Politecnico di Torino

Master's Degree in Electronics Engineering

Master's Degree Thesis

Implementation of Neural Network on FPGA Using HLS

Supervisor: Professor Mihai Lazarescu

Candidate: Bilal Awad (S320760)

Academic Year 2024–2025

Acknowledgments

I would like to express my sincere gratitude to my thesis supervisor, Professor Mihai Lazarescu, for his expert guidance and unwavering support throughout the course of this research. His insights and expertise were invaluable in shaping the direction and execution of my thesis on "Implementation of Neural Network on FPGA Using HLS."

I am also immensely thankful to the faculty and staff at the Politecnico di Torino. Their continuous encouragement and the intellectually stimulating environment they provided were fundamental to my academic growth.

Completing this thesis has been a challenging yet rewarding experience, and it would not have been possible without the support I received from my mentors and the broader university community. I am grateful to have had the opportunity to learn and conduct my research at such a prestigious institution.

Grazie

Summary

This thesis presents a full and technologically elaborate pipeline for mapping the architecture of a Temporal Convolutional Network (TCN) into FPGA hardware for low-energy and low-latency inference for applications in edge computing. The pipeline begins from the problem definition, wherein multivariate time-series input is split into sliding windows of size 15 time steps and 4 input features. The supervised learning label is taken as the mid-value of each window, which is equivalent to the model's receptive field.

We have implemented two TCN architectures in PyTorch: a basic dilated one and a more sophisticated one with skip connections, trainable normalization, dropout, and residual blocks. The one based on residuals was selected after comparing trials based on its temporal information modeling capacity, resistance to noise, and architectural flexibility.

One of the prime concerns of this project was hardware deployment efficiency through quantization. Post-Training Quantization (PTQ) via ONNX Runtime and Quantization-Aware Training (QAT) via Brevitas were both considered. QAT had the benefit of maintaining the performance of the model but lost out ultimately in terms of compatibility with NN2FPGA and support for symbolic operators. PTQ was instead utilized for converting the trained 32-bit model into its 8-bit integer ONNX equivalent. Weights and activation values were exported via Python scripts into fixed-point C++ header files for hardware synthesis, as `Q4.4` format.

The PyTorch model was hand-translated into synthesizable C++ based on needs for full architectural control, Xilinx HLS support, and visibility of all arithmetic-related operations. Fixed-point precision had been achieved with `ap_fixed<8,4>` types and intermediate summations using `ap_int<32>`. Dropout and normalization had been removed from hardware implementation for synthesis needs. The C++ model had contained one input layer of projection with size 1×1, three dilation residual blocks, and the last fully connected output layer for prediction of 2D coordinates.

Vivado HLS High-Level Synthesis was used for converting the C++ model into hardware IP. Loop unrolling, pipelining, array partitioning pragmas, and AXI4-Lite interface directives were used for optimization. The hardware has been synthesized in two levels: one non-pipelined for minimizing resources and one partly pipelined for optimized performance. Functional correctness was verified through comparison of generated traces of the C++ model with the target PyTorch implementation.

The final IP core was integrated into Vivado using IP Integrator, alongside essential components like the ZYNQ7 Processing System, Clocking Wizard, Processor System Reset, AXI BRAM Controller, and AXI GPIO. Connection Automation and manual wiring were used to establish AXI and clock/reset routing, followed by memory-mapped address assignment via the Address Editor. Once the system was implemented, Vivado's Report Power tool was used to conduct post-implementation power analysis with vectorless activity (12.5% toggle rate). The total power consumption was estimated at 1.871 W, broken

down into 0.365 W of static power, 0.892 W of dynamic logic power, 0.422 W of dynamic clock power, and 0.192 W for BRAM and DSP.

The pipelined design achieved a latency of approximately 9.65 μs per inference, translating to an energy cost of around 18.05 μJ. The non-pipelined design consumed less dynamic power but incurred significantly higher latency, illustrating a key trade-off between performance and efficiency. Accuracy evaluation across the deployment pipeline revealed an MSE of 0.065 m^2 for the original TensorFlow model, 0.1199 m^2 for the PyTorch implementation, and 0.2365 m^2 for the final C++ fixed-point inference system—highlighting quantization-induced precision loss while still ensuring practical performance for real-time applications.

In essence, this work proposes a reproducible, hardware-conscious, and scalable deployment strategy for deep temporal models using fixed-point arithmetic on FPGA platforms, with strong implications for future edge AI use cases such as indoor localization and sensor fusion.

Contents

List of Figures

Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
ARM	Advanced RISC Machine
ARRAY	Array Data Structure
ASCII	American Standard Code for Information Interchange
AXI	Advanced eXtensible Interface
BLE	Bluetooth Low Energy
BRAM	Block Random Access Memory
CPU	Central Processing Unit
CTRL	Control
DATAFLOW	Data Flow Architecture
DDR	Double Data Rate
DNN	Deep Neural Network
DRAM	Dynamic Random Access Memory
DSE	Design Space Exploration
DSP	Digital Signal Processing
DVFS	Dynamic Voltage and Frequency Scaling
FF	Flip-Flop
FINN	Fast Inference of Neural Networks
FPGA	Field-Programmable Gate Array
FULL	Full Precision
GPIO	General Purpose Input/Output
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HLS	High-Level Synthesis
II	Initiation Interval
III	Third Iteration
INLINE	Inline Function or Operation
INPUT	Input Signal/Data
INTERFACE	Hardware or Software Interface
IO	Input/Output
IP	Intellectual Property
IV	Initial Vector
LOOP	Loop Operation
LSTM	Long Short-Term Memory

LUT	Look-Up Table
MAC	Multiply-Accumulate
MAE	Mean Absolute Error
MSE	Mean Squared Error
MTC	Model Training Configuration
NAS	Neural Architecture Search
NN	Neural Network
OFF	Power-Off State
ONNX	Open Neural Network Exchange
OUTPUT	Output Signal/Data
PIPELINE	Pipelined Architecture
PS	Processing System
PTQ	Post-Training Quantization
QAT	Quantization Aware Training
QONNX	Quantized ONNX
RAM	Random Access Memory
RESOURCE	Hardware or Software Resource
RGB	Red Green Blue Color Space
RMSE	Root Mean Squared Error
RSSI	Received Signal Strength Indicator
RTL	Register Transfer Level
SAIF	Switching Activity Interchange Format
SCALE	Scaling Factor
TCN	Temporal Convolutional Network
TEX	Typesetting System
TPU	Tensor Processing Unit
TS	Timestamp
UNROLL	Loop Unrolling
UWB	Ultra-Wideband
ZYNQ	Zynq Programmable SoC

Chapter 1

Introduction

1.1 Introduction

Indoor location positioning — the determination of the exact location of users or devices indoors — has become increasingly important in many industries. Applications encompass healthcare monitoring and industrial automation, emergency response, smart building systems, surveillance, as well as supporting emerging paradigms such as the Internet of Things (IoT), smart city infrastructures, smart buildings, smart grids, and Machine-Type Communication (MTC) networks. High-accuracy indoor positioning allows real-time context-aware services, operational efficiencies, as well as improved safety in dynamic settings. Although conventional close-range communication technologies such as Wi-Fi, Bluetooth, and Ultra-Wideband (UWB) have encouraging localization capabilities, delivering these services with low latency as well as energy efficiency at the edge remains an engineering challenge [20].

To overcome such constraints, FPGAs have also been suggested as a reconfigurable hardware platform to accommodate machine learning models at the edge in indoor localization applications. As compared to general-purpose sequential execution-based CPUs, FPGAs provide high parallelism with the capability to process continuous streams of sensors in real time — an important enabler of fast and accurate location inference. Reprogrammability makes them very adaptable to iterative development as well as after-deployment adjustment, and the presence of embedded processing blocks allows small, energy-efficient system-on-chip designs. Such an amalgamation of flexibility, efficiency, and energy awareness makes the FPGAs an efficient way to implement the deep learning-based localization at the edge [21].

Particularly, deep models such as Temporal Convolutional Networks (TCNs) are well-suited to time series analysis in particular — thereby an appealing option to indoor localization systems. TCNs employ dilated causal convolutions to capture long-range effective memory with sequential dependencies, without the need to resort to recurrent structures. The deployment of TCNs to resource-constrained edge devices is, however, plagued with technical challenges related to inference speed, memory usage, as well as hardware availability.

This thesis presents the entire TCN-based model development and hardware implementation flow in the application of indoor localization with the aim of real-time inference on FPGA platforms. The flow starts with the creation and training of the TCN model in the software with TensorFlow, then converting it to PyTorch in order to gain more control over the quantization process. Quantization of the model is performed with fixed-point representation in order to achieve reduced computational overhead along with memory footprint. The obtained quantized network is re-implemented manually in C++ with fixed-point arithmetic to make it compatible with the HLS tool.

High-Level Synthesis, using tools like the Xilinx Vitis HLS, allows generation from high-level C++ descriptions to hardware. With the application of the HLS optimizations like loop unrolling, pipelining, and parallelization, the C++ design is converted to highly optimized hardware. The deployment is then completed with the integration into an FPGA platform using Vivado's IP integrator, thereby readying the path from high-level model to deployable hardware.

The two primary ambitions of the thesis are, one, to demonstrate an end-to-end deployment process of a deep learning model using quantization and HLS; two, to examine the inference accuracy, execution time, as well as hardware resource consumption tradeoffs. Throughout the course of the thesis, it contributes to the existing body of work related to the deployment of neural networks in edge configurations, with the special emphasis being placed on their deployment in energy-constrained real-time indoor localization systems.

Chapter 2

Background and Related Work

The rapid development of artificial intelligence (AI) and deep learning has created the need for seeking advancements in the hardware realizations of neural networks (NNs). Due to their high power efficiency, flexibility, and ability to perform tasks in parallel, Field Programmable Gate Arrays (FPGAs) have emerged as an active area of interest along with other hardware platforms like CPUs and GPUs. The use of NN through FPGAs has been eased by High-Level Synthesis (HLS) whereby engineers are able to implement utilizing higher-level languages like C, C++, or OpenCL. The subsequent literature review considers existing research on NN implementations over FPGAs, the role of HLS in hardware acceleration, and comparative reviews with other hardware platforms.

2.1 Literature Review

2.1.1 Neural Networks on FPGA

Neural networks (NNs) may be deployed on field-programmable gate arrays (FPGAs), which provide a good trade-off in terms of energy efficiency and computation speed. There have been some studies that have investigated FPGA-based implementations to go around the limitations of CPUs and GPUs, especially where low-latency inference or energy-efficient applications are required. Improving computational efficiency through parallelization and tailored hardware implementations, minimizing resource usage through model compression and quantization, and maximizing throughput via pipelining and memory hierarchy optimizations are the three general objectives of optimization research for this domain.

The other notable contribution in this direction is that of Guo et al. (2015) [1], which systematically explored FPGA-based deep neural network (DNN) inference with particular focus on hardware design methods such as quantization, pruning, and pipelining. Their findings established that FPGAs could achieve dramatic speedups relative to their CPU equivalents while maintaining reconfigurability. Subsequent to this, Suda et al. (2016) [2] proposed an optimized method to accelerating convolutional neural networks (CNNs) on FPGAs using fine-grained parallelism and optimization of memory bandwidth, with near real-time performance on vision processing applications. Contrarily, Li et al. (2017) [3] investigated the acceleration of recurrent neural networks (RNNs) on FPGAs, both for their latency as well as power efficiency in comparison with GPU-based implementations. Their research showed the advantages of FPGAs in sequential data processing, particularly in edge-computing when energy efficiency is crucial.

Combined, the above research identifies that FPGAs present a balanced middle level between application-specific integrated circuits (ASICs) high power efficiency and GPUs' programmability. While GPUs are particularly well-suited for raw throughput when there is large batch processing, FPGAs have higher energy efficiency and lower latency when there is streaming or single-sample inference, thus being best suited for embedded and real-time applications.

2.1.2 High-Level Synthesis (HLS) for FPGA Implementation

Industry-standard software such as Xilinx Vivado HLS (now Vitis HLS) and the Intel HLS Compiler are now a requirement in transforming high-level neural network models into optimized FPGA implementations. These tools abstract the low-level hardware design difficulties, enabling software engineers and AI practitioners to leverage FPGA acceleration without having to be hardware description language experts. This abstraction not only enables quicker prototyping and design iteration but also opens up FPGA development to a broader audience of engineers.

For instance, Qiu et al. (2016) [4] demonstrated how HLS can simplify the deployment of deep learning models onto FPGAs through automatic handling of resource allocation, scheduling, and dataflow architecture synthesis. Their research underscored the power of HLS to extract parallelism and improve memory access, both critical for sustaining high throughput in inference pipelines. Similarly, Wei et al. (2019) [5] introduced an HLS-acceleration framework tailored to image processing-based applications, demonstrating that HLS has the ability to cut down development time and effort by substantially while providing similar performance levels as hand-optimized RTL designs.

Along with these, community-developed open-source tool and framework advancements such as hls4ml and Torch2Vitis have also enriched the HLS ecosystem in a similar way. These tools target the auto-conversion of widely used neural network models—more specifically, those trained in PyTorch or Keras—into synthesizable C++ or SystemC forms. Supporting quantized and binarized operators, they enhance the machine learning model's compatibility with resource constraints of FPGA and bring configurable design knobs for area and latency tuning.

Despite these positive trends, HLS adoption remains encumbered by several trade-offs. The abstraction layer typically leads to inefficient use of hardware, particularly with logic packing and critical path timing, than well-designed HDL. Moreover, HLS tools are also prone to struggle with deeply pipelined or highly parallel designs with intricate control flows or irregular memory access patterns. These constraints can turn into performance bottlenecks in large-scale deployment scenarios.

However, the advantage of HLS in enhancing development productivity, coupled with the continuous enhancement of its compiler technology and template libraries, makes it a compelling and future-proof choice for FPGA-based acceleration of neural networks. As domain-specific languages and hardware-aware training models gain widespread popularity, HLS can potentially find itself at the forefront of making adaptive, high-performance AI systems edge- and embedded-friendly.

2.1.3 Comparison of FPGA with Other Hardware Architectures

While FPGAs offer strong advantages of reconfigurability and power efficiency, the case of accelerating neural networks is varied across hardware platforms of differing strengths and weaknesses. Modern deep learning software leverages graphics processing units (GPUs), tensor processing units (TPUs), and application-specific integrated circuits (ASICs) alongside FPGAs, with the optimal solution depending on application-specific criteria like computation throughput, power consumption, and flexibility requirements. Comparative studies in this area reveal substantial architectural compromises between programmability, performance, and power consumption.

One of the first pioneering works by Jouppi et al. (2017) [6] provided a detailed examination of Google's TPU architecture, displaying its higher performance-per-watt for large-scale cloud-based inference workloads. The authors noted the way the systolic array architecture of the TPU and its 8-bit quantization enable extreme throughput over matrix operations but at the cost of inflexibility in fixed-function capabilities. On the other hand, Han et al. (2016) [7] conducted an exhaustive comparison of FPGA and GPU mappings of deep learning accelerators and arrived at the conclusion that while GPUs excel in processing large-scale data sets, FPGAs are far more energy efficient - so much so that they are extremely suitable for edge devices and power-constrained environments. It is noteworthy in their work that FPGAs can be 5-10 times more energy efficient than GPUs for particular convolutional neural network workloads.

Scalability was the query addressed by Cong et al. (2018) [8], who extensively reviewed FPGA-based accelerators for more extensive and complex neural structures. They determined that while FPGAs offer unmatched versatility for model adjustment and model refreshes, they inherently suffer from memory bandwidth and logic resource constraints when executing extremely large models (like transformers with billions of parameters). This paper proposed hybrid methods, under which FPGAs perform preprocessing and compact models with offloading big computations to GPU/TPU clusters.

These comparison studies overall depict that hardware choice for accelerating neural networks is multidimensional trade-offs: Upcoming heterogeneous computing trends suggest that upcoming systems will combine these methodologies more and more, using FPGAs for low latency inference and input preprocessing and GPUs/TPUs for training as well as batch processing. This design co-design technique has especial potential for edge-cloud systems and real-time AI usage.

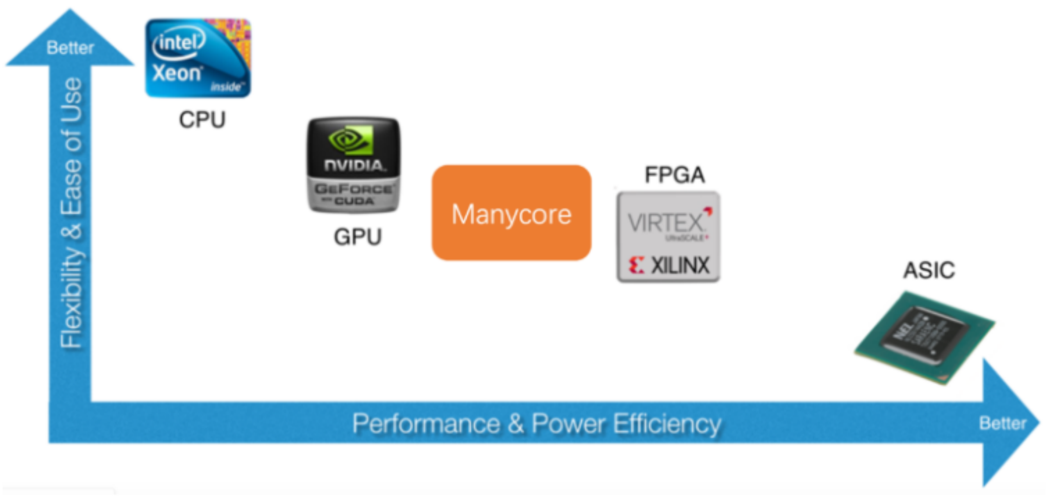

Figure 2.1. Comparison Between Hardware Architectures [27]

2.1.4 Optimization Techniques for FPGA-based NN Implementation

In order to meet the performance, resource, and power demands of a real-time application, various optimization techniques have been proposed to enhance the implementation of neural networks (NNs) on field-programmable gate arrays (FPGAs). The following sub-sections introduce some significant techniques that are thoroughly researched in the literature.

1. **Quantization and Pruning**
   Quantization reduces the bit-width of neural network weights and activations, usually from high-precision data types such as 32-bit floating point to lower-precision data types such as 16-bit fixed-point, 8-bit integers, or even binary (1-bit) type. Reducing them lowers the arithmetic intensity of operations such as matrix multiplications and convolutions, which are the workhorse of neural network computations. With smaller numerical values, quantization decelerates both computation time and memory consumption, hence facilitating faster inference and lower energy cost—properties particularly valuable in FPGA implementations where resources are limited and power budgets are constrained.

   Besides reducing hardware overhead, quantization further increases data throughput and facilitates more efficient patterns of memory accesses since more data can be encoded and transferred within the same memory space. These benefits render quantization a crucial way of deploying deep models for real-time and embedded applications. However, reduction of precision should be carefully managed to avoid compromising the model severely in terms of accuracy loss. This is typically accomplished through quantization-aware training (QAT) where the model is trained so as to counteract the effect of low-precision computations.

Pruning is a helping technique that minimizes the complexity of the neural network by removing weights, neurons, or full filters with the least impact on the network predictions. Pruning techniques can either be unstructured, removing individual weights, or structured, removing larger units such as channels or layers. The result is a slimmed-down network consuming less computational resources and memory bandwidth and hence more deployable on FPGAs.

Both quantization and pruning are used together extensively to compress models for low-hardware implementation. As demonstrated by Han et al. (2015) [9], joining these approaches can decrease model size by an order of magnitude and enhance performance at no great loss in precision. Umuroglu et al. (2017) [10] also demonstrated that binarized neural networks (BNNs) are high throughput and energy-efficient if executed on FPGA platforms using certain hardware architectures. These approaches are gaining momentum in applications such as edge computing, mobile devices, and real-time inference scenarios.

2. **Pipelining and Parallelism**
   FPGAs incorporate inherent flexibility for engineers to leverage spatial as well as temporal parallelism, the key enablers for accelerating neural network inference. Spatial parallelism is the concurrent execution of independent operations such as calculating multiple neuron or layer calculations in parallel. Temporal parallelism is distributing computation into sequential phases with pipeline registers in between. Each pipeline stage can handle an alternate data item in each clock cycle, allowing for a steady flow of data in the pipeline.

   Pipelining is especially helpful in reducing latency and increasing the throughput, since it allows the system to proceed with new inputs before completing the computations of the previous ones. In neural networks, where convolution, matrix multiplication, and activation functions are being repeatedly computed, pipelining keeps different parts of the computation running all the time and avoids idle hardware cycles.

   Parallelism does better than this by permitting a number of operations to be performed simultaneously. In convolution neural networks (CNNs), for example, all convolution filters along with the respective region of the input feature map can be computed in parallel, which greatly speeds up inference. Parallelism levels could be adjusted based on the amount of resources available in FPGAs, offering a scalable design paradigm well-suited to the application.

Zhang et al. (2015) [11] demonstrated that through accurate mapping of convolutional layers to pipelined and parallel FPGA hardware, highly significant inference speed and efficiency gains are possible over traditional sequential counterparts. Such techniques form the foundation for optimizing the computing capacity of FPGAs for real-time and high-performance deep learning systems.

3. **Memory Optimization**
   Effective memory management is a cornerstone of high-performance NN architectures on FPGAs, due mainly to the scarcity of on-chip memory and higher latency of memory access from off-chip memory. The bandwidth between FPGA and external memory can become a bottleneck unless efficiently managed, especially in memory-intensive operations like matrix multiplications and convolutions involving large volumes of data.

   To overcome these limitations, a variety of memory optimization techniques are commonly employed. Tiling is one technique that partitions input data and model parameters into tractable blocks or tiles that can be stored in the on-chip memory. The architecture processes a single tile at a time, minimizing expensive off-chip memory accesses, enabling more efficient reuse of data. It also facilitates the achievement of high data throughput while working under resource-constrained environments.

   Double buffering is another technique used to overlap memory transfer with computation. One buffer is being used in computation, while the other one is being filled with the forthcoming block of data. This method of usage aids in concealing memory latency and maintaining the computational pipeline full, thereby improving overall efficiency.

   On-chip caching also lowers the delays due to memory access. Frequently accessed data, such as intermediate feature maps or filter weights, can be store temporarily in high-speed on-chip memory blocks in order to prevent unnecessary data movement. The caching technique is able to function efficiently in convolutional layers where the same filter is convolved with different spatial locations of input.

   Ma et al. (2017) [12] showed that the combination of these approaches in a carefully designed memory access pattern facilitates beneficial enhancements in energy efficiency and performance. It highlights the importance of the optimized memory hierarchy design as per the computational characteristics of the NN model to ensure maximum utilization of the FPGA.

4. **Reconfigurable Architectures**
   The reprogrammability of FPGAs is hugely advantageous in the deployment of neural networks, especially where flexibility and agility are crucial. One of the most robust features in this area is dynamic or partial reconfiguration, where specific

segments of the FPGA fabric are reprogrammed and continue in operation while the remainder of the system keeps running uninterrupted. This enables updates to the components on a per-component basis, e.g., neural network layers or processing pipelines, without requiring a complete system halt or re-synthesis.

