize to unseen data and improve overall performance.
- **Performance Optimization:** Further optimization of the tool's performance, especially in resource-intensive environments, should be explored. Techniques such as parallel processing, or distributed computing could be investigated to enhance execution speed and efficiency.
- **Model Interpretability:** Future work could focus on incorporating techniques for model interpretability and understanding. Explaining the decision-making process of the deep learning model would enhance transparency and trustworthiness, especially in safety-critical applications.
- **Integration of Additional Machine Learning Algorithms:** Expanding the tool to support other machine learning algorithms beyond deep learning would increase its versatility and usefulness in a wider range of applications.

XV. REFERENCES

1. Asic-World. (n.d.). SystemC tutorial. Retrieved from <https://www.asic-world.com/systemc/tutorial.html>
2. Ng, A. (n.d.). Convolutional Neural Networks [Online course]. Coursera. Retrieved from <https://www.coursera.org/learn/convolutional-neural-networks>
3. NVDLA. (n.d.). NVIDIA deep learning accelerator. Retrieved from <http://nvdla.org/>
4. Ahmed, A. N., Fadel, K. A., Ahmed, K. H., Ali, M. A. (2021). An Automated Flow for Configuration and Generation of CNN based AI accelerators for HW Emulation & FPGA Prototyping (Unpublished master's thesis). Cairo University. Supervisors: AbdElSalam, M. Ossama, K. A.
5. Ibrahim, A. E., Younes, A. H. M., Abd-Elbadee, A. K., Mohammed, A. T., Abd-ElSadeq, B. E. A., Mohammed, M. A. (2022). An Automated Flow for Configuration and Generation of CNN based AI accelerators for HW Emulation & FPGA Prototyping (Unpublished master's thesis). Cairo University. Supervisors: Ossama, K. A. AbdElSalam, M.
6. SystemC: From the Ground Up (4th Edition) by David C. Black, Jack Donovan, Bill Bunton, and Anna Keist.
7. "Deep Learning: A Practitioner's Approach" by Josh Patterson and Adam Gibson.
8. LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11), 2278-2324.
9. Simonyan, K., & Zisserman, A. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556*.