Such dynamic reconfiguration is particularly valuable in applications with changing workloads or being able to implement multiple models of neural networks on a common hardware platform. For instance, in edge devices or embedded systems for application in automotive or surveillance scenarios, the ability to rapidly switch between models for object detection, facial recognition, or gesture interpretation can substantially enhance system value and responsiveness.

Moreover, this flexibility leads to improved hardware utilization. Instead of statically allocating resources for the worst-case, designers can allocate a smaller, low-power core and dynamically reconfigure it on-the-fly as a function of the arrived task. This dynamic approach leads to a better organization of the limited logic, DSP block, and memory resources of the FPGA.

Liu et al. (2018) [13] demonstrated how convolutional neural networks can be configured with dynamically reconfigurable architectures for the sharing and adapting hardware across layers with different computational needs. It was determined from the results that it reduces both logic overhead and energy consumption, which makes it suitable for energy-constrained applications where flexibility cannot be afforded by compromising efficiency.

5. **Efficient Dataflow Architectures**
   Custom dataflow structures are the most important enablers of high-speed neural network inference on FPGAs. They are designed with precision to optimize the movement of data among processing elements, memory, and interconnects for removing idle cycles and reducing the off-chip memory accesses that are costly. A classic example is the use of systolic arrays, an extensively adopted architecture style in hardware acceleration.

   Systolic arrays are a one-dimensional array of processing elements (PEs) that shuffle data rhythmically across the array, simulating the contraction and relaxation of a heart. This paradigm is best suited for the computation of dense linear algebra operations such as matrix multiplications that are the backbone of the computation of fully connected and convolutional layers of neural networks. By keeping data flowing in a pipelined manner, systolic arrays minimize memory bottlenecks and maximize throughput.

   On FPGAs, systolic arrays may be configured to the target neural network model's bit-width and precision, allowing for highly optimized usage of resources. Their regular structure also makes efficient deployment on reconfigurable logic possible, allowing high parallelism and deterministic execution.

Jouppi et al. (2017) [14] illustrated the capability of systolic array-based architectures in their research on Google's TPU, where they are used to accelerate deep learning operations. The concepts used in the TPU can, of course, be extended to FPGAs as well. The research demonstrated how the approach significantly enhances performance and efficiency over general-purpose designs and is therefore a valuable approach to FPGA-based acceleration of neural networks.

6. **Hybrid Computing Strategies**
Besides offering improved performance and flexibility, modern computing systems are increasingly resorting to heterogeneous designs, which integrate FPGAs along with CPUs and GPUs. By so doing, system architects can leverage the unique capabilities of each processing engine to address the disparate computational requirements of neural network tasks. In such applications, FPGAs tend to be tasked with executing deterministic, latency-bound tasks due to the fact that they can exhibit fine-grained control and pipeline well. Their reconfigurable nature also enables customized hardware design to be optimized for a specific task, which has strong application in time-critical or resource-constrained applications.

   GPUs, on the other hand, are inherently capable of doing highly parallelizable, floating-point heavy computation. They are most effective in the training of neural networks and large-batch inference owing to their enormous amount of parallel processing cores and general-purpose programming model support such as CUDA and OpenCL. CPUs serve to supplement the system, handling control logic, irregular computation, and orchestration tasks that include deep decision-making or branching logic.

   Nurvitadhi et al. (2017) [15] presented a comprehensive report on this heterogeneous approach, demonstrating how hybrid systems of FPGAs together with GPUs and CPUs can deliver significant performance per watt and end-to-end throughput improvements. Outcomes demonstrated how through proper workloads partitioning, best energy efficiency, computational performance, and system flexibility trade-offs can be achieved. These architectures are increasingly being employed in both edge computing systems and data centers, where power efficiency and flexibility are central. As complexity and depth in neural networks grow, heterogeneous architectures become an even more central phenomenon, motivating the development of efficient hardware-software co-design methodologies.

2.1.5 Power Efficiency and Resource Utilization

Power efficiency is a critical design constraint for NN accelerators on FPGAs, particularly for edge computing, Internet of Things (IoT) devices, and battery-powered systems. These have limited energy resources and, as such, need hardware-level as well as architectural optimizations in order not to compromise on computational performance while saving power. Necessary strategies are:

1. **Dynamic Power Management**
   Dynamic power reduction techniques are widely present in FPGA implementations with a focus on saving power, especially in those systems with a need for continuous, extended computation under limited energy conditions, i.e., in IoT edge devices powered by batteries and edge AI systems. One such extremely efficient, yet extremely well-liked method is clock gating [16], where the clock signal is turned off for parts of the computation not involved at a given moment. This prevents the unnecessary registers as well as combinational logic from switching, hence conserving dynamic power waste, including an enormous amount of total energy power in high-frequency digital systems. Fine-grained clock gating has very accurate control over an extremely huge number of functional units in an FPGA, with very specific, focused control over idle resources being possible without any impact on the remainder of the system.

   In conjunction with clock gating, dynamic voltage and frequency scaling (DVFS)[17] is yet another extremely useful power optimization method. DVFS dynamically, in real time, varies the supply voltage as well as operating frequency in accordance with workload patterns. As a specific example, voltage as well as clock frequency can be reduced in low computational intensity in an effort to conserve power but boosted in high performance requirements. DVFS is well suited for fluctuating performance demands, such as sensor-driven AI or bursty inferencing workloads, thanks to its adaptability.

   Clock gating and dynamic voltage scaling become a foundation for a combined, high-granularity approach to power saving. Merged, next-generation FPGA devices should, in an architectural way, provide these techniques, allowing real-time logic dynamically implement power-saving capabilities while it observes usage patterns. In a number of realizations, even feedback-based power governors or policy managers using machine learning are included. Together, these techniques provide enormous potential for power saving without affecting functional correctness or workload requirements for real-time systems.

2. **Energy-Efficient Dataflow Architectures**

   Dataflow optimization for FPGA accelerators is one of the most important aspects in power as well as system efficiency minimization. Since memory access as well as data movement are some of the most energy-intensive operations in deep learning, data-handling mechanisms need to be efficient in order to allow sustainable FPGA implementation. Low-energy dataflow implementations should take utmost precaution not to cause unnecessary data movement and ensure utmost on-chip data reuse such that off-chip memory accesses for power-hungry memory resources are eliminated.

One of the most efficient methods, tiling, partitions huge computations into little tiles with boundaries inside an FPGA's limited on-chip memory. Thus, the method keeps hot-accessed data on-chip across several computations, significantly lowering memory transaction time and energy cost. Likewise, data reuse methods, such as caching weights or feature maps into on-chip local memories, take advantage of neural network computation temporal locality in order to lower memory bandwidth demands. Scheduling methods also enhance energy efficiency by controlling task execution order not to incur data stalls and balance hardware utilization. As an example, computation might be scheduled ahead of data transmission, hiding memory latency and eliminating idle processing units.

Second, specific processing elements (PEs) for specific neural network operations such as matrix-vector multiplication or convolution would be implemented in a pipelined fashion to ensure constant data flow. Pipelined interconnects not only eliminate idle cycles but also address overheads of a general-purpose architecture for low dynamic power usage and performance optimization.

These architectural as well as algorithmic innovations together yield a highly optimized hardware architecture with vast parallelism as well as controllability under limited power budgets—the key facilitator for deep learning model executions in edge devices as well as real-time embedded systems [18].

3. **Trade-offs Between Performance and Power Consumption**
A balance between performance and power is a top-level concern for power-aware FPGA-based neural network accelerators. While increasing the number of processing elements or data word widths—raising parallelism—and frequencies would significantly boost performance throughputs and reduce latency, these approaches typically have a cost in higher dynamic power dissipation as a result of higher switching activity as well as higher heat dissipation. Contrasting with these are extremely power-saving methods, i.e., scaling down voltage levels, frequency, or using conservative scheduling, lowering power dissipation at the cost of lower processing rate, reduced resource utilization, as well as compromised real-time capability.

To meet these competing demands in design, researchers and hardware designers typically rely on Design Space Exploration (DSE). DSE involves intensively benchmarking various architecture configurations as well as implementation parameters in order to decide on the best available trade-offs for specified application requirements. These configurations can include levels of parallelism, formats for precision (e.g., INT8 and FLOAT32), pipeline depth, and memory hierarchy.

Of specific interest in this conversation is whether parallelism is coarse-grained or fine-grained. Fine-grained parallelism gives detailed control but with potential overheads in control, whereas coarse-grained parallelism improves throughput with possibly reduced flexibility and power requirement. Numerical precisions similarly have direct influences on power, as well as on accuracies. Lower-precision numerics, i.e., fixed-point or binarized data, reduce computational cost as well as memory access cost but at the cost of needing calibration so accuracy is ensured.
Finally, memory hierarchies should be structured with great diligence in order to balance power with performance. More on-chip utilization of small buffers can significantly reduce energy spent on accesses to external memory, while locality and reuse are supported by tiling, as well as data caching.

In general, efficient use of power in FPGA NN implementation is a matter of seeking out and leveraging the sweet spot between computational performance, power, and model accuracy—a necessarily application-dependent compromise only resolvable through attentive empirical observation with iterative tuning [19].

2.1.6 Toolchain Interoperability and Framework Support

The other significant but often overlooked factor for deploying neural networks on FPGAs is interoperability between FPGA toolchain and deep learning frameworks. Inasmuch as FPGAs offer huge energy efficiencies as well as performance, they are often hindered from entering mainstream machine learning practice due to insufficient deep integration with mainstream AI frameworks such as TensorFlow, PyTorch, and ONNX.

More recent research has built on this with a push towards achieving interoperability through intermediate representations as well as standardized compilation pipelines. Umuroglu et al. (2017) [22], for instance, introduced FINN, where it is made possible to run binarized and quantized neural networks (BNNs) on FPGAs through compilation from high-level network descriptions into extremely optimized hardware descriptions. FINN demonstrates how it is possible to automatically model-to-hardware translation using ONNX as an intermediary format with application of domain-specific hardware templates for synthesis. Along with this, frameworks like hls4ml as well as Xilinx Vitis AI are also being created, which are streamlining the deployment of deep learning models on hardware. These frameworks permit users to convert pre-trained models into HLS-compatible code or directly execute them on programmable logic using pre-optimized IP cores. These frameworks lower the entry cost for new users since they abstract out general neural network layers as well as patterns in quantization.

However, there remain a number of concerns. One major bottleneck is incomplete operator coverage as well as non-standard or customer-defined layer support within toolchains. In addition, symbolic methods of quantization—while beneficial for hardware efficiency—create numerical discrepancies between hardware execution and software simulation. These mismatches can decrease model accuracy or make validation as well as debugging procedures more difficult.

To address these limitations, future research has to concentrate on developing end-to-end automated, verified model-to-hardware pipelines with strong fallbacks for unsupported layers. Open benchmark development and verification suites for guaranteeing consistent behavior across training, inference simulation, as well as synthesized hardware are also necessary. Further integration of machine learning environments with FPGA toolflows is achievable, promoting broader adoption as well as innovation in low-power, real-time AI.

2.1.7 Binarized and Low-Bitwidth Neural Networks on FPGAs

Another approach in FPGA-based neural network acceleration is using binarized and low-bitwidth neural networks. These networks limit weights and/or activations to a small set of discrete values, e.g., $\{-1, +1\}$ for Binary Neural Networks (BNNs), which significantly reduce memory footprint as well as computational complexity. Reconfigurability as well as bit-level processing, inherent in FPGAs, make them an apt candidate for hosting these networks since they provide custom tailored circuits for ultra-low precision computation.

One of the leading frameworks in this field is FINN, presented by Umuroglu et al. (2017), [22], which allows quick and scalable inference of binarized neural networks on FPGAs. FINN employs a dataflow model in which layers are represented as streaming operators, and memory accesses are optimized using on-chip buffers. The framework demonstrated how, with binarized weights, as well as activations, the resource usage is drastically reduced, yet high-throughput, low-latency inference is still achieved using mid-range FPGAs.

Subsequent development has made FINN applicable for QNNs with greater than 1-bit precision (e.g., 2–8 bit) for more detailed dictionaries, as well as accuracy at no compromise on hardware optimization. These developments are also characteristic of hybrid precision methods, where significant layers are marginally more accurate than binary, but sensitive ones remain binary, making a sensible model accuracy versus hardware efficiency compromise.

2.1.8 Security and Reliability Considerations

Security and trustworthiness take top priority in FPGA-based neural network implementations, especially in safety-critical areas such as those for autonomous systems, industrial automation, and medical devices. The reprogrammability and hardware-transparency provided by FPGAs introduce new attack surfaces with rather distinct characteristics compared with traditional CPU- and GPU-based architectures.

Shayan Moini et al. (2020) [23] demonstrated that accelerators for CNNs on FPGAs are susceptible to remote power side-channel attacks. They presented how attackers would be able to deduce input information as well as reconstruct some weights from CNNs using power consumption fluctuations. That calls for establishing secure design practice as well as on-chip countermeasures such as power balancing, or power isolation, or random masking for information leakage mitigation.

Furthermore, Likun Kang et al. (2019) [24] explored how neural network hardware Trojans can be mounted on FPGA platforms. These Trojans have the potential to be stealthily embedded within bitstreams or IP blocks, leading to malicious actions such as misclassification or inference manipulation. Formal verification, bitstream authentication, and logic locking were presented as countermeasures against these threats.

In respects to reliability, Pierluigi Civera et al. (2002) [25] presented an FPGA-based framework for speeding up fault injection campaigns on safety-critical circuits. They propose a method for fast simulation of soft errors and SEUs, permitting researchers to quantify different network architectures' resilience under faults. This is valuable research that presents a benchmarking tool for resilience on FPGA-based deep learning.

Ultimately, Jiale Zhang et al. (2019) [26] resolved FPGA-based inference privacy using a privacy-preserving framework with cryptographic protocols and specialized FPGA accelerators. Its method is supportive of inference on encryption data with decent performance, demonstrating FPGAs practical in confidential environments.

These two bodies of work, side by side, highlight how, while FPGAs provide end-to-end performance and flexibility for AI acceleration, they also demand a security- and reliability-aware co-design methodology. Security verification-enabling standardization of design flows, tamper-proof bitstream loading, and runtime monitoring for trusted neural network execution on FPGAs are some areas that need to be researched going forward.

2.1.9 Challenges and Future Directions

Nevertheless beneficial FPGA implementations of neural networks certainly are, there are a number of issues they present that must be given serious consideration. Among those problems is the complicated nature of programming FPGAs even with High-Level Synthesis tools. One would require enormous amounts of hardware structure knowledge in order to efficiently implement neural network computation on FPGA hardware as well as attain optimum performance. One constraint is memory on-chip limitations on most FPGAs. Deep networks, in particular, require a lot of storage for weights and activations. Low latency and fast memory handling is a current research area. Various memory optimization techniques, including compression, memory reuse techniques, as well as interfacing memory with external memory, have been proposed as solutions for bypassing these limitations. Power consumption is also a concern, especially for edge computing and real-time systems. While FPGAs possess a more energy-efficient nature than GPUs in some uses, very deep neural networks are still power-hungry. Power scaling, clock gating, as well as architecture modifications, are being researched as a way towards increasing energy efficiency.

Scalability is a serious concern for FPGA-based neural network architectures as well. While small- to medium-sized neural nets can be implemented on one FPGA, larger ones typically require multi-FPGA architectures or hybrid architectures with other accelerators such as GPUs and CPUs. Integration of these seamlessly without loss of performance remains a research concern.

Hardware-software codesign is also extremely crucial for usability, next only to performance, for FPGA-based NN programs. Popular deep learning frameworks such as TensorFlow and PyTorch don't provide significant native FPGA acceleration. Limited standardization, coupled with a need for custom solutions, is a hindrance for wide-scale acceptance. Further openness of integration with these frameworks in FPGA toolcrafts would be extremely beneficial not just for developers but even for researchers.

More recent FPGA innovations, such as adaptive and reconfigurable architectures dynamically, offer promise as a resolution for these problems. These would permit FPGA hardware to reconfigure dynamically with a change in computational workload, leading to enhanced overall performance as well as efficiency. Work in this area is still in its nascent stages, and more research effort has to be focused on bridging ease of software development with hardware-level optimality.

2.2 Technical background

2.2.1 Architecture of the Float-Based TCN Model (FloatTCN)

Overview

The adopted model in the current thesis is FloatTCNModel, which is a Temporal Convolutional Network. It is a sequence model used in the library PyTorch. It uses the use of 1D convolutional layers with the technique of dilation to model long-term temporal dependencies with efficient computations.

Architecture Description

FloatTCN Model is comprised of a sequence of consecutive blocks of convolutions in which consecutive blocks modify the temporal shape of the input sequence to capture progressively wider context information. Each such block is composed of carefully designed layers in order to promote representational capability as well as regularization. Each of the convolutional blocks contains the following key elements:

- A one-dimensional convolutional layer (nn.Conv1d): It is used to apply temporal filtering to the input. By using the exponentially large receptive field with the growing rates in the superposed blocks, the model is capable of learning long-range dependencies without going deep in the layers or using large filter sizes. Dilation merely "spaces the elements" in the kernel apart, such that every filter gets to look further in the sequence.

- A layer normalization unit (nn.LayerNorm): We employ this technique of normalization after convolution, across the feature dimension in every step to enhance training stability as well as faster convergence. It is particularly apt in sequence models where the batches can differ in size, in contrast to batch normalization.
- Rectified linear unit activation function (ReLU): After normalization, the non-linearity enables the model to capture complex temporal patterns as well as the inter-action among the sequence features.
- A dropout layer (nn.Dropout): Included conditionally with the given dropout probability, the regularization strategy sets some fraction of the activations to zero at random during training to impose generalization in the model as well as to avoid overfitting..

Prior to passing the input to the convolutional stack, the model reshapes the input tensor from shape $[B, T, C]$ to $[B, C, T]$, where the batch size is denoted as B, the sequence length as T, and the input channels as C. This reshapes the data to adhere to the input convention in PyTorch for use with `Conv1d`, which expects the channel dimension to be in the second position.

It then passes through the stack of dilated blocks with enlarged convolutions and employs a pointwise convolutional layer with the kernel size being 1 to perform the channel-wise transformation. It mixes features in the feature dimension without modifying the dimension in time as the final step in the projection. The additional layer with normalization followed by ReLU activation is included to further enhance the representation.

Then the model uses the middle step in the sequence output representation, which is calculated using the sequence length floor division

$$\text{middle\_index} = \left\lfloor \frac{T}{2} \right\rfloor$$

This uses the middle section of the sequence to have the richest temporal context from the past and future due to the symmetric padding in the previous layers. The output is given as a tensor with shape $[B, C]$, one latent feature vector per sequence input.

Dilation and Receptive Field

One of the important advantages of the `FloatTCNModel` is the use in its convolutional blocks of adaptive dilations. Dilation is one technique used to put gaps in-between kernel elements, allowing the network to expand the size of the receptive field in an exponential manner with the depth, without additional parameters or increasing filter size. The effective receptive field in the case where the kernel is of size k and the dilation is d is given mathematically as $k + (k - 1)(d - 1)$. The model can capture long-term dependencies in the sequence input using few layers with the dilation rates increasing in subsequent layers (for instance, 1, 2, 4, 8, . . .).

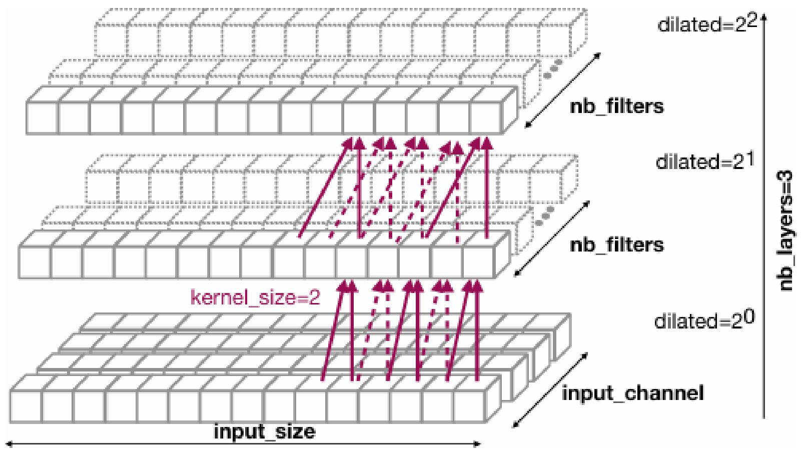

Figure 2.2. Dilated Causal Convolutional Blocks in a Temporal Convolutional Network (TCN) with Exponentially Increasing Dilation [28]

In order to preserve the input and output sequence length consistency, the model employs a symmetric padding strategy. That is, the input is padded on either side in every convolutional layer such that the output tensor has the same temporal dimension as the input tensor. It is convenient to use with applications where temporal resolution must be preserved (e.g., sequence-to-sequence modelling or classification per step). It is also easier to interpret the output because no part is cropped or lost in the forward pass.

By combining dilation with symmetric padding, the model is entirely convolutional. That is, it does not depend on the input sequence length in an absolute way and can therefore accommodate sequences of any length without modifying the architecture in any way. The architecture is also parallelizable over time steps, which can accelerate both training as well as inference in comparison with recurrent models, yet still capture global as well as local temporal dependencies.

Output Representation

We use the final output from the `FloatTCNModel` as the activation vector at the middle time index in the sequence after processing. Its mathematical form is

$$\text{middle\_index} = \left\lfloor \frac{T}{2} \right\rfloor$$

where the input sequence length is given as T. The model takes the given index from the feature representation in the last activation tensor, which has passed through the series of dilated convolutions, normalization, and activation.

This is motivated from the fact that the model can use symmetric padding and bidirectional receptive fields, thanks to the non-causal, dilated convolutional model. Each output timestep has influence from the future as well as the past, with the central timestep particularly able to leverage the most symmetric context. It is specifically equidistant from the sequence boundaries, and can hence draw from the most symmetric temporal aggregate.

This is especially suitable for applications where one is interested in producing one output per input sequence, such as sequence-level classification or regression. With the middle-level representation, the model is immune to the issue with boundaries, and the output is derived from the temporally rich and contextually full latent representation.

In addition, the output mode supplies the same interface to be used with any downstream decision or dense layers, facilitating simpler downstream processing. It also obviates the need for global pooling or recurrent summarization layers, preserving the full convolutional nature of the model while conserving computational efficiency.

2.2.2 Architecture of the ResidualBlock-Based TCN Model and Comparison

Overview

One other model is a more powerful version of the Temporal Convolutional Network (TCN) based on the residual learning principles. The normal TCN employs ResidualBlocks in it to enable the training of deeper networks with the aid of identity skip connections, normalization layers that are optional, and causal padding that is adjustable during training. The more powerful and flexible model, especially with very long sequential inputs, is the stronger variant TCN.

Architecture Description

In the second TCN model employed in the thesis, architecturally the model is structured as a stack of modular `ResidualBlock` layers to achieve improved expressiveness in learning along with learning stability. Every `ResidualBlock` is a composite building block carrying out the task of feature transformation with residual connections for retaining information. The blocks form the TCN backbone and are stacked with progressively growing dilation rates to allow the model to capture long-term temporal dependencies in lengthy sequences Each `ResidualBlock` includes the following components:

- **Two dilated 1D convolutional layers (`nn.Conv1d`):**They are mostly applied for temporal feature extraction. Dilation makes the kernel skip input positions, which has the effect of increasing the receptive field of the layer without kernel size or depth increase. This is important in order to model long-range relations in time-series data.
- **Zero-padding using `F.pad`:** Instead of employing the built-in padding argument in PyTorch convolutional layers, padding is performed manually with `F.pad`. This

yields explicit control over causal or symmetric ("same") padding, based on the programmed setting. Causal padding ensures that each output timestep will be caused by only present and past inputs, a requirement for real-time prediction problems.

- **Normalization:** There are different normalization schemes in the model. Batch normalization (`BatchNorm1d`)normalizes along the batch dimension and is appropriate for most common use cases. Layer normalization (`LayerNorm`) normalizes along the feature dimension of each time step and is thus more appropriate for variable length sequences. If none of these is given, normalization is entirely omitted.
- **Dropout:** Used after every convolution, dropout adds regularization by randomly dropping out a portion of the units during training. It prevents overfitting and makes the model learn more generalizable temporal patterns.
- **Activation function:** The nonlinearity applied after each normalization and dropout stage is configurable, with `ReLU` being the default. This flexibility allows experimentation with other activations such as `tanh`, `leaky_relu`, or `gelu` depending on the nature of the task and dataset.
- **Residual (skip) connection:** To make gradient propagation easier and to allow for deeper networks, each block adds its output to a shortcut connection from the input. A 1x1 convolution is used in order to have matching input and output channel dimensions if they differ. The residual mechanism makes it easier to learn identity mappings and speeds up convergence.

Beyond the internal structure of one block, the macro-architecture of the model consists of multiple `ResidualBlocks` stacked with exponentially growing dilation rates (e.g., 1, 2, 4, 8, .). This approach causes the network to have an exponentially growing receptive field in terms of depth, enabling it to effectively model both short- and long-range dependencies at no corresponding cost in parameters or computation.

Besides, the model includes skip connections among blocks that are optional, where the intermediate results from each residual layer are concatenated and passed into the last output layer. This facilitates deep supervision as well as enhances signal passage throughout the network, particularly for extremely deep TCNs.

Padding Strategy and Temporal Alignment

One of the most important design choices in temporal convolutional models is the padding scheme, which has direct consequences on the temporal alignment of features and the directionality of the receptive field.The TCN architecture presented in this chapter implements two kinds of padding schemes, each one tuned for various application requirements: *causal padding* and *same (symmetric) padding.*Padding impacts not only the information flow through the network but also the applicability of the model to certain applications like forecasting or sequence classification.

- **Causal Padding:**
  This guarantees the convolutional output at time step t is computed based only on the current and past time steps, i.e., $[0, t]$, and not on any future information. This is important when doing real-time or forecasting applications—e.g., time-series prediction, streaming signal processing, and anomaly detection—where the model should not "peek" into the future inputs. By maintaining this strict temporal causality, the model is safe to apply in situations where future data does not exist at inference time.

- **Same Padding:**
  This padding method pads equally on either side of the input sequence, with the effect of centering the convolutional kernel around each timestep. Thus, the output at time t is determined by a bidirectional context, i.e., future, present, and past inputs. This is viable for sequence classification or modeling where the whole input sequence is known beforehand, for instance, offline signal processing, action recognition, or sentiment analysis. Surprisingly, same padding also allows the original sequence length to be retained and makes input and output sequence alignment easier.

In order to have exact control over the effective receptive field and output alignment, padding is not performed with the native `padding` argument of `nn.Conv1d`. In implementation, padding is performed manually using PyTorch's `F.pad()` method. This allows for conditioning on using left-only (causal) or symmetric padding preceding every convolution operation. This also prevents version-number ambiguity within PyTorch for automatic padding behavior and enables deterministic, reproducible input shifting control when dilating.

By accommodating both padding schemes, the model is made extremely flexible as it can be applied to a wide range of temporal tasks with high accuracy in feature alignment and causal structure. The dual-mode flexibility is particularly useful in experimental settings, where the same base model can be re-used for either causal prediction or sequence-level classification without modifying the architecture.

Skip Connections and Deep Supervision

The Temporal Convolutional Network (TCN) model utilized in this project has optional support for skip connections between residual blocks, a design decision that greatly benefits the representational power of the network and the training process. Skip connections, when activated, operate by gathering the intermediate residual outputs of each `ResidualBlock` in the network and adding them before the last output projection layer. Specifically, following the computation of each residual block's transformed output and the addition of the residual (shortcut) connection, the resulting tensor—is often referred to as the residual output—is inserted into a list of intermediate representations. Following the forward pass through all the residual blocks, they are added element-wise, giving a compound feature map that aggregates information from all hierarchical levels of the network.

This skip connection strategy offers several important benefits:

- **Improved Gradient Flow:** Skip connections circumvent the vanishing gradient issue ubiquitous in deep networks by providing shortcut paths from early layers to the output. Gradients now find it easier to flow through these additive shortcuts during backpropagation and thereby speeding up convergence as well as learning meaningful weights in early layers
- **Multi-Scale Feature Learning:** Every residual block corresponds to a different receptive field size because of the incremental dilation strategy. By adding up the output of every block, the network is in effect doing multi-scale temporal aggregation, which enables the network to capture local short-term features (from shallow layers with small receptive fields) and global long-range dependencies (from deep layers with large receptive fields). This hierarchical structure is particularly advantageous in problem-solving where substantial patterns occur at a range of temporal scales.
- **Robustness and Redundancy:** The combination of intermediate outputs provides a type of redundancy that introduces a level of robustness into the network to noisy or unusual input sequences. A salient feature that is not captured by a block can potentially still be retained by other blocks working at different temporal resolutions.
- **Feature Reuse:** By blending outputs of every residual step, the model does not throw away possibly useful intermediate features. This promotes feature reuse, a concept widely used in modern deep architectures such as ResNets and DenseNets, where mid-level activations are good signals for later steps.

The last merged tensor, which is created by adding up all residual outputs, is fed through a fully connected output layer(`nn.Linear`) that transforms the merged temporal features into the desired output space. Regardless of whether the task demands an individual output vector or an entire output sequence, this skip connection strategy guarantees that the model generates predictions from a rich and varied set of temporal abstractions.

Output Layer

After the convolutional backbone, Temporal Convolutional Network (TCN) gives flexible output settings to allow experimentation with a broad variety of downstream tasks, which is supported by conditional control of the behavior of the model's final layer using two parameters:`return_sequences` and `regression`.

I. Output Modes

- **Full Sequence Output (`return_sequences=True`):**
  In this mode, the model produces a prediction for every timestep in the input sequence. The output tensor has the shape $[B, T, C]$, where B is the batch size, T is the sequence length, and C is the number of output features. This configuration

is ideal for tasks such as sequence labeling, multi-step forecasting, or temporal segmentation, where the model is expected to make localized predictions at each time point. For example, in sensor signal analysis, each timestep might be labeled as belonging to a specific activity or event class.

- **Last Timestep Output (`return_sequences=False`):**
  In this mode, the model takes the feature vector of the last timestep of the sequence and gives it as an individual output for each input sample. The output shape is $[B, C]$. This is particularly helpful for sequence-level classification or regression problems, in which one needs a general sense of the full sequence in order to make one prediction (e.g., classifying overall activity type, forecasting total energy usage, or summarizing sentiment).

II. Output Projection Layer

Irrespective of the chosen output mode, temporal features are fed into a fully connected (linear) layer (`nn.Linear`). The layer transforms hidden feature representations into target output size and is the last transformation step prior to prediction. Its input size equals that of the number of filters in the last convolutional layer, and its output size equals that of the number of target classes (in classification) or scalar quantities (in regression).

III. Task-Specific Activation

To support both classification and regression tasks within a unified framework, the model optionally applies a log-softmax activation to the final output. This behavior is governed by the `regression` flag:

- **When `regression=False`:**
  The model assumes a classification setting and applies `F.log_softmax` over the output features, transforming the raw logits into log-probabilities. This is appropriate for use with loss functions such as negative log-likelihood loss (`nn.NLLLoss`), commonly used in multi-class classification problems.

- **When `regression=True`:**
  The model skips the log-softmax activation, and the output is raw linear. This is appropriate for regression problems, when we are interested in predicting continuous-valued targets, and loss functions like mean squared error (MSE) or mean absolute error (MAE) are employed.

IV. Design Implications

This adaptive output learning interface allows the same model architecture to be applied across a range of temporal learning tasks, merely by adjusting a few configuration parameters. It also allows for multi-task experimentation, whereby multiple output heads can be used for various targets, depending on whether full sequence predictions or otherwise aggregated outputs are desired.

Model Selection

Following thorough analysis of both architectures, the second model—the Temporal Convolutional Network (TCN) based on ResidualBlocks—was chosen to be utilized in the final system.

This was selected due to the heightened architectural flexibility, improved stability via residual learning, and support for essential features like causal padding, skip connections, and programmable normalization schemes.
These properties render it superior for modeling multi-scale temporal structures and achieving robustness for a wide range of sequence modeling tasks.

In addition, its support for variable output form and both classification and regression targets gives the flexibility needed for the experimental work of this thesis.

Hence, all the following experiments, tests, and deployment activities rely on this residual TCN framework.

Chapter 3

Methodology

This chapter describes the end-to-end process to transform a high-level Temporal Convolutional Network (TCN) model to one where it is hardware-friendly to implement in FPGAs. We aimed to co-optimize the model to perform real-time inference with the least computational latency and resource consumption.

The process began with the quantization of the TCN to reduced-precision integer to minimize memory and computational complexity (Section 3.1). The baseline TensorFlow build then had to be translated to PyTorch to have better control over the quantization flows in order to simplify further translation to C++ (Section 3.2).

The quantized PyTorch model had to then be implemented in fixed-point C++ with custom logic hand-optimized to the constraints in the High-Level Synthesis (HLS) to ensure the model structure as well as numerical correctness (Section 3.3). The C++ codebase then underwent co-optimization with high-level synthesis (HLS) using pragma directives as well as structural modifications to optimize the C++ code to achieve minimum latency, improved parallelism as well as balanced HW resource consumption (Section 3.4).

The optimized full design then had to be implemented in the entire end-to-end FPGA system in Vivado block-level designing tools with AXI interfaces as well as system-level IPs to allow real-time inference in embedded devices (Section 3.5). All these steps combined constitute an end-to-end software-to-hardware workflow optimized to implement deep learning models in real-time resource-constrained embedded applications.

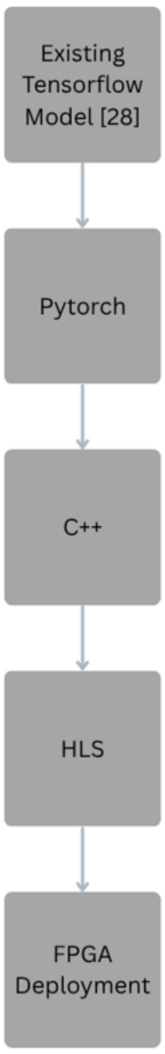

Figure 3.1. Flow Diagram of The Work

3.1 Conversion from TensorFlow to PyTorch with C++ Deployment

This section outlines the procedure undertaken to quantize and optimize the deployment-ready version of the Temporal Convolutional Network (TCN) onto hardware-restricted devices. The goal with the procedure is to reduce the model's computational and memory footprint without impacting the predictability of the model. Two methods were explored to perform the quantization: Quantization-Aware Training (QAT) with the library Brevitas, and Post-Training Quantization (PTQ) with ONNX Runtime. Although the two methods were explored, only PTQ underwent to the final deployment stage due to restrictions in the target FPGA toolchain. The motivation with the choice is explained in the subsections.

Beyond the C++ deployment for fixed-point inference and hardware, the framework had to be ported to PyTorch initially. This is the reason it had to be done in the first place, as two models—one baseline float32-based TCN, and one further flexible ResidualBlock-based TCN—had to be ported and experimented with in a platform with higher transparency, control over quantization, as well as C++ exportability.

The subsequent subsections first describe the motivation as well as the translation procedure from the TensorFlow to the PyTorch models, followed by the procedure followed to perform the quantization, and then the procedure to transform the resulting models to the deployable C++ form appropriate to accept HLS-based synthesis to the targeted FPGA hardware.

3.1.1 Motivation for Converting the TCN Model from TensorFlow to PyTorch

The primary motivation to migrate to PyTorch was the need to perform manual fixed-point quantization (for example, Q4.8) as well as structured weight export to hand-coded C++ inference. TensorFlow is capable of deployment formats such as TensorFlow Lite (TFLite) as well as ONNX, but the formats essentially target usage with opaque run-time interpreters or edge deployment to ARM hardware. The formats, while optimized for mobile inference efficiency, encapsulate much of the inner mechanisms, and it is not easy to achieve fine-grained control over numerical computations as well as weight representations required in fixed-point implementations.

Particularly when compiling to embedded targets such as STM32 microcontrollers or high-level synthesis (HLS) flows to run in FPGAs, programmers usually require full visibility and determinism in the execution of arithmetic operations, particularly scaling, clipping, and saturation. These go beyond the abstraction levels provided by TensorFlow automatic quantization and export tools. Also, TensorFlow's quantized model representations (e.g., .tflite files) package weights as binary blobs, which can't be directly read or exported to standard C++ data structures, and hence can't readily be included into custom testbenches or hardware accelerators.

In contrast, PyTorch exposes underlying model parameters in a straightforward manner and allows fine-grained scripting both during export and quantization. This is especially important when deploying models to environments in which memory, bitwidth, and latency constraints demand some level of optimization and visibility that is not achievable within TensorFlow's deployment pipeline.

3.1.2 Advantages of Using PyTorch as an Intermediate Step Before C++

PyTorch offers a variety of useful features to support this mid-level functionality. With the "state_dict" giving straightforward access to the model parameters, PyTorch facilitates transferring the weights directly into NumPy arrays and to writing them to human-readable C++ header files. Such structured access is necessary while implementing fixed-point representations as well as corresponding data formats across platforms.

Second, PyTorch has native support for post-training quantization (PTQ) as well as quantization-aware training (QAT), alongside external tools like Brevitas. These offer high-fine-grained control over the process of quantization, i.e., bit width, scaling factors, as well as options for per-channel or per-layer quantization. This is required to optimize the model to the particular constraints of specific hardware targets, e.g., aligning accumulator ranges as well as saturation characteristics in the C++ fixed-point engine.

Third, native interoperability with NumPy, as well as eager execution mode, simplify debugging. Developers can now easily examine intermediate tensors, observe the activation levels for every layer, and directly look at outputs in comparison to equivalent C++ calculations. This greatly increases traceability and verifiability, especially in case of overflow, round-off, or data format mismatch.

Lastly, the entire weight exportation and conversion process can be automated using short Python scripts. The scripts accept weights from PyTorch models, perform the quantization, reshape the array, and produce C-style header files which can be included in embedded or in in HLS C++ projects. The automatic process eliminates the possibility of human error, allows reproducibility, and allows iterative design cycles as well as debugging cycles

3.1.3 Challenges in Direct TensorFlow-to-C++ Translation

Converting directly to C++ from TensorFlow proved even tougher. The standard TensorFlow export formats, such as .pb, .tflite, or .onnx, are primarily designed for runtime inference engines and do not lend themselves to straightforward manual inspection or special customization at the lower levels.

Such formats tend to encapsulate weights and metadata in binary forms difficult to read or specialize to specific embedded requirements. Developers hence enjoy limited flexibility in automated deployment tools with little room to optimize numeric precision or add custom fixed-point semantics. TensorFlow likewise lacks robust intrinsic fixed-point arithmetic simulation capabilities at the operator-level.

It is not possible to directly support simulating clipping, saturation, or fixed-point arithmetic rounding in the context of TensorFlow graphs. That is serious in the case where exact, reproducible arithmetic pipelined models must be used as is normally required in the case of hardware synthesis or microcontroller applications. Moreover, the quantization operations offered in TensorFlow Lite et al. are intended to be used in deployment to embedded or mobile runtimes, as opposed to manual control or debugging. They obfuscate the middle representations, making it difficult to see where the Python and C++ implementations diverge.

In practice, developers have to either accept the opaque model representations or spend large amounts of effort reverse-engineering the exported models, which neither allows for agile, transparent development. Generally, the lack of flexibility, interoperability, and visibility in the hand-coded fixed-point flows renders the TensorFlow-to-C++ route unsuitable to the needs of the work.

3.2 Quantization Techniques for Efficient TCN Deployment

This chapter describes the approach taken to quantize a Temporal Convolutional Network (TCN) to deploy on low-power hardware in embedded and FPGA-based environments. Quantization of the model was aimed at reducing the computational complexity and memory requirement of the model at the cost of not compromising on prediction accuracy to a great extent. Quantization-Aware Training (QAT) using the Brevitas library and Post-Training Quantization (PTQ) using ONNX Runtime were the two primary approaches that were considered initially. The workflow also provides validation and evaluation steps of every quantized model regarding accuracy, numerical behavior, and deployability. But because of the toolchain restriction, only the PTQ approach was actually realized for deployment, as elaborated below.

3.2.1 Applying Quantization-Aware Training (QAT) Using Brevitas

In order to mimic low-bit inference behavior at training time, Quantization-Aware Training (QAT) was carried out through the **Brevitas** library, a PyTorch development framework for quantized neural networks. The rationale for doing so was to get the model ready for eventual deployment on hardware accelerators—i.e., FPGAs—by subjecting it to the same numerical constraints at training time that would be imposed at inference time. In this approach, all the default layers within the Temporal Convolutional Network (TCN) were substituted one by one with their quantized versions offered by Brevitas:

- `QuantConv1d` was used in place of traditional `Conv1d` layers to apply 8-bit weight and activation quantization with configurable scale and zero-point encoding.
- `QuantReLU` replaced standard ReLU activations, simulating non-linear activation behavior under low-bit constraints.
- `QuantLinear` substituted the final dense layer, enforcing quantized weight matrices and activations at the output stage.

These quantized layers introduced simulated quantization impacts in both training passes forward and backward so that the model learned quantization-resilient parameters. Quantization noise and reduced precision were emulated using fake quantization nodes, allowing the optimizer to update the network's parameters appropriately. In this way, training-time and inference-time numerical behavior are consistent, and this generally produces more accurate quantized models than post-training quantization.

Upon completion of training, the Brevitas model was exported to the **QONNX** (Quantized Open Neural Network Exchange) format using the `export_qonnx()` function. Unlike traditional ONNX models that rely purely on `float32` computations, the QONNX format embeds symbolic quantization semantics through specialized operator nodes such as:

- `Quant` (for quantization of tensors),
- `Mul` (for scale factor application), and
- `Trunc` (for integer rounding or clipping).

This quantization symbolic encoding enables downstream hardware synthesis tools—i.e., Xilinx FINN—to identify and implement the low-precision mathematics as actual hardware operations. Nevertheless, while exported FINN-compatible, the QONNX model continues to have weights in `float32` format; low-bit operation is realized through symbolic graph structure, rather than through explicit bit-width truncation. Unfortunately, this QAT-based quantization process pipeline was not deployable for this project.

The main limitation was that the NN2FPGA toolchain, which was chosen to synthesise neural networks to RTL, does not yet support Temporal Convolutional Network (TCN) architectures.

TCN-specific layers, including dilated 1D convolutions and residual structures, were not addressed by NN2FPGA's layer conversion or mapping.

As a result, the QONNX export—although structurally accurate and functionally accurate—was not deployable to the target FPGA workflow.

Therefore, while the QAT path successfully demonstrated the feasibility of training low-precision models with negligible loss in accuracy (average RMSE degradation $< 0.3\%$ relative to the `float32` baseline), it was excluded from the final deployment strategy. Nevertheless, the experiments validated that QAT using Brevitas is a hardware-aware and technologically solid solution, and it can remain a viable candidate for possible future deployment use cases with complete toolchain support. On the other hand, the deployment of the current project was solely based on Post-Training Quantization (PTQ) using manual C++ fixed-point conversion, as explained in the following sections.

3.2.2 Post-Training Quantization (PTQ) for Export to C++

PTQ offers an architecture-agnostic, non-intrusive alternative to Quantization-Aware Training (QAT), enabling the quantization of an already trained model without exposing it to architectural changes or retraining. The process of quantization started with the initial `float32` PyTorch model, which was trained and tested on the indoor localization dataset. The model was first exported to the ONNX (Open Neural Network Exchange) representation, as a common intermediate representation. Once in ONNX format, ONNX Runtime's dynamic quantization API—specifically, `quantize_dynamic()` from the `onnxruntime.quantization` module—was applied. This tool automatically converted eligible weight tensors of type `float32` to real `int8` initializers, as well as re-writing parts of the computation graph to quantized operations wherever possible. The conversion mainly involved linear layers and matrix multiplications, which are the most computationally demanding operations in ordinary neural networks.

The quantized model generated was inspected using Netron, which is a graphical neural network model visualizer. The inspection confirmed that the quantized ONNX model contained literal `int8` tensors and replaced float-based operations with quantized hardware-efficient counterparts. Unlike QONNX export used in QAT—which introduces symbolic quantization operators while keeping `float32` weights—the PTQ model contained true integer weights and was structurally adapted for low-bit inference. One of the most significant benefits of PTQ is that it is non-intrusive. It neither modifies the model's architecture nor adds more training cycles, which makes it very useful for deployment environments where retraining is unfeasible or expensive. Empirical evaluations showed that the prediction accuracy of the PTQ model remained very close to the original `float32` baseline. In fact, in some validation folds, the quantized model achieved slightly better performance, a phenomenon often attributed to the implicit regularization effect introduced by quantization noise during inference.

Crucially, this quantized ONNX model also served as the reference blueprint for the downstream C++ fixed-point implementation. After confirming the structure and weights, the `int8` quantized weights were converted into Q4.4 fixed-point format (`ap_fixed<8,4>`) and manually embedded into C++ header files. In the C++ implementation, all tensor operations were re-written using fixed-point types such as `ap_fixed<8,4>` for multipliers and `ap_int<32>` for accumulators. Care was taken to faithfully replicate the behavior of quantized inference, including `int8`-style multiplication, rounding, truncation, and saturation. This ensured consistency between the PTQ ONNX model and the synthesized C++ version intended for FPGA deployment.

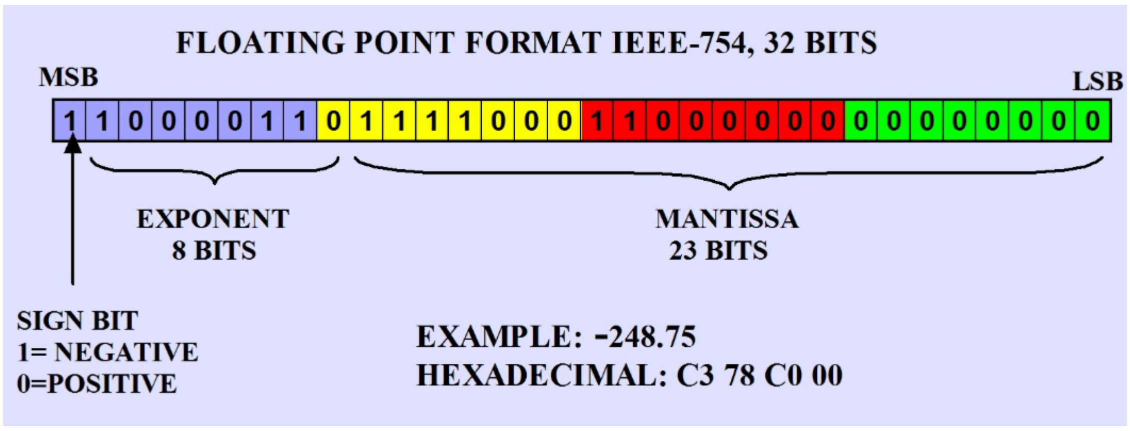

Figure 3.2. Floating Point Representation IEEE 754 [29]

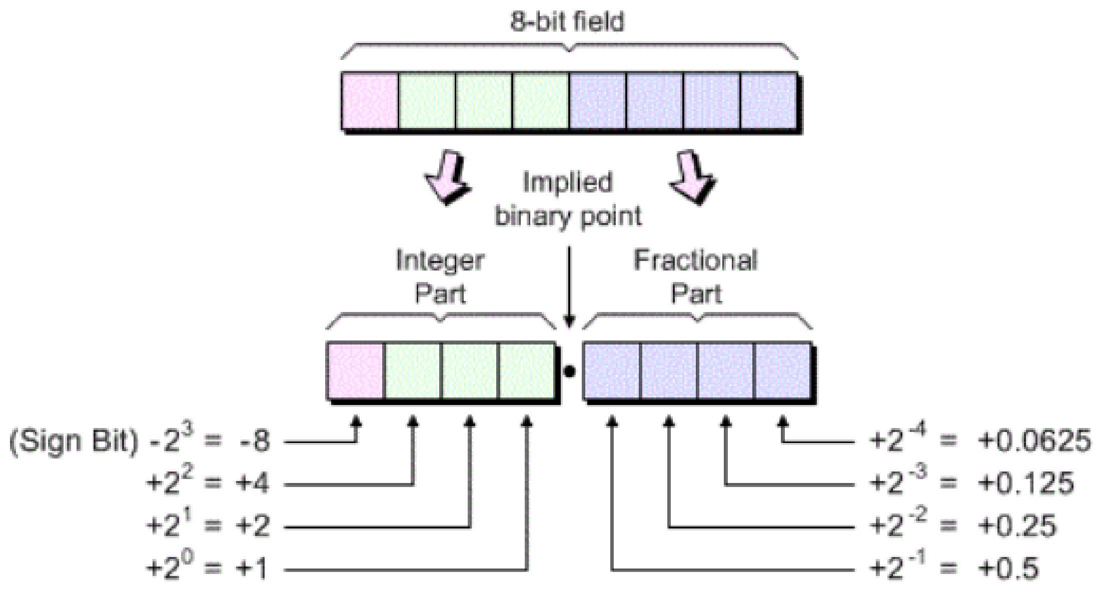

Figure 3.3. Q4.4 Fixed Point Representation [30]

Thus, PTQ not only facilitated a smooth and accurate quantization process but also enabled a seamless transition to hardware-level fixed-point computation, making it a central component of the deployment pipeline.

Due to deployment limitations and incompatibilities between toolchains—i.e., NN2FPGA's inability to take QONNX models obtained from Temporal Convolutional Networks (TCNs)—Post-Training Quantization (PTQ) was done manually. Further technical details regarding the fixed-point transformation, C++ kernel implementation, and synthesis for FPGA are provided in the subsequent sections of this thesis.

3.2.3 Structure and Design of the PyTorch Quantized TCN Model

The original neural network model was implemented in PyTorch and based on the Temporal Convolutional Network (TCN) architecture, which is well-suited for processing sequential time-series data such as indoor localization signals. TCNs are designed to handle temporal dependencies through the use of causal and dilated convolutions, making them effective for tasks that require learning from patterns over time. The model in this work was specifically tailored to handle input windows of length 15, each consisting of 4 parallel sensor features, resulting in an input tensor of shape `[batch size, 15, 4]`.

Model Input and Data Structure

The input to the TCN was a time-series sequence with a fixed window of 15-time steps and 4 channels per step. Each sample represented a temporal snapshot of sensor readings, where channels could represent RSSI values, UWB data, or other time-synchronized signal features. This input was transposed in PyTorch to match the expected shape for 1D convolutions—`[batch, channels, sequence]`, i.e., `[B, 4, 15]`.

1×1 Downsampling Convolution

The first layer in the TCN architecture was a 1×1 convolution, also known as a pointwise convolution. Its role was to map the input feature space from 4 channels to the internal hidden representation dimension of 8 channels (matching the depth used throughout the residual blocks). This operation also served as a downsampling or projection layer, ensuring that the dimensionality matched across the network's depth, especially for the residual connections that require consistent shapes between input and output. The use of a 1×1 kernel ensures that no temporal context is blended in this step—only feature transformation is performed.

Residual Blocks with Dilated Convolutions

Following the downsampling layer, the model consisted of three stacked residual blocks. Each residual block contained two dilated 1D convolutional layers, enabling the network to model long-range dependencies without increasing the number of layers or parameters significantly. The dilation rates were set to `[1, 2, 4]`, meaning that the receptive field of the model grew exponentially with each block. The kernel size used was 5, and the padding was set in a causal manner (i.e., future time steps were not used to predict the current one), ensuring temporal consistency.

Each convolution in the residual block was followed by:

- **Normalization Layer:** Either `BatchNorm1d`, `LayerNorm`, or an identity operation depending on configuration. In this project, normalization was kept minimal to facilitate quantization and FPGA deployment.

- **ReLU Activation:** Non-linearity was introduced using the ReLU function (`max(0, x)`), enabling the model to capture complex patterns.
- **Dropout Layer:** Applied only during training to prevent overfitting by randomly masking neurons. Dropout was disabled during inference and not implemented in the final hardware model.

To preserve gradient flow and enable deeper models, skip connections were added between the block's input and output. When input and output channel dimensions matched, a direct identity connection was used. Otherwise, the skip path also passed through a 1×1 convolution.

Final Output Layer

After the final residual block, the output tensor was of shape `[batch, 8, 15]`. As the model was configured for non-sequential output mode, only the last time step from the sequence (i.e., index 14) was extracted. This tensor slice of shape `[batch, 8]` was passed through a fully connected `Linear` layer mapping the 8 hidden units to 2 output units, representing the 2D spatial coordinates (`x`, `y`) of the predicted position.

In PyTorch, the final output was passed through a `log_softmax()` function if classification was required. However, in this work—where the task is regression (coordinate prediction)—the final activation was omitted, and the raw outputs were treated as continuous values.

Training and Quantization

The model was trained using 32-bit floating-point precision and optimized using standard loss functions like RMSE or MSE. Following training, the model was quantized using Post-Training Quantization (PTQ) to convert weights and activations to 8-bit integers. This step was critical to preparing the model for deployment on FPGA hardware, where memory and compute resources are limited and low-precision inference is essential for real-time operation. Quantized weights and biases were exported and embedded into the final C++ implementation as `int8_t` arrays, enabling efficient fixed-point computation.

3.2.4 Exporting Quantized Weights from PyTorch to C++ Headers

To enable efficient fixed-point inference of the Temporal Convolutional Network (TCN) on FPGA hardware, the model weights and biases trained in floating-point precision were quantized and exported to a C++-compatible format. This process was essential for bridging the software-defined PyTorch implementation and the hardware-synthesizable C++ code used with Vitis HLS.

Motivation for Quantization

Quantization is the process of mapping high-precision numerical values (typically `float32`) to lower-precision representations (e.g., `int8`) in a way that reduces memory footprint, bandwidth requirements, and arithmetic complexity while retaining acceptable accuracy. On FPGA, quantization is critical because:

- Fixed-point arithmetic is significantly cheaper than floating-point in terms of logic and power.
- DSP blocks and BRAMs are limited resources, and using 8-bit data maximizes parallelization.
- Latency and power consumption can be drastically reduced by avoiding float operations.

In this project, quantization enabled deployment of a deep TCN model in an embedded environment without violating the real-time constraints of indoor localization.

Quantization Strategy

The chosen quantization strategy was symmetric linear quantization using a fixed scale factor of 16. This means all `float32` weights and biases were mapped to 8-bit integers via the following formula:

$$\text{quantized\_val} = \text{round}(\text{float\_val} \times \text{scale})$$

where `scale` = 16.

This effectively shifts the floating-point range $[-8.0, +7.9375]$ into the `int8` range $[-128, +127]$, assuming no clipping beyond bounds. The choice of a power-of-two scale factor ($2^4 = 16$) simplifies the hardware implementation, as division and multiplication by 16 can be replaced with bit shifting (`» 4` and `« 4`), which are much faster and more efficient in hardware logic.

Dequantization in C++

To convert intermediate outputs from fixed-point accumulations back to approximate float values in the final layer, dequantization was performed by applying the inverse of the scale:

$$\text{float\_val} = \frac{\text{quantized\_val}}{\text{scale}}$$

In the C++ implementation, this was done by dividing the final accumulator by `scale^2 = 256`, since both the input and weight were scaled by 16. This scaling was integrated directly into the logic of the output layer:

```
output[o] = static_cast<float>(acc) / (SCALE * SCALE);
```

Header File Generation: `tcn_weights_int8.h`

Once quantized, the weights and biases were exported from Python into a C-compatible header file named `tcn_weights_int8.h`. This file contained:

- `int8_t` arrays for each convolutional layer's weights and biases.

- `int8_t` arrays for the downsampling layer and final linear layer.
- All arrays were flattened into 1D buffers to simplify indexing in C++ and minimize synthesis complexity.

Each array followed the naming convention: `tcn_network_<layer>_<conv1/conv2/downsample>_<weig`
These weights were indexed manually in the convolution loops using:

```
int w_idx = oc * in_ch * KERNEL_SIZE + ic * KERNEL_SIZE + k;
```

This ensured that the structure and layout of weights in C++ exactly mirrored those used in the PyTorch model, preserving correctness in forward propagation.

Advantages of Manual Export

While many toolchains (e.g., Brevitas → QONNX → FINN) support automatic export pipelines, manual export was chosen for the following reasons:

- Complete control over precision and scaling behavior
- Compatibility with custom C++ inference engine
- Simplicity in debugging and verification, especially when validating numerical equivalence between PyTorch and C++ inference outputs

Export scripts were implemented using NumPy, and quantized arrays were saved using `np.round()`, followed by casting to `int8`. A custom code generation step then wrote these arrays as C-style initializers into the `.h` file.

Limitations and Considerations

- A single global scale factor was used for all layers, simplifying implementation but limiting the ability to fine-tune quantization per layer.
- Asymmetric quantization or per-channel scales were not used to avoid complex dequantization logic in C++ and to minimize logic overhead.
- Rounding errors and clipping were carefully controlled to avoid inference instability in edge cases.

3.2.5 Implementation of Fixed-Point Arithmetic in C++

To enable hardware-efficient inference on FPGA, the model was entirely implemented using fixed-point arithmetic, eliminating the need for floating-point operations that are typically resource-intensive and power-hungry on programmable logic devices. The numerical format used was based on Xilinx's `ap_fixed` and `ap_int` types, which are optimized for synthesis using Vitis HLS.

Data Representation

All intermediate and input/output values in the model were represented using:

- `ap_fixed<8, 4>` for 8-bit signed fixed-point numbers, where:
  - 8 is the total number of bits
  - 4 are integer bits (including the sign bit)
  - The remaining 4 bits represent the fractional part (resolution = 0.0625)

This format allows encoding values in the range of approximately $[-8.0, +7.9375]$ with a granularity of $\frac{1}{2^4} = 0.0625$, which aligns precisely with the symmetric quantization scheme using a scale factor of 16.

Accumulations during convolution and fully connected layers were performed using:

- `ap_int<32>`, a 32-bit signed integer type, to prevent overflow and preserve intermediate precision before final output scaling.

ReLU Activation with Saturation

The ReLU (Rectified Linear Unit) activation function, defined as $\text{ReLU}(x) = \max(0, x)$, was implemented in fixed-point logic with additional saturation logic. This ensured that:

- Negative values were clipped to zero.
- Positive values exceeding the maximum representable `int8` value (+127) were saturated.

C++ implementation:

```
int8_t relu(acc_t x) {
    return (x > 127) ? 127 : (x < 0 ? 0 : static_cast<int8_t>(x));
}
```

This behavior mimics standard ReLU while preventing overflows during inference, preserving numerical stability across layers.

Fixed-Point Accumulation and Scaling

All core mathematical operations — such as convolutions and linear transformations — were performed using integer multiply-accumulate (MAC) operations. Each operation multiplied an 8-bit quantized input with an 8-bit quantized weight and accumulated the result into a 32-bit accumulator:

$$\text{acc}+ = w \times x$$

where both w and x are `int8_t`, and the result is held in an `ap_int<32>` accumulator `acc`.

After summing all contributions, the accumulator was scaled back down by dividing by the square of the scale factor (i.e., $\text{scale}^2 = 256$) to simulate dequantization. Since divisions are expensive in hardware, this was implemented using a bitwise right shift:

$$\text{dequantized\_val} = \frac{\text{acc}}{16} \approx \text{acc} >> 4$$

This efficient approximation retained performance while adhering to the quantization scheme's numeric logic.

Final Output Scaling and Conversion

In the final linear layer, which outputs `float32` coordinates, the accumulated `int32` values were converted back to approximate floating-point by applying:

$$\text{output}[o] = \frac{\text{acc}}{\text{scale}^2}$$

C++ implementation:

```
output[o] = static_cast<float>(acc) / (SCALE * SCALE);
```

This converts the integer sum back into a meaningful real-world prediction by reversing the quantization effect. It also ensures compatibility with systems expecting float output (e.g., visualization or post-processing modules).

Advantages of Fixed-Point Arithmetic

- **Hardware Efficiency:** Avoids the need for floating-point IP cores, saving FPGA area and power.
- **Predictable Latency:** Integer operations have deterministic execution cycles.
- **Scalability:** Enables processing multiple channels in parallel using array partitioning and pipelining.

Challenges Addressed

- **Overflow Management:** 32-bit accumulators were chosen to ensure sufficient headroom during MAC operations.
- **Quantization Noise:** The choice of scale (16) and symmetric quantization provided a good trade-off between range and resolution, minimizing accuracy loss.
- **Bit-Accurate Consistency:** Ensured that inference outputs matched PyTorch (float32) outputs within a small RMSE margin after quantization.

3.2.6 Layer-by-Layer Translation of the Model to C++

This section describes how each layer of the original PyTorch TCN model was manually implemented in C++ using fixed-point arithmetic and synthesized for FPGA using Vitis HLS. Each layer is functionally equivalent to its software counterpart but rewritten to use `int8_t` operations, static array indexing, and pragmas for hardware optimization.

1. Downsampling Layer

The first stage in the hardware TCN model is a 1×1 convolution, implemented in the function `downsample_fixed()`. In the PyTorch model, this layer projects the input from 4 channels to the hidden representation of 8 channels, ensuring compatibility with the subsequent residual blocks.

In the C++ implementation:

- The convolution uses a kernel size of 1, so each output channel at each time step is computed as a weighted sum across the 4 input channels at the same time step.
- The corresponding C++ loop performs this sum using quantized weights and biases:

```
acc_t acc = static_cast<acc_t>(bias[oc]) * SCALE;
for (int ic = 0; ic < IN_CH; ++ic) {
    acc += static_cast<acc_t>(weights[w_idx]) * input[ic][t];
}
```

The result is passed through a ReLU activation with saturation and stored in the `x0` buffer for input to the first residual block. This implementation maps directly to the PyTorch operation while optimizing for low-latency execution via loop pipelining and array partitioning.

2. Residual Blocks with Dilated Convolutions

Each residual block is implemented in the function `residual_block_fixed()` and consists of:

- Two dilated 1D convolutional layers
- A residual skip connection that adds the input back to the output

Dilated Convolution Logic To replicate PyTorch's `nn.Conv1d` dilation behavior with dilation factor d, the convolution logic accesses past inputs spaced by the dilation step:

```
int idx = t - dilation * k;
```

This ensures that for kernel position k, the convolution accesses time step $t - d \cdot k$. If the index is negative (i.e., before the start of the sequence), a zero is used to simulate causal padding.

This method guarantees:

- Exponential receptive field growth across layers
- Preservation of temporal causality (no future information leaks into predictions)

Layer Execution Each residual block follows this sequence:

1. Apply first dilated convolution with bias and ReLU
2. Store output in a temporary buffer
3. Apply second dilated convolution with bias and ReLU
4. Add original input via residual skip connection:

```
acc = static_cast<acc_t>(output[oc][t]) + static_cast<acc_t>(input[oc][t]);
output[oc][t] = relu(acc);
```

The result is written into a new activation buffer (e.g., `x1`, `x2`, `x3`). Residual blocks are called with increasing dilation rates of 1, 2, and 4, consistent with the PyTorch model.

3. Final Linear Layer

After the last residual block (`x3`), the output tensor is of shape `[8][15]`, corresponding to 8 channels across 15 time steps. Since the model is configured for non-sequential regression, only the last time step (index 14) is used:

```
last_step[i] = x3[i][SEQ_LEN - 1];
```

This 8-element vector is passed to `linear_output_fixed()`, which implements a fully connected layer using quantized weights and biases:

```
acc += static_cast<acc_t>(weights[idx]) * input[i];
```

The output consists of 2 values (predicted x and y coordinates in float). Since both inputs and weights are scaled by 16, the accumulator is divided by $\text{scale}^2 = 256$ to dequantize:

$$\text{output[o]} = \frac{\text{acc}}{\text{SCALE}^2} \tag{3.1}$$

C++ equivalent:

```
output[o] = static_cast<float>(acc) / (SCALE * SCALE);
```

This ensures compatibility with downstream modules expecting floating-point predictions, such as tracking systems or evaluation metrics.

3.2.7 Benefits of Manual Quantized Model Translation to C++

Opting for a manual translation of the quantized PyTorch model into synthesizable C++ code — rather than relying on automated tools like NN2FPGA or FINN — provided several significant advantages. These benefits span numerical control, optimization flexibility, validation clarity, and long-term adaptability of the hardware design.

1. Full Control Over Numerical Behavior

Manual C++ implementation allowed precise control over how arithmetic was performed, especially in terms of fixed-point quantization behavior. With this approach:

- The quantization scale factor, saturation limits, and rounding strategy were explicitly defined in code rather than abstracted away in opaque conversion tools.
- Custom logic was introduced for ReLU activations with saturation, clipping logic for overflow, and dequantization by shift-scaling — all critical for maintaining consistent behavior across layers.
- Accumulators were carefully managed using 32-bit signed integers (`ap_int<32>`) to avoid overflow, a level of control that most high-level translation tools do not expose or optimize for.

This explicit handling of numerical behavior ensured that the fixed-point version closely mirrored the float32 behavior of the original PyTorch model with minimal loss in accuracy.

2. Platform-Specific Optimizations (to be discussed later)

The translation process was tailored to the requirements and capabilities of Xilinx Vitis HLS and the specific FPGA platform used for deployment. This level of tuning enabled:

- Optimized memory layouts via `ARRAY_PARTITION` and `ARRAY_RESHAPE`, which reduced access latency and enabled parallel memory reads from BRAM.
- Fine-tuned compute loops using `#pragma HLS PIPELINE`, `UNROLL`, and `DATAFLOW`, which maximized throughput and reduced latency.
- Interface pragmas (`AXI4-FULL` and `AXI-Lite`) were customized for efficient integration with the target platform's memory and control interfaces.

Such hardware-specific adaptations are often impossible or suboptimal when using general-purpose auto-conversion tools, which tend to generate generic but inefficient HDL.

3. Enhanced Debug Visibility and Validation

Manual translation provided transparent visibility into every layer and operation of the inference pipeline:

- Each function (e.g., convolution, ReLU, residual connection) could be individually tested using C++ testbenches or co-simulation.
- Intermediate activation maps and accumulator states could be logged and compared directly against PyTorch outputs.

- The debug process was significantly more manageable because code logic and data flow were explicit, unlike in automated hardware exports where the mapping from high-level model layers to hardware logic is often non-trivial.

This improved debug visibility was crucial during iterative design, validation, and performance profiling phases.

4. Modularity and Future Extensibility

The handcrafted C++ implementation is modular by design, enabling flexible experimentation and extension. Some advantages include:

- Easy replacement of activation functions (e.g., swapping ReLU with LeakyReLU)
- Support for mixed-precision formats (e.g., `int4`, `ap_fixed<6,2>`) simply by adjusting data types and quantization logic
- Integration of hardware-specific acceleration units, such as DSP blocks or custom systolic arrays, without being constrained by the abstraction layers of automated toolchains
- Simplified adaptation for future models with different numbers of layers, channels, or kernel sizes

This extensibility makes the C++ implementation not just a single-use converter, but a reusable foundation for future FPGA-based deep learning applications.

3.2.8 Limitations and Design Trade-offs in the Quantization Pipeline

While manual translation of the quantized PyTorch model into synthesizable C++ code provided a high level of control and optimization potential, it also introduced several non-trivial challenges. These limitations highlight the engineering trade-offs involved in bypassing automated toolchains and should be taken into account for similar future implementations.

1. Time-Consuming and Error-Prone Development

Manually implementing each layer, operation, and buffer in C++—especially in a way that conforms to the requirements of Vitis HLS—demanded a significant amount of development time. Unlike automated toolchains, which abstract much of the hardware generation and validation process, manual development required:

- Writing explicit loops for all convolutional, activation, and linear layers
- Managing array dimensions, data types, and interface specifications
- Inserting and tuning HLS pragmas (e.g., `#pragma HLS PIPELINE`, `ARRAY_PARTITION`) for performance and synthesis correctness

This low-level design process increased the risk of indexing errors, data misalignment, or overflow issues, all of which could silently degrade model accuracy or cause synthesis failures. Debugging such issues often required close inspection of both functional behavior and hardware synthesis logs, which added to the complexity.

2. Explicit Simulation of Quantization Behavior

Unlike toolchains such as Brevitas or QONNX, which embed quantization logic through symbolic operators and layer wrappers, the manual C++ approach required all aspects of quantization to be explicitly simulated in code:

- Multiplication by a fixed scale factor during quantization
- Manual implementation of ReLU with saturation for `int8_t` values
- Use of integer shift operations for dequantization

This increased the number of operations that had to be verified, and added an additional burden in ensuring bit-accurate consistency with the original floating-point model. Any deviation from the intended quantization flow (e.g., forgetting to apply a shift after accumulation) could lead to unpredictable inference behavior.

Moreover, since no standardized quantization library was available in this environment, the behavior of rounding, clipping, and overflow had to be replicated manually and consistently across the entire pipeline.

3. Manual Weight Mapping and Indexing

All weights and biases extracted from the trained PyTorch model were exported as `int8_t` arrays and manually written into a C++ header file (`tcn_weights_int8.h`). While this provided direct access to quantized parameters, it also required:

- Flattening multi-dimensional tensors into 1D arrays
- Reconstructing the original dimensionality in code via manual indexing
- Strict layout consistency between PyTorch and C++

For example, indexing into convolutional weight arrays required detailed knowledge of the weight layout:

$$\text{index} = \text{output\_channel} \times \text{input\_channel} \times \text{kernel\_size} + \text{input\_channel} \times \text{kernel\_size} + k$$

Errors in this mapping could lead to subtle yet catastrophic misalignments, especially in deep or multi-branch networks. No automated consistency checking existed, so the burden of validation was fully on the developer.

4. Limited Flexibility in Model Portability

Because the implementation was hardcoded to a specific model architecture (e.g., 3 residual blocks, dilation rates of [1, 2, 4], 8 channels), any changes in the model design would require:

- Rewriting parts of the C++ inference code
- Re-exporting and relabeling weight arrays
- Re-validating all buffer sizes and pipeline behavior

This makes it difficult to generalize or reconfigure the hardware implementation for dynamic models or varying input/output sizes, in contrast to more abstracted deployment tools which can often handle structural variations with minor modifications.

3.2.9 Summary of Quantization Workflow and Insights

The manual translation of the quantized PyTorch Temporal Convolutional Network (TCN) into a synthesizable C++ implementation proved to be a highly effective approach for deploying deep learning models on FPGA using High-Level Synthesis (HLS). Unlike automated toolchains that abstract away much of the conversion process, the handcrafted implementation enabled precise control over numerical behavior, layer functionality, memory layout, and hardware resource usage — all of which are critical in constrained edge computing environments.

Every component of the model — from the downsampling projection layer to the series of residual blocks with dilated convolutions, and finally the fully connected linear output layer — was faithfully reimplemented using `int8` fixed-point arithmetic. This ensured that quantization effects were fully preserved and that the model's behavior remained functionally consistent with the original `float32` PyTorch implementation. The use of `ap_fixed<8,4>` and `ap_int<32>` types allowed efficient multiply-accumulate operations while avoiding the overhead of floating-point units.

Further, the C++ implementation was carefully structured to match the TCN's original architecture, including:

- Causal, dilated convolutions that preserved temporal context
- Residual connections to maintain gradient flow and representational depth
- Last-step slicing to produce non-sequential, real-valued regression outputs

This structural fidelity was essential for maintaining prediction accuracy while reducing the computational burden through quantization.

The manual process also enabled deep integration with Vitis HLS, where architectural optimizations such as loop pipelining, array partitioning, and memory banking were applied to meet real-time performance constraints. These optimizations were tailored specifically to the FPGA's logic and memory architecture, resulting in an implementation that was not only correct and lightweight but also synthesizable and deployable on a real hardware target.

While the development effort was significantly more involved compared to automated deployment flows, the benefits of full transparency, debug visibility, and platform-aware tuning made the manual approach ideal for research and prototyping scenarios requiring maximum control and reproducibility.

Ultimately, the final design demonstrated that a manually quantized and optimized neural network could achieve high inference performance on FPGA while maintaining fidelity to its software origin. This methodology serves as a reference blueprint for future implementations of fixed-point deep learning models targeted at embedded hardware platforms.

3.3 High-Level Synthesis Co-Optimization

In order to achieve real-time inference performance on FPGA for the Temporal Convolutional Network (TCN) model, several high-level synthesis (HLS) co-optimization techniques were employed. These techniques targeted the efficient transformation of C/C++ descriptions into synthesizable RTL while preserving the model's functionality and accuracy. This section discusses in detail the optimizations performed via HLS directives (pragmas), the manual and controlled implementation of dilated convolutions, and memory architecture tuning via partitioning and reshaping for parallel data access.

3.3.1 Leveraging HLS Pragmas for Structural and Performance Optimization

High-Level Synthesis (HLS) pragmas serve as essential directives to guide the compiler in translating C/C++ behavioral descriptions into optimized hardware structures without modifying the algorithmic logic. In this project, these pragmas were strategically applied to maximize the parallelism, minimize latency, and enhance throughput of the Temporal Convolutional Network (TCN) implementation on FPGA. The primary pragmas utilized included `#pragma HLS pipeline`, `#pragma HLS unroll`, and `#pragma HLS dataflow`, each targeting specific aspects of performance optimization. Rather than applying them indiscriminately, each directive was iteratively introduced based on synthesis feedback and performance profiling to achieve an optimal balance between speed and resource usage.

Loop Pipelining

Loop pipelining was a key optimization, particularly within the outer temporal loop of the convolution operations, which process data across sequence time steps. Applying `#pragma HLS pipeline II=1` allowed the compiler to initiate a new loop iteration every clock cycle, thus overlapping operations and reducing idle cycles. A typical pattern used in the convolution loops was:

```
for (int i = 0; i < SEQ_LEN; i++) {
    #pragma HLS pipeline II=1
    // computation logic
}
```

This dramatically improved throughput, especially for long sequences.

Loop Unrolling

For inner loops—such as those iterating over input channels or kernel taps—`#pragma HLS unroll` was applied to increase computation parallelism. Loop unrolling allowed simultaneous operations across different channels, enhancing DSP and logic utilization:

```
for (int ch = 0; ch < IN_CH; ch++) {
    #pragma HLS unroll
    // channel-wise operations
}
```

The unroll factor was implicitly derived from loop bounds and resource availability reported in the synthesis tool.

Dataflow Scheduling

To achieve high-level task parallelism, the `#pragma HLS dataflow` directive was employed across function boundaries to decouple major stages such as input preprocessing, convolutional computation, and output postprocessing. This enabled concurrent execution of these stages, each operating as an independent thread using internal FIFOs for data transfer:

```
#pragma HLS dataflow
read_input();
compute_tcn();
write_output();
```

Function-level pipelining provided by dataflow significantly reduced total inference latency. To ensure correctness and avoid pipeline hazards, the codebase was carefully structured to remove interdependencies and ensure that all inputs and outputs were fully buffered.

Function Inlining Control

`#pragma HLS INLINE OFF` was selectively applied to modular functions like `conv1d_fixed`, `residual_block_fixed`, and `downsample_fixed`. This directive prevents the compiler from flattening these functions into their callers, preserving modular scheduling. In combination with `dataflow`, it allowed each function to be synthesized and executed in parallel, ensuring efficient pipeline separation and resource management without sacrificing functional clarity.

Summary

These pragmas formed the backbone of the HLS co-optimization strategy, allowing the transformation of the high-level TCN model into a hardware-efficient, high-performance design suitable for FPGA deployment.

3.3.2 Custom Implementation of Dilated Convolutions in HLS

To increase the receptive field of the Temporal Convolutional Network (TCN) without inflating the kernel size or model complexity, dilated convolutions were employed. Dilation introduces controlled gaps between filter elements, allowing the network to capture longer temporal dependencies while keeping the number of parameters constant. However, in the context of High-Level Synthesis (HLS), implementing dilation is not straightforward and requires explicit loop manipulation to guide hardware generation.

Manual Loop Indexing for Dilation

To implement dilation in a hardware-friendly manner, the inner convolution loop was manually modified to introduce the dilation spacing through adjusted indexing. This approach avoids abstract or dynamic access patterns that could hinder the synthesis tool's ability to optimize memory scheduling and pipeline the loops effectively. The modified loop structure is shown below:

```
for (int k = 0; k < KERNEL_SIZE; ++k) {
    int input_index = t - k * dilation;
    if (input_index >= 0) {
        acc += weight[k] * input[input_index];
    }
}
```

Here, the dilation parameter is statically applied to the index calculation, resulting in a sparse access pattern. This code structure provides the HLS compiler with clear, analyzable dependencies and memory access patterns, making it easier to pipeline and optimize the loop.

Static Scheduling and Optimization

The dilation factor itself was defined as a compile-time constant (either passed as a template parameter or hardcoded) rather than being dynamically configurable at runtime. This allowed the HLS tool to perform static scheduling, leading to more aggressive optimizations such as loop pipelining and partial unrolling. Furthermore, this static approach avoids runtime control logic overhead and ensures deterministic hardware behavior.

Avoiding Loop-Carried Dependencies

By explicitly encoding the dilation logic, the implementation retains full visibility over the memory read addresses and avoids potential loop-carried dependencies, which could prevent pipelining. This method proved essential to maintaining predictable hardware structure, meeting timing closure, and optimizing latency in the final FPGA implementation.

3.3.3 Memory Banking and Efficient Data Access in FPGA Logic

The Temporal Convolutional Network (TCN) architecture involves processing multiple channels and temporal steps in parallel, leading to high simultaneous memory access demands. To mitigate potential memory access bottlenecks, especially in bandwidth-constrained FPGA environments, memory banking techniques were applied through array partitioning and optimized AXI interfaces.

Array Partitioning for Parallel Channel Access

To allow concurrent access to activation arrays and intermediate buffers, array partitioning was applied using the following directive:

```
#pragma HLS ARRAY_PARTITION variable=input complete dim=1
```

This partitions the array along the channel dimension, effectively creating independent memory banks for each channel. As a result, multiple channels can be read or written in parallel, which is particularly beneficial for loop unrolling and pipelining across channels. This technique was applied to the input feature maps as well as intermediate tensors like `x0`, `x1`, and others.

Partitioning Intermediate Buffers

Similar partitioning was performed on internal buffers that played a crucial role in residual computations and output generation:

```
#pragma HLS ARRAY_PARTITION variable=temp complete dim=1
#pragma HLS ARRAY_PARTITION variable=last_step complete dim=1
```

By fully partitioning the `temp` and `last_step` arrays along the channel dimension, the design enabled all channel elements to be accessed concurrently in a single cycle. This significantly enhanced throughput in both the intermediate residual blocks and the final linear classification layer.

AXI Interface Optimization

In addition to internal memory optimizations, external communication was handled through carefully assigned AXI interfaces on the top-level function:

```
#pragma HLS INTERFACE m_axi depth=60 port=input_f offset=slave bundle=INPUT
#pragma HLS INTERFACE m_axi depth=2 port=output_f offset=slave bundle=OUTPUT
#pragma HLS INTERFACE s_axilite port=return bundle=CTRL
```

These pragmas map the input and output data arrays to AXI4-FULL interfaces for high-speed burst transfers, while the control logic is exposed through a lightweight AXI4-Lite interface. Grouping these ports into logical bundles (`INPUT`, `OUTPUT`, `CTRL`) facilitates clear separation and seamless integration within a block design in Vivado.

Performance Gains and Real-Time Operation

Together, these optimizations ensured low-latency, high-throughput memory access while staying within BRAM and DSP resource limits. The use of `ARRAY_PARTITION` and AXI configuration was crucial for enabling real-time inference in the deployed FPGA accelerator.

3.3.4 Additional Pragmas and Constructs to Optimize HLS Inference Logic

Beyond the core optimization directives such as `pipeline`, `unroll`, and `dataflow`, several supporting pragmas and constructs were applied to improve modularity, concurrency, and hardware synthesis efficiency throughout the Temporal Convolutional Network (TCN) implementation.

Function Modularization via INLINE OFF

The following pragma was used on major compute blocks:

```
#pragma HLS INLINE OFF
```

This directive was applied to functions such as `conv1d_fixed`, `residual_block_fixed`, `downsample_fixed`, and `linear_output_fixed`. It prevents the compiler from flattening function calls, ensuring each is synthesized as an independent hardware block. This enabled effective `dataflow` between stages and allowed parallel scheduling across model layers.

Enhanced Array Partitioning

To support concurrent channel access, `ARRAY_PARTITION` pragmas were applied to internal arrays:

```
#pragma HLS ARRAY_PARTITION variable=temp complete dim=1
#pragma HLS ARRAY_PARTITION variable=last_step complete dim=1
```

This facilitated simultaneous reads/writes across all channels, boosting throughput in both intermediate computations and the final linear layer.

Multi-Dimensional Partitioning for Deep Concurrency

In performance-critical loops, full partitioning was extended along multiple dimensions:

```
#pragma HLS ARRAY_PARTITION variable=input complete dim=0
#pragma HLS ARRAY_PARTITION variable=input complete dim=1
```

This enabled concurrent access across both time steps and channels, essential in deeply nested loops such as convolutions or residual computations.

Static Indexing for Dilation Support

Dilation logic was implemented via static indexing to avoid dynamic memory access hazards:

```
int index = t - k * dilation;
```

Static, compile-time constant indexing ensured that all access patterns were analyzable by the HLS compiler, enabling pipelining, unrolling, and timing closure.

Intermediate Buffer Reuse in Dataflow Pipeline

Buffers such as `x0, x1, x2, x3` were reused across residual blocks. These were declared as local memory and partitioned:

```
#pragma HLS ARRAY_PARTITION variable=x0 complete dim=1
```

This buffer reuse enabled tightly coupled dataflow while avoiding redundant storage, reducing BRAM usage and increasing throughput.

Tool-Driven Resource Binding

Although Vivado HLS supports explicit binding via:

```
#pragma HLS RESOURCE
```

no manual resource binding was enforced. Multiply-accumulate operations were inferred to DSP slices, and memory buffers were inferred to BRAMs or LUTRAMs by the synthesis tool. This approach improved portability and allowed Vivado's heuristics to optimize binding for the specific FPGA device in use.

3.3.5 Summary of HLS Optimization Strategies for the TCN

Loop Pipelining and Unrolling

The successful hardware realization of the Temporal Convolutional Network (TCN) model was made possible through a carefully crafted set of High-Level Synthesis (HLS) co-optimization techniques. At the core of this effort were loop pipelining and loop unrolling, which enabled the concurrent execution of operations across time steps and channels. These directives allowed the design to reach high throughput, particularly critical for handling sequential data in real-time scenarios.

Dilated Convolution

Dilated convolutions were manually implemented by modifying loop index calculations, enabling the expansion of the receptive field without increasing the kernel size or computational cost. This manual control over dilation logic ensured predictable memory access and preserved compatibility with synthesis optimizations such as pipelining.

Memory Partitioning

Memory bandwidth and access contention — common performance bottlenecks in parallel architectures — were alleviated using array partitioning techniques. Internal buffers, including inputs and intermediate tensors, were fully partitioned across the channel dimension using `#pragma HLS ARRAY_PARTITION`, allowing simultaneous multi-channel access and seamless integration with unrolled loops.

Function Inlining Control

Function inlining was selectively disabled via `#pragma HLS INLINE OFF` to preserve modular hierarchy and facilitate efficient task scheduling within dataflow regions.

AXI Interface Mapping

To integrate the model into a full system on FPGA, top-level I/O ports were mapped using AXI4-FULL and AXI4-Lite interface pragmas. This ensured high-speed burst communication with external memory and provided lightweight control mechanisms for deployment within a processing system. Interfaces were grouped into logical bundles for clean integration in Vivado block design.

Conclusion

Altogether, the combination of structural pragmas (pipeline, unroll, dataflow, inline), custom dilation handling, and memory banking strategies enabled the synthesis of a resource-efficient, low-latency TCN accelerator. These optimizations formed the backbone of a hardware-aware design process, crucial for deploying deep learning models on FPGAs with tight performance and area constraints.

3.4 FPGA Block Design and System Integration

Vivado Project Initialization and Block Design Creation

The process began by launching Vivado and creating a new RTL project targeting the appropriate FPGA development board, such as the ZedBoard or Zybo. No design sources were added during the initial setup. After configuring the project settings, we transitioned to the IP Integrator environment, where a new block design—named tcn_fpga_wrapper—was created. This block design served as the structural foundation for integrating the exported HLS IP core with key system components, including memory controllers, the processing system, clock/reset modules, and AXI-based communication interfaces. The objective of this stage was to provide a functional and modular platform that supports the fixed-point inference model in hardware by enabling communication, synchronization, and memory interaction across all components. Notably, the HLS core was synthesized using partial pipelining, meaning that while some loops were pipelined to enhance throughput, full loop unrolling or extreme pipelining was intentionally avoided to maintain a balanced trade-off between area, power, and latency. This moderate pipelining strategy ensured efficient resource utilization while still enabling low-latency inference suitable for real-time embedded applications.

3.4.1 Configuring and Integrating Custom IP Blocks in Vivado

The following IP blocks were instantiated and configured in the Vivado block design environment:

1. ZYNQ7 Processing System

The ZYNQ7 Processing System was the first block added to the design. This IP block provides the interface to the on-chip ARM Cortex-A9 processor, memory controllers (e.g., DDR), and communication buses. After inserting the IP, Block Automation was executed. This Vivado feature automatically configures the Zynq PS with appropriate clock, reset, and AXI General Purpose Master (M_AXI_GP0) ports. The M_AXI_GP0 port was enabled to allow the processor to communicate with AXI slave peripherals, including the custom HLS IP and memory-mapped controllers. The DDR and fixed IO pins were automatically mapped based on the selected board.

2. Clocking Wizard

To supply a stable clock signal to the HLS IP core and other synchronous components, a Clocking Wizard IP was instantiated. It was configured to generate a 100 MHz output clock from the default onboard oscillator (typically 125 MHz). The clock output of this IP was connected to the `ap_clk` input of the custom HLS core and also distributed to other modules that required synchronous operation.

3. Processor System Reset

The Processor System Reset IP was inserted to manage reset signals throughout the design. It takes the output clock from the Clocking Wizard and the external reset input from the ZYNQ Processing System and generates appropriately synchronized reset signals for other IP blocks. Its outputs were connected to the `ap_rst_n` input of the HLS IP and other peripherals such as the BRAM controller and AXI GPIO.

4. AXI Interconnect

To route AXI transactions between the Zynq PS (acting as master) and multiple AXI slave devices, an AXI Interconnect IP was instantiated. This block supports multiple master/slave ports and handles address decoding, data multiplexing, and handshaking.

It was connected to the M_AXI_GP0 port of the ZYNQ PS and routed to the slave interfaces of the HLS IP, AXI BRAM Controller, and AXI GPIO.

5. AXI BRAM Controller + Block Memory Generator

For temporary on-chip data storage, the AXI BRAM Controller and Block Memory Generator (BRAM) IP blocks were used. The AXI BRAM Controller was connected to the AXI Interconnect and configured to serve as a memory-mapped AXI slave. The BRAM Generator was attached to the controller's output ports and configured with suitable depth and data width (e.g., 32-bit).

6. AXI GPIO

An AXI GPIO module was inserted to provide simple general-purpose I/O control. It was configured as an output-only interface, connected to the AXI Interconnect, and mapped to physical pins such as onboard LEDs. This enabled debugging and status signaling.

7. Custom HLS IP Core

The fixed-point TCN model synthesized in Vitis HLS was imported into the IP Catalog in Vivado and instantiated. The `s_axi_control` port was connected to the AXI Interconnect, `ap_clk` was connected to the Clocking Wizard, and `ap_rst_n` to the Processor System Reset output.

3.4.2 Automated and Manual Signal Routing for Design Integration

Once the IP blocks were added and configured, Vivado's Connection Automation tool was used to automatically wire compatible AXI interfaces and clock/reset lines. This included routing the ZYNQ PS's M_AXI_GP0 port to the AXI Interconnect and distributing the clock and reset signals from the Clocking Wizard and Processor System Reset, respectively.

For non-standard IPs, such as the custom HLS module, manual connections were made:

- `clk_out1` from the Clocking Wizard to `ap_clk` inputs of the HLS IP, AXI BRAM Controller, and other synchronous IPs.
- `peripheral_aresetn` from the Processor System Reset to `ap_rst_n` and other reset inputs.
- AXI Interconnect master interface to ZYNQ PS M_AXI_GP0; slave interfaces to HLS IP, GPIO, and BRAM Controller.

The Address Editor was used to assign unique address ranges to each AXI slave:

- HLS IP: `0x43C00000 - 0x43C0FFFF`
- AXI GPIO: `0x41200000`
- AXI BRAM Controller: a separate non-overlapping range

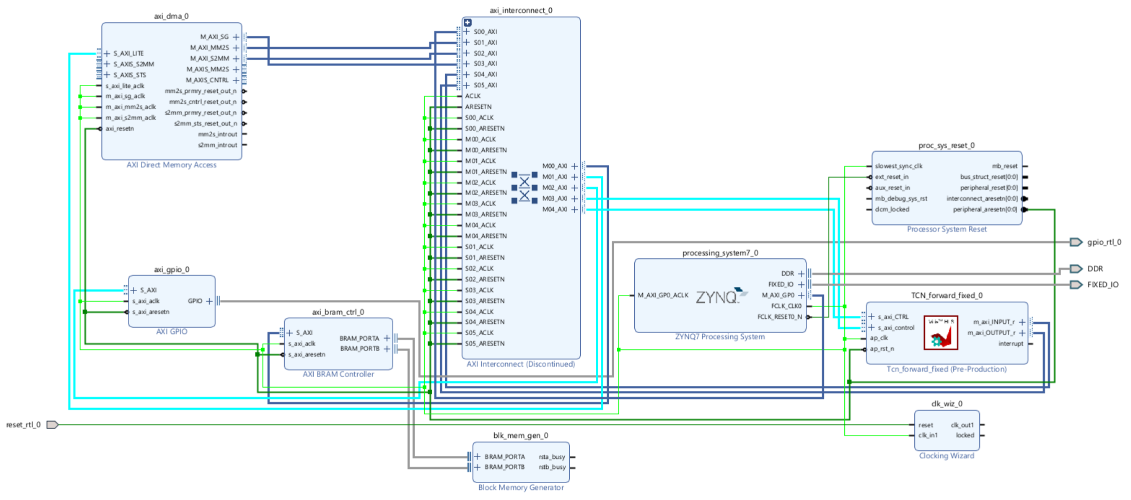

Figure 3.4. Vivado Block Design

3.4.3 Synthesis, Implementation, and Setup for Power Analysis

After verifying all connections and assigning addresses, the block design was synthesized and implemented using Vivado tools. Synthesis converted the design into a netlist, while implementation performed placement and routing. Once implementation completed successfully, Vivado's power analysis tool was used to estimate dynamic and static power consumption. These results—along with performance, latency, and resource metrics—are presented and discussed in the Results section.

Chapter 4

Experimental Validation

In this part, fixed-point TCN implementation analysis with respect to accuracy, latency, as well as resource utilization is presented. Its objective is to measure how much a human-defined, optimized, as well as quantized model retains a float32 PyTorch reference model's prediction accuracy with FPGA-deployment constraints. Functional correctness as well as hardware efficiency are prioritized equally, with a focus towards ensuring effectiveness for proposed quantization as well as synthesis pipeline.

4.1 Experimental Benchmark and Evaluation Framework

4.1.1 Description of the Dataset Used for Model Training and Testing

The data collected in this research was collected from a controlled 3x3 meters indoor environment with four single-plate load-mode capacitive sensors mounted at the middle point of every virtual wall. Movement as well as presence in the environment was sensed by the sensors with ground truth two-dimensional (x, y) location coordinate labeling for every sample. Sampling was at a frequency of 3 Hz, resulting in sequential time-series data with temporal movement information. One input sample is a 15 consecutive time step sliding window with four parallel channels on every sensor for a given input shape [15, 4].

To enhance signal quality as well as eliminate noise, raw sensor data were processed using a two-stage filtering pipeline. First, a 50-second window median filter was applied to eliminate spikes as well as outliers. Secondly, a low-pass filter within a transition band of 0.3 to 0.4 Hz was applied to eliminate high-frequency components not required to detect human movement. Following preprocessing, data was segmented into a training, a validation set, as well as a test set in temporal order not to destroy temporal consistency as well as cause information leakage. Specifically, 60% of data (910 samples) was held out for training as well as 20% for validation (310 samples) as well as tests. Temporal splitting of this type ensures model tests are run on unseen portions of data, promoting generalizability as well as serious performance testing.

4.1.2 Overview and Role of the Baseline Model in Comparison

The Temporal Convolutional Network (TCN) model implemented in this project was created in TensorFlow by Dr. Giorgia Subbicini. We optimized it with Neural Architecture Search (NAS) in order to minimize parameters but maintain high inference speed. It consists of three 1D convolutional layers with a 5xkernel, a total of eight filters, dilated convolutions for capturing temporal context, and an output dense with eight units. We trained it with an Adamax optimizer with a learning rate of 0.0001.

To achieve quantization-aware training as well as fixed-point inferencing for FPGA implementation, re-implementation of the model using PyTorch was done. Architecture as well as hyperparameters remained identical as those in the base model. Adam as an optimizer, as well as Mean Squared Error (MSE) loss, was utilized in order to train the model. Dataset (explained in Section 5.1.1) was divided temporally into training, test, and validation datasets. PyTorch model acted as reference software float32 for subsequent quantized as well as hardware-accelerated variants.

4.1.3 Metrics for Evaluation and Hardware Deployment Setup

To compare TCN model performance at different steps of training, as well as how well they perform, a standard procedure for regular evaluation was maintained.

Evaluation Metric:

The key performance metric utilized to approximate model accuracy was Mean Squared Error (MSE), a common performance metric in regression problems. MSE is an estimation of average squared differences between predicted pairs and actual (x, y) coordinate pairs. We used it uniformly with respect to PyTorch (float32), quantized (int8), and FPGA-deployed (fixed-point C++) models for making a direct comparison.

FPGA Deployment Setup:

The quantized model was then implemented in an HLS-compliant version using Vitis HLS. The version utilized fixed-point arithmetic (`ap_fixed<8,4>` for values, `ap_int<32>` for accumulations) and was implemented on an FPGA manufactured by Xilinx. Toolchains from Vivado were utilized for synthesis, IP generation, as well as hardware verification, along with Vitis.

Measurement Procedure:

- For PyTorch as well as quantized int8 models, test set MSE was calculated using standard Python scripting.
- Inference predictions were read from FPGA via AXI interfaces and offline compared against test set ground truths. Fixed-point dequantization and error computation were introduced as post-processing.

- Synthesis reports were also used in resource utilization, latency, as well as throughput while determining whether or not the design satisfied real-time needs.

This common evaluation framework consequently offered a solid foundation for gauging the performance along with precision of all model versions in the workflow.

4.2 Analysis of Results Across Software and Hardware Implementations

4.2.1 Quantitative Comparison of Models: PyTorch, Quantized, and C++

In order to compare performance and stability of TCN at different development, as well as deployment, stages, three implementations were in consideration: native TensorFlow, PyTorch reimplementation, and a fixed-point implementation using C++ for synthesis on FPGA. All three implementations used exactly the same architecture as well as hyperparameters given in Section 5.1.2, such that performance differs mainly due to framework as well as numerical precision.

Original TensorFlow Model

The reference model was ported into TensorFlow by Dr. Giorgia Subbicini [28] and optimized via Neural Architecture Search (NAS). It was a high-accuracy reference port with an MSE=0.065 m^2 in 32-bit floating-point precision. The model also acted as a reference for subsequent reimplementations' cross-checking as well as for hardware porting.

Figure 4.1 and **Figure** 4.2 illustrate the X and Y coordinate predictions, respectively, of the TensorFlow model on the test set.

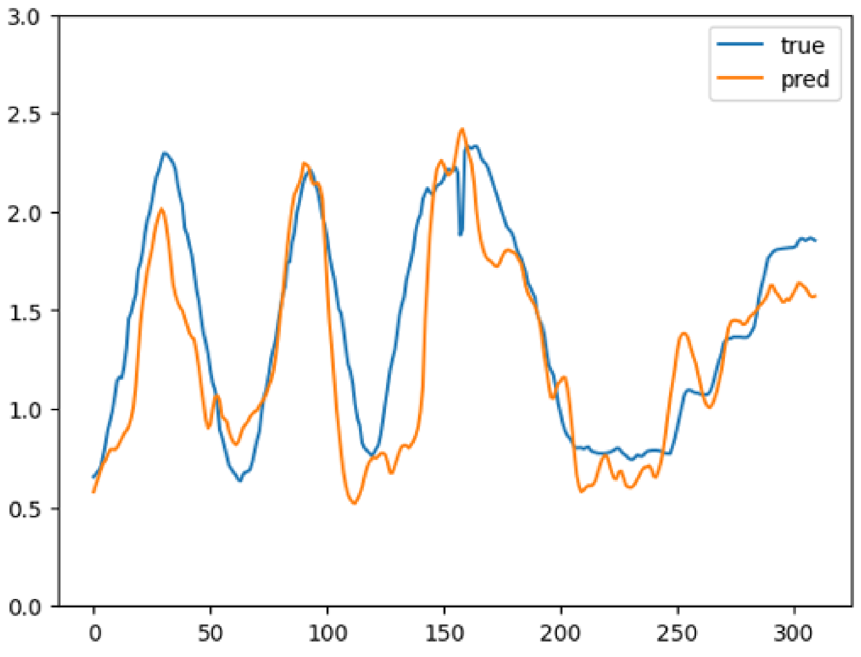

Figure 4.1. Test X coordinates – TensorFlow model

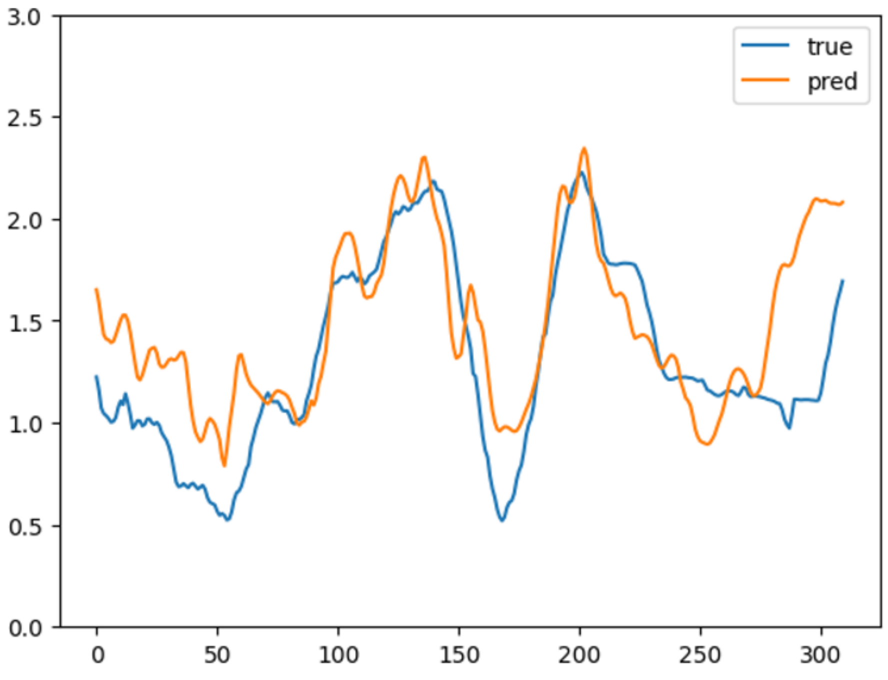

Figure 4.2. Test Y coordinates – TensorFlow model

PyTorch Reimplementation

The PyTorch version was developed to enable greater control over quantization and to facilitate export to C++ for FPGA deployment. It strictly followed the TensorFlow configuration and was trained using the same dataset and hyperparameters. The resulting model achieved an MSE of 0.1199 m^2 on the test set, maintaining comparable accuracy to the original while enabling downstream quantization and optimization.

The PyTorch model's test performance is shown in **Figure 4.3** for the X coordinates and **Figure 4.4** for the Y coordinates.

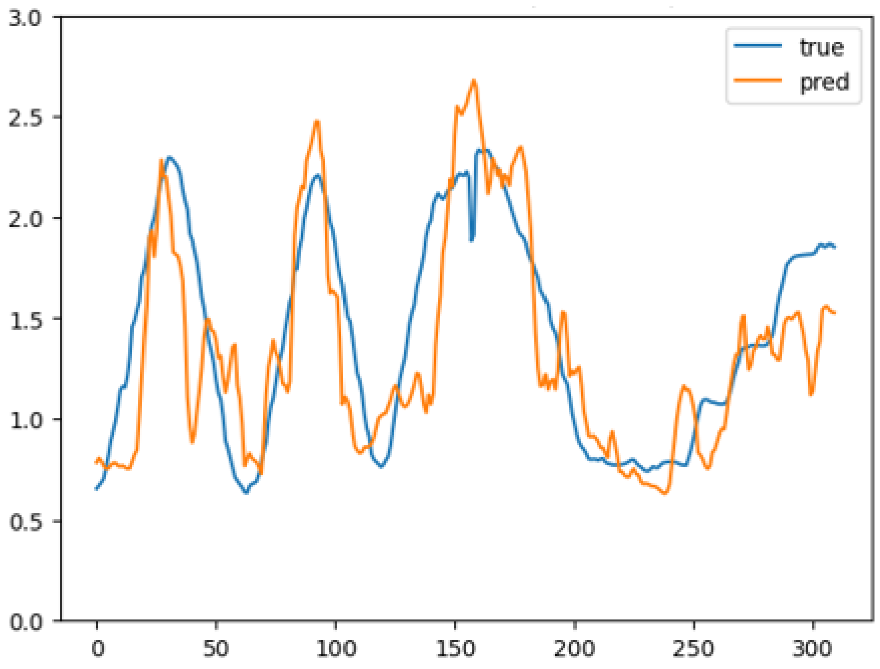

Figure 4.3. Test X coordinates – PyTorch model

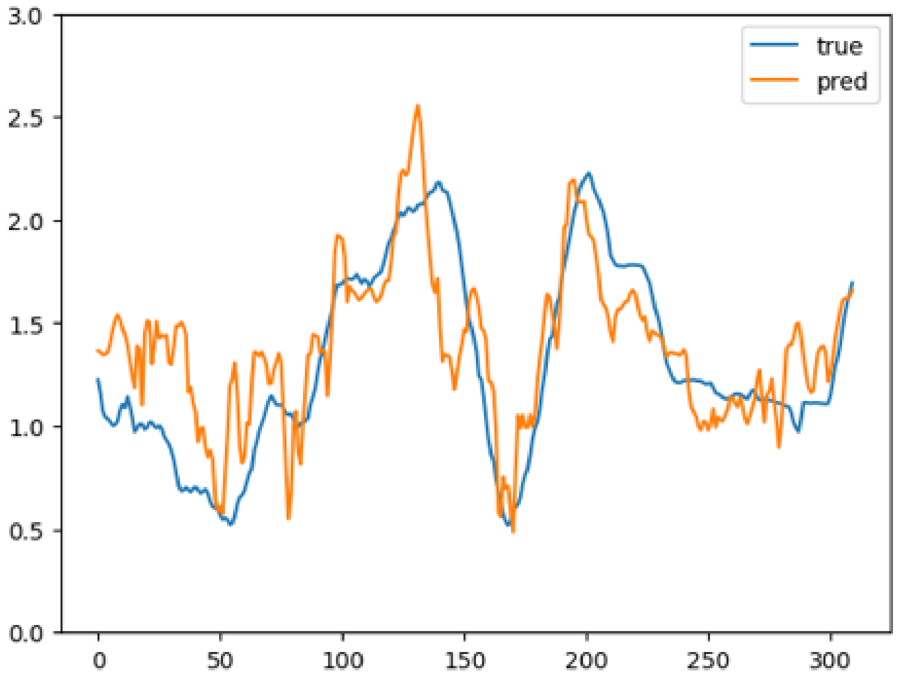

Figure 4.4. Test Y coordinates – PyTorch model

Fixed-Point C++ Model for FPGA

The final C++ implementation was a manual fixed-point reimplementation of the PyTorch model using `ap_fixed<8,4>` for 8-bit arithmetic and `ap_int<32>` accumulators. This model was designed specifically for High-Level Synthesis (HLS) using Vitis HLS and targeted at FPGA deployment. After quantization and translation, the C++ model achieved an MSE of 0.2365 m^2, reflecting the accuracy trade-offs inherent in using low-precision fixed-point arithmetic on embedded hardware.

The test predictions of the fixed-point C++ model are shown in **Figure** 4.5 (X coordinates) and **Figure** 4.6 (Y coordinates).

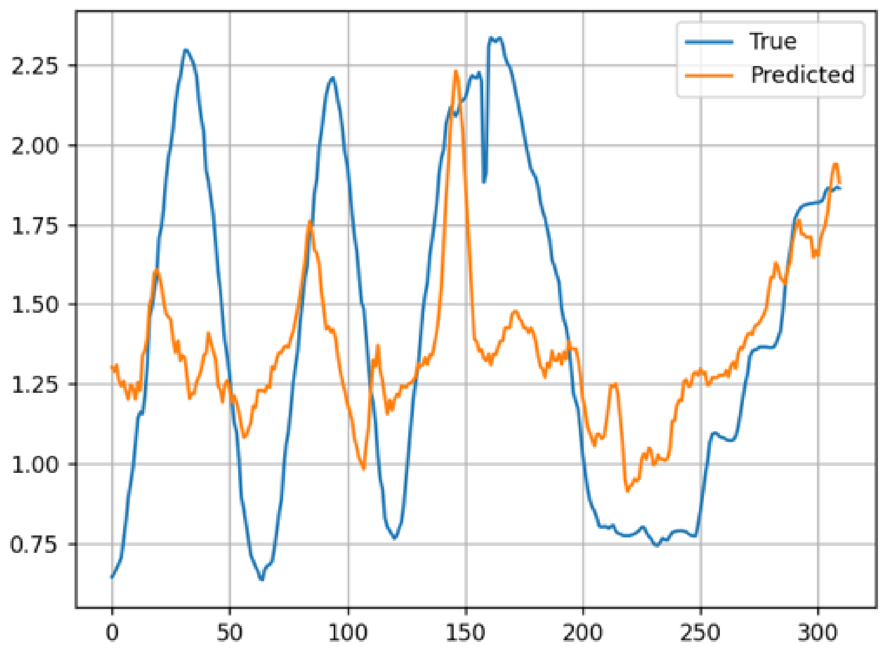

Figure 4.5. Test X coordinates – Fixed-point C++ model

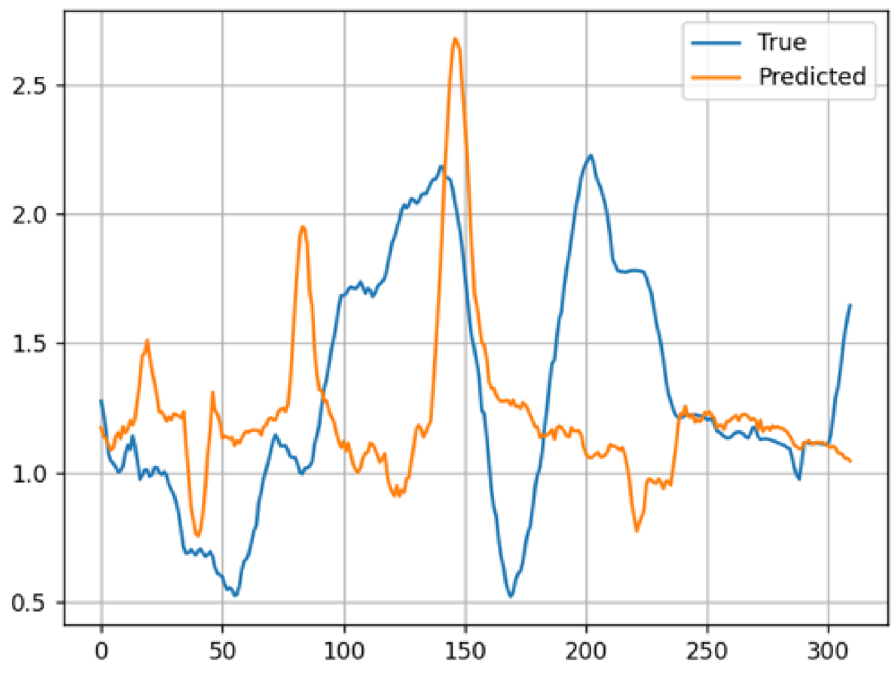

Figure 4.6. Test Y coordinates – Fixed-point C++ model

4.2.2 HLS Synthesis Results and the Effect of Code Optimizations

Initial HLS Implementation Without Pipelining

The TCN's initial model in C++ was synthesized into FPGA without any explicit pipelining directives such as `#pragma HLS dataflow` or intra-loop pipelining. Instead, the model was synthesized as a reference hardware implementation with functional correctness as well as fixed-point compatibility given higher preference than architectural optimization.

At its optimum, function `TCN_forward_fixed` with all pipeline stages repeated through in 1651 clocks, or 16.51 micro-seconds on a 100MHz clock. The function itself was not pipelined, with an initiation interval (II) identical to its latency value of 1651, such that only one sample at a time could be processed. Its sequential processing procedure limited overall throughput as well as prohibited computation overlap per input sample.

Individual residual blocks—`residual_block_fixed`, `residual_block_fixed_1`, and `residual_block_fixed_2`—were implemented as separate modules, each performing the function of two dilated convolutional layers with a single residual. Each required 401, 572, and 556 cycles, respectively. As these modules were not pipelined, i.e., they executed sequentially and waited until all computation was complete before consuming more data, there was a lot of latency introduced because they contained deeply nested convolution loops with no concurrency.

In terms of resource utilization, most of the design was in the residual blocks. The initial block (`residual_block_fixed`) utilized 112 DSPs, 6190 FFs, and 13,145 LUTs. The second one and third residual block exhibited similar trends with 122 and 92 DSPs, and 4172-5419 FFs and 8030-9564 LUTs in logic usage. These outcomes were reflective of arithmetic-heavy procedures of convolution operations as well as structural complexity from residual learning.

The other elements within the design were quite light. The `downsample_fixed`, which implemented the initial $1{\times}1$ projection convolution, was only 126 cycle-latency deep with an aperture requirement of only 3 DSPs, 1006 FFs, but 1361 LUTs. To minimize it even further, it was synthesized under automatic loop pipelining (loop auto-rew) with an II equal to 120. Similarly, the `linear_output_fixed`, calculating output regression, was 22 cycle-latency deep with an II of 2, with a very modest hardware demand of 2 DSPs, 307 FFs, but 600 LUTs.

The synthesis report also uncovered auxiliary pipeline loops within the top function, i.e., `TCN_forward_fixed_Pipeline_LOOP_92`, `LOOP_61`, and `LOOP_113`, with 69, 15, and 10-cycle latencies, respectively. These small loops handled input quantization and tensor extraction and were optimized automatically using loop rewinding, adding negligibly to total latency.

Finally, its non-pipelined HLS version acted as a functionally accurate hardware reference. Its performance, however, was constrained due to its sequential structure and lack of intra-module or between-module pipelining, especially with the high-throughput implementations. Its resource utilization was maintained modest, a result of its serialized computation as well as lack of model layer parallelism.

Partially Pipelined HLS Implementation

Selective pipelining techniques were integrated into this fixed-point C++ model in this implementation flow using Vivado-compliant directives. Incremental performance enhancement as well as latency reduction were handled through the addition of loop-level pipelining in low-complexity units without compromising on the structure integrity of the initial model. Implementation synthesis was done on Xilinx Vitis HLS, with results demonstrating improvement predominantly for low-complexity functions.

At its top, the `TCN_forward_fixed` function, responsible for controlling the entire inference process, incurred a latency of 1458 clock cycles (14.58 μs) with an initiation interval (II) of 1459 cycles. This verified that the function was not pipelined, i.e., it was processing only one input at a time. Nonetheless, it was still the largest hardware resource consumer, using 4 BRAM blocks, 157 DSP slices, 16,652 flip-flops (FFs), and 45,498 look-up tables (LUTs).

Residual blocks (`residual_block_fixed`, _1, and _3) with two dilated convolutions and residual addition were not pipelined in this stage. They all took a latency of 380-446 cycles, with the highest computational requirements to utilize 50 DSPs, 4912 FFs, and 15,851 LUTs. These blocks, despite being involved with temporal dependency mapping, could be optimized since they were still running sequentially.

Contrarily, the `downsample_fixed` module, being the initial 1×1 convolution layer to decrease input dimensionality, was pipeline-ed automatically by using automatic rewrites for loops in this module. The computation was completed within 126 cycles, using just 3 DSPs, 385 FFs, and 570 LUTs, demonstrating a power-efficient, light-weight architecture.

Similarly, the last `linear_output_fixed` transformation of fully connected output consumed only 23 cycles with low resource utilization: 2 DSPs, 346 FFs, and 800 LUTs. Both modules exhibited loop pipelining characteristics with initiation intervals of 121 and 23, respectively, characteristic of their streaming nature.

There were two other automatically pipelined loops (`TCN_forward_fixed_Pipeline_VITIS_LOOP_112_1_113_2` and _136_3) with latencies of 69 and 10 cycles, respectively. These are supporting operations like input quantization and data slicing, where optimizations at the loop.

The partially pipelined approach, on average, is a compromise between design simplicity and optimality. It achieves reasonable gains in terms of module efficiency as well as latency, particularly in respect of elementary stages, without leaving basic computation blocks behind. It serves as a stepping-stone for fully pipelined implementations as well as an introduction into performance as well as resource implications of more aggressive optimization methods to be executed in the subsequent steps.

Comparison Between Non-Pipelined and Partially Pipelined Implementations

A non-pipelined partially pipelined Temporal Convolutional Network (TCN) implementation demonstrated concrete improvement on latency as well as modularity efficiency, particularly for outer model modules. Contrast performs real gain through selective usage of loop-level pipelining, along with other potential optimizations and bottlenecks.

At a high level, total latency fell from 1651 clock cycles in a non-pipelined system down to 1458 clock cycles in a partially pipelined system—approximately an 11.7% decrease. The initiation interval (II) was still equivalent to the latency (1459), however, suggesting that optimum throughput had not yet been obtained due to the serial nature of critical-modules such as residual blocks, as shown in **Figure 4.7**.

Significant efficiency gains were realized in the `downsample_fixed` and `linear_output_fixed` modules.These were helped with auto-loop pipelining, under which they were capable of initiating new operations at every 121 and 23 cycles, as compared to sequential ones in the non-pipelined implementation. Though these optimizations were worthwhile in isolation, they were not significant in terms of making a contribution towards overall latency because, in comparison, they made a negligible contribution towards overall exection time.

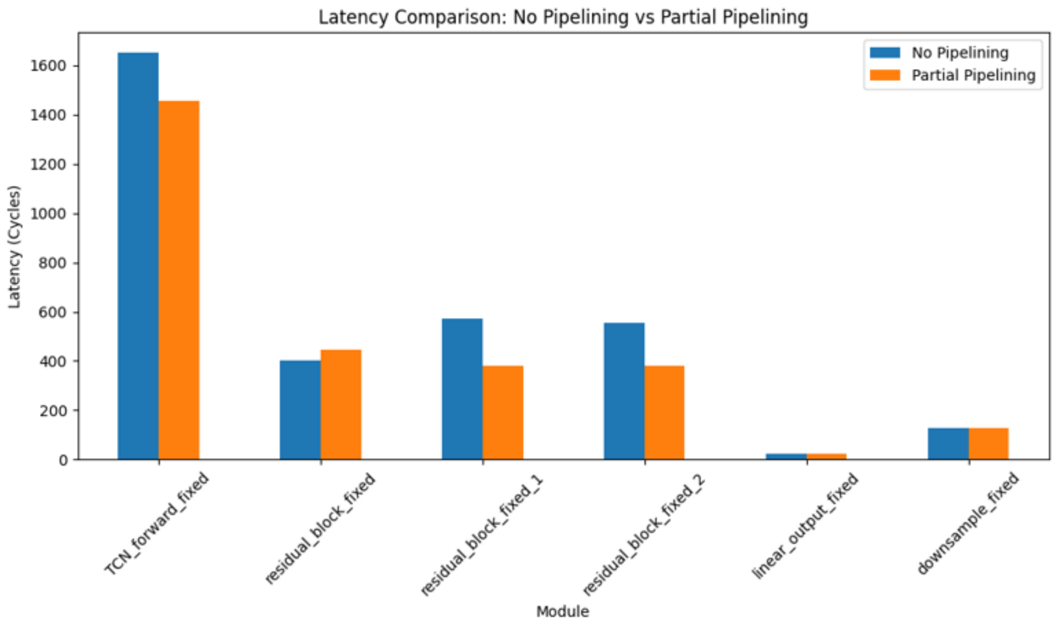

Figure 4.7. Latency Comparison: No Pipelining vs Partial Pipelining

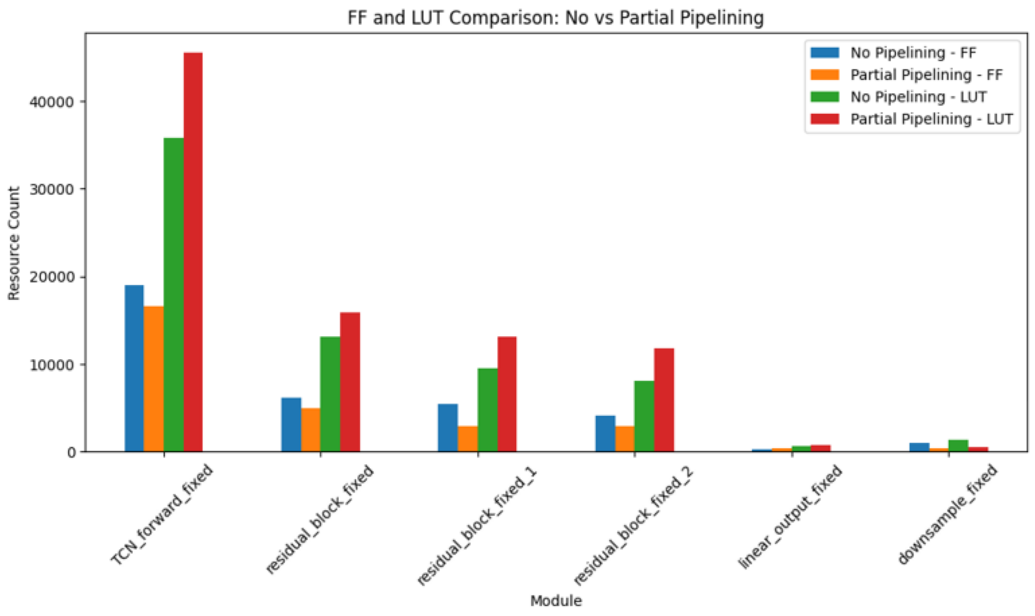

Figure 4.8. Resource Comparison: No Pipelining vs Partial Pipelining

Resource utilization also had minimal trade-offs, as is evident from **Figure 4.8**. Slice usage for DSPs reduced marginally from 376 (not pipelined) to 157 (partially pipelined) due to better utilization of computation logic within loops. FF and LUT usage were comparatively constant for the two variants—approximately 16,652 FFs vs. 19,039 FFs, and 45,498 LUTs vs. 35,859 LUTs—demonstrating no significant cost overhead for elementary pipelining.

Instead, residual blocks were not pipelined as part of these architectures. Latencies between 380-446 cycles with greater than 50 slice utilization rates in some blocks made these modules remain predominant along the end-to-end inference pipeline. Being sequential, they imposed a constraint on model initiation interval as well as curtailed further latency optimization as well as an uptick in throughput.

At a high level, partial pipelining implementation yielded modest but considerable performance improvement compared to the baseline, with significant benefits in low-complexity blocks. Limitation in pipelining between residual blocks, as well as functions, limited overall performance improvement. Results provided foundation as well as motivation for further optimization via full pipelining at intra-block as well as inter-block concurrency.

Fully Pipelined HLS Implementation

In this hardware design flow phase, fixed-point C++ TCN implementation was fully optimized with respect to pipelining using Xilinx Vitis HLS. Latency was reduced and performance was optimized using intra-module pipelining within loops as well as between-module streaming with `#pragma HLS DATAFLOW`. Results from synthesis show that the architecture presented here is well-suited for real-time high-performance inference on FPGA hardware.

In top-level function `TCN_forward_fixed`, governing the entire inference pipeline, the implementation had a 965-clock-cycle (or 9.65 μ) latency but an initiation interval (II) of 242. Whereas it could have only started inner modules sequentially, with each new input beginning its traversal through the pipeline every 242 clocks—even as previous computation continued downstream—the use of compiler directive `dataflow` freed it from this constraint. Overlapped execution has a very big impact in terms of increasing throughput, particularly in streaming or real-time scenarios.

Each of the three residual blocks—`residual_block_fixed`, `residual_block_fixed_4`, and `residual_block_fixed_7` were internally pipelined as well as externally connected with a `dataflow` interface. They had a latency of 393, 284, and 286 cycles, with respective IIs of 242, 133, and 153. These figures reflect successful intra-block pipelining, with two convolution stages as well as residual summation being executed in an overlapped manner.

Further, these residual blocks were likewise most intensive in terms of resources. They required 49 DSPs for block one, 45 for block two, and 104 DSPs for block three, mirroring dilated convolution density in multiply-accumulate operations. Flip-flop utilization ranged between 30,770 and 53,757, and LUT utilization ranged from 32,300 to 46,712, mirroring combinational as well as sequential logic needs for parallel data processing as well as for buffering.

The `downsample_fixed` block, being the initial 1×1 convolutional projection from input data, preserved its hardware efficiency with 126 cycles of latency and II of 120. It still was frugal on hardware resources with just 3 DSPs, 2,284 FFs, and 2,743 LUTs. Further, the `linear_output_fixed` function, being the last dense mapping into (x, y) coordinates, was extremely hardware efficient with only 22 cycles of latency, II of 2, and negligible resource usage.

As a whole, this pipeline-based version exploits all concurrency offered by current FPGA architectures. Each module is in a separate temporal phase, with data flowing readily across the design via block RAM and pipelined buffers. What is achieved is a low-latency, high-performing inference pipeline structurally accurate with respect to its PyTorch model equivalent but performing real-time inference in hardware.

This final optimized implementation serves as the foundation for deployment in latency-critical edge computing environments and showcases the effectiveness of combining fixed-point quantization with HLS-based pipelining and streaming techniques.

Comparison Between Partially and Fully Pipelined Implementations

The transition from partially pipelined architecture to fully pipelined architecture provided huge performance as well as hardware utilization enhancements. Though both variants made use of loop pipelining in selected modules, the fully pipelined variant employed module-level streaming using `#pragma HLS DATAFLOW`, permitting deeper concurrency as well as better Temporal Convolutional Network (TCN) performance on FPGA.

In the partially pipelined version, some, but not all, internal loops—such as those within `downsample_fixed` and `linear_output_fixed`—were pipelined, but not residual blocks or top-level dataflow. From **Figure 4.9**, it is evident that this partial pipelining obtained a total latency of 1458 cycles (14.58 μs) for top-level function `TCN_forward_fixed` with an initiation interval (II) of zero—i.e., only a single input could be processed at a time. Computationally costly residual blocks were not pipelined and contained latencies of 380–446 cycles. They were sequential with no streaming interfaces, adding significant latency to overall delay.

Contrastingly, fully pipelined version had an end-to-end processing architecture: all residual blocks internally pipelined, coupled with top-level function re-organized using `DATAFLOW` directives for streaming one module into another. Re-organization is visible with ease in considerably better performance results in **Figure 4.9**, with latency falling down to 965 cycles (9.65 μs), a reduction of 34%. In addition, initiation interval was reduced down to 242 cycles, making it a system capable of a new input acceptance every 242 cycles—improving throughput in batched as well as real-time configurations. However, this improvement in performance was at a cost, i.e., higher utilization of hardware resources. As illustrated in **Figure 4.10**, this fully pipelined architecture required significantly more resources. Flip-flop utilization went from 16,652 in the partial architecture to 178,049, LUT utilization from 45,498 to 156,405, and DSP block utilization from 157 to 203. These are because they required replication of buffers, registers, as well as control logic for parallel execution of modules.

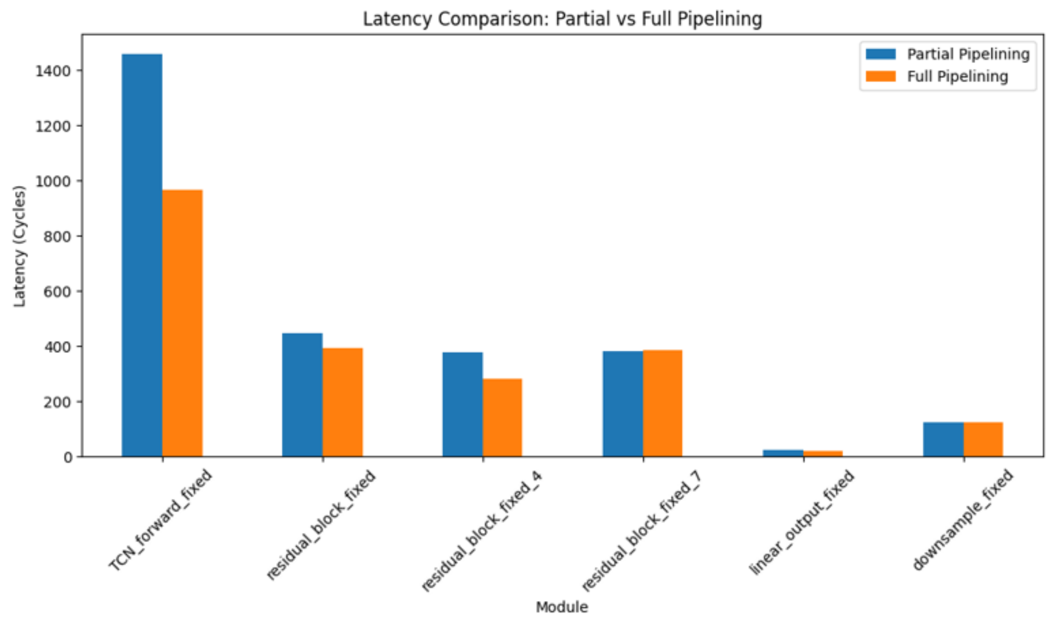

Figure 4.9. Latency Comparison: Partial vs Full Pipelining

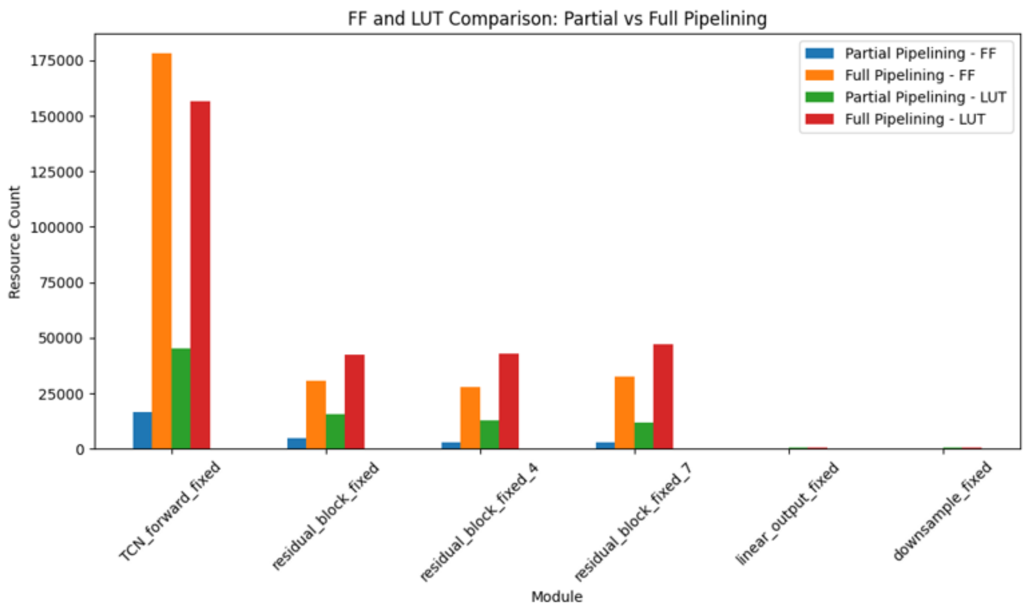

Figure 4.10. Resource Comparison: Partial vs Full Pipelining

Despite the increased resource footprint, the fully pipelined architecture offers a clear advantage in applications requiring low latency and high throughput. Its modular, streamable design is particularly well-suited for real-time environments where minimizing processing delay and maximizing parallelism are paramount. In contrast, the partially pipelined version, though more resource-efficient, lacks the parallel execution capabilities necessary for such demanding applications.

In summary, this comparison underscores a fundamental trade-off in FPGA design: **performance versus area**. The fully pipelined implementation excels in latency and throughput but requires significantly more hardware. Choosing between the two architectures ultimately depends on system constraints—specifically, the latency requirements and the available FPGA resources.

4.3 Power Efficiency

To evaluate the energy efficiency of the fixed-point TCN hardware accelerator deployed on the FPGA, a post-implementation power analysis was conducted using Vivado's `Report Power` tool. This tool performs a detailed estimation of power consumption based on the placed and routed netlist, FPGA resource utilization, and switching activity. Since no simulation-based SAIF (Switching Activity Interchange Format) file was imported, the analysis adopted a *vectorless estimation* approach, which assumes a default average toggle rate of 12.5% for all registers and nets. While this methodology is not cycle-accurate, it offers a standardized and practical baseline for comparing the relative energy demands of architectural configurations.

The analysis yielded a total on-chip power consumption of **1.871 W**, broken down into:

- **Static Power**: 0.365 W
- **Dynamic Logic Power**: 0.892 W
- **Dynamic Clock Power**: 0.422 W
- **BRAM and DSP Dynamic Power**: 0.192 W (combined)

This distribution highlights the dynamic components as the primary contributors to overall power, particularly in the logic and clock domains. This behavior is expected, given the architectural choices made during High-Level Synthesis (HLS), where *partial loop pipelining* was applied. Specifically, the inner loops of convolutional and residual operations were pipelined, enabling intermediate streaming between layers and overlapping execution across stages. However, full loop unrolling or aggressive pipelining across all layers was avoided to preserve logic and routing resources, resulting in a design that balances throughput and area without incurring extreme power overhead.

The implemented design achieved a post-routing latency of approximately **9.65 μs per inference**, which was inferred from HLS performance estimates and confirmed via Vivado timing reports. Using this latency figure, the energy per inference can be approximated as:

$$E_{\text{inference}} = P \times t = 1.871\,\text{W} \times 9.65\,\mu\text{s} \approx 18.05\,\mu\text{J} \tag{4.1}$$

This low energy footprint is a direct consequence of the fixed-point quantization, which significantly reduces computational complexity and switching activity compared to floating-point alternatives. Moreover, the design fully leverages on-chip BRAM for both input and output buffering, eliminating the need for external DRAM access during inference. This not only reduces dynamic power due to off-chip memory traffic but also contributes to tighter real-time performance guarantees by avoiding memory access latency.

In the context of edge computing, where power budgets are often limited to sub-2W operation, this implementation proves to be highly suitable. It demonstrates that FPGAs—when combined with quantization, efficient HLS strategies, and localized memory—can offer an optimal trade-off between latency, throughput, and energy consumption.

In summary, the reported power efficiency validates the practicality of deploying fixed-point TCN inference pipelines on embedded FPGAs. The results support the use of moderately pipelined, BRAM-centric architectures for real-time applications that demand both responsiveness and low energy usage in constrained environments.

Table 4.1. Post-Implementation Power and Energy Metrics

Metric	**Value**
Static Power	0.365 W
Dynamic Logic Power	0.892 W
Dynamic Clock Power	0.422 W
BRAM and DSP Dynamic Power	0.192 W
Total On-Chip Power	**1.871 W**
Energy per Inference	**18.05 μJ**

Chapter 5

Conclusion and Future Work

Conclusion

This thesis demonstrated an end-to-end flow to realizing an end-to-end Temporal Convolutional Network (TCN) model in FPGA hardware using High-Level Synthesis (HLS) for real-time indoor localization. The solution proposed in this thesis combined multiple steps such as model creation, quantization, translation to C++, hardware generation, and power estimation. Beginning with the TCN models in PyTorch, the work explored Quantization-Aware Training (QAT) as well as Post-Training Quantization (PTQ) to reduce the model complexity without a loss in performance. The entire C++ equivalent of the quantized model was realized and verified to be identical to its PyTorch equivalent with numerical co-equivalency guaranteed. Two implementations in the FPGA—a pipelined and non-pipelined one—were synthesized in Vivado HLS, their execution, latency, as well as efficiency in terms of power exhaustively compared.

Results showed the resulting FPGA-based inference engine to achieve an effective tradeoff in the form of efficiency in hardware as compared to accuracy. The TensorFlow baseline model attained an MSE value of 0.065 m^2, while the PyTorch model as well as the quantized C++ variant had MSEs of 0.1199 $m^2$and 0.2365 m^2, respectively. Though the accuracy decreased by an intermediate degree, the C++ variant effectively attained low energy usage with high velocity, which is quite conducive to edge computing applications.

Future Work

While this work lays a robust foundation, several avenues remain for future exploration:

- **QAT Deployment:** Future versions of the NN2FPGA toolchain may support TCN layers and symbolic quantization. Once available, QAT-based models could be directly deployed, potentially enhancing accuracy and reducing quantization error.
- **Multi-FPGA and SoC Integration:** Extending the current design to support multi-FPGA systems or integration with heterogeneous System-on-Chip (SoC) platforms (e.g., Zynq Ultrascale+) could improve scalability and support larger models.

- **Adaptive Reconfiguration:** Implementing dynamic partial reconfiguration could allow switching between different model configurations or resolutions on-the-fly, optimizing for accuracy or power based on real-time requirements.
- **FPGA-AI Framework Integration:** Closer integration with mainstream deep learning frameworks (e.g., PyTorch or TensorFlow) using intermediate representations like QONNX or TVM would simplify model conversion and deployment.
- **Broader Applications:** The pipeline can be generalized to support other time-series applications, such as gesture recognition, health monitoring, or anomaly detection, where temporal modeling and edge deployment are essential.

By addressing these challenges and exploring these extensions, the proposed framework can evolve into a more versatile and scalable solution for efficient neural network inference in embedded and real-time environments.

Bibliography

[1] K. Guo, L. Sui, J. Zhuang, N. Zheng, and W. Luk, “A Survey of FPGA-Based Neural Network Inference Accelerators,” *ACM Transactions on Reconfigurable Technology and Systems*, 2017.

[2] N. Suda et al., “Throughput-Optimized OpenCL-Based FPGA Accelerator for Large-Scale Convolutional Neural Networks,” *Proceedings of the ACM/SIGDA FPGA Symposium*, 2016.

[3] S. Li, C. Wu, H. Li, B. Li, and Y. Shi, “FPGA Acceleration of Recurrent Neural Network Based Language Model,” 2017.

[4] C. Zhang et al., “Going Deeper with Embedded FPGA Platform for Convolutional Neural Network,” *Proceedings of the ACM/SIGDA FPGA Symposium*, 2016, pp. 26–35.

[5] Anonymous, “HLS-Based Acceleration Framework for Deep Convolutional Neural Networks,” *IEEE Access*, vol. 7, pp. 7823–7859, 2019.

[6] N. P. Jouppi et al., “In-Datacenter Performance Analysis of a Tensor Processing Unit,” *Proceedings of the ISCA*, 2017.

[7] S. Han et al., “EIE: Efficient Inference Engine on Compressed Deep Neural Network,” *Proceedings of the ISCA*, 2016.

[8] Anonymous, “Understanding Performance Differences of FPGAs and GPUs,” *International Symposium on Field-Programmable Gate Arrays*, 2018.

[9] S. Han, J. Pool, J. Tran, and W. J. Dally, “Learning Both Weights and Connections for Efficient Neural Networks,” *NeurIPS*, 2015.

[10] Y. Umuroglu et al., “FINN: A Framework for Fast, Scalable Binarized Neural Network Inference,” *ACM/SIGDA FPGA Symposium*, 2017.

[11] C. Zhang et al., “Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks,” *ACM/SIGDA FPGA Symposium*, 2015.

[12] Y. Ma et al., “Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks,” *ACM/SIGDA FPGA Symposium*, 2017.

[13] D. Liu, T. Zhang, and L. Shi, “Dynamic Configuration of FPGA-Based CNN Accelerators for Real-Time Performance Tuning,” *ACM Transactions on Reconfigurable Technology and Systems*, 2018.

[14] N. P. Jouppi, C. Young, N. Patil, and D. Patterson, “In-Datacenter Performance Analysis of a Tensor Processing Unit,” *ISCA*, 2017.

[15] E. Nurvitadhi et al., “Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?,” *ACM/SIGDA FPGA Symposium*, 2017.

[16] N. Srinivasana et al., “Power Reduction by Clock Gating Technique,” *Journal/Conference Unknown*, Year Unknown.

[17] R. Awati, “Dynamic Voltage and Frequency Scaling (DVFS),” *TechTarget*, Year Unknown.

[18] Y.-H. Chen, J. Emer, and V. Sze, “Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks,” *IEEE/ACM*, Year Unknown.

[19] Y. Chen, T. Krishna, J. Emer, and V. Sze, “Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks,” *IEEE Journal of Solid-State Circuits*, 2017.

[20] F. Zafari, A. Gkelias, and K. K. Leung, “A Survey of Indoor Localization Systems and Technologies,” *IEEE Communications Surveys Tutorials*, Year Unknown.

[21] HardwareBee, “Top 10 FPGA Advantages,” 2 Oct. 2019. [Online]. Available: https://www.hardwarebee.com/top-10-fpga-advantages

[22] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, “FINN: A framework for fast, scalable binarized neural network inference,” *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017, pp. 65–74.

[23] S. Moini, S. Tian, J. Szefer, and D. Holcomb, “Remote Power Side-Channel Attacks on CNN Accelerators in FPGAs,” *arXiv preprint arXiv:2011.07603*, Nov. 2020.

[24] L. Kang, Y. Li, Y. Zhao, Z. Yan, and H. Li, “Hardware Trojan Attacks on Neural Networks: Theoretical Analysis and Application to FPGA-Based Accelerators,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 11, pp. 1960–1973, Nov. 2019.

[25] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and M. Violante, “An FPGA-based approach for speeding-up fault injection campaigns on safety-critical circuits,” *Journal of Electronic Testing*, vol. 18, pp. 261–271, 2002.

[26] J. Zhang, X. Chen, J. Chen, C. Chen, and M. Li, “Privacy-Preserving Machine Learning with Fully Homomorphic Encryption for Deep Neural Network on FPGA,” *IEEE Access*, vol. 7, pp. 136962–136971, 2019.

[27] Elektrologi, “Advantages of FPGAs Over Microprocessors,” *Elektrologi*, [Online]. Available: https://elektrologi.iptek.web.id/en/advantages-of-fpgas-over-microprocessors/. [Accessed: Jun. 2025].

[28] G. Subbicini, L. Lavagno, and M. T. Lazarescu, “Enhanced Exploration of Neural Network Models for Indoor Human Monitoring,” in *Proc. 9th Int. Workshop on Advances in Sensors and Interfaces (IWASI)*, 2023, pp. 1–6, doi: 10.1109/IWASI58316.2023.10164436.

[29] “Floating Point Format IEEE 754,” *Punto Flotante*. [Online]. Available: https://www.puntoflotante.net/FLOATING-POINT-FORMAT-IEEE-754.htm

[30] “Fixed-Point Representation: The Q Format and Addition Examples,” *All About Circuits*. [Online]. Available: https://www.allaboutcircuits.com/technical-articles/fixed-point-representation-the-q-format-and-addition-examples/

Appendix

C++ Model Code

```cpp
// Vivado HLS-compatible TCN model using ap_fixed Q4.4
#include <ap_fixed.h>
#include "tcn_weights_int8.h"

#define IN_CH 4
#define OUT_CH 8
#define KERNEL_SIZE 5
#define SEQ_LEN 15
#define OUTPUT_SIZE 2
#define SCALE 16

typedef ap_fixed<8, 4> fixed8_t;
typedef ap_int<32> acc_t;

int8_t relu(acc_t x) {
    return (x > 127) ? 127 : (x < 0 ? 0 : static_cast<int8_t>(x));
}

int8_t saturate(acc_t x) {
    return (x > 127) ? 127 : (x < -128 ? -128 : static_cast<int8_t>(x));
}

void conv1d_fixed(const int8_t input[][SEQ_LEN], const int8_t* weights, const int8_t* bias,
                  int8_t output[][SEQ_LEN], int in_ch, int out_ch, int dilation) {
#pragma HLS INLINE off
#pragma HLS ARRAY_PARTITION variable=input complete dim=0
#pragma HLS ARRAY_PARTITION variable=input complete dim=1
    for (int oc = 0; oc < out_ch; ++oc) {
        for (int t = 0; t < SEQ_LEN; ++t) {
#pragma HLS PIPELINE II=1
            acc_t acc = static_cast<acc_t>(bias[oc]) * SCALE;
            for (int ic = 0; ic < in_ch; ++ic) {
                for (int k = 0; k < KERNEL_SIZE; ++k) {
                    int idx = t - dilation * k;
                    int8_t val = 0;
                    if (idx >= 0)
                        val = input[ic][idx];
                    int w_idx = oc * in_ch * KERNEL_SIZE + ic * KERNEL_SIZE + k;
                    acc += static_cast<acc_t>(weights[w_idx]) * val;
                }
            }
            output[oc][t] = relu(acc >> 4);
        }
    }
}
```

```

void residual_block_fixed(int8_t input[][SEQ_LEN],
                          const int8_t* w1, const int8_t* b1,
                          const int8_t* w2, const int8_t* b2,
                          int8_t output[][SEQ_LEN], int dilation) {
#pragma HLS INLINE off
    int8_t temp[OUT_CH][SEQ_LEN];
#pragma HLS ARRAY_PARTITION variable=temp complete dim=1
    conv1d_fixed(input, w1, b1, temp, OUT_CH, OUT_CH, dilation);
    conv1d_fixed(temp, w2, b2, output, OUT_CH, OUT_CH, dilation);

    for (int oc = 0; oc < OUT_CH; ++oc) {
        for (int t = 0; t < SEQ_LEN; ++t) {
#pragma HLS PIPELINE II=1
            acc_t acc = static_cast<acc_t>(output[oc][t]) + static_cast<acc_t>(input[oc][t]);
            output[oc][t] = relu(acc);
        }
    }
}

void downsample_fixed(const int8_t input[][SEQ_LEN], const int8_t* weights, const int8_t* bias,
                      int8_t output[][SEQ_LEN]) {
#pragma HLS INLINE off
    for (int oc = 0; oc < OUT_CH; ++oc) {
        for (int t = 0; t < SEQ_LEN; ++t) {
#pragma HLS PIPELINE II=1
            acc_t acc = static_cast<acc_t>(bias[oc]) * SCALE;
            for (int ic = 0; ic < IN_CH; ++ic) {
                int w_idx = oc * IN_CH + ic;
                acc += static_cast<acc_t>(weights[w_idx]) * input[ic][t];
            }
            output[oc][t] = relu(acc >> 4);
        }
    }
}

void linear_output_fixed(const int8_t input[OUT_CH], const int8_t* weights, const int8_t* bias, f
#pragma HLS INLINE off
    for (int o = 0; o < OUTPUT_SIZE; ++o) {
        acc_t acc = static_cast<acc_t>(bias[o]) * SCALE;
        for (int i = 0; i < OUT_CH; ++i) {
            int idx = o * OUT_CH + i;
            acc += static_cast<acc_t>(weights[idx]) * input[i];
        }
        output[o] = static_cast<float>(acc) / (SCALE * SCALE);
    }
}

void TCN_forward_fixed(float input_f[IN_CH][SEQ_LEN], float output_f[OUTPUT_SIZE]) {
#pragma HLS INTERFACE s_axilite port=return bundle=CTRL
#pragma HLS INTERFACE m_axi depth=60 port=input_f offset=slave bundle=INPUT
#pragma HLS INTERFACE m_axi depth=2 port=output_f offset=slave bundle=OUTPUT

    int8_t input[IN_CH][SEQ_LEN];
    int8_t x0[OUT_CH][SEQ_LEN];
    int8_t x1[OUT_CH][SEQ_LEN];
    int8_t x2[OUT_CH][SEQ_LEN];
    int8_t x3[OUT_CH][SEQ_LEN];

#pragma HLS ARRAY_PARTITION variable=input complete dim=1
#pragma HLS ARRAY_PARTITION variable=x0 complete dim=1
#pragma HLS ARRAY_PARTITION variable=x1 complete dim=1
#pragma HLS ARRAY_PARTITION variable=x2 complete dim=1
#pragma HLS ARRAY_PARTITION variable=x3 complete dim=1

    for (int c = 0; c < IN_CH; ++c)
        for (int t = 0; t < SEQ_LEN; ++t)
#pragma HLS PIPELINE II=1
            input[c][t] = saturate(static_cast<acc_t>(input_f[c][t] * SCALE));

    downsample_fixed(input, tcn_network_0_downsample_weight, tcn_network_0_downsample_bias, x0);

    residual_block_fixed(x0,
                         tcn_network_0_conv1_weight, tcn_network_0_conv1_bias,
                         tcn_network_0_conv2_weight, tcn_network_0_conv2_bias,
                         x1, 1);

    residual_block_fixed(x1,
                         tcn_network_1_conv1_weight, tcn_network_1_conv1_bias,
                         tcn_network_1_conv2_weight, tcn_network_1_conv2_bias,
                         x2, 2);

    residual_block_fixed(x2,
                         tcn_network_2_conv1_weight, tcn_network_2_conv1_bias,
                         tcn_network_2_conv2_weight, tcn_network_2_conv2_bias,
                         x3, 4);

    int8_t last_step[OUT_CH];
#pragma HLS ARRAY_PARTITION variable=last_step complete dim=1
    for (int i = 0; i < OUT_CH; ++i)
        last_step[i] = x3[i][SEQ_LEN - 1];

    linear_output_fixed(last_step, output_layer_weight, output_layer_bias, output_f);
}

```

TCN Residual based Tensorflow Code

```python
import inspect
from typing import List

```

```
import tensorflow as tf
from keras import backend as K, Model, Input, optimizers, regularizers
from keras import layers
from keras.layers import (
    Activation, SpatialDropout1D, Lambda, Layer, Conv1D,
    Dense, BatchNormalization, LayerNormalization
)


def is_power_of_two(num: int) -> bool:
    return num != 0 and ((num & (num - 1)) == 0)


def adjust_dilations(dilations: List[int]) -> List[int]:
    return dilations if all(is_power_of_two(d) for d in dilations) else [2 ** i for i in dilation


class ResidualBlock(Layer):
    def __init__(self, dilation_rate, nb_filters, kernel_size, padding,
                 activation='relu', dropout_rate=0.0, kernel_initializer='he_normal',
                 use_batch_norm=False, use_layer_norm=False, use_weight_norm=False, **kwargs):
        super().__init__(**kwargs)
        self.dilation_rate = dilation_rate
        self.nb_filters = nb_filters
        self.kernel_size = kernel_size
        self.padding = padding
        self.activation = activation
        self.dropout_rate = dropout_rate
        self.kernel_initializer = kernel_initializer
        self.use_batch_norm = use_batch_norm
        self.use_layer_norm = use_layer_norm
        self.use_weight_norm = use_weight_norm
        self.layers = []
        self.shape_match_conv = None
        self.res_output_shape = None
        self.final_activation = None

    def _build_layer(self, layer):
        self.layers.append(layer)
        layer.build(self.res_output_shape)
        self.res_output_shape = layer.compute_output_shape(self.res_output_shape)

    def build(self, input_shape):
        self.res_output_shape = input_shape
        for k in range(2):
            conv = Conv1D(filters=self.nb_filters, kernel_size=self.kernel_size,
                          dilation_rate=self.dilation_rate, padding=self.padding,
                          name=f'conv1D_{k}', kernel_initializer=self.kernel_initializer,
                          kernel_regularizer=regularizers.l2(0.0001))
            if self.use_weight_norm:
                from keras.layers import WeightNormalization
                conv = WeightNormalization(conv)
            self._build_layer(conv)

            if self.use_batch_norm:
                self._build_layer(BatchNormalization())
            elif self.use_layer_norm:
                self._build_layer(LayerNormalization())

            self._build_layer(Activation(self.activation, name=f'Act_Conv1D_{k}'))
            self._build_layer(SpatialDropout1D(rate=self.dropout_rate, name=f'SDropout_{k}'))

        if self.nb_filters != input_shape[-1]:
            self.shape_match_conv = Conv1D(filters=self.nb_filters, kernel_size=1, padding='same'
                                           name='matching_conv1D', kernel_initializer=self.kernel
                                           kernel_regularizer=regularizers.l2(0.0001))
        else:
            self.shape_match_conv = Lambda(lambda x: x, name='matching_identity')

        self.shape_match_conv.build(input_shape)
        self.res_output_shape = self.shape_match_conv.compute_output_shape(input_shape)

        self._build_layer(Activation(self.activation, name='Act_Conv_Blocks'))
        self.final_activation = Activation(self.activation, name='Act_Res_Block')
        self.final_activation.build(self.res_output_shape)

        for layer in self.layers:
            setattr(self, layer.name, layer)
        setattr(self, self.shape_match_conv.name, self.shape_match_conv)
        setattr(self, self.final_activation.name, self.final_activation)

        super().build(input_shape)

    def call(self, inputs, training=None, **kwargs):
        x = inputs
        for layer in self.layers:
            x = layer(x, training=training) if 'training' in inspect.signature(layer.call).parame
        res = self.shape_match_conv(inputs)
        return [self.final_activation(layers.add([res, x], name='Add_Res')), x]

    def compute_output_shape(self, input_shape):
        return [self.res_output_shape, self.res_output_shape]


class TCN(Layer):
    def __init__(self, nb_filters=64, kernel_size=3, nb_stacks=1, dilations=(1, 2, 4, 8, 16, 32),
                 padding='causal', use_skip_connections=True, dropout_rate=0.0, return_sequences=
                 activation='relu', kernel_initializer='he_normal', use_batch_norm=False,
                 use_layer_norm=False, use_weight_norm=False, go_backwards=False, return_state=Fa
        super().__init__(**kwargs)

        self.nb_filters = nb_filters
        self.kernel_size = kernel_size
        self.nb_stacks = nb_stacks
        self.dilations = dilations
        self.padding = padding
        self.use_skip_connections = use_skip_connections
        self.dropout_rate = dropout_rate
        self.return_sequences = return_sequences
        self.activation_name = activation
        self.kernel_initializer = kernel_initializer
        self.use_batch_norm = use_batch_norm
        self.use_layer_norm = use_layer_norm
        self.use_weight_norm = use_weight_norm
        self.go_backwards = go_backwards
        self.return_state = return_state

        if sum([use_batch_norm, use_layer_norm, use_weight_norm]) > 1:
            raise ValueError("Only one normalization method can be used at a time.")
        if padding not in {'causal', 'same'}:
            raise ValueError("Padding must be 'causal' or 'same'.")

        self.residual_blocks = []
        self.skip_connections = []
        self.layers_outputs = []
        self.output_slice_index = None
        self.padding_same_and_time_dim_unknown = False

    @property
    def receptive_field(self):
        return 1 + 2 * (self.kernel_size - 1) * self.nb_stacks * sum(self.dilations)

    def build(self, input_shape):
        self.build_output_shape = input_shape
        self.residual_blocks = []

        for s in range(self.nb_stacks):
            for i, d in enumerate(self.dilations):
                filters = self.nb_filters[i] if isinstance(self.nb_filters, list) else self.nb_fi
                block = ResidualBlock(d, filters, self.kernel_size, self.padding,
                                      self.activation_name, self.dropout_rate,
                                      self.kernel_initializer, self.use_batch_norm,
                                      self.use_layer_norm, self.use_weight_norm,
                                      name=f'residual_block_{len(self.residual_blocks)}')
                block.build(self.build_output_shape)
                self.build_output_shape = block.res_output_shape
                self.residual_blocks.append(block)
                setattr(self, block.name, block)

        time = self.build_output_shape[1]
        self.output_slice_index = (time // 2 if time is not None else None) if self.padding == 's
        self.padding_same_and_time_dim_unknown = self.padding == 'same' and time is None

        self.slicer_layer = Lambda(lambda tt: tt[:, self.output_slice_index, :], name='Slice_Outp
        self.slicer_layer.build(self.build_output_shape.as_list())

    def call(self, inputs, training=None, **kwargs):
        x = tf.reverse(inputs, axis=[1]) if self.go_backwards else inputs
        self.layers_outputs = [x]
        self.skip_connections = []

        for block in self.residual_blocks:
            x, skip = block(x, training=training)
            self.skip_connections.append(skip)
            self.layers_outputs.append(x)

        if self.use_skip_connections:
            x = layers.add(self.skip_connections, name='Add_Skip_Connections') if len(self.skip_c
            self.layers_outputs.append(x)

        if not self.return_sequences:
            if self.padding_same_and_time_dim_unknown:
                self.output_slice_index = K.shape(x)[1] // 2
            x = self.slicer_layer(x)
            self.layers_outputs.append(x)
        return x

    def compute_output_shape(self, input_shape):
        if not self.built:
            self.build(input_shape)
        if self.return_sequences:
            return [v.value if hasattr(v, 'value') else v for v in self.build_output_shape]
        else:
            return [self.build_output_shape[0], self.build_output_shape[-1]]

    def get_config(self):
        config = super().get_config()
        config.update({
            'nb_filters': self.nb_filters,
            'kernel_size': self.kernel_size,
            'nb_stacks': self.nb_stacks,
            'dilations': self.dilations,
            'padding': self.padding,
            'use_skip_connections': self.use_skip_connections,
            'dropout_rate': self.dropout_rate,
            'return_sequences': self.return_sequences,
            'activation': self.activation_name,
            'kernel_initializer': self.kernel_initializer,
            'use_batch_norm': self.use_batch_norm,
            'use_layer_norm': self.use_layer_norm,
            'use_weight_norm': self.use_weight_norm,
            'go_backwards': self.go_backwards,
            'return_state': self.return_state
        })
        return config


def compiled_tcn(num_feat, num_classes, nb_filters, kernel_size, dilations, nb_stacks, max_len,
                 output_len=1, padding='causal', use_skip_connections=False, return_sequences=Tru
                 regression=False, dropout_rate=0.05, name='tcn', kernel_initializer='he_normal',
                 activation='relu', opt='adam', lr=0.002, use_batch_norm=False, use_layer_norm=Fa
                 use_weight_norm=False) -> Model:

    dilations = adjust_dilations(dilations)
    input_layer = Input(shape=(max_len, num_feat))
    x = TCN(nb_filters, kernel_size, nb_stacks, dilations, padding, use_skip_connections,
            dropout_rate, return_sequences, activation, kernel_initializer,
            use_batch_norm, use_layer_norm, use_weight_norm, name=name)(input_layer)

    def get_optimizer():
        if opt == 'adam':
            return optimizers.Adam(lr=lr, clipnorm=1.)
        elif opt == 'rmsprop':
            return optimizers.RMSprop(lr=lr, clipnorm=1.)
        raise ValueError("Unsupported optimizer. Use 'adam' or 'rmsprop'.")

    if regression:
        x = Dense(output_len, kernel_regularizer=regularizers.l2(0.0001))(x)
        x = Activation('linear')(x)
        model = Model(input_layer, x)
        model.compile(optimizer=get_optimizer(), loss='mean_squared_error')
    else:
        x = Dense(num_classes)(x)
        x = Activation('softmax')(x)
        model = Model(input_layer, x)

        def accuracy(y_true, y_pred):
            if K.ndim(y_true) == K.ndim(y_pred):
                y_true = K.squeeze(y_true, -1)
            y_pred_labels = K.argmax(y_pred, axis=-1)
            return K.cast(K.equal(y_true, K.cast(y_pred_labels, K.floatx())), K.floatx())

        model.compile(optimizer=get_optimizer(), loss='sparse_categorical_crossentropy', metrics=

    return model


def tcn_full_summary(model: Model, expand_residual_blocks=True):
    versions = [int(v) for v in tf.__version__.split('-')[0].split('.')]
    if versions[0] <= 2 and versions[1] < 5:
        original_layers = model._layers.copy()
        model._layers.clear()
        for layer in original_layers:
            if isinstance(layer, TCN):
                for sublayer in layer._layers:
                    if isinstance(sublayer, ResidualBlock) and expand_residual_blocks:
                        for l in sublayer._layers:
                            model._layers.append(l)
                    else:
                        model._layers.append(sublayer)
            else:
                model._layers.append(layer)
        model.summary()
        model._layers.clear()
        model._layers.extend(original_layers)
    else:
        print('WARNING: tcn_full_summary is compatible only with TensorFlow 2.5.0 or below. Use T


```

TCN Residual based Pytorch Code

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from typing import List, Union


def is_power_of_two(num: int):
    return num != 0 and ((num & (num - 1)) == 0)


def adjust_dilations(dilations: List[int]) -> List[int]:
    if all([is_power_of_two(i) for i in dilations]):
        return dilations
    return [2 ** i for i in dilations]


class Chomp1d(nn.Module):
    def __init__(self, chomp_size):
        super().__init__()
        self.chomp_size = chomp_size

    def forward(self, x):
        return x[:, :, :-self.chomp_size].contiguous()


class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, dilation,
                 activation='relu', dropout=0.05,
                 use_batch_norm=False, use_layer_norm=False, use_weight_norm=False):
```

```
        super().__init__()

        self.activation = getattr(F, activation)
        self.use_batch_norm = use_batch_norm
        self.use_layer_norm = use_layer_norm

        self.conv1 = nn.Conv1d(in_channels, out_channels, kernel_size,
                               padding=0, dilation=dilation)
        self.conv2 = nn.Conv1d(out_channels, out_channels, kernel_size,
                               padding=0, dilation=dilation)

        if use_weight_norm:
            self.conv1 = nn.utils.weight_norm(self.conv1)
            self.conv2 = nn.utils.weight_norm(self.conv2)

        self.norm1 = nn.BatchNorm1d(out_channels) if use_batch_norm else (
            nn.LayerNorm(out_channels) if use_layer_norm else nn.Identity())
        self.norm2 = nn.BatchNorm1d(out_channels) if use_batch_norm else (
            nn.LayerNorm(out_channels) if use_layer_norm else nn.Identity())

        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)

        self.downsample = nn.Conv1d(in_channels, out_channels, 1) if in_channels != out_channels

    def forward(self, x):
        # x shape: [batch, channels, time]
        residual = self.downsample(x)

        pad1 = (self.conv1.dilation[0] * (self.conv1.kernel_size[0] - 1), 0)
        out = F.pad(x, pad1)
        out = self.conv1(out)
        out = self.norm1(out.transpose(1, 2)).transpose(1, 2)
        out = self.activation(out)
        out = self.dropout1(out)

        pad2 = (self.conv2.dilation[0] * (self.conv2.kernel_size[0] - 1), 0)
        out = F.pad(out, pad2)
        out = self.conv2(out)
        out = self.norm2(out.transpose(1, 2)).transpose(1, 2)
        out = self.activation(out)
        out = self.dropout2(out)

        out += residual
        return self.activation(out), out


class TCN(nn.Module):
    def __init__(self, input_size, output_size, num_channels, kernel_size=2,
                 dropout=0.01, dilations=None, nb_stacks=1, padding_type='causal',
                 use_skip_connections=True, activation='relu',
                 use_batch_norm=False, use_layer_norm=False, use_weight_norm=False,
                 return_sequences=True, regression=False):
        super().__init__()

        if dilations is None:
            dilations = [1, 2, 4, 8, 16, 32]

        layers = []
        self.skip_connections = use_skip_connections
        self.return_sequences = return_sequences
        self.regression = regression

        in_channels = input_size
        for s in range(nb_stacks):
            for d in dilations:
                pad = (kernel_size - 1) * d if padding_type == 'causal' else (kernel_size - 1) //
                block = ResidualBlock(
                    in_channels=in_channels,
                    out_channels=num_channels,
                    kernel_size=kernel_size,
                    dilation=d,
                    activation=activation,
                    dropout=dropout,
                    use_batch_norm=use_batch_norm,
                    use_layer_norm=use_layer_norm,
                    use_weight_norm=use_weight_norm
                )
                layers.append(block)
                in_channels = num_channels

        self.network = nn.ModuleList(layers)

        self.linear = nn.Linear(num_channels, output_size)
        if not return_sequences:
            self.output_slice = -1

    def forward(self, x):
        # x: [batch, seq, features] => [batch, features, seq]
        x = x.transpose(1, 2)
        skip_outputs = []

        for block in self.network:
            x, skip = block(x)
            if self.skip_connections:
                skip_outputs.append(skip)

        if self.skip_connections:
            x = torch.stack(skip_outputs, dim=0).sum(dim=0)

        # [batch, features, seq] => [batch, seq, features]
        x = x.transpose(1, 2)

        if self.return_sequences:
            out = self.linear(x)
        else:
            out = self.linear(x[:, self.output_slice, :])

        if not self.regression:
            out = F.log_softmax(out, dim=-1)
        return out
